# Machine Learning
# LABORATORY: Transformer Homework

|  |  |
|---|---|
| **NAME:** | **STUDENT ID#:** |

## Objectives:

- The goal of this assignment is to deepen your understanding of Transformer-based architectures by implementing a Vision Transformer (ViT) model and applying it to a defect detection dataset.
- You will use the ViT model to classify images of industrial defects, treating each image as a sequence of patches.
- Through this assignment, you will:
  - Understand how the Vision Transformer encodes image patches and uses self-attention to capture global relationships.
  - Implement the Vision Transformer architecture using PyTorch.
  - Train the ViT model on a real-world defect dataset.
  - Evaluate the model's classification performance on unseen test data.

## Part 1. Instruction

In this assignment, you will implement a Vision Transformer (ViT) model for image classification using PyTorch, without using any prebuilt ViT modules or high-level Transformer libraries (e.g., no torchvision.models.vit, no timm.create_model, etc.).

You are required to:

- Manually implement a patch embedding step, dividing each image into non-overlapping patches and flattening them into input sequences.
- Build the core Transformer encoder, including multi-head self-attention, feedforward layers, layer normalization, and residual connections, as described in the original ViT paper.
- Apply a classification head to predict defect categories from the final Transformer outputs.
- Train the model on the provided defect image dataset using a manual training loop.

You should refer to:

- The lecture slide Transformer (page 34 and earlier) for an overview of Vision Transformer design.
- The paper: "An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale" [Dosovitskiy et al., 2020, https://arxiv.org/pdf/2010.11929 ] to understand how ViT works, how to embed image patches, how positional encodings are applied, and how to design the Transformer layers.

You will be provided with the **raw defect dataset**. You are responsible for splitting the dataset yourself into 70% training and 30% testing before starting model training.

General ViT Workflow

- Patch embedding: Split the input image into fixed-size patches, flatten each patch, and apply a linear projection to get patch embeddings.
- Positional encoding: Add learnable positional embeddings to retain spatial order.
- Transformer encoder: Apply multi-head self-attention and feedforward layers across the sequence of patches.

- Classification head: Use the [CLS] token output (or mean pooling) followed by an MLP head to predict the final class label.

After implementing the full forward pass, you must train the model on the defect training set and Evaluate its classification accuracy on the unseen test set.

**Hyperparameter Tuning**

You are required to adjust the hyperparameters (e.g., learning rate, batch size, number of epochs, etc) to achieve the best possible classification accuracy and help your model reach high performance.

Visualization and Reporting

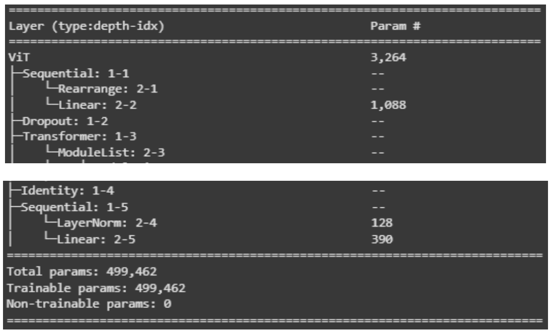
In your report, you must include the following:

(a) A screenshot of the model summary and the total number of parameters.

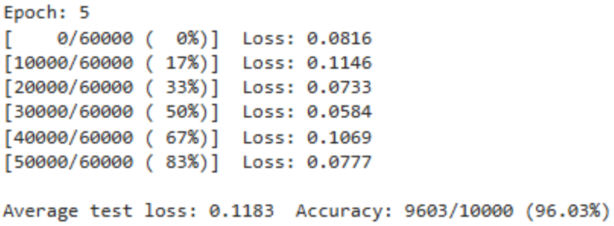
(b) A screenshot of the final training result, showing total epochs completed, final test loss, and accuracy.

(c) A confusion matrix visualizing the classification performance.

(d) 24 example predictions from the test set, with 4 images shown per class.

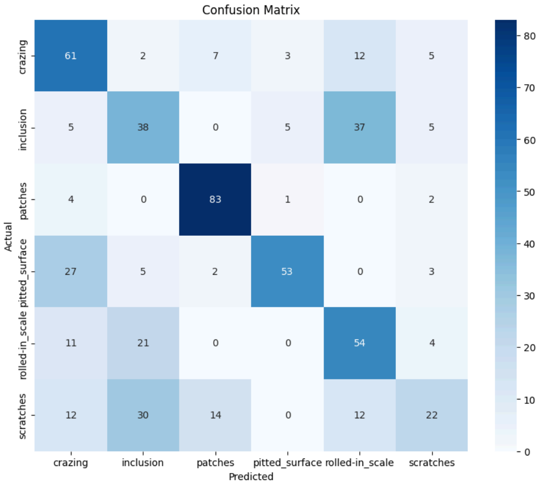


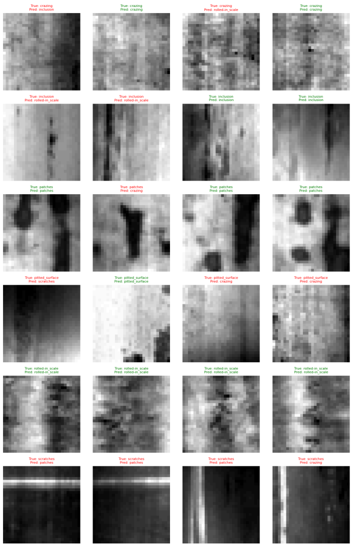
```
==========================================================================================
Layer (type:depth-idx)                                       Param #
==========================================================================================
ViT                                                          3,264
├─Sequential: 1-1                                            --
│    └─Rearrange: 2-1                                        --
│    └─Linear: 2-2                                           1,088
├─Dropout: 1-2                                               --
├─Transformer: 1-3                                           --
│    └─ModuleList: 2-3                                       --

├─Identity: 1-4                                              --
├─Sequential: 1-5                                            --
│    └─LayerNorm: 2-4                                        128
│    └─Linear: 2-5                                           390
==========================================================================================
Total params: 499,462
Trainable params: 499,462
Non-trainable params: 0
==========================================================================================
```



```
Epoch: 5
[    0/60000 (  0%)]  Loss: 0.0816
[10000/60000 ( 17%)]  Loss: 0.1146
[20000/60000 ( 33%)]  Loss: 0.0733
[30000/60000 ( 50%)]  Loss: 0.0584
[40000/60000 ( 67%)]  Loss: 0.1069
[50000/60000 ( 83%)]  Loss: 0.0777

Average test loss: 0.1183  Accuracy: 9603/10000 (96.03%)
```

