Open in Colab

Implementing Vision Transformers in PyTorch from Scratch on Any Dataset!

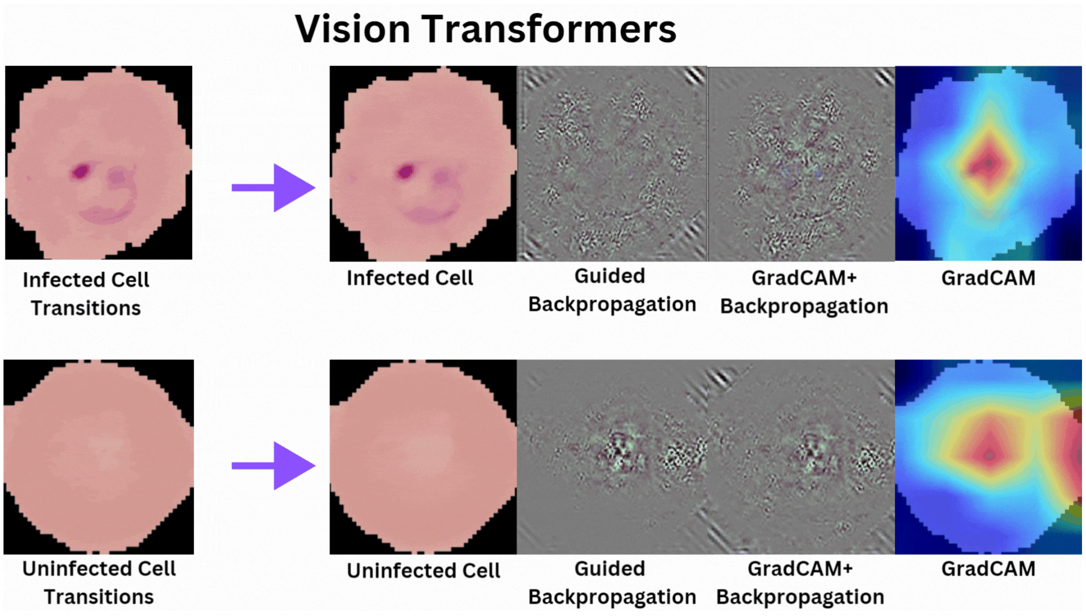


Table of Contents

Why this Tutorial?

Vision Transformers (ViTs) were introduced in 2020 to present a classification approach for images. Having being inspired by the Transformer architecture used prevalently in text classification, ViTs present a robust approach powered by multi-head self-attention to compete with CNNs for image recognition tasks. There have been numerous implementations of ViTs so far that capture the performance of the model perfectly. However, they are often implementable through external APIs such as HuggingFace or Keras, providing easier computations at the expense of lower architectural transparency.

In fact, while these APIs are a Godsend for developers and engineers to facilitate model implementation and fine-tuning at the hit of a button, they are often not the best resort for students starting with Deep Learning. For learners, the best approach to get hands-on deep learning practice is through implementing SOTA models from scratch. In this repository, we have created the most lucid and understandable tutorial for students to implement ViTs from scratch on any dataset of their choice any dataset of their choice.

Introduction to ViTs

Conventional research demonstrated replacing CNN layers or network components with attention mechanisms to aid image recpognition. The emergence of ViT proved that transformer models independently can also yield performances synonymous to those of CNNs. Let us discuss the ViT architecture that enables this.

ViT Architecture

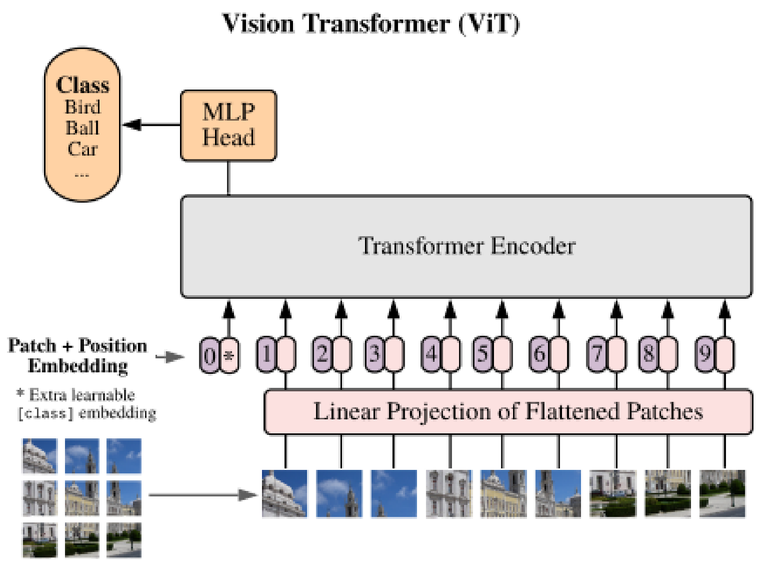

Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0 * 1 2 3 4 5 6 7 8 9
Linear Projection of Flattened Patches

Fig.1 Vision Transformer Architecture - Source

Let us discuss the architecture step-by-step.

