Lampiran 3 Source Code

```
!pip install torchinfo

# Data Handling

import pandas as pd

import numpy as np

# Data Visualization

import matplotlib.pyplot as plt

import seaborn as sns

from PIL import Image

import cv2

# Preprocessing

from sklearn.model_selection import train_test_split

# Torch

import torch

from torch.utils.data import Dataset, DataLoader

from torch import nn, optim

from torchinfo import summary

from torchvision.models import vit_b_16, ViT_B_16_Weights

# Metrics

from sklearn.metrics import accuracy_score

from sklearn.metrics import confusion_matrix

# os

import os

# Path

from pathlib import Path
```

```
# tqdm

from tqdm.auto import tqdm

# OrderedDict

from collections import OrderedDict

# random

import random

# warnings

import warnings

warnings.filterwarnings("ignore")

import os

from google.colab import drive

# Mount Google Drive

drive.mount('/content/drive')

# Path dataset

data_path = Path('/content/drive/MyDrive/penyakit-daun-
padi/penyakit-daun-padi/Train)

data_path_list = list(data_path.glob("*/*.jpg"))

print(f'Total Images = {len(data_path_list)}')

classes = os.listdir(data_path)

classes = sorted(classes)

print('==' * 20)

print(' ' * 10, f'Total Classes = {len(classes)}')

print('==' * 20)

for c in classes:

total_images_class =
list(Path(os.path.join(data_path,c)).glob("*.jpg"))

print(f'* {c}: {len(total_images_class)} images')
```

```
NUM_IMAGES = 3

fig,ax = plt.subplots(nrows = len(classes), ncols = NUM_IMAGES,
figsize = (10,30))

p = 0

for c in classes:

total_images_class =
list(Path(os.path.join(data_path,c)).glob("*.jpg"))

images_selected = random.choices(total_images_class, k =
NUM_IMAGES)

for i,img_path in enumerate(images_selected):

img_bgr = cv2.imread(str(img_path))

img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

ax[p,i].imshow(img_rgb)

ax[p,i].axis('off')

ax[p,i].set_title(f'Class: {c}\nShape: {img_rgb.shape}')

p += 1

fig.tight_layout()

fig.show()

images_path = [None] * len(data_path_list)

labels = [None] * len(data_path_list)

for i,img_path in enumerate(data_path_list):

images_path[i] = img_path

labels[i] = img_path.parent.stem

df_path_and_label = pd.DataFrame({'path':images_path,

'label':labels})

df_path_and_label.head()

SEED = 42
```

```
df_train, df_rest = train_test_split(df_path_and_label, test_size
= 0.3,

random_state = SEED, stratify = df_path_and_label["label"])

df_valid, df_test = train_test_split(df_rest, test_size = 0.5,

random_state = SEED, stratify = df_rest["label"])

# We have to define the mapping of the classes to convert the
labels to numbers.

label_map = dict(zip(classes, range(len(classes))))

label_map

# Now we define the transformations that we are going to apply.

weights = ViT_B_16_Weights.DEFAULT

auto_transforms = weights.transforms()

auto_transforms

class CustomImageDataset(Dataset):

def __init__(self, df:pd.DataFrame, label_map:dict, transforms):

self.df = df

self.label_map = label_map

self.transforms = transforms

def __len__(self):

return len(self.df)

def __getitem__(self,idx):

df_new = self.df.copy()

df_new = df_new.reset_index(drop = True)

df_new["label"] = df_new["label"].map(self.label_map)

image_path = df_new.iloc[idx, 0]

image = Image.open(image_path).convert("RGB")

image = self.transforms(image)

label = df_new.iloc[idx, 1]
```

```
return image,label



train_dataset = CustomImageDataset(df_train, label_map,
auto_transforms)

valid_dataset = CustomImageDataset(df_valid, label_map,
auto_transforms)



BATCH_SIZE = 8

NUM_WORKERS = os.cpu_count(

train_dataloader = DataLoader(dataset = train_dataset,

batch_size = BATCH_SIZE,

shuffle = True,

num_workers = NUM_WORKERS)

valid_dataloader = DataLoader(dataset = valid_dataset,

batch_size = BATCH_SIZE,

shuffle = True,

num_workers = NUM_WORKERS)