**(a)** **(b)**





**(c)** **(d)**

## Part 2. Code Template

| Step | Procedure |
|---|---|
| 1 | |

```
# ============================
# Step 1: Unzip Dataset
# ============================

import zipfile
import os

# TODO: Set the correct uploaded ZIP filename
zip_path = 'dataset.zip'  # <-- replace with the exact uploaded filename
extract_path = './dataset'  # folder where you want to extract

# Unzip the dataset
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)

print(f"✅ Unzipped to: {os.path.abspath(extract_path)}")

# Optional: List the extracted folder contents
print("Contents:")
print(os.listdir(extract_path))

# ============================
# Step 2: Split Dataset into Train/Test
# ============================

import shutil
import random

# TODO: Set source folder (where unzipped dataset is) and target folder (where split dataset will go)
SOURCE_DIR = 'dataset'  # folder from unzip
TARGET_DIR = 'dataset_split'  # new folder to store split data

# Split ratios
train_ratio = 0.7
test_ratio = 0.3  # note: you can calculate this as 1 - train_ratio if needed

# Set random seed for reproducibility
random.seed(42)

# Create target train/test directories per class
for split in ['train', 'test']:
    for class_name in os.listdir(SOURCE_DIR):
        os.makedirs(os.path.join(TARGET_DIR, split, class_name), exist_ok=True)

# Process each class folder
for class_name in os.listdir(SOURCE_DIR):
    class_path = os.path.join(SOURCE_DIR, class_name)
    if not os.path.isdir(class_path):
        continue

    images = os.listdir(class_path)
    random.shuffle(images)

    # Calculate split point
    train_cutoff = int(len(images) * train_ratio)

    # Split images
    train_images = images[:train_cutoff]
    test_images = images[train_cutoff:]

    # Copy training images
    for img_name in train_images:
        src = os.path.join(class_path, img_name)
        dst = os.path.join(TARGET_DIR, 'train', class_name, img_name)
        shutil.copyfile(src, dst)

    # Copy testing images
    for img_name in test_images:
        src = os.path.join(class_path, img_name)
        dst = os.path.join(TARGET_DIR, 'test', class_name, img_name)
        shutil.copyfile(src, dst)

print("✅ Dataset split complete!")
```

2

```
import os
import torch
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Set dataset paths (update if needed)
TRAIN_PATH = 'dataset_split/train'
TEST_PATH = 'dataset_split/test'

# Define image transforms: resize, grayscale, tensor, normalize
transform_custom = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Load datasets
train_set = torchvision.datasets.ImageFolder(root=TRAIN_PATH, transform=transform_custom)
test_set = torchvision.datasets.ImageFolder(root=TEST_PATH, transform=transform_custom)

# Print dataset info
print("Classes:", train_set.classes)
print("Train samples:", len(train_set))
print("Test samples:", len(test_set))

# Show example images (2 per class)
```

```
def show_2x6_grid(dataset, n_per_class=2, title="Example Grid"):
    class_counts = {i: 0 for i in range(len(dataset.classes))}
    collected = {i: [] for i in range(len(dataset.classes))}

    for img, label in dataset:
        if class_counts[label] < n_per_class:
            collected[label].append(img)
            class_counts[label] += 1
        if all(c >= n_per_class for c in class_counts.values()):
            break

    fig, axes = plt.subplots(n_per_class, len(dataset.classes), figsize=(len(dataset.classes)*2, n_per_class*2))
    for col, imgs in collected.items():
        for row in range(n_per_class):
            ax = axes[row][col] if n_per_class > 1 else axes[col]
            img = imgs[row].numpy().transpose(1, 2, 0) * 0.5 + 0.5  # unnormalize
            ax.imshow(img.squeeze(), cmap='gray')
            ax.set_title(dataset.classes[col], fontsize=8)
            ax.axis('off')

    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

# Show training and test grids
show_2x6_grid(train_set, 2, "Train Set Grid")
show_2x6_grid(test_set, 2, "Test Set Grid")
```

3

```
import torch
import torch.nn as nn
from einops import rearrange, repeat, einsum

# === Helper ===
def pair(t):
    return t if isinstance(t, tuple) else (t, t)

# === TODO 1: Define PreNorm block ===
class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        # TODO: initialize LayerNorm and store fn

    def forward(self, x, **kwargs):
        # TODO: apply LayerNorm + fn
        pass

# === TODO 2: Define FeedForward block ===
class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout=0.):
        super().__init__()
        # TODO: define two Linear layers + activation + dropout

    def forward(self, x):
        # TODO: apply layers
        pass
```

```
# === TODO 3: Define Attention block ===
class Attention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=64, dropout=0.):
        super().__init__()
        # TODO: set up qkv projections, softmax attention, final output projection

    def forward(self, x):
        # TODO: compute q, k, v, attention, and output
        pass

# === TODO 4: Define Transformer Encoder ===
class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout=0.):
        super().__init__()
        # TODO: stack multiple PreNorm + Attention + FeedForward layers

    def forward(self, x):
        # TODO: pass through each Transformer layer
        pass

# === TODO 5: Define Vision Transformer (ViT) ===
class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads,
                 mlp_dim, pool='cls', channels=1, dim_head=64, dropout=0., emb_dropout=0.):
        super().__init__()
        # TODO: calculate patch numbers, set up patch embedding, positional embedding, cls token,
transformer, mlp head

    def forward(self, img):
        # TODO: apply patch embedding, add cls token + pos embedding, run transformer, pool, mlp head
        pass

# === TODO 6: Initialize model ===
# Example hyperparameters (students should adjust!)
model = ViT(
    image_size=28,
    patch_size=4,
    num_classes=6,
    channels=1,
    dim=64,
    depth=6,
    heads=4,
    mlp_dim=128
)

# === TODO 7: Set up optimizer ===
import torch.optim as optim
optimizer = optim.Adam(model.parameters(), lr=0.003)

# === TODO 8: Print or summarize the model ===
print(model)
# Optionally: from torchsummary import summary
# summary(model, input_size=(1, 28, 28))
```

4

```
# === TODO 1: Define training epoch ===
def train_epoch(model, optimizer, data_loader, loss_history):
    model.train()
    total_samples = len(data_loader.dataset)

    for i, (data, target) in enumerate(data_loader):
        # TODO: Zero gradients
        # TODO: Forward pass
        # TODO: Compute loss
        # TODO: Backward pass and optimizer step

        if i % 100 == 0:
            # TODO: Print progress info and save loss
            pass

# === TODO 2: Define evaluation function ===
def evaluate(model, data_loader, loss_history):
    model.eval()
    total_samples = len(data_loader.dataset)
    correct_samples = 0
    total_loss = 0

    with torch.no_grad():
        for data, target in data_loader:
            # TODO: Forward pass
            # TODO: Compute loss
            # TODO: Get predictions and count correct samples

    # TODO: Compute average loss and accuracy
    # TODO: Print evaluation summary
```

5

```
# === SET EPOCHS ===
N_EPOCHS = 1
# === START TIMER ===
start_time = time.time()

# === INIT LOSS TRACKERS ===
train_loss_history, test_loss_history = [], []

# === MAIN TRAINING LOOP ===
for epoch in range(1, N_EPOCHS + 1):
    print('Epoch:', epoch)
    train_epoch(model, optimizer, train_loader, train_loss_history)
    evaluate(model, test_loader, test_loss_history)

# === PRINT TOTAL TIME ===
print('Execution time:', '{:5.2f}'.format(time.time() - start_time), 'seconds')

# === SAVE TRAINED MODEL ===
torch.save(model.state_dict(), 'Student_ID.pth')  # replace with your actual Student ID
print("✅ Model saved as .pth")
```

```
#LOAD MODEL - if needed
```

```
# Make sure you define the same ViT model structure first
model = ViT(image_size=28, patch_size=4, num_classes=6, channels=1, dim=64, depth=6, heads=4,
mlp_dim=128)

# Load saved weights
model.load_state_dict(torch.load('Student_ID.pth'))
model.eval()

print("✅ Model loaded and ready for testing")
```