1. **Split the images into fixed-sized (16x16) patches**. The patch dimensions have been mentioned in the original ViT paper and are taken as standard.
2. **Flatten the image patches**. A standard Transformer receives inputs in the form of a 1D sequence of embeddings. Since image patches are 2D, we must reshape the patches into a sequence of flattened embeddings (patches, in our case).
3. **Add linear and positional embeddings to the patch sequence**. Linear embedding is a dimensionality reduction technique that projects high-dimensional data onto a lower-dimensional space while preserving the local features of the data. Positional embeddings are added to the image patches to provide knowledge about the position of the input vectors. This ensures that upon 2D image reconstruction from the patches, the positional integrity of the patches is not lost and a recognizable image is formed.
4. **Feed the flattened patch sequence into a Transformer Encoder**. This transforms input tokens into contextualized representations.
5. **An extra "classification token" or an MLP head is added** to perform classification and recognition.

Import Modules and Libraries

I have created this GitHub repository that can be cloned directly on your Colab/IPython environment to implement ViTs with PyTorch from scratch. It only contains the files you require for this code.

this code.

In [ ]:
```
#@title Clone, Install, Import
!git clone https://github.com/ssanya942/MICCAI-Educational-Challenge-2024.git
```

In [2]:
```
%cd /content/MICCAI-Educational-Challenge-2024
```

```
/content/MICCAI-Educational-Challenge-2024
```

In [ ]:
```
!pip install -r requirements.txt
import matplotlib.pyplot as plt
import torch
import torchvision
torch.__version__
from torch import nn
from torchvision import transforms
""" Remember to restart the session after this cell runs to use the newly-installed deper
```

Data Processing

Once the requirements have been installed, it is now time to process the dataset and create train/test dataloaders.

For this implementation, we have used the openly-accessible NiH Malaria Cell Image Dataset.

The dataset contains 27,558 images divided into 2 categories:

1. **parasitized** (13,779 images), and
2. **uninfected** (13,779 images)

To speed up the training process, we have reduced the size of the dataset and structured it as:

```
malaria(500 images)
├── train (400)/
│ ├── uninfected(200)
│ └── infected(200)
└── test (100)/
├── uninfected(50)
└── infected(50)
```

The dataset contains images of thin blood smear slides, obtained from 150 patients with *P. falciparum* malaria infection and 50 healthy individuals

In [4]:
```
""" Specify the train and test image directories and proceed to create Dataloaders"""
train_dir = '/content/MICCAI-Educational-Challenge-2024/train'
test_dir = '/content/MICCAI-Educational-Challenge-2024/test'

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
):

  # 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 train and test data loaders
  train_dataloader = DataLoader(
      train_data,
      batch_size=batch_size,
      shuffle=True,
      num_workers=num_workers,
      pin_memory=True,
  )
  test_dataloader = DataLoader(
      test_data,
      batch_size=batch_size,
      shuffle=False,
      num_workers=num_workers,
      pin_memory=True,
  )

  return train_dataloader, test_dataloader, class_names
```

In [5]:

```
""" Create Manual Data Transformation Pipeline"""

IMG_SIZE = 224

# Create transform pipeline manually
manual_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)), ##Resize to image size
    transforms.ToTensor(),  ## Convert np array to tensor
   transforms.RandomHorizontalFlip(p=0.5), ## Perform horizontal flipping
    transforms.RandomVerticalFlip(p=0.5),  ## Perform vertical flipping


])
print(f"Manually created transforms: {manual_transforms}")
```

```
Manually created transforms: Compose(
    Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=True)
    ToTensor()
    RandomHorizontalFlip(p=0.5)
    RandomVerticalFlip(p=0.5)
)
```

The code above defines a function ***create_dataloaders*** that sets up data loaders for training and testing datasets using PyTorch. It utilizes the **ImageFolder** class from **torchvision.datasets** to load image data from the **train_dir** and **test_dir** directories, applying a given set of manually-specified transformations to each image through **transforms.Compose()**. The function then creates **DataLoader objects** for the training and testing datasets for efficient data loading and batching during model training and evaluation. The ***batch_size*** parameter determines how many samples per batch to load, and ***num_workers*** specifies how many subprocesses to use for data loading, the default value for which is the number of available CPU cores. The function also retrieves the class names from the training dataset and returns the training data loader, test data loader, and class names.

Setting up the dataloader in the code below:

In [6]:

```
BATCH_SIZE = 32

# Create data loaders
train_dataloader, test_dataloader, class_names = create_dataloaders(
    train_dir=train_dir,
    test_dir=test_dir,
    transform=manual_transforms, ## defined above
    batch_size=BATCH_SIZE
)

train_dataloader, test_dataloader, class_names
```

Out[6]:
```
(<torch.utils.data.dataloader.DataLoader at 0x7db4a5cde1a0>,
 <torch.utils.data.dataloader.DataLoader at 0x7db3c9bea6b0>,
 ['infected', 'uninfected'])
```

Now, we extract and visualize a single image from a batch of images obtained from a PyTorch data loader. First, we retrieve the next batch of images and their corresponding labels from ***train_dataloader*** using the **next** and **iter** functions. We display the shapes of the image and label for sanity checking. The image is then plotted, but first, it is permuted to rearrange its dimensions from **[color_channels, height, width] to [height, width, color_channels]**, which is required by matplotlib for display.

This is shown in the code below.

