

about Machine Learning

```

make_Task()
make_Learner()
make_wrappers(wrappers), resample(), tune_Param(), setHyperParams()
setHyperParams() → lmn, param_vals=tpf1n
input_Learner()

make_Paramset()
    make_Numerical_Param()
    make_Resample_Descriptor()
    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 sk
import feature_engine
from sklearn.model_selection import StratifiedKFold, GridSearchCV, RandomizedSearchCV, cross_val_score
{Train(), predict(), performance}

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

data and target split korba
lmn = {it model} for select attr
# from feature_engine.wrappers import SklearnTransformerWrapper,
select SKLearn Transformer Wrappers()
gridsearch.best_params_, gridsearch.best_estimator_()
gridsearch.best_model_.fit(x,y) → best_model_.predict(), accuracy, score
PS = {"x": [1, 2, 3]}

Stratified Kfold()
GridSearch CV() = grid search
best_model_.fit(x,y) → best_model_.predict(), accuracy, score

```

→ PCA: ~~prcomp~~ \rightarrow `pca$x[1]`, `pca$x[2]`

→ t-SNE / UMAP: SIT ~~use~~ library (tsne) | Library (umap)

("most likely" Rtsne())

perplexity = ~~width~~ width (n)
 probability (n)
 convert ~~size~~ (5-50)

`Rtsne$Y[1]`,
`Rtsne$Y[2]` \rightarrow $\theta = N(0, 0.25)$
 theta = trade off between
 Umap\$layout[1] speed and accuracy
 Umap\$layout[2] eta = learning rate

max iter = number of iterations
 too slow.

? Rtsne.defaults

→ SOM / ULE: library (som)

somGrid()
 nIterations = 100000
 xdim = 2, ydim = 2
 so som
 topo = "hexagonal"
 neighbourhood = "gaussian"
 fct method = 0
 alpha = learning rate = c(min, max)
 nlen = iteration number

grid = somGrid()

plot(som, type = ., shape)

from sklearn.decomposition import PCA
 import umap.umap_

import matplotlib.pyplot as plt

Assume 'X' is your data matrix
~~PCA~~
 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()

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)

df = ~~wizy~~ #
 as.matrix() %>%
 umpl()

n_neighbours = ~~for~~ near value
 manifold fit,
 min.dist = ~~0.1~~
 case of 0.05
 min.y = ~~0.05~~

metric = "euclidean"

n_epochs = number of iterations

? umap.defaults

doULE(df, matrix, stop = ~~500000~~, ndim = 2)

df = ~~matrix~~
 ndim = ~~2~~ dimensions
 convert = ~~0.1~~

dUle\$Y[1], pUle\$Y[2]

```
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)
```

Clustering

- analysis -

library(clvalid)
library(cluster)
library(factoextra)
library(mclust); library(fpc//dbSCAN)

clValid(df, nClust=2:6; Funcclus = "agnes",
method = "euclidean",
metric = "internal")

summary (clValid)

(optimal) → use.

Facto Extra

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

agnes/

hierarchical

dian

fviz_dend()

fviz_cluster()

(mclust)

mclust() → automatics

dbSCAN
FPC

map(2:10, function(x){

fpc::dbScan(df, eps=kNNdistplot(),
minpts=x)

)

↓ same

X=dbScan(df, minpts=) → eps នៅណា?

X → រាយជានួយ នៅពីរ select ផ្សេងៗ

cbind(pca.data, cluster=cluster %>% cluster)
(precomp(pca[,1]), pca[,2])

DL work

flow:

① train test val dataset ↗

↳ list ↗ ,

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

② Import tensorflow as tf

import tensorflow as tf

③ Import tensorflow as keras

from keras import layers

(Sequential functioning)

↳ input=keras.Input(shape=depends)

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

for i in range(n):

residual = x

n2 = layers.Dense(shape, ..., activation="relu")

n = layers.Dropout(0.5) (n)

n2 = layers.Add([feature, residual])

n2 = layers.Activation("relu") (n)

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

↳ output = layers.Dense(1, activation="linear")

↳ output = layers.Dense(1, activation="linear")

↳ output = layers.Dense(1, activation="linear")

model = keras.Model(input, output)

④ model.summary()

ekhane mone rakhbo residual er shape & dense
layer er shape same thakbe and residual network
protect 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)

```

Shape এর জুড়ে রেসিউল নেটওর্ক