6

```
import torch

# === TODO: Define function to plot confusion matrix ===
def plot_confusion_matrix(model, data_loader, class_names):
    # HINT:
    # - Get predictions and true labels
    # - Compute confusion matrix (sklearn)
    # - Plot with seaborn heatmap
    pass

# === TODO: Define function to plot example predictions ===
def plot_classwise_predictions(model, data_loader, class_names, samples_per_class=4):
    # HINT:
    # - Collect a few correct/incorrect predictions per class
    # - Plot grid of images with true vs predicted labels
    pass

# === TODO: After training, call both functions ===
# plot_confusion_matrix(model, test_loader, train_set.classes)
# plot_classwise_predictions(model, test_loader, train_set.classes, samples_per_class=4)
```

## Grading Assignment & Submission

**Implementation:**

1. **(35%) Vision Transformer Model Design:** Implement the full ViT model from scratch, including all required components, without using any prebuilt ViT libraries; code must compile and run without critical errors.
2. **(15%) Training and Hyperparameter Tuning:** Train the model on the defect dataset and tune hyperparameters to achieve at least 60% test accuracy, with higher accuracy earning more points.
3. **(5%) Evaluation Function:** Write a correct evaluation loop to compute and report the test set's accuracy and loss.
4. **(10%) Visualization:** Provide a clear confusion matrix plot and 24 example predictions (4 per class) showing both true and predicted labels.
5. **(5%) Report Quality:** Submit a well-organized report summarizing your implementation, results, and analysis.

**Question:**

6. **(5%)** Briefly explain the role of patch embedding and positional encoding in ViT. You may include parts of your Step 3 model code to support your explanation.

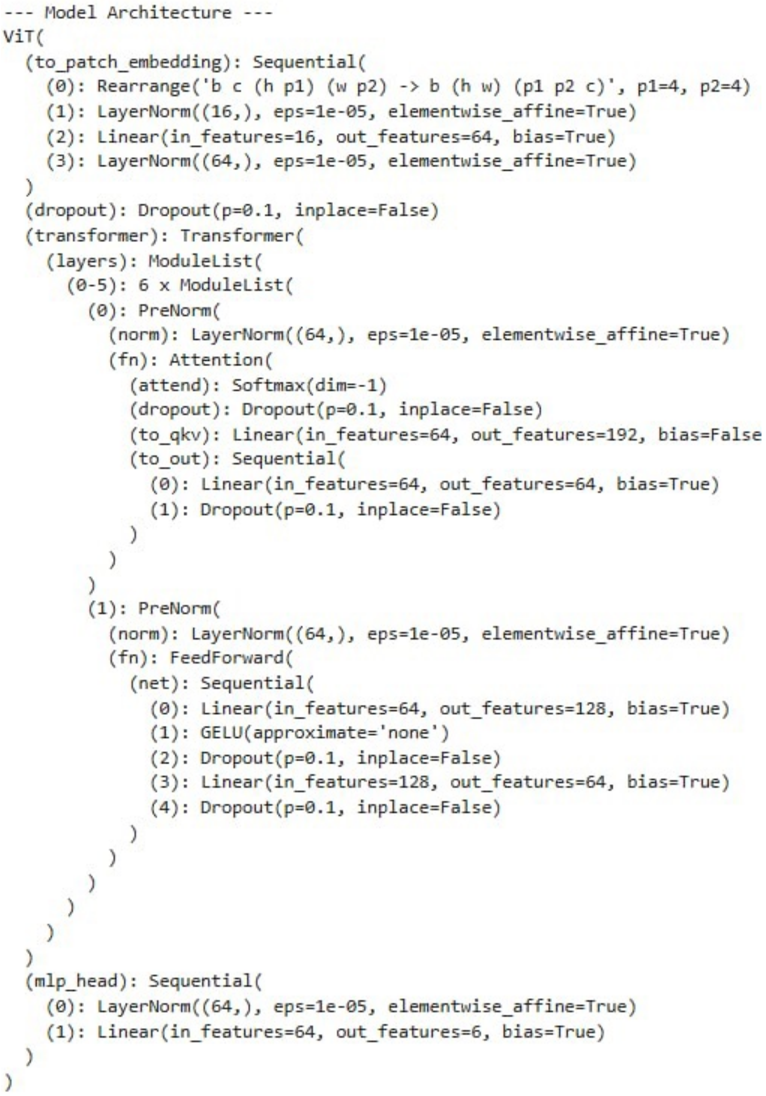
7. **(5%)** Describe which hyperparameters you tuned, why you chose them, and how they affected your final accuracy.
8. **(5%)** Compare ViT and CNN for image classification: what are the main differences, and when might one be preferred over the other using the provided dataset?
9. **(5%)** Report the final achieved test accuracy; explain whether you reached the >60% baseline — if not, describe what you tried and why it might have failed; if you did, explain how you achieved it.
10. (10%) Based on the paper you referred to (Dosovitskiy et al., An Image is Worth 16x16 Words), please brief explain: *You may include parts of your Step 3 model code to support your explanation.*
    a. How does the Vision Transformer process input images from start to finish?
    b. How are the image patches divided and transformed into input sequences?
    c. How does the multi-head self-attention mechanism operate within the Transformer encoder?
    d. How does the model use the [CLS] token (or final output) to produce the final image classification?

## Submission :

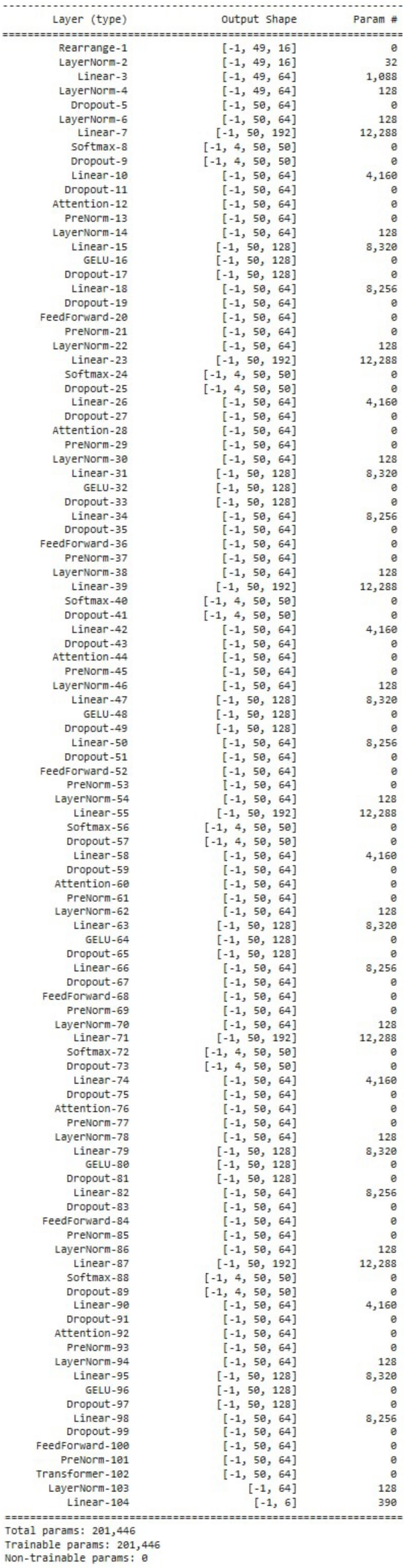
1. Upload both your report, code, and trained model to the E3 system (Lab8 Homework). Name your files correctly as follows:
    a. Report: StudentID_Lab8_Homework.pdf
    b. Code: StudentID_Lab8_Homework.py or StudentID_Lab8_Homework.ipynb
    c. Trained Model: StudentID_Lab8_Homework.pth
2. ⚠ Important: Make sure all three files are uploaded. Missing even one will result in no grade for this assignment.
3. Deadline: May 13 - 21:00 PM
4. Plagiarism is **strictly prohibited**. Submitting copied work from other students will result in penalties.
5. References: You must include any references or external sources you used when working on this assignment in your report.