In [7]:
```
image_batch, label_batch = next(iter(train_dataloader))

# Get a single image from the batch
image, label = image_batch[0], label_batch[0]

# View the batch shapes
print(image.shape, label)

# Plot image with matplotlib
plt.imshow(image.permute(1, 2, 0)) # rearrange image dimensions to suit matplotlib [colo
plt.title(class_names[label])
plt.axis(False);
```

```
torch.Size([3, 224, 224]) tensor(1)
```

Creating ViT Network Components

We assume the ViT architecture to be a sum of multiple functional components stacked onto one another synchronously. We create multiple, independent network components that formulate the model architecture when amalgamated together. These network components include:

1. **A Patch Embedding Block**
2. **A Multi-head Self-attention Block**
3. **An MLP Head**, and
4. **A Transformer Encoder**

In [8]:

```python
# 1. Create a class which subclasses nn.Module
class PatchEmbedding(nn.Module):
    """Turns a 2D input image into a 1D sequence learnable embedding vector.

    Args:
        in_channels (int): Number of color channels (3).
        patch_size (int): Size of the patches generated from the input image (16x16).
        embedding_dim (int): Size of embedding to turn image into (768).
    """
    # 2. Initialize the class with appropriate variables
    def __init__(self,
                 in_channels:int=3,
                 patch_size:int=16,
                 embedding_dim:int=768):
        super().__init__()

        # 3. Create a layer to turn an image into patches
        self.patcher = nn.Conv2d(in_channels=in_channels,
                                 out_channels=embedding_dim,
                                 kernel_size=patch_size,
                                 stride=patch_size,
                                 padding=0)

        # 4. Create a layer to flatten the patch feature maps into a single dimension
        self.flatten = nn.Flatten(start_dim=2, # only flatten the feature map dimensions
                                  end_dim=3)

    # 5. Define the forward method
    def forward(self, x):
        # Create assertion to check that inputs are the correct shape
        image_resolution = x.shape[-1]
        assert image_resolution % patch_size == 0, f"Input image size must be divisble by

        # Perform the forward pass
        x_patched = self.patcher(x)
        x_flattened = self.flatten(x_patched)

        # 6. Make sure the output shape has the right order
        return x_flattened.permute(0, 2, 1)
```

The ***PatchEmbedding class***, a subclass of **nn.Module** in PyTorch, converts a 2D input image into a 1D sequence of learnable embedding vectors. It takes parameters for:

1. the number of input channels (**in_channels**),
2. patch size (**patch_size**), and
3. embedding dimension (**embedding_dim**).

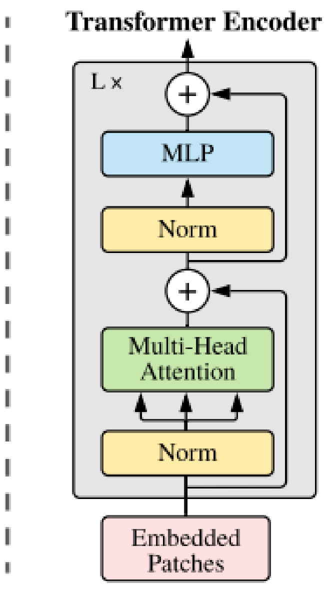
The class utilizes a convolutional layer (**self.patcher**) to divide the image into non-overlapping patches and project each patch into a vector of size **embedding_dim**.

The **nn.Flatten** layer then reshapes the patch feature maps into one dimension. An assertion ensures the image resolution is divisible by the patch size, preventing runtime errors. In the **forward** method, the image is passed through the patcher and flatten layers, and the output dimensions are permuted to match the expected input format of transformers, yielding an output shape of (**batch_size**, **num_patches**, **embedding_di**m). This process allows the image data to be efficiently transformed for sequential processing in neural network architectures.

In [9]:
```python
# Let's test it on single image
patch_size =16

# Set seeds
def set_seeds(seed: int=42):
    """Sets random sets for torch operations (42). """
    # Set the seed for general torch operations (CPU)
    torch.manual_seed(seed)
    # Set the seed for CUDA torch operations (GPU)
    torch.cuda.manual_seed(seed)



set_seeds()

# Create an instance of patch embedding layer
patchify = PatchEmbedding(in_channels=3,
                          patch_size=16,
                          embedding_dim=768)

# Pass a single image through
print(f"Input image shape: {image.unsqueeze(0).shape}")
patch_embedded_image = patchify(image.unsqueeze(0)) # add an extra batch dimension on th
print(f"Output patch embedding shape: {patch_embedded_image.shape}")
print(patch_embedded_image)
print(f"Patch embedding shape: {patch_embedded_image.shape} -> [batch_size, number_of_pa
```

```
Input image shape: torch.Size([1, 3, 224, 224])
Output patch embedding shape: torch.Size([1, 196, 768])
tensor([[[-0.0188, -0.0114, -0.0311,  ...,  0.0105, -0.0033,  0.0033],
         [-0.0188, -0.0114, -0.0311,  ...,  0.0105, -0.0033,  0.0033],
         [-0.0188, -0.0114, -0.0311,  ...,  0.0105, -0.0033,  0.0033],
         ...,
         [-0.3176,  0.1695,  0.2036,  ...,  0.2856, -0.2756, -0.0553],
         [-0.8642,  0.1169,  0.0788,  ...,  0.5396, -0.4044, -0.2918],
         [-0.4942,  0.1559,  0.1007,  ...,  0.1938, -0.1650,  0.0930]]],
       grad_fn=<PermuteBackward0>)
Patch embedding shape: torch.Size([1, 196, 768]) -> [batch_size, number_of_patches, embedd
ing_dimension]
```

Now, we create the Multi-head Self Attention and Multi-layer Perceptron blocks present within the



Transformer Encoder.