```
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',
batch_size=batch_size, verbose=1)
loss='mse',
metrics=['mae'])

callbacks = [keras.callbacks.ModelCheckpoint('best.keras', save_best_only=True)]

callbacks in keras, callbacks, Early Stopping

monitor = "val1805",
ani.2015, patience = 20)

5. ~~histology~~ ~~knows~~

history, model.fit(x_train, y_train)

callbacks = [callbacks],

validation-data = (x-val, y-val),

```
tumor_model=keras.models.load_model("tumor.keras")
#tumor_model.save("tumor.h5")
final_model=keras.models.load_model("tumor.h5")
```

(batch_size=50) batch_size=50

```
best_model_mae = keras.Model.load_model
```

"Avowal" - avitovitsa (13 9253) - best. keross

+ val, mae = best_model_.mae . evaluate
encoder_inputs = Input(shape=(sequence_length, features))
x_test ,

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

Decoder
x = RepeatVector(sequence_length)(latent)

The RepeatVector layer in Keras is used to

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, replicate or "repeat" the input data across a specified number of timesteps. It's often used in sequence generation tasks, particularly in

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

— PyVision —

```
import torch
import torch.nn as nn
import optim
import torch.nn.functional as F
import torch.utils.data import DataLoader, randomsplit
from torchvision.dataset import ImageFolder
from torchvision import transforms
transform = transforms.Compose([
    transforms.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])
```

out F.00

x-train: [img[i] for img in train_dataset]
y-train: [img[i] for img in train_dataset]

train_loader = Data Loader (train_dataset,
batch_size=32,

shuffle=True)

test_loader = Data Loader (test_dataset,

batch_size=32,
shuffle=False)

val_loader = Data Loader (val_dataset,

batch_size=32,
shuffle=False)

class Residual Network (nn.Module):

def __init__(self, input_size=784,
hidden_size=128,
num_classes=10):

super(Residual Network, 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)

*parceme test
self
self
self*

*or
layer
define*

model ensemble

```
def forward(self, x):
```

*input pass
functional
unit
layer
of CNN
structure*

```
x = x.view(-1, 784) # if flatten  
out = self.fc1(x)  
out = F.relu(out)
```

then residual connection:

```
residual = out
```

```
out = self.fc2(out)
```

```
out = F.relu(out)
```

```
out += residual [out = out + residual  
size same]
```

```
out = self.fc3(out)
```

```
return out.
```

(gradient descent - back propagation)
(grad of loss w.r.t. weights)

#1. Instantiate the model, define loss and optimizer:

```
model = Residual Network
```

```
model = Residual Network().to(device)
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.Adam(model.parameters(),
```

```
lr=learning_rate)
```

```
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, train_loader,
          val_loader,
          criterion,
          optimizer,
          num_epochs):
```

- for epochs in range(num_epochs):

```
    model.train()
    running_loss = 0.0
    connect = 0
```

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

```
    for image, labels in train_loader:
```

```
        image, labels = images.to(device),
                        labels.to(device)
```

```
        gradient = optimizers.zero_grad()
        loop_start = time.time()
        model_input = model(images)
        image_input = loss_calculate = criterion(output, labels)
        loss = loss_calculate
        loss_fn = Loss.backwards()
        calculate = optimizers.step()
        optimizers.step()
        running_loss += loss.item()
        connect += 1
```

-> predicted = outputs.max(1)

total += labels.size(0)

connect += predicted.eq(labels)

• sum.1 = connect

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

val_loss, val_acc = evaluate_model(model,
                                    val_loader,
                                    criterion,
                                    print=True)

def evaluate_model(model, data_loader,
                   criterion):
    model.eval()
    running_loss = 0.0
    correct = 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 = running_loss / len(data_load_en)

accuracy = 100.0 * connect / total

return avg_loss, accuracy

train_model()

evaluate_model()

torch.save(model.state_dict(),

"xyz.pth")

Load model:

load = ResidualNetwork().to(device)

load.load_state_dict(torch.load

("xyz.pth")

model.eval() # Set model to evaluation mode

Forward pass with training=False to set the
model in inference mode

outputs = model(inputs) # Forward pass

loss_fn = criterion(outputs, targets) # Calculate the loss if needed

loss = loss_fn(targets, predictions)

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

with torch.inference_mode():

outputs = model(inputs) # Forward pass

loss = criterion(outputs, targets) # Calculate the loss if needed

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

~~train test song~~

```

#train
for epoch in range
    go model.train()
    do the forward pass
        [y_pred = model(x-train)]
    calculate the loss
        loss_fn(y_pred, ytrain)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

```

#test
go model.eval()
with torch.inference_mode():
    do the forward pass
        [test_pred = model(x-test)]
    calculate the loss
        test_loss = loss_fn(
            test_pred,
            y-test)
    print out what's happenin'

```

choosing loss_fn:

multiclass-classification : CrossEntropy Loss

Binary-classification : BCE Loss

Regression : MSE Loss

multi-label classification : MultiLabelSoftMargin 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.