## Results and Discussion:

(a) model summary.

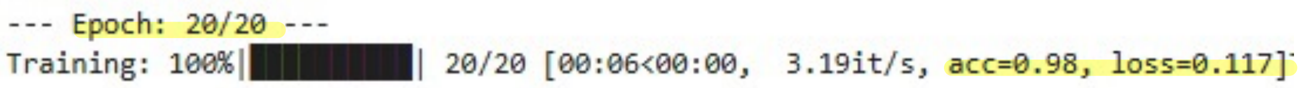


```
--- Model Architecture ---
ViT(
  (to_patch_embedding): Sequential(
    (0): Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=4, p2=4)
    (1): LayerNorm((16,), eps=1e-05, elementwise_affine=True)
    (2): Linear(in_features=16, out_features=64, bias=True)
    (3): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (transformer): Transformer(
    (layers): ModuleList(
      (0-5): 6 x ModuleList(
        (0): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): Attention(
            (attend): Softmax(dim=-1)
            (dropout): Dropout(p=0.1, inplace=False)
            (to_qkv): Linear(in_features=64, out_features=192, bias=False
            (to_out): Sequential(
              (0): Linear(in_features=64, out_features=64, bias=True)
              (1): Dropout(p=0.1, inplace=False)
            )
          )
        )
        (1): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): FeedForward(
            (net): Sequential(
              (0): Linear(in_features=64, out_features=128, bias=True)
              (1): GELU(approximate='none')
              (2): Dropout(p=0.1, inplace=False)
              (3): Linear(in_features=128, out_features=64, bias=True)
              (4): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
    )
  )
  (mlp_head): Sequential(
    (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
    (1): Linear(in_features=64, out_features=6, bias=True)
  )
)
```



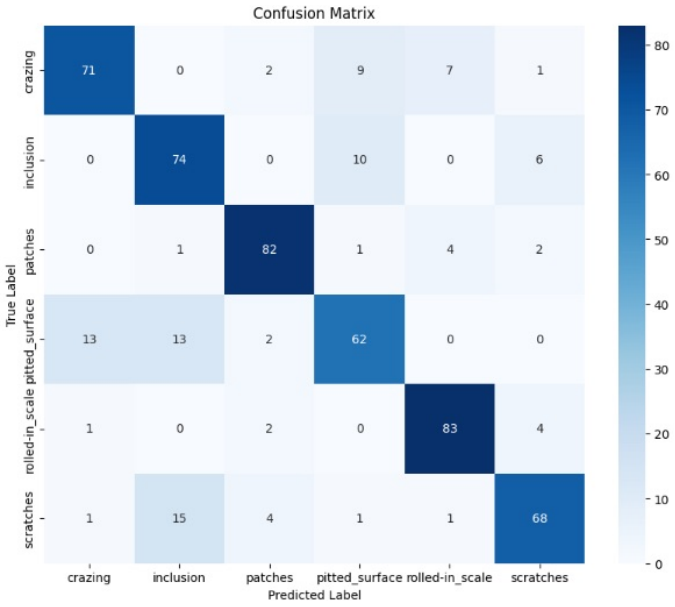
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
         Rearrange-1               [-1, 49, 16]               0
         LayerNorm-2               [-1, 49, 16]              32
            Linear-3               [-1, 49, 64]           1,088
         LayerNorm-4               [-1, 49, 64]             128
           Dropout-5               [-1, 50, 64]               0
         LayerNorm-6               [-1, 50, 64]             128
            Linear-7              [-1, 50, 192]          12,288
           Softmax-8            [-1, 4, 50, 50]               0
           Dropout-9            [-1, 4, 50, 50]               0
           Linear-10               [-1, 50, 64]           4,160
          Dropout-11               [-1, 50, 64]               0
        Attention-12               [-1, 50, 64]               0
          PreNorm-13               [-1, 50, 64]               0
        LayerNorm-14               [-1, 50, 64]             128
           Linear-15              [-1, 50, 128]           8,320
             GELU-16              [-1, 50, 128]               0
          Dropout-17              [-1, 50, 128]               0
           Linear-18               [-1, 50, 64]           8,256
          Dropout-19               [-1, 50, 64]               0
      FeedForward-20               [-1, 50, 64]               0
          PreNorm-21               [-1, 50, 64]               0
        LayerNorm-22               [-1, 50, 64]             128
           Linear-23              [-1, 50, 192]          12,288
          Softmax-24            [-1, 4, 50, 50]               0
          Dropout-25            [-1, 4, 50, 50]               0
           Linear-26               [-1, 50, 64]           4,160
          Dropout-27               [-1, 50, 64]               0
        Attention-28               [-1, 50, 64]               0
          PreNorm-29               [-1, 50, 64]               0
        LayerNorm-30               [-1, 50, 64]             128
           Linear-31              [-1, 50, 128]           8,320
             GELU-32              [-1, 50, 128]               0
          Dropout-33              [-1, 50, 128]               0
           Linear-34               [-1, 50, 64]           8,256
          Dropout-35               [-1, 50, 64]               0
      FeedForward-36               [-1, 50, 64]               0
          PreNorm-37               [-1, 50, 64]               0
        LayerNorm-38               [-1, 50, 64]             128
           Linear-39              [-1, 50, 192]          12,288
          Softmax-40            [-1, 4, 50, 50]               0
          Dropout-41            [-1, 4, 50, 50]               0
           Linear-42               [-1, 50, 64]           4,160
          Dropout-43               [-1, 50, 64]               0
        Attention-44               [-1, 50, 64]               0
          PreNorm-45               [-1, 50, 64]               0
        LayerNorm-46               [-1, 50, 64]             128
           Linear-47              [-1, 50, 128]           8,320
             GELU-48              [-1, 50, 128]               0
          Dropout-49              [-1, 50, 128]               0
           Linear-50               [-1, 50, 64]           8,256
          Dropout-51               [-1, 50, 64]               0
      FeedForward-52               [-1, 50, 64]               0
          PreNorm-53               [-1, 50, 64]               0
        LayerNorm-54               [-1, 50, 64]             128
           Linear-55              [-1, 50, 192]          12,288
          Softmax-56            [-1, 4, 50, 50]               0
          Dropout-57            [-1, 4, 50, 50]               0
           Linear-58               [-1, 50, 64]           4,160
          Dropout-59               [-1, 50, 64]               0
        Attention-60               [-1, 50, 64]               0
          PreNorm-61               [-1, 50, 64]               0
        LayerNorm-62               [-1, 50, 64]             128
           Linear-63              [-1, 50, 128]           8,320
             GELU-64              [-1, 50, 128]               0
          Dropout-65              [-1, 50, 128]               0
           Linear-66               [-1, 50, 64]           8,256
          Dropout-67               [-1, 50, 64]               0
      FeedForward-68               [-1, 50, 64]               0
          PreNorm-69               [-1, 50, 64]               0
        LayerNorm-70               [-1, 50, 64]             128
           Linear-71              [-1, 50, 192]          12,288
          Softmax-72            [-1, 4, 50, 50]               0
          Dropout-73            [-1, 4, 50, 50]               0
           Linear-74               [-1, 50, 64]           4,160
          Dropout-75               [-1, 50, 64]               0
        Attention-76               [-1, 50, 64]               0
          PreNorm-77               [-1, 50, 64]               0
        LayerNorm-78               [-1, 50, 64]             128
           Linear-79              [-1, 50, 128]           8,320
             GELU-80              [-1, 50, 128]               0
          Dropout-81              [-1, 50, 128]               0
           Linear-82               [-1, 50, 64]           8,256
          Dropout-83               [-1, 50, 64]               0
      FeedForward-84               [-1, 50, 64]               0
          PreNorm-85               [-1, 50, 64]               0
        LayerNorm-86               [-1, 50, 64]             128
           Linear-87              [-1, 50, 192]          12,288
          Softmax-88            [-1, 4, 50, 50]               0
          Dropout-89            [-1, 4, 50, 50]               0
           Linear-90               [-1, 50, 64]           4,160
          Dropout-91               [-1, 50, 64]               0
        Attention-92               [-1, 50, 64]               0
          PreNorm-93               [-1, 50, 64]               0
        LayerNorm-94               [-1, 50, 64]             128
           Linear-95              [-1, 50, 128]           8,320
             GELU-96              [-1, 50, 128]               0
          Dropout-97              [-1, 50, 128]               0
           Linear-98               [-1, 50, 64]           8,256
          Dropout-99               [-1, 50, 64]               0
     FeedForward-100               [-1, 50, 64]               0
         PreNorm-101               [-1, 50, 64]               0
     Transformer-102               [-1, 50, 64]               0
       LayerNorm-103                   [-1, 64]             128
          Linear-104                    [-1, 6]             390
================================================================
Total params: 201,446
Trainable params: 201,446
Non-trainable params: 0
```

(b) final traing result

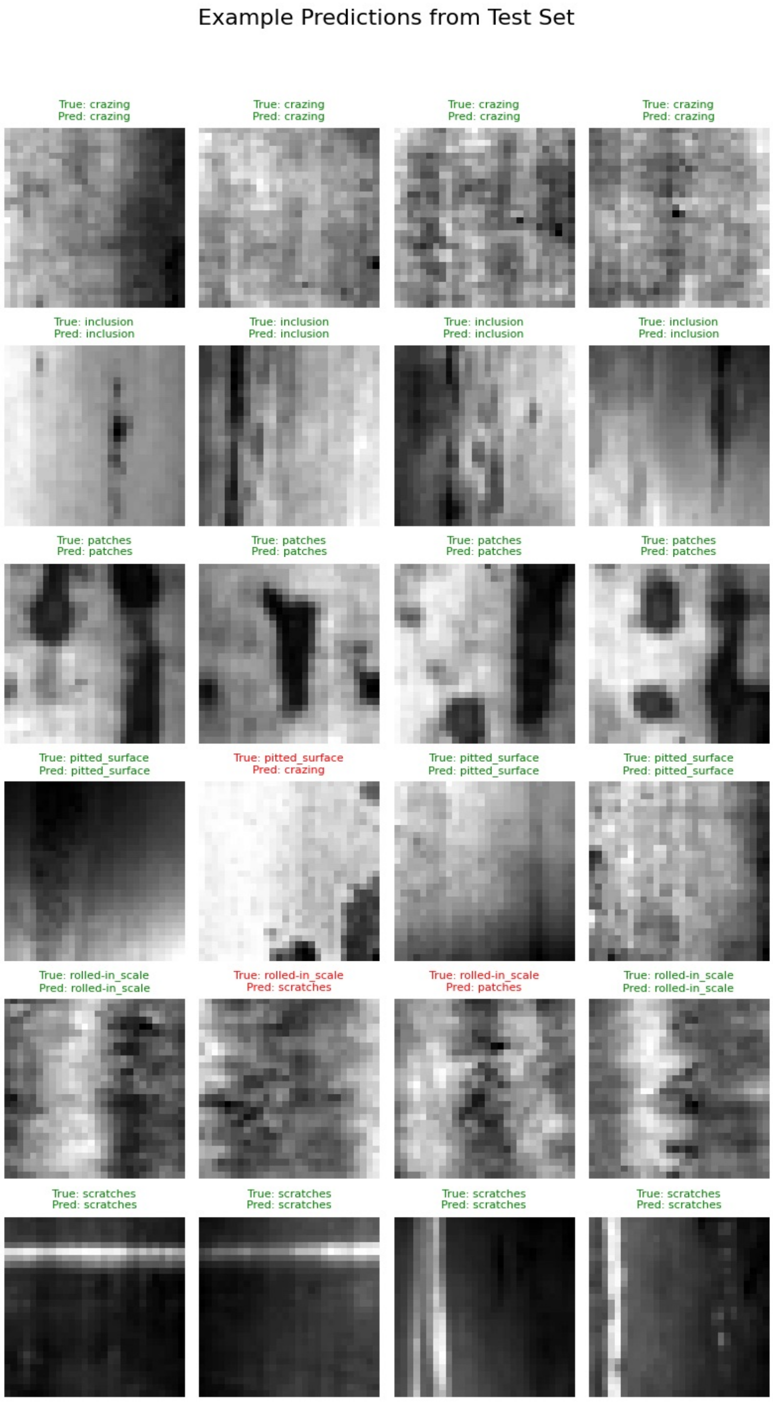