> Fig.2. Transformer Encoder

In [10]:
```python
# 1. Create a new nn.Module class
```

```
class MultiheadSelfAttentionBlock(nn.Module):
    """Creates a multi-head self-attention block ("MSA block").
    Args:
        embedding_dim (int): Size of embedding to turn image into (768).
        num_heads(int): Number of heads to use in the multi-head attention layer.
        attn_dropout (float): Dropout rate in the attention layer.
         """
    # 2. Initialize the class with hyperparameters from Table 1 in ViT-Paper
    def __init__(self,
                 embedding_dim:int=768, # Data from Table 1 for ViT-Base
                 num_heads:int=12, # Heads from Table 1 for ViT-Base
                 attn_dropout:float=0): # Defaults to 0 in ViT-Paper for MSABlocks
        super().__init__()

        # 3. Create the Norm layer (LN)
        self.layer_norm = nn.LayerNorm(normalized_shape=embedding_dim)

        # 4. Create MSA layer
        self.multihead_attn = nn.MultiheadAttention(embed_dim=embedding_dim,
                                                    num_heads=num_heads,
                                                    dropout=attn_dropout,
                                                    batch_first=True) # if our batch dim

    # 5. Create a forward() method to pass the data throguh the layers
    def forward(self, x):
        x = self.layer_norm(x)
        attn_output, _ = self.multihead_attn(query=x, # query embeddings
                                             key=x, # key embeddings
                                             value=x, # value embeddings
                                             need_weights=False) # do we need the weight
        return attn_output
```

The ***MultiheadSelfAttentionBlock*** class implements a multi-head self-attention mechanism, a core component of ViT. Instead of a single attention mechanism for capturing long-range dependencies and contextual features, multiple attention heads (e.g., 12 for ViT-Base) operate in parallel. Each head has its own set of learned Query-Key-Value (QKV) transformations, enabling the model to capture different types of relationships and patterns within the data. It initializes with hyperparameters:

1. **embedding_dim (768)**,
2. **num_heads (12)**, and
3. **attn_dropout** for dropout rate (**defaulting to 0**).

The class consists of two main components:

1. **Layer Normalization (nn.LayerNorm)** is applied to the input to stabilize and speed up training.
2. **Multi-Head Attention (nn.MultiheadAttention)** is configured with the specified embedding dimension and number of heads.

In the **forward** method, the input tensor `x` is normalized, then passed through the multi-head attention layer as **query, key, and value**, producing the attention output. The **batch_first=True** parameter ensures that the batch dimension is the first dimension of the input tensor, adhering to the typical PyTorch conventions. As **need_weights=False** is specified, the attention weights are computed internally but not returned,. This class facilitates efficient attention-based transformations of input data for sequential modeling tasks.

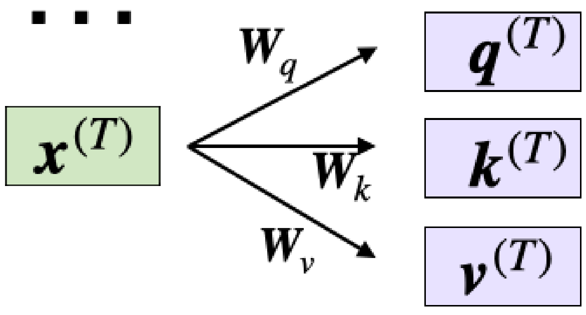
If you don't have prior experience with self-attention mechanisms, please take a look at this explanation for QKV-pairs [2].

Understanding Query-Key-Value in Self-Attention

All self-attention networks use a system of 3 weight matrices which are adjusted as model parameters during training. These matrices are W_q, W_k, and W_v.

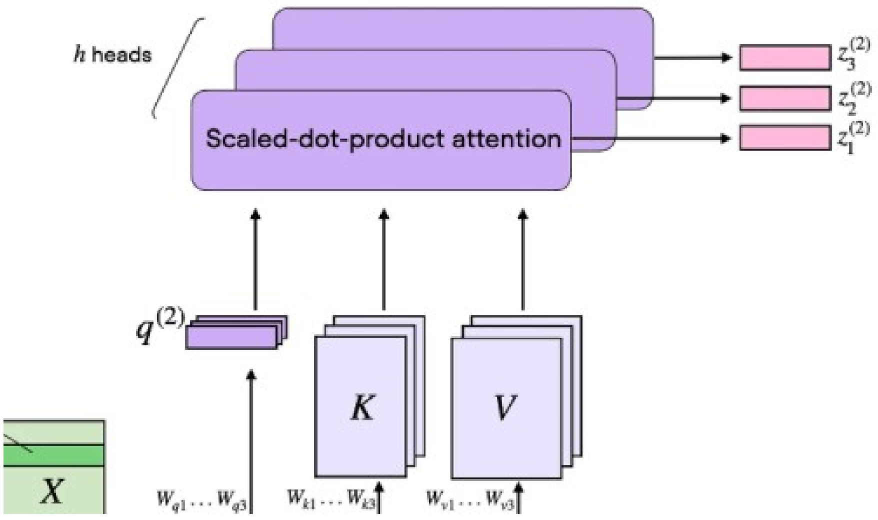
These learned weight matrices are multiplied with the input embeddings (x) to yield QKV input sequences. This is calculated through a scaled dot-product multiplication.