# Let's visualize the dimensions of a batch.

batch_images, batch_labels = next(iter(train_dataloader))

batch_images.shape, batch_labels.shape

# Augmentasi Data

from torchvision import transforms

# Augmentasi dan Normalisasi

data_transforms = transforms.Compose([

transforms.RandomRotation(degrees=45), # Increase rotation angle

transforms.RandomHorizontalFlip(),

transforms.RandomVerticalFlip(),
```

```
transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.2), # Increase magnitude

transforms.RandomResizedCrop(224, scale=(0.7, 1.0)), # Wider range
for crop scale

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),

])

# GPU

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

device

# VIT 16

model = vit_b_16(weights = weights)

output_shape = len(classes) # total classe

model.heads = nn.Sequential(OrderedDict([('head',
nn.Linear(in_features = 768,

out_features = len(classes))])))

for param in model.parameters():

param.requires_grad = False

for param in model.heads.parameters():

param.requires_grad = True

# Let's see if the parameters were frozen.

summary(model = model,

input_size = [8,3,224,224],

col_width = 15,

col_names = ["input_size",
"output_size","num_params","trainable"],

row_settings = ["var_names"])
```

```
for param in model.conv_proj.parameters():

param.requires_grad = False

for param in model.encoder.parameters():

param.requires_grad = False

# One last time let's take a look if the last layer was modified.

summary(model = model,

input_size = [8,3,224,224],

col_width = 15,

col_names = ["input_size",
"output_size","num_params","trainable"],

row_settings = ["var_names"])

loss_fn = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr = 0.001)

def train_step(model:torch.nn.Module,
dataloader:torch.utils.data.DataLoader,

loss_fn:torch.nn.Module, optimizer:torch.optim.Optimizer):

model.train()

train_loss = 0.

train_accuracy = 0.

for batch, (X,y) in enumerate(dataloader):

X,y = X.to(device), y.to(device)

optimizer.zero_grad()

y_pred_logit = model(X)

loss = loss_fn(y_pred_logit, y)

train_loss += loss.item()

loss.backward()

optimizer.step()

y_pred_prob = torch.softmax(y_pred_logit, dim = 1)
```

```
y_pred_class = torch.argmax(y_pred_prob, dim = 1)

train_accuracy += accuracy_score(y.cpu().numpy(),
y_pred_class.detach().cpu().numpy())

train_loss = train_loss / len(dataloader)

train_accuracy = train_accuracy / len(dataloader)

return train_loss, train_accuracy

def save_checkpoint(filename, model, epoch, loss, optimizer,
metric):

state = {'filename':filename,

'model':model.state_dict(),

'epoch':epoch,

'loss':loss,

'optimizer':optimizer.state_dict(),

'metric':metric}

torch.save(state, filename)

def valid_step(model:torch.nn.Module,
dataloader:torch.utils.data.DataLoader,

loss_fn:torch.nn.Module):

model.eval()

valid_loss = 0.

valid_accuracy = 0.