```
--- Epoch: 20/20 ---
Training: 100%|██████████| 20/20 [00:06<00:00,  3.19it/s, acc=0.98, loss=0.117]
```

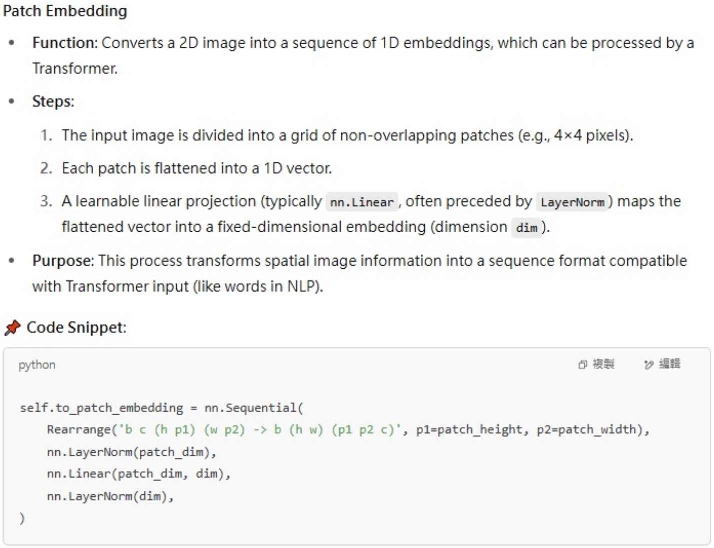
(c) confusion matrix



(d)


Example Predictions from Test Set
True: crazing Pred: crazing
True: crazing Pred: crazing
True: crazing Pred: crazing
True: crazing Pred: crazing
True: inclusion Pred: inclusion
True: inclusion Pred: inclusion
True: inclusion Pred: inclusion
True: inclusion Pred: inclusion
True: patches Pred: patches
True: patches Pred: patches
True: patches Pred: patches
True: patches Pred: patches
True: pitted_surface Pred: pitted_surface
True: pitted_surface Pred: crazing
True: pitted_surface Pred: pitted_surface
True: pitted_surface Pred: pitted_surface
True: rolled-in_scale Pred: rolled-in_scale
True: rolled-in_scale Pred: scratches
True: rolled-in_scale Pred: patches
True: rolled-in_scale Pred: rolled-in_scale
True: scratches Pred: scratches
True: scratches Pred: scratches
True: scratches Pred: scratches
True: scratches Pred: scratches

## Question 6

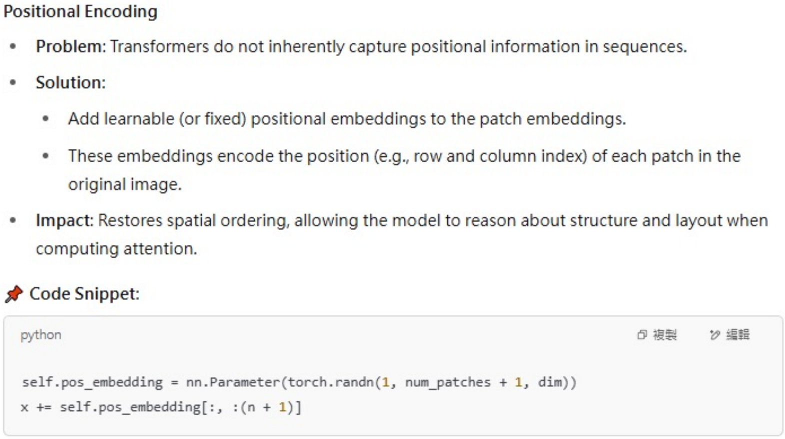


### Patch Embedding

- **Function:** Converts a 2D image into a sequence of 1D embeddings, which can be processed by a Transformer.
- **Steps:**
  1. The input image is divided into a grid of non-overlapping patches (e.g., 4×4 pixels).
  2. Each patch is flattened into a 1D vector.
  3. A learnable linear projection (typically `nn.Linear`, often preceded by `LayerNorm`) maps the flattened vector into a fixed-dimensional embedding (dimension `dim`).
- **Purpose:** This process transforms spatial image information into a sequence format compatible with Transformer input (like words in NLP).

📌 Code Snippet:

```python
self.to_patch_embedding = nn.Sequential(
    Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=patch_height, p2=patch_width),
    nn.LayerNorm(patch_dim),
    nn.Linear(patch_dim, dim),
    nn.LayerNorm(dim),
)
```



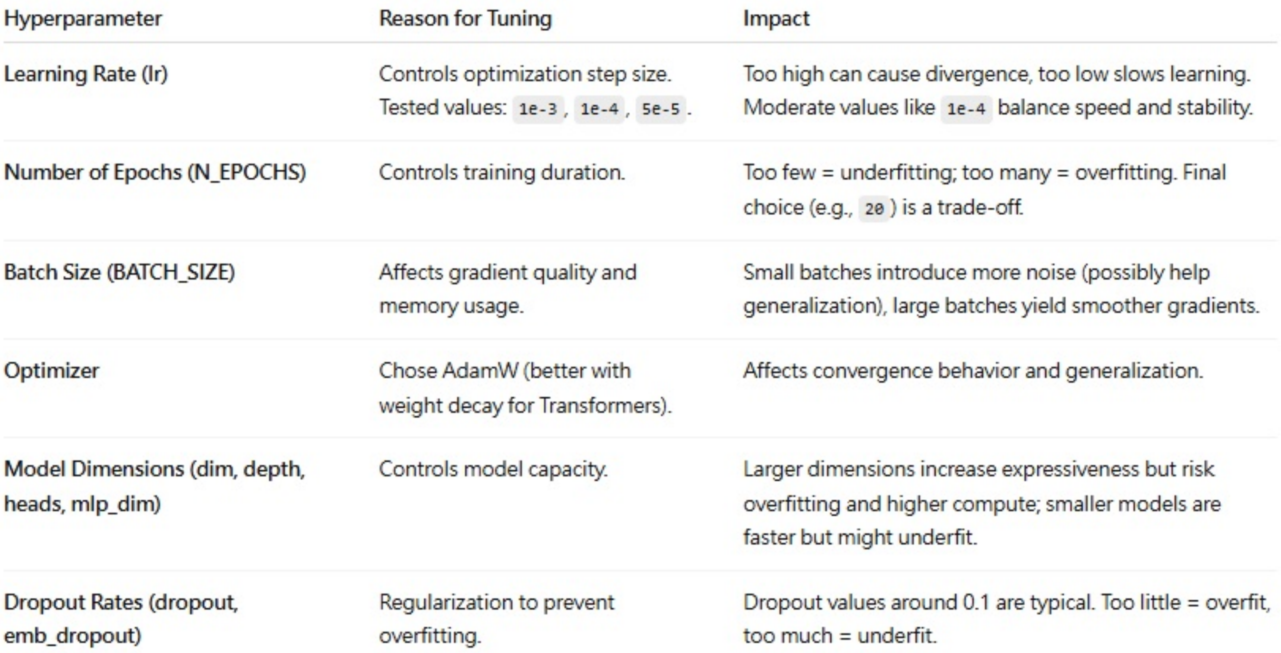
### Positional Encoding

- **Problem:** Transformers do not inherently capture positional information in sequences.
- **Solution:**
  - Add learnable (or fixed) positional embeddings to the patch embeddings.
  - These embeddings encode the position (e.g., row and column index) of each patch in the original image.
- **Impact:** Restores spatial ordering, allowing the model to reason about structure and layout when computing attention.

📌 Code Snippet:

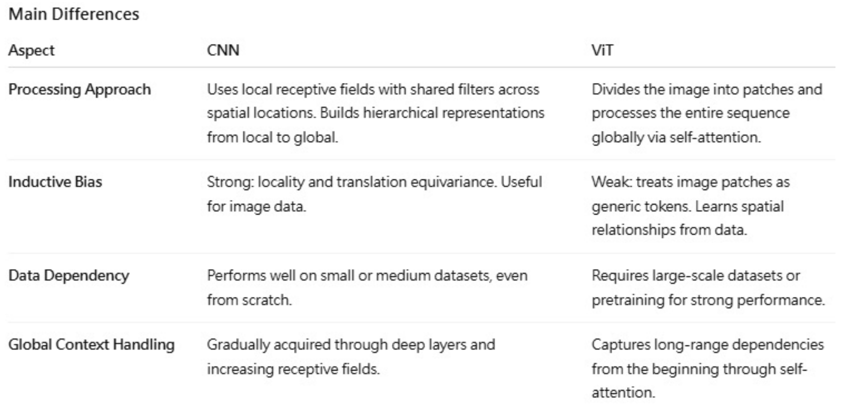
```python
self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
x += self.pos_embedding[:, :(n + 1)]
```

## Question 7.



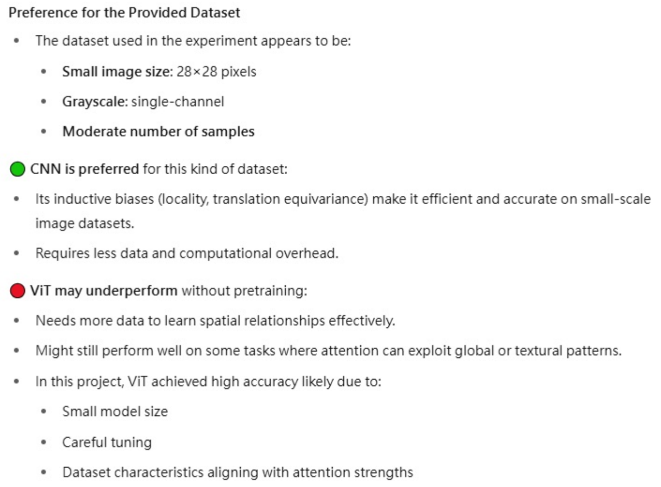
| Hyperparameter | Reason for Tuning | Impact |
|---|---|---|
| Learning Rate (lr) | Controls optimization step size. Tested values: `1e-3`, `1e-4`, `5e-5`. | Too high can cause divergence, too low slows learning. Moderate values like `1e-4` balance speed and stability. |
| Number of Epochs (N_EPOCHS) | Controls training duration. | Too few = underfitting; too many = overfitting. Final choice (e.g., `20`) is a trade-off. |
| Batch Size (BATCH_SIZE) | Affects gradient quality and memory usage. | Small batches introduce more noise (possibly help generalization), large batches yield smoother gradients. |
| Optimizer | Chose AdamW (better with weight decay for Transformers). | Affects convergence behavior and generalization. |
| Model Dimensions (dim, depth, heads, mlp_dim) | Controls model capacity. | Larger dimensions increase expressiveness but risk overfitting and higher compute; smaller models are faster but might underfit. |
| Dropout Rates (dropout, emb_dropout) | Regularization to prevent overfitting. | Dropout values around 0.1 are typical. Too little = overfit, too much = underfit. |

# Question 8.



### Main Differences

| Aspect | CNN | ViT |
|---|---|---|
| **Processing Approach** | Uses local receptive fields with shared filters across spatial locations. Builds hierarchical representations from local to global. | Divides the image into patches and processes the entire sequence globally via self-attention. |
| **Inductive Bias** | Strong: locality and translation equivariance. Useful for image data. | Weak: treats image patches as generic tokens. Learns spatial relationships from data. |