1. Query sequence: $q(i) = x(i)W_q$ for i in sequence $1 \dots T$
2. Key sequence: $k(i) = x(i)W_k$ for i in sequence $1 \dots T$
3. Value sequence: $v(i) = x(i)W_v$ for i in sequence $1 \dots T$



> Fig.3. QKV Dot Product Calculation-Source

In the case of MSA, multiple self-attention heads are placed in parallel.



> Fig.4 Multi-head Self-Attention Mechanism-Source

Here, z_i is a weighted context vector of the original input x obtained through the scaled dot product multiplication.

In [11]:
```
# 1. Create another nn.Module Class
class MLPBlock(nn.Module):
    """Creates a layer normalized multilayer perceptron block (MLP) .
        Args:
          embedding_dim (int): Size of embedding to turn image into (768).
          mlp_size (int): Size of MLP block.
          dropout (float): Dropout rate
        """
    # 2. Initialize the class with hyperparameters from Table 1 and Table 3 of the ViT-Pa
    def __init__(self,
                 embedding_dim:int=768, # Dimensions from Table 1 for ViT-Base Paper
                 mlp_size:int=3072, # Size of the hidden layer in the MLP block from Tabl
                 dropout:float=0.1): # Dropout from Table 3 for ViT-Base Paper
        super().__init__()
        # 3. Create the Norm layer (LN)
        self.layer_norm = nn.LayerNorm(normalized_shape=embedding_dim)

        # 4. Create the MLP Layer
        self.mlp = nn.Sequential(
            nn.Linear(in_features=embedding_dim,
                      out_features=mlp_size),
            nn.GELU(), # The MLP contains two GELU Layers
            nn.Dropout(p=dropout),
            nn.Linear(in_features=mlp_size, # input of this layer is the output of the pi
                      out_features=embedding_dim), # take back to embedding_dim
            nn.Dropout(p=dropout) # Dropout after Dense Layer
        )

    # 5. Create a forward() method to pass the data throguh the layers
    def forward(self, x):
        x = self.layer_norm(x)
        x = self.mlp(x)
        return x
```

The above code defines a PyTorch module class called ***MLPBlock***, which is a Multi-layer Perceptron block of the ViT architecture. This block implements a layer-normalized MLP with parameters based on the ViT-Base configuration from the ViT paper.

For those who are unaware, MLP is one of the simplest artificial neural network architecturesfor classification tasks [3].

The class initializes with the hyperparameters

1. **embedding_dim**,
2. **mlp_size**, and
3. **dropout**, which default to values specified in the ViT paper.

The MLP block consists of a **LayerNorm** layer followed by a sequential model of two linear layers, followed by a **Gaussian Error Linear Unit (GELU)** activation function and a dropout for regularization. The **forward** method applies layer normalization and passes the input through the MLP layers, returning the output, which helps in enhancing the model's expressiveness and improving training stability.

In [12]:
```
# 1. Create another nn.Module class
class TransformerEncoderBlock(nn.Module):
    """Creates a Transformer Encoder block.
    Args:
        embedding_dim (int): Size of embedding to turn image into (768).
        num_heads (int): Number of heads to use in the multi-head attention layer.
        mlp_size (int): Size of MLP block.
        mlp_dropout (float): Dropout rate in the MLP block.
        attn_dropout (float): Dropout rate in the attention layer.
        """
    # 2. Initialize the class with hyperparameters from Table 1 and Table 3
    def __init__(self,
                 embedding_dim:int=768, # Embedding size from Table 1 in ViT-Base Paper
                 num_heads:int=12, # No. of Heads from Table 1 in ViT-Base Paper
                 mlp_size:int=3072, # MLP size from Table 1 in ViT-Base Paper
                 mlp_dropout:float=0.1, # Amount of dropout for dense layers from Table 
                 attn_dropout:float=0): # Amount of dropout for attention layers (defaul
        super().__init__()

        # 3. Create MSA block
        self.msa_block = MultiheadSelfAttentionBlock(embedding_dim=embedding_dim,
                                                     num_heads=num_heads,
                                                     attn_dropout=attn_dropout)

        # 4. Create MLP block
        self.mlp_block =  MLPBlock(embedding_dim=embedding_dim,
                                     mlp_size=mlp_size,
                                     dropout=mlp_dropout)

    # 5. Create a forward() method
    def forward(self, x):

        # 6. Create residual connection for MSA block
        x =  self.msa_block(x) + x

        # 7. Create residual connection for MLP block
        x = self.mlp_block(x) + x

        return x
```

The code defines a ***TransformerEncoderBlock class*** inherited from the PyTorch **nn.Module** class. It can handle input sequences through self-attention and feedforward neural networks. This is the block that processes an input sequence of vectors and creates representations that contain contextual information, which are then passed through a classification head. The block is initialized with the hyperparameters

1. **embedding_dim**,
2. **num_heads**,
3. **mlp_size**,
4. mlp_dropout, and
5. attn_dropout, all sourced from the ViT-Base architecture specifications.

Within the block, a multi-head self-attention (MSA) mechanism is implemented using a ***MultiheadSelfAttentionBlock***, which facilitates learning contextual relationships in the input data. Additionally, an ***MLPBlock*** is utilized to apply a multi-layer perceptron for further processing of the attended features. A normalization layer is added before the MSA and MLP blocks. The **forward** method includes residual connections around both the MSA and MLP blocks, which help maintain gradient flow and improve model stability during training, helping in encoding complex input features.