with torch.inference_mode():

for batch, (X,y) in enumerate(dataloader):

X,y = X.to(device), y.to(device)

y_pred_logit = model(X)

loss = loss_fn(y_pred_logit, y)

valid_loss += loss.item()

y_pred_prob = torch.softmax(y_pred_logit, dim = 1)
```

```
y_pred_class = torch.argmax(y_pred_prob, dim = 1)

valid_accuracy += accuracy_score(y.cpu().numpy(),
y_pred_class.detach().cpu().numpy())

valid_loss = valid_loss / len(dataloader)

valid_accuracy = valid_accuracy / len(dataloader)

return valid_loss, valid_accuracy

def train(model:torch.nn.Module,
train_dataloader:torch.utils.data.DataLoader,

valid_dataloader:torch.utils.data.DataLoader,
loss_fn:torch.nn.Module,

optimizer:torch.optim.Optimizer, epochs:int = 10, patience:int =
5):

results = {"train_loss":[], "train_accuracy":[], "valid_loss":[],
"valid_accuracy":[]}

best_valid_loss = float('inf')

epochs_no_improve = 0 # Untuk melacak jumlah epoch tanpa
peningkata

for epoch in tqdm(range(epochs)):

train_loss, train_accuracy = train_step(model = model, dataloader
= train_dataloader,

loss_fn = loss_fn, optimizer = optimizer)

valid_loss, valid_accuracy = valid_step(model = model, dataloader
= valid_dataloader,

loss_fn = loss_fn)

if valid_loss < best_valid_loss:

best_valid_loss = valid_loss

file_name = "best_model.pth"

save_checkpoint(file_name, model, epoch, best_valid_loss,
optimizer, valid_accuracy)

epochs_no_improve = 0 # Reset jumlah epoch tanpa peningkatan

else:

epochs_no_improve += 1 # Tambahkan jumlah epoch tanpa peningkatan
```

```
if epochs_no_improve == patience:

print(f'Early stopping triggered after {patience} epochs without
improvement.')

break

print(f'Epoch: {epoch + 1} | ',

f'Train Loss: {train_loss:.4f} | ',

f'Train Accuracy: {train_accuracy:.4f} | ',

f'Valid Loss: {valid_loss:.4f} | ',

f'Valid Accuracy: {valid_accuracy:.4f}')

results["train_loss"].append(train_loss)

results["train_accuracy"].append(train_accuracy)

results["valid_loss"].append(valid_loss)

results["valid_accuracy"].append(valid_accuracy)

return results

# Training!!

EPOCHS = 50

torch.cuda.manual_seed(SEED)

torch.manual_seed(SEED)

MODEL_RESULTS = train(model, train_dataloader, valid_dataloader,
loss_fn, optimizer, EPOCHS)

# Function to plot the loss and metric during each training epoch.

def loss_metric_curve_plot(model_results:dict):

train_loss = model_results["train_loss"]

valid_loss = model_results["valid_loss"]

train_accuracy = [float(value) for value in
model_results["train_accuracy"]]

valid_accuracy = [float(value) for value in
model_results["valid_accuracy"]]

fig,axes = plt.subplots(nrows = 1, ncols = 2, figsize = (10,4))
```

```
axes = axes.flat

axes[0].plot(train_loss, color = "red", label = "Train")

axes[0].plot(valid_loss, color = "blue", label = "Valid",
linestyle = '--')

axes[0].spines["top"].set_visible(False)

axes[0].spines["right"].set_visible(False)

axes[0].set_title("CrossEntropyLoss", fontsize = 12, fontweight =
"bold", color = "black")

axes[0].set_xlabel("Epochs", fontsize = 10, fontweight = "bold",
color = "black")

axes[0].set_ylabel("Loss", fontsize = 10, fontweight = "bold",
color = "black")

axes[0].legend()

axes[1].plot(train_accuracy, color = "red", label = "Train")

axes[1].plot(valid_accuracy, color = "blue", label = "Valid",
linestyle = '--')

axes[1].spines["top"].set_visible(False)

axes[1].spines["right"].set_visible(False)

axes[1].set_title("Metric of performance: Accuracy", fontsize =
12, fontweight = "bold", color = "black")

axes[1].set_xlabel("Epochs", fontsize = 10, fontweight = "bold",
color = "black")

axes[1].set_ylabel("Score", fontsize = 10, fontweight = "bold",
color = "black")

axes[1].legend()

fig.tight_layout()

fig.show()

loss_metric_curve_plot(MODEL_RESULTS)

def predictions(test_dataloader:torch.utils.data.DataLoader)

checkpoint = torch.load('/content/best_model.pth')

loaded_model = vit_b_16()
```

```
loaded_model.heads =
nn.Sequential(OrderedDict([('head',nn.Linear(in_features = 768,

out_features = output_shape))]))

loaded_model.load_state_dict(checkpoint["model"]