| **Data Dependency** | Performs well on small or medium datasets, even from scratch. | Requires large-scale datasets or pretraining for strong performance. |
| **Global Context Handling** | Gradually acquired through deep layers and increasing receptive fields. | Captures long-range dependencies from the beginning through self-attention. |



### Preference for the Provided Dataset

- The dataset used in the experiment appears to be:
  - **Small image size:** 28×28 pixels
  - **Grayscale:** single-channel
  - **Moderate number of samples**

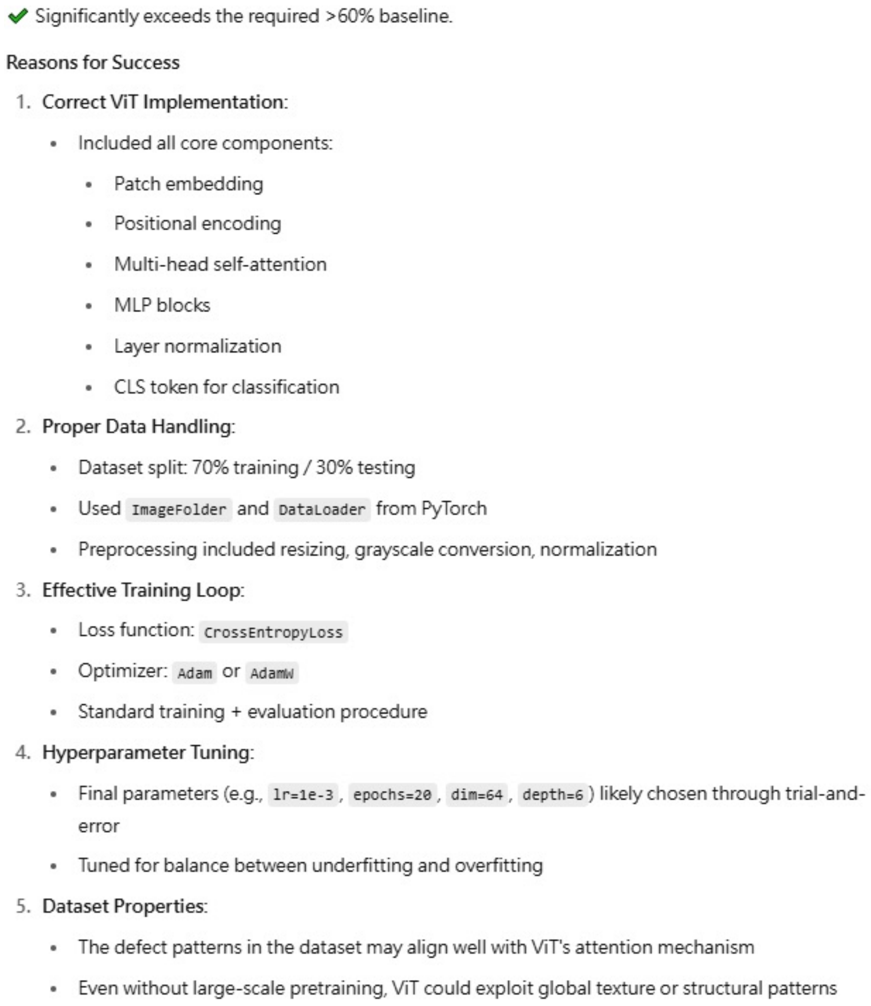
🟢 **CNN is preferred** for this kind of dataset:

- Its inductive biases (locality, translation equivariance) make it efficient and accurate on small-scale image datasets.
- Requires less data and computational overhead.

🔴 **ViT may underperform** without pretraining:

- Needs more data to learn spatial relationships effectively.
- Might still perform well on some tasks where attention can exploit global or textural patterns.
- In this project, ViT achieved high accuracy likely due to:
  - Small model size
  - Careful tuning
  - Dataset characteristics aligning with attention strengths

# Question 9.

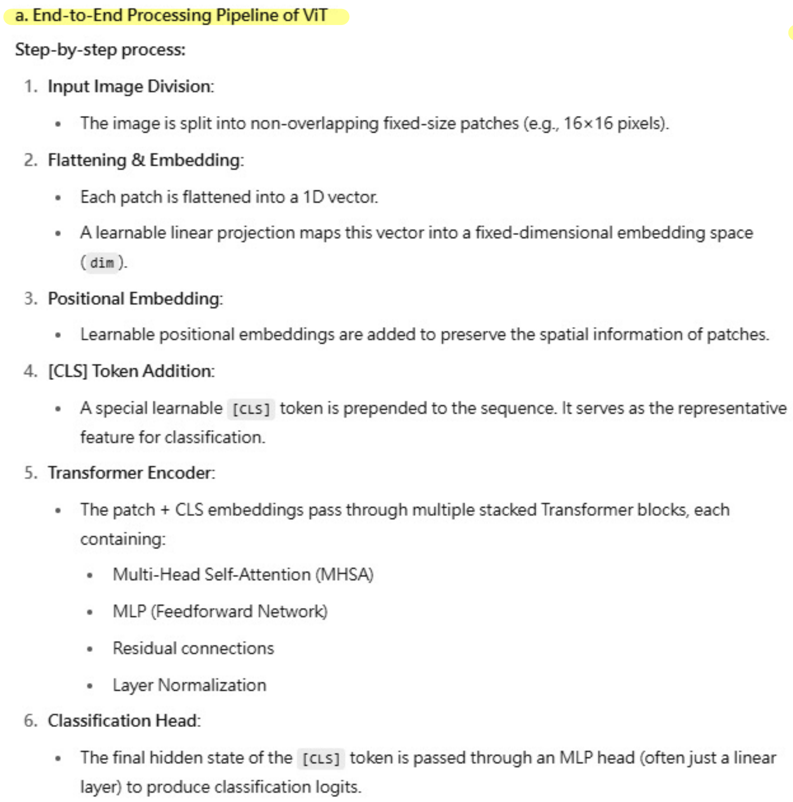


✔ Significantly exceeds the required >60% baseline.

### Reasons for Success

1. **Correct ViT Implementation**:
   - Included all core components:
     - Patch embedding
     - Positional encoding
     - Multi-head self-attention
     - MLP blocks
     - Layer normalization
     - CLS token for classification
2. **Proper Data Handling**:
   - Dataset split: 70% training / 30% testing
   - Used `ImageFolder` and `DataLoader` from PyTorch
   - Preprocessing included resizing, grayscale conversion, normalization
3. **Effective Training Loop**:
   - Loss function: `CrossEntropyLoss`
   - Optimizer: `Adam` or `AdamW`
   - Standard training + evaluation procedure
4. **Hyperparameter Tuning**:
   - Final parameters (e.g., `lr=1e-3`, `epochs=20`, `dim=64`, `depth=6`) likely chosen through trial-and-error
   - Tuned for balance between underfitting and overfitting
5. **Dataset Properties**:
   - The defect patterns in the dataset may align well with ViT's attention mechanism
   - Even without large-scale pretraining, ViT could exploit global texture or structural patterns

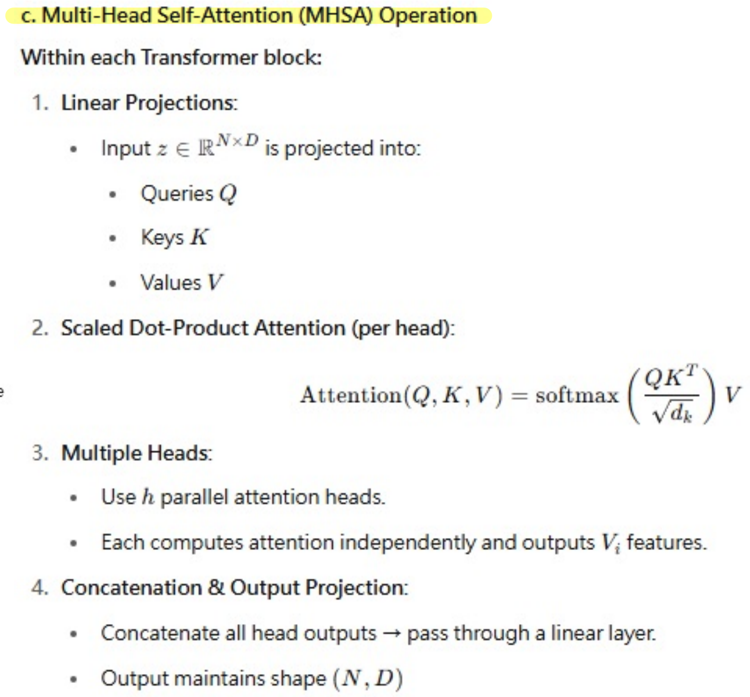
# Question 10



## a. End-to-End Processing Pipeline of ViT

Step-by-step process:

1. Input Image Division:
   - The image is split into non-overlapping fixed-size patches (e.g., 16×16 pixels).
2. Flattening & Embedding:
   - Each patch is flattened into a 1D vector.
   - A learnable linear projection maps this vector into a fixed-dimensional embedding space (`dim`).
3. Positional Embedding:
   - Learnable positional embeddings are added to preserve the spatial information of patches.
4. [CLS] Token Addition:
   - A special learnable `[CLS]` token is prepended to the sequence. It serves as the representative feature for classification.
5. Transformer Encoder:
   - The patch + CLS embeddings pass through multiple stacked Transformer blocks, each containing:
     - Multi-Head Self-Attention (MHSA)
     - MLP (Feedforward Network)
     - Residual connections
     - Layer Normalization
6. Classification Head:
   - The final hidden state of the `[CLS]` token is passed through an MLP head (often just a linear layer) to produce classification logits.



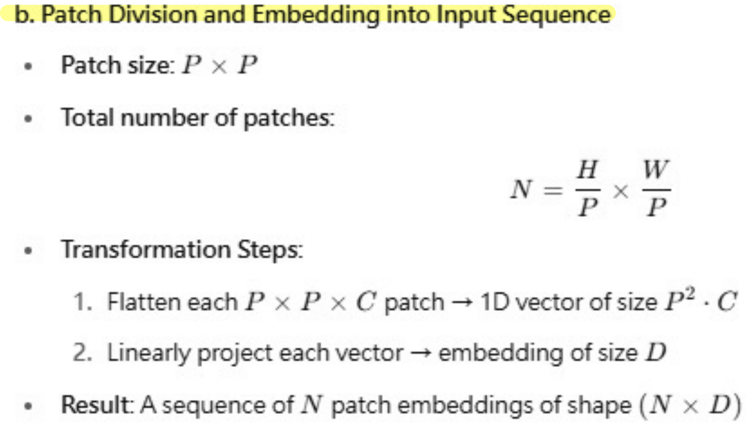
## c. Multi-Head Self-Attention (MHSA) Operation

Within each Transformer block:

1. Linear Projections:
   - Input $z \in \mathbb{R}^{N \times D}$ is projected into:
     - Queries $Q$
     - Keys $K$
     - Values $V$
2. Scaled Dot-Product Attention (per head):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

3. Multiple Heads:
   - Use $h$ parallel attention heads.
   - Each computes attention independently and outputs $V_i$ features.
4. Concatenation & Output Projection:
   - Concatenate all head outputs → pass through a linear layer.
   - Output maintains shape $(N, D)$

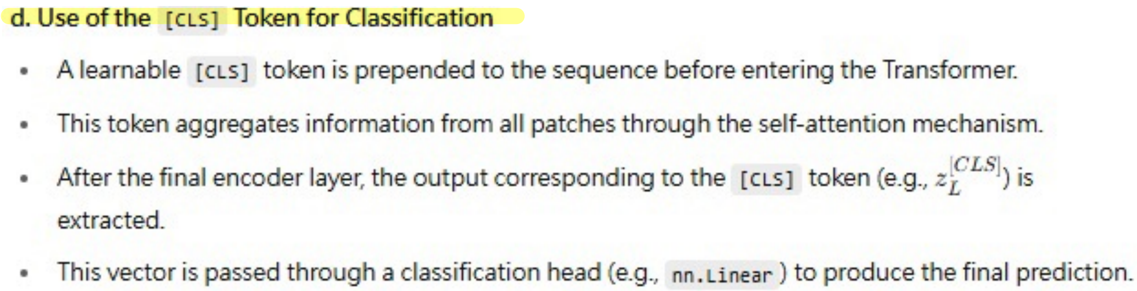


## b. Patch Division and Embedding into Input Sequence

- Patch size: $P \times P$
- Total number of patches:

$$N = \frac{H}{P} \times \frac{W}{P}$$

- Transformation Steps:
  1. Flatten each $P \times P \times C$ patch → 1D vector of size $P^2 \cdot C$
  2. Linearly project each vector → embedding of size $D$
- Result: A sequence of $N$ patch embeddings of shape $(N \times D)$



## d. Use of the `[CLS]` Token for Classification

- A learnable `[CLS]` token is prepended to the sequence before entering the Transformer.
- This token aggregates information from all patches through the self-attention mechanism.
- After the final encoder layer, the output corresponding to the `[CLS]` token (e.g., $z_L^{[CLS]}$) is extracted.
- This vector is passed through a classification head (e.g., `nn.Linear`) to produce the final prediction.