Let us display a quick model summary of our Transformer Encoder block.

In [13]:
```
transformer_encoder_block = TransformerEncoderBlock()

from torchinfo import summary
# # Print an input and output summary of the Transformer Encoder
summary(model=transformer_encoder_block,
        input_size=(1, 197, 768), # (batch_size, num_patches, embedding_dimension)
        col_names=["input_size", "output_size", "num_params", "trainable"],
        col_width=25,
       row_settings=["var_names"])
```

Out[13]:
```
==========================================================================================
==============================================================
Layer (type (var_name))                            Input Shape               Output Shap
e               Param #                   Trainable
==========================================================================================
==============================================================
TransformerEncoderBlock (TransformerEncoderBlock)  [1, 197, 768]             [1, 197, 76
8]              --                        True
├─MultiheadSelfAttentionBlock (msa_block)          [1, 197, 768]             [1, 197, 76
8]              --                        True
│    └─LayerNorm (layer_norm)                      [1, 197, 768]             [1, 197, 76
8]              1,536                     True
│    └─MultiheadAttention (multihead_attn)         --                        [1, 197, 76
8]              2,362,368                 True
├─MLPBlock (mlp_block)                             [1, 197, 768]             [1, 197, 76
8]              --                        True
│    └─LayerNorm (layer_norm)                      [1, 197, 768]             [1, 197, 76
8]          1,536                      True
│    └─Sequential (mlp)                         [1, 197, 768]             [1, 197, 76
8]          --                         True
│    │    └─Linear (0)                          [1, 197, 768]             [1, 197, 30
72]         2,362,368                  True
│    │    └─GELU (1)                            [1, 197, 3072]            [1, 197, 30
72]         --                         --
│    │    └─Dropout (2)                         [1, 197, 3072]            [1, 197, 30
72]         --                         --
│    │    └─Linear (3)                          [1, 197, 3072]            [1, 197, 76
8]          2,360,064                  True
│    │    └─Dropout (4)                         [1, 197, 768]             [1, 197, 76
8]          --                         --
=========================================================================================
==============================================================
Total params: 7,087,872
Trainable params: 7,087,872
Non-trainable params: 0
Total mult-adds (M): 4.73
=========================================================================================
==============================================================
Input size (MB): 0.61
Forward/backward pass size (MB): 8.47
Params size (MB): 18.90
Estimated Total Size (MB): 27.98
=========================================================================================
==============================================================
```

Assembling ViT

Now that we are done creating individual model components for ViT, it is now time for us to assemble these components together to form the complete architecture.

In [14]:

```
class ViT(nn.Module):
    """Creates a Vision Transformer architecture with ViT-Base hyperparameters by default
    Args:
    img_size(int): Size of the input image
    in_channels(int): Number of color channels of the input image
    patch_size(int): Size of the patches to be extracted from the input
    num_transformer_layers(int): Number of transformer blocks in the encoder
    embedding_dim(int): Number of hidden dimensions in the transformer
    mlp_size(int): Number of hidden dimensions in the MLP of the transformer
    num_heads(int): Number of attention heads in the transformer
    attn_dropout(float): Dropout in the attention layers
    mlp_dropout(float): Dropout in the MLP layers
    embedding_dropout(float): Dropout in the embedding layers
    num_classes(int): Number of classes to predict
    """
    # 2. Initialize the class with hyperparameters from Table 1 and Table 3 in ViT-Base 
    def __init__(self,
                 img_size:int=224, # Training resolution from Table 3 in ViT paper
                 in_channels:int=3, # Number of channels in input image
                 patch_size:int=16, # Patch size
                 num_transformer_layers:int=12, # Layers from Table 1 for ViT-Base
                 embedding_dim:int=768, # Hidden size D from Table 1 for ViT-Base
                 mlp_size:int=3072, # MLP size from Table 1 for ViT-Base
                 num_heads:int=12, # Heads from Table 1 for ViT-Base
                 attn_dropout:float=0, # Dropout for attention projection
                 mlp_dropout:float=0.1, # Dropout for dense/MLP layers
                 embedding_dropout:float=0.1, # Dropout for patch and position embedding
                 num_classes:int=1000): # Default for ImageNet
        super().__init__() # don't forget the super().__init__()!

        # 3. Make the image size divisble by the patch size
        assert img_size % patch_size == 0, f"Image size must be divisible by patch size,

        # 4. Calculate number of patches (height * width/patch^2)
        self.num_patches = (img_size * img_size) // patch_size**2
        # 5. Create learnable class embedding (needs to go at front of sequence of patch
        self.class_embedding = nn.Parameter(data=torch.randn(1, 1, embedding_dim),
                                            requires_grad=True)

        # 6. Create learnable position embedding
        self.position_embedding = nn.Parameter(data=torch.randn(1, self.num_patches+1, em
                                               requires_grad=True)

        # 7. Create embedding dropout value
        self.embedding_dropout = nn.Dropout(p=embedding_dropout)

        # 8. Create patch embedding layer
        self.patch_embedding = PatchEmbedding(in_channels=in_channels,
                                              patch_size=patch_size,
                                              embedding_dim=embedding_dim)

        # 9. Create Transformer Encoder blocks (stacked)
        self.transformer_encoder = nn.Sequential(*[TransformerEncoderBlock(embedding_dim=
                                                                            num_heads=nur
                                                                            mlp_size=mlp_
                                                                            mlp_dropout=r

        # 10. Create classifier head
        self.classifier = nn.Sequential(
            nn.LayerNorm(normalized_shape=embedding_dim),
            nn.Linear(in_features=embedding_dim,
                      out_features=num_classes)
        )

    # 11. Create a forward() method
    def forward(self, x):

        # 12. Get batch size
        batch_size = x.shape[0]

        # 13. Create class token embedding and expand it to match the batch size (equatic
        class_token = self.class_embedding.expand(batch_size, -1, -1) # "-1" means to in

        # 14. Create patch embedding
        x = self.patch_embedding(x)

        # 15. Concatenate class embedding and patch embedding
        x = torch.cat((class_token, x), dim=1)

        # 16. Add positional embedding to patch embedding
        x = self.position_embedding + x

        # 17. Run embedding dropout
        x = self.embedding_dropout(x)

        # 18. Pass patch, positional and class embeddings through transformer encoder la
        x = self.transformer_encoder(x)

        # 19. Put 0 index logit through classifier
        x = self.classifier(x[:, 0]) # run on each sample in a batch at 0 index

        return x
```

