Aim

To develop an image classifier capable of distinguishing between images of daisies and dandelions using the Swin Transformer architecture.

Objective

1. To preprocess images of daisies and dandelions for model input.
2. To implement the Swin Transformer model for image classification.
3. To train the model on the daisy and dandelion image dataset.
4. To evaluate the model's performance and visualize the results.

Summary

This project involves building an image classifier to differentiate between daisies and dandelions. The classifier leverages the Swin Transformer, a powerful model for computer vision tasks, known for its efficient handling of image patches and hierarchical feature representation. The dataset consists of images of daisies and dandelions, and the project is implemented using Jupyter Lab on a local machine with an RTX 3060 GPU.

Tools and Libraries Used

- Python
- PyTorch
- timm (for Swin Transformer)
- NumPy
- PIL (Python Imaging Library)
- torchvision
- matplotlib

Procedure

1. **Import Necessary Libraries:**

CODE:

```
import torch

import torch.nn as nn

from timm.models.layers import DropPath, to_2tuple, trunc_normal_

from PIL import Image

import torchvision.transforms as transforms
```

```
import matplotlib.pyplot as plt
```

2. **Load and Transform Image:**

CODE:

```
image_path = 'custom_dataset/test/daisy/sample_image.jpg'
image = Image.open(image_path)
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])
tensor_image = transform(image).unsqueeze(0)
```

3. **Partition Image into Patches:**

CODE:

```
def window_partition(x, patch_size=4):
    B, H, W, C = x.shape
    x = x.view(B, H // patch_size, patch_size, W // patch_size, patch_size, C)
    patches = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, patch_size, patch_size, C)
    return patches

tensor_image = tensor_image.permute(0, 2, 3, 1)
windows = window_partition(tensor_image, 4)
```

4. **Patch Embedding:**

CODE:

```
class PatchEmbed(nn.Module):
    def __init__(self, img_size=224, patch_size=4, in_chans=3, embed_dim=96, norm_layer=None):
        super().__init__()
        self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)
        self.norm = norm_layer(embed_dim) if norm_layer else None

    def forward(self, x):
```

```
        x = self.proj(x).flatten(2).transpose(1, 2)
        if self.norm:
            x = self.norm(x)
        return x

patch_embed = PatchEmbed()
embedding = patch_embed(tensor_image.permute(0, 3, 1, 2))
```

5. **Model Implementation and Training:**

CODE:

```
from swin_functions_and_classes import *
stage_1 = BasicLayer(dim=96, input_resolution=(56, 56), depth=2, num_heads=4, window_size=7)
output = stage_1(embedding)

merge_layer = PatchMerging(input_resolution=(56, 56), dim=96)
merged_output = merge_layer(output)

block_1 = SwinTransformerBlock(dim=96, input_resolution=(56, 56), num_heads=4, window_size=7)
output = block_1(embedding)

model = SwinTransformer(num_classes=2)
optimizer = torch.optim.Adam(model.parameters(), lr=3e-3, weight_decay=0.3)
loss_fn = nn.CrossEntropyLoss()

results = engine.train(model, train_dataloader, test_dataloader, optimizer, loss_fn, 10, device)
```

6. **Evaluation and Visualization:**

CODE:

```
from helper_functions import plot_loss_curves, pred_and_plot_image
plot_loss_curves(results)
custom_image_path = 'custom_dataset/test/daisy/sample_image.jpg'
```

pred_and_plot_image(model, custom_image_path, class_names)