```
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras import backend as K
from pycocotools.coco import COCO
```

```
train_dir = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/train'
val_dir = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/valid'
test_dir = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/test'

train_annotation_file = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/train/_annotations.coco.json'
test_annotation_file = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/test/_annotations.coco.json'
val_annotation_file = '/kaggle/input/brain-tumor-image-dataset-semantic-segmentation/valid/_annotations.coco.json'
```

```
train_coco = COCO(train_annotation_file)
val_coco = COCO(val_annotation_file)
test_coco = COCO(test_annotation_file)
```

```
def load_image_and_mask(coco, image_dir, image_id):
    image_info = coco.loadImgs(image_id)[0]
    image_path = os.path.join(image_dir, image_info['file_name'])
    image = Image.open(image_path)
    image = np.array(image)

    ann_ids = coco.getAnnIds(imgIds=image_id)
    anns = coco.loadAnns(ann_ids)
    mask = np.zeros((image_info['height'], image_info['width']))
    for ann in anns:
        mask = np.maximum(mask, coco.annToMask(ann))

    return image, mask
```

```
def create_tf_dataset(coco, image_dir, image_ids):
    def generator():
        for image_id in image_ids:
```

```
yield load_image_and_mask(coco, image_dir, image_id)

return tf.data.Dataset.from_generator(generator,
                                     output_signature=(tf.TensorSpec(shape=(None, None, 3), dtype=tf.uint8),
                                                       tf.TensorSpec(shape=(None, None), dtype=tf.uint8)))

train_dataset = create_tf_dataset(train_coco, train_dir, train_coco.getImgIds())
val_dataset = create_tf_dataset(val_coco, val_dir, val_coco.getImgIds())
test_dataset = create_tf_dataset(test_coco, test_dir, test_coco.getImgIds())
```

```
def preprocess(image, mask):

    image = tf.image.resize(image, (256, 256))

    mask = tf.expand_dims(mask, axis=-1)
    mask = tf.image.resize(mask, (256, 256))

    image = tf.cast(image, tf.float32) / 255.0

    return image, mask

train_dataset = train_dataset.map(preprocess)
val_dataset = val_dataset.map(preprocess)
test_dataset = test_dataset.map(preprocess)
```

```
def visualize_dataset(dataset, num_samples=5):
    for i, (image, mask) in enumerate(dataset.take(num_samples)):
        plt.figure(figsize=(10, 5))

        plt.subplot(1, 2, 1)
        plt.imshow(image.numpy())
        plt.title("Image")
        plt.axis("off")

        plt.subplot(1, 2, 2)
        plt.imshow(mask.numpy().squeeze(), cmap="gray")
        plt.title("Mask")
```

```
plt.axis("off")
plt.show()

visualize_dataset(train_dataset)
visualize_dataset(val_dataset)
```

BATCH_SIZE = 16
AUTOTUNE = tf.data.AUTOTUNE

```
train_dataset =
train_dataset.repeat().batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)
val_dataset = val_dataset.repeat().batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)

steps_per_epoch = len(train_coco.getImgIds()) // BATCH_SIZE
validation_steps = len(val_coco.getImgIds()) // BATCH_SIZE
test_steps = len(test_coco.getImgIds()) // BATCH_SIZE
```

```
history = model.fit(
    train_dataset,
    validation_data=val_dataset,
    epochs=5,
    steps_per_epoch=steps_per_epoch,
    validation_steps=validation_steps,
    callbacks=[early_stopping, reduce_lr]
)
```

vectorized_text = vectorizer(X.values)
Adapting the Dataset before Training

```
dataset = tf.data.Dataset.from_tensor_slices((vectorized_text, y))
dataset = dataset.cache()
dataset = dataset.shuffle(160000)
dataset = dataset.batch(16)
dataset = dataset.prefetch(8)
```

Splitting the Dataset to Train, Validation and Test Datasets