Finally, all the network components of ViT are put together and a **ViT** class is initialized with all the parameters described in the code above.

To summarize the code, the model divides an input image into non-overlapping patches, which are linearly embedded into a sequence of patch embeddings. A learnable class token and position embeddings are added to this sequence. The sequence is then passed through multiple transformer encoder blocks, each consisting of multi-head self-attention and a feed-forward neural network, followed by layer normalization. A dropout layer is applied to the embeddings to prevent overfitting. Finally, the class token's embedding is extracted and passed through a classifier head, which includes a layer normalization and a fully connected layer, to produce class logits.

Initialize the model as follows and start training.

In [15]:
```
vit = ViT(num_classes=len(class_names))
```

I have trained the model for only 30 epochs using Google Colab's Tesla A100 GPU, but you can also train it with the free T4 GPU. The *engine.py* file is provided in my repository for direct import. It is originally from the Pytorch deep learning repo.

Set up the

1. **optimizer (Adam)**, and
2. **the loss function (Cross entropy loss)**

Now, train the model.

Training

In [16]:
```
%cd /content/MICCAI-Educational-Challenge-2024
```

```
/content/MICCAI-Educational-Challenge-2024
```

In [17]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
""" Set up the loss function and the optimizer and begin training on cuda"""

import engine
optimizer = torch.optim.Adam(params=vit.parameters(),
                             lr=3e-3, # Base LR from Table 3 for ViT-ImageNet-1k
                             betas=(0.9, 0.999), # default values mentioned in ViT-Base
                             weight_decay=0.3) # Default from ViT-Base Training & Fine-tu

# Setup the loss function for multi-class classification
loss_fn = torch.nn.CrossEntropyLoss()

# Set the seeds
set_seeds()

# Train the model and save the training results to a dictionary
results = engine.train(model=vit,
                       train_dataloader=train_dataloader,
                       test_dataloader=test_dataloader,
                       optimizer=optimizer,
                       loss_fn=loss_fn,
                       epochs=30,
                       device=device)
```