loaded_model.to(device)

loaded_model.eval()

y_pred_test = []

with torch.inference_mode():

for X,_ in tqdm(test_dataloader):

X = X.to(device)

y_pred_logit = loaded_model(X)

y_pred_prob = torch.softmax(y_pred_logit, dim = 1)

y_pred_class = torch.argmax(y_pred_prob, dim = 1)

y_pred_test.append(y_pred_class.detach().cpu())

y_pred_test = torch.cat(y_pred_test)

return y_pred_tes

test_dataset = CustomImageDataset(df_test, label_map,
auto_transforms)

test_dataloader = DataLoader(dataset = test_dataset,

batch_size = BATCH_SIZE,

shuffle = False,

num_workers = NUM_WORKERS)

y_pred_test = predictions(test_dataloader)

print(f'Accuracy Test =
{round(accuracy_score(df_test["label"].map(label_map),
y_pred_test.numpy()), 4)}'

confusion_matrix_test =
confusion_matrix(df_test["label"].map(label_map),
y_pred_test.numpy())

fig,ax = plt.subplots(figsize = (10,4.5))
```

```
sns.heatmap(confusion_matrix_test,

cmap = 'Oranges',

annot = True,

annot_kws = {"fontsize":9, "fontweight":"bold"},

linewidths = 1.2,

fmt = ' ',

linecolor = "white",

square = True,

xticklabels = classes,

yticklabels = classes,

cbar = False,

ax = ax)

ax.set_title("Confusion Matrix Test", fontsize = 15, fontweight =
"bold", color = "darkblue")

ax.tick_params('x',rotation = 90)

fig.show()



from sklearn.metrics import classification_report



def evaluate_model(model, dataloader):

model.eval()

y_true = []

y_pred = []

with torch.no_grad():

for images, labels in dataloader:

images = images.to(device)

labels = labels.to(device)
```

```
outputs = model(images)

_, predicted = torch.max(outputs, 1)

y_true.extend(labels.cpu().numpy())

y_pred.extend(predicted.cpu().numpy())

# Print classification report

print("\nClassification Report:")

print(classification_report(y_true, y_pred, target_names=classes))

# Evaluasi model dengan data validasi

evaluate_model(model, valid_dataloader)



# Function to load the trained model and move it to the
appropriate device

def load_model(model_path, device):

model = vit_b_16(weights=ViT_B_16_Weights.DEFAULT)

model.heads = nn.Sequential(OrderedDict([('head',
nn.Linear(in_features=768, out_features=len(classes)))]))

checkpoint = torch.load(model_path, map_location=device) # Ensure
model is loaded to the correct device

model.load_state_dict(checkpoint['model'])

return model.to(device) # Move model to the appropriate device

# Function to preprocess the image and move it to the appropriate
device

def preprocess_image(image_path):

image = Image.open(image_path).convert("RGB")

transform = transforms.Compose([

transforms.Resize((224, 224)), # Resize to match model input size

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
```

```
])

input_tensor = transform(image).unsqueeze(0) # Add batch dimension

return input_tensor, image

# Function to perform inference on single image

def predict_image(image_path, model, device):

# Preprocess the image and move it to the appropriate device

input_tensor, image = preprocess_image(image_path)

input_tensor = input_tensor.to(device)

# Convert model parameters to the same data type as input tensor

model.to(input_tensor.dtype)

# Set model to evaluation mode

model.eval()

# Perform inference

with torch.no_grad():

output = model(input_tensor)

probabilities = torch.softmax(output, dim=1)

predicted_class = torch.argmax(probabilities, dim=1).item()

predicted_label = classes[predicted_class]

return predicted_label, image

# Load the trained model

model_path = '/content/model-vit.pt' # Path to the trained model

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

loaded_model = load_model(model_path, device)

# Input image path for prediction

image_path = "/content/drive/MyDrive/penyakit-daun-padi/penyakit-
daun-padi/test/Blast(361).jpg"

# Perform prediction
```

```
predicted_label, image = predict_image(image_path, loaded_model,
device)

# Display the image with predicted label

plt.imshow(image)

plt.title(f'Predicted Label: {predicted_label}')

plt.axis('off')

plt.show()
```