```
train = dataset.take(int(len(dataset)*.7))
val = dataset.skip(int(len(dataset)*.7)).take(int(len(dataset)*.2))
test = dataset.skip(int(len(dataset)*.9)).take(int(len(dataset)*.1))
```

```
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 condensed 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 major 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'll 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  
varaible

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):
```

*Super (tiny vgg)*

```
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)
```



```
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.

```
# Example synthetic data: 100 time steps, 5 features
np.random.seed(42)
data = np.random.rand(100, 5).astype(np.float32)
lookback = 10
# Specify which columns to use as inputs and targets
input_features = [0, 1, 2]    # e.g., columns 0,1,2 as inputs
target_features = [3, 4]      # e.g., columns 3 and 4 as targets
def create_seq2seq_dataset(data, input_feats, target_feats, lookback):
    X, y = [], []
    max_i = len(data) - lookback
    for i in range(max_i):
        X.append(data[i : i + lookback, input_feats])
        y.append(data[i + 1 : i + lookback + 1, target_feats])
    # Predict only the next step y.append(data[i + lookback])
    # Predict a sequence (e.g. next 5 days) y.append(data[i+1:i+lookback+1])
    return np.array(X), np.array(y)
# Create windowed dataset
X, y = create_seq2seq_dataset(data, input_features, target_features, lookback)
print("X shape:", X.shape) # (samples, lookback, num_input_features)
print("y shape:", y.shape) # (samples, lookback, num_target_features)
# Train-test split
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

# Create tf.data.Dataset objects with batching and shuffling
batch_size = 8
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
train_dataset = train_dataset.shuffle(1024).batch(batch_size).prefetch(tf.data.AUTOTUNE)
```

```
# train-test split for time series
train_size = int(len(timeseries) * 0.67)
test_size = len(timeseries) - train_size
train, test = timeseries[:train_size], timeseries[train_size:]

def create_dataset(dataset, lookback):
    """Transform a time series into a prediction dataset
    Args:
        dataset: A numpy array of time series, first dimension is the time steps
        lookback: Size of window for prediction
    """
    X, y = [], []
    for i in range(len(dataset)-lookback):
        feature = dataset[i:i+lookback]
        target = dataset[i+1:i+lookback+1]
        X.append(feature)
        y.append(target)
    return torch.tensor(X), torch.tensor(y)

lookback = 4
X_train, y_train = create_dataset(train,
lookback=lookback)
X_test, y_test = create_dataset(test, lookback=lookback)
loader = data.DataLoader(data.TensorDataset(X_train,
y_train), shuffle=True, batch_size=8)
```

```

-- 1. Create Sample Tables
DROP TABLE IF EXISTS employees, departments, sales;

CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary NUMERIC(10,2),
    hire_date DATE,
    dept_id INT REFERENCES departments(dept_id)
);

CREATE TABLE sales (
    sale_id SERIAL PRIMARY KEY,
    emp_id INT,
    sale_date DATE,
    amount NUMERIC(10,2)
);

-- 2. Insert Sample Data
INSERT INTO departments (dept_name) VALUES
('Marketing'),
('IT'),
('Sales');

INSERT INTO employees (first_name, last_name, salary, hire_date, dept_id) VALUES
('John', 'Doe', 60000, '2020-01-15', 1),
('Jane', 'Smith', NULL, '2021-03-20', 2),
('Bob', 'Johnson', 75000, '2019-11-01', 3);

INSERT INTO sales (emp_id, sale_date, amount) VALUES
(1, '2023-01-05', 1500),
(1, '2023-02-12', 2000),
(3, '2023-01-20', 3000),
(3, '2023-03-08', 2500);

-- 3. Basic SQL Queries
-- Simple SELECT with WHERE and ORDER BY
SELECT first_name || ' ' || last_name AS full_name, salary
FROM employees
WHERE salary > 50000
ORDER BY hire_date DESC;

-- 4. Aggregations and Grouping
-- GROUP BY with HAVING and COALESCE
SELECT

```

```

def create_seq2seq_dataset(data, input_feats,
                           target_feats, lookback, delay, horizon):
    X, y = [], []
    for i in range(len(data) - lookback - delay - horizon + 1):
        x_seq = data[i : i + lookback, input_feats]
        y_seq = data[i + lookback + delay : i + lookback + delay + horizon, target_feats]
        X.append(x_seq)
        y.append(y_seq)
    return np.array(X), np.array(y)

Args:
data: NumPy array, shape (timesteps, features)
input_feats: list of indices of input features
target_feats: list of indices of target features
lookback: input sequence length
delay: how far into the future to begin predicting
horizon: how many timesteps to predict