```
  0%|          | 0/30 [00:00<?, ?it/s]
Epoch: 1 | train_loss: 2.0587 | train_acc: 0.4976 | test_loss: 0.6874 | test_acc: 0.6094
Epoch: 2 | train_loss: 0.7750 | train_acc: 0.4808 | test_loss: 0.6695 | test_acc: 0.6094
Epoch: 3 | train_loss: 0.6962 | train_acc: 0.5144 | test_loss: 0.6786 | test_acc: 0.6094
Epoch: 4 | train_loss: 0.6993 | train_acc: 0.5240 | test_loss: 0.6748 | test_acc: 0.6094
Epoch: 5 | train_loss: 0.7037 | train_acc: 0.4880 | test_loss: 0.6697 | test_acc: 0.6094
Epoch: 6 | train_loss: 0.7250 | train_acc: 0.5024 | test_loss: 0.6691 | test_acc: 0.6094
Epoch: 7 | train_loss: 0.7350 | train_acc: 0.4784 | test_loss: 0.8184 | test_acc: 0.3906
Epoch: 8 | train_loss: 0.7273 | train_acc: 0.5072 | test_loss: 0.7912 | test_acc: 0.3906
Epoch: 9 | train_loss: 0.7519 | train_acc: 0.4591 | test_loss: 0.6846 | test_acc: 0.6094
Epoch: 10 | train_loss: 0.7300 | train_acc: 0.5457 | test_loss: 0.7424 | test_acc: 0.3906
Epoch: 11 | train_loss: 0.7531 | train_acc: 0.4736 | test_loss: 0.6746 | test_acc: 0.6094
Epoch: 12 | train_loss: 0.7323 | train_acc: 0.4784 | test_loss: 0.6905 | test_acc: 0.6094
Epoch: 13 | train_loss: 0.7184 | train_acc: 0.4760 | test_loss: 0.6719 | test_acc: 0.6094
Epoch: 14 | train_loss: 0.7177 | train_acc: 0.5048 | test_loss: 0.7026 | test_acc: 0.3906
Epoch: 15 | train_loss: 0.7115 | train_acc: 0.4856 | test_loss: 0.7252 | test_acc: 0.3906
```

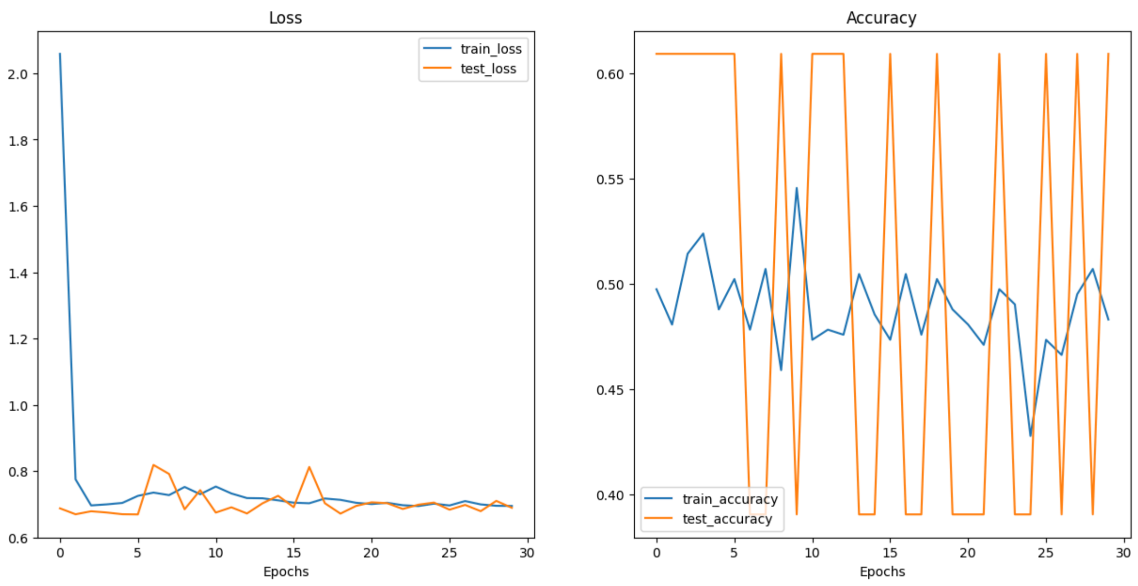
```
Epoch: 15 | train_loss: 0.7115 | train_acc: 0.4856 | test_loss: 0.7252 | test_acc: 0.3906
Epoch: 16 | train_loss: 0.7044 | train_acc: 0.4736 | test_loss: 0.6908 | test_acc: 0.6094
Epoch: 17 | train_loss: 0.7027 | train_acc: 0.5048 | test_loss: 0.8123 | test_acc: 0.3906
Epoch: 18 | train_loss: 0.7171 | train_acc: 0.4760 | test_loss: 0.7028 | test_acc: 0.3906
Epoch: 19 | train_loss: 0.7131 | train_acc: 0.5024 | test_loss: 0.6715 | test_acc: 0.6094
Epoch: 20 | train_loss: 0.7041 | train_acc: 0.4880 | test_loss: 0.6947 | test_acc: 0.3906
Epoch: 21 | train_loss: 0.7002 | train_acc: 0.4808 | test_loss: 0.7055 | test_acc: 0.3906
Epoch: 22 | train_loss: 0.7041 | train_acc: 0.4712 | test_loss: 0.7029 | test_acc: 0.3906
Epoch: 23 | train_loss: 0.6968 | train_acc: 0.4976 | test_loss: 0.6856 | test_acc: 0.6094
Epoch: 24 | train_loss: 0.6943 | train_acc: 0.4904 | test_loss: 0.6984 | test_acc: 0.3906
Epoch: 25 | train_loss: 0.7014 | train_acc: 0.4279 | test_loss: 0.7047 | test_acc: 0.3906
Epoch: 26 | train_loss: 0.6961 | train_acc: 0.4736 | test_loss: 0.6833 | test_acc: 0.6094
Epoch: 27 | train_loss: 0.7093 | train_acc: 0.4663 | test_loss: 0.6977 | test_acc: 0.3906
Epoch: 28 | train_loss: 0.6992 | train_acc: 0.4952 | test_loss: 0.6789 | test_acc: 0.6094
Epoch: 29 | train_loss: 0.6951 | train_acc: 0.5072 | test_loss: 0.7098 | test_acc: 0.3906
Epoch: 30 | train_loss: 0.6946 | train_acc: 0.4832 | test_loss: 0.6891 | test_acc: 0.6094
```

Congratulations! You have successfully trained your model! Now, you can plot the accuracy and loss curves of your model through *helper_functions.py*. You can also check the performance of your model on custom images present in your dataset and obtain a probability score through the *predictions.py* file. Both of these are already present in my repository, so you don't need to worry!

Results

In [18]:

```
from helper_functions import plot_loss_curves

# Plot our ViT model's loss curves
plot_loss_curves(results)
```



In [19]:

```
import requests
import predictions
from predictions import pred_and_plot_image
 #Setup custom image path
custom_image_path = "/content/MICCAI-Educational-Challenge-2024/test/uninfected/C240ThinI

# Predict on custom image
pred_and_plot_image(model=vit,
                    image_path=custom_image_path,
                    class_names=class_names)
```