```

```

dept_id,
COUNT(*) AS employee_count,
COALESCE(AVG(salary), 0) AS avg_salary
FROM employees
GROUP BY dept_id
HAVING COUNT(*) > 0;

-- 5. Joins Demonstration
-- All Join Types
SELECT e.first_name, d.dept_name
FROM employees e
FULL OUTER JOIN departments d ON e.dept_id = d.dept_id;

-- 6. Subqueries and CTEs
-- Subquery in WHERE clause
SELECT first_name, last_name
FROM employees
WHERE emp_id IN (
    SELECT emp_id
    FROM sales
    WHERE amount > 2000
);

-- CTE Example
WITH sales_cte AS (
    SELECT emp_id, SUM(amount) AS total_sales
    FROM sales
    GROUP BY emp_id
)
SELECT e.first_name, s.total_sales
FROM employees e
JOIN sales_cte s ON e.emp_id = s.emp_id;

-- 7. Window Functions
-- Ranking and Running Total
SELECT
    sale_date,
    amount,
    RANK() OVER (ORDER BY amount DESC) AS sales_rank,
    SUM(amount) OVER (ORDER BY sale_date) AS running_total
FROM sales;

-- 8. Data Transformation
-- String and Date Functions
SELECT
    UPPER(first_name) || ' ' || LOWER(last_name) AS formatted_name,
    SUBSTRING(dept_name FROM 1 FOR 3) AS dept_code,
    hire_date + INTERVAL '1 year' AS anniversary_date,
    EXTRACT(YEAR FROM hire_date) AS hire_year,
    ROUND(salary, -3) AS rounded_salary
FROM employees

```

Now, I have to build a LSTM model that takes past 30 samples (i.e. past 30 hour data in this example) and predicts some or all of the parameters for the next 5 samples (i.e. next 5 hours in this example).

Or, input is from samples at t-30,t-29,t-28,...,t-1 and output is sample at t,t+1,t+2,t+3,t+4.

Also someone may need prediction of 5 samples but not immediately after instead after some delay like the next 5 samples after 3 hours. That means input is sample at t-30,t-29,t-28,...,t-1 and output is t+3,t+4,t+5,t+6 and t+7.

Let l,m and n as lag, delay and next steps respectively.

So input will be samples from t-l,t-l+1,t-l+2, ...,t-1 and output is samples from t+m,t+m+1,...,t+m+n-1.

So, one example of LSTM input will have shape of (1,l,d) and output shape will have (1,n,d)

Here, d = dimension of data (i.e. number of columns in data)

So if there are total N samples in data then the input and output shape of the entire data set would be (k,l,d) and (k,n,d) respectively.

Where, k = N-(l+m+n)

```
JOIN departments USING (dept_id);
```

```

class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 8, 3, padding=1)
        self.batchnorm1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(8, 8, 3, padding=1)
        self.batchnorm2 = nn.BatchNorm2d(8)
        self.pool2 = nn.MaxPool2d(2)
        self.conv3 = nn.Conv2d(8, 32, 3, padding=1)
        self.batchnorm3 = nn.BatchNorm2d(32)
        self.conv4 = nn.Conv2d(32, 32, 3, padding=1)
        self.batchnorm4 = nn.BatchNorm2d(32)
        self.pool4 = nn.MaxPool2d(2)
        self.conv5 = nn.Conv2d(32, 128, 3, padding=1)
        self.batchnorm5 = nn.BatchNorm2d(128)
        self.conv6 = nn.Conv2d(128, 128, 3, padding=1)
        self.batchnorm6 = nn.BatchNorm2d(128)
        self.pool6 = nn.MaxPool2d(2)
        self.conv7 = nn.Conv2d(128, 2, 1)
        self.pool7 = nn.AvgPool2d(3)

    def forward(self, x):
        #-----
        # INPUT
        #-----
        x = x.view(-1, 3, 32, 32)

        #-----
        # LAYER 1
        #-----
        output_1 = self.conv1(x)
        output_1 = F.relu(output_1)
        output_1 = self.batchnorm1(output_1)

        #-----
        # LAYER 2
        #-----
        output_2 = self.conv2(output_1)
        output_2 = F.relu(output_2)
        output_2 = self.pool2(output_2)
        output_2 = self.batchnorm2(output_2)

        #-----
        # LAYER 3
        #-----
        output_3 = self.conv3(output_2)
        output_3 = F.relu(output_3)
        output_3 = self.batchnorm3(output_3)

        #-----
        # LAYER 4
        #-----
        output_4 = self.conv4(output_3)
        output_4 = F.relu(output_4)
        output_4 = self.pool4(output_4)
        output_4 = self.batchnorm4(output_4)

        #-----
        # LAYER 5
        #-----
```

During training use model.checkpoint()
after training use model.save()

```

output_5 = self.conv5(output_4)
output_5 = F.relu(output_5)
output_5 = self.batchnorm5(output_5)

#-----
# LAYER 6
#-----
output_6 = self.conv6(output_5)
output_6 = F.relu(output_6)
output_6 = self.pool6(output_6)
output_6 = self.batchnorm6(output_6)

#-----
# OUTPUT LAYER
#-----
output_7 = self.conv7(output_6)
output_7 = self.pool7(output_7)
output_7 = output_7.view(-1, 2)

return F.softmax(output_7, dim=1)

```

```

def train(network, training_set, batch_size, optimizer, loss_function):
    """
    This function optimizes the convnet weights
    """
    # creating list to hold loss per batch
    loss_per_batch = []

    # defining dataloader
    train_loader = DataLoader(training_set, batch_size)

    # iterating through batches
    print('training...')
    for images, labels in tqdm(train_loader):
        #-----
        # sending images to device
        #-----
        images, labels = images.to(device), labels.to(device)

        #-----
        # zeroing optimizer gradients
        #-----
        optimizer.zero_grad()

        #-----
        # classifying instances
        #-----
        classifications = network(images)

        #-----
        # computing loss/how wrong our classifications are
        #-----
        loss = loss_function(classifications, labels)
        loss_per_batch.append(loss.item())

        #-----
        # computing gradients/the direction that fits our objective
        #-----
        loss.backward()

```

```
#-----  
# optimizing weights/slightly adjusting parameters  
#-----  
optimizer.step()  
print('all done!')  
  
return loss_per_batch
```

```
def validate(network, validation_set, batch_size, loss_function):  
    """  
    This function validates convnet parameter optimizations  
    """  
    # creating a list to hold loss per batch  
    loss_per_batch = []  
  
    # defining model state  
    network.eval()  
  
    # defining dataloader  
    val_loader = DataLoader(validation_set, batch_size)  
  
    print('validating...')  
    # preventing gradient calculations since we will not be optimizing  
    with torch.no_grad():  
        # iterating through batches  
        for images, labels in tqdm(val_loader):  
            #-----  
            # sending images and labels to device  
            #-----  
            images, labels = images.to(device), labels.to(device)  
  
            #-----  
            # making classifications  
            #-----  
            classifications = network(images)  
  
            #-----  
            # computing loss  
            #-----  
            loss = loss_function(classifications, labels)  
            loss_per_batch.append(loss.item())  
  
    print('all done!')  
  
    return loss_per_batch
```

```
def accuracy(network, dataset):  
    """  
    This function computes accuracy  
    """  
    # setting model state  
    network.eval()  
  
    # instantiating counters  
    total_correct = 0
```

```

total_instances = 0

# creating dataloader
dataloader = DataLoader(dataset, 64)

# iterating through batches
with torch.no_grad():
    for images, labels in tqdm(dataloader):
        images, labels = images.to(device), labels.to(device)

        #-----
        # making classifications and deriving indices of maximum value via argmax
        #-----
        classifications = torch.argmax(network(images), dim=1)

        #-----
        # comparing indicies of maximum values and labels
        #-----
        correct_predictions = sum(classifications==labels).item()

        #-----
        # incrementing counters
        #-----
        total_correct+=correct_predictions
        total_instances+=len(images)
    return round(total_correct/total_instances, 3)

```

```

# instantiating model
model = ConvNet()
# defining optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=3e-4)

# training/optimizing parameters
training_losses = train(network=model, training_set=training_set,
                        batch_size=64, optimizer=optimizer,
                        loss_function=nn.CrossEntropyLoss())

# validating optimizations
validation_losses = validate(network=model, validation_set=validation_set,
                            batch_size=64, loss_function=nn.CrossEntropyLoss())

# deriving model accuracy on the training set
training_accuracy = accuracy(model, training_set)
print(f'training accuracy: {training_accuracy}')

# deriving model accuracy on the validation set
validation_accuracy = accuracy(model, validation_set)
print(f'validation accuracy: {validation_accuracy}')

```