Swin_functions_and_classes.py

CODE:

```
import torch

import torch.nn as nn

from timm.models.layers import DropPath, to_2tuple, trunc_normal_


def window_partition(x, patch_size=4):
    """
    Args:
        x: (B, H, W, C)
        patch_size (int): patch size (Default: 4)

    Returns:
        patches: (num_patches * B, patch_size, patch_size, C)
                (num_windows * B, patch_size, patch_size, C)
    """

    B, H, W, C = x.shape

    # Calculate the number of patches in each dimension
    num_patches_h = H // patch_size
    num_patches_w = W // patch_size

    # Convert to (B, num_patches_h, patch_size, num_patches_w, patch_size, C)
    x = x.view(B, num_patches_h, patch_size, num_patches_w, patch_size, C)

    # Convert to (B, num_patches_h, num_patches_w, patch_size, patch_size, C)
    patches = x.permute(0, 1, 3, 2, 4, 5).contiguous()
```

```
    # Efficient Batch Computation - Convert to (B * num_patches_h * num_patches_w, patch_size, patch_size, C)
    patches = patches.view(-1, patch_size, patch_size, C)

    return patches

# Lets use PatchEmbed

class PatchEmbed(nn.Module):
    """ Convert image to patch embedding

    Args:
        img_size (int): Image size (Default: 224)
        patch_size (int): Patch token size (Default: 4)
        in_channels (int): Number of input image channels (Default: 3)
        embed_dim (int): Number of linear projection output channels (Default: 96)
        norm_layer (nn.Module, optional): Normalization layer (Default: None)
    """

    def __init__(self, img_size=224, patch_size=4, in_chans=3, embed_dim=96, norm_layer=None):
        super().__init__()
        img_size = to_2tuple(img_size) # (img_size, img_size) to_2tuple simply convert t to (t,t)
        patch_size = to_2tuple(patch_size) # (patch_size, patch_size)
        patches_resolution = [img_size[0] // patch_size[0], img_size[1] // patch_size[1]] # (num_patches, num_patches)

        self.img_size = img_size
        self.patch_size = patch_size
```

```
    self.patches_resolution = patches_resolution
    self.num_patches = patches_resolution[0] * patches_resolution[1]

    self.in_chans = in_chans
    self.embed_dim = embed_dim

    # proj layer: (B, 3, 224, 224) -> (B, 96, 56, 56)
    self.proj = nn.Conv2d(in_chans, embed_dim, kernel_size=patch_size, stride=patch_size)

    if norm_layer is not None:
        self.norm = norm_layer(embed_dim)
    else:
        self.norm = None

def forward(self, x):
    """
    x: (B, C, H, W) Default: (B, 3, 224, 224)
    returns: (B, H//patch_size * W//patch_size, embed_dim) (B, 56*56, 96)
    """
    B, C, H, W = x.shape
    assert H == self.img_size[0] and W == self.img_size[1], \
        f"Input image size ({H}*{W}]) doesn't match model ({self.img_size[0]}*{self.img_size[1]})."

    # (B, 3, 224, 224) -> (B, 96, 56, 56)
    x = self.proj(x)

    # (B, 96, 56, 56) -> (B, 96, 56*56)
    x = x.flatten(2)
```

```
        # (B, 96, 56*56) -> (B, 56*56, 96): 56 refers to the number of patches
        x = x.transpose(1, 2)

        if self.norm is not None:
            x = self.norm(x)

        return x




def window_reverse(windows, window_size, H, W):
    """
    Args:
        windows: (num_windows * B, window_size, window_size, C)
                (8*8*B, 7, 7, C)
        window_size (int): window size (default: 7)
        H (int): Height of image (patch-wise)
        W (int): Width of image (patch-wise)

    Returns:
        x: (B, H, W, C)
    """

    # Get B from 8*8*B
    B = int(windows.shape[0] / (H * W / window_size / window_size))
```

```
    # Convert to (B, 8, 8, 7, 7, C)
    x = windows.view(B, H // window_size, W // window_size, window_size, window_size, -1)

    # Convert to (B, 8, 7, 8, 7, C)
    x = x.permute(0, 1, 3, 2, 4, 5).contiguous()

    # Convert to (B, H, W, C)
    x = x.view(B, H, W, -1)

    return x



# MLP of tranformer Block

class Mlp(nn.Module):
    def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU,
drop=0.):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features

        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act_layer = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)
```

```
        x = self.act_layer(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x



# W-MSA


class WindowAttention(nn.Module):
    """ Window based multi-head self attention(W-MSA) module with relative position bias.
        Used as Shifted-Window Multi-head self-attention(SW-MSA) by providing shift_size parameter in
        SwinTransformerBlock module


    Args:
        dim (int): Number of input channels (C)
        window_size (tuple[int]): The height and width of the window (M)
        num_heads (int): Number of attention heads for multi-head attention
        qkv_bias (bool, optional): If True, add a learnable bias to q, k, v (Default: True)
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
        attn_drop (float, optional): Dropout ratio of attention weight (Default: 0.0)
        proj_drop (float, optional): Dropout ratio of output (Default: 0.0)
    """


    def __init__(self, dim, window_size, num_heads, qkv_bias=True, qk_scale=None, attn_drop=0.,
proj_drop=0.):
        super().__init__()
        self.dim = dim
```

```
self.window_size = window_size # Wh(M), Ww(M) (7, 7)
self.num_heads = num_heads
head_dim = dim // num_heads
self.scale = qk_scale or head_dim ** -0.5

# Parameter table of relative position bias: B_hat from the paper
# (2M-1, 2M-1, num_heads) or (2*Wh-1 * 2*W-1, num_heads)
self.relative_position_bias_table = nn.Parameter(
    torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1), num_heads)
)

# Pair-wise relative position index for each token inside the window
coords_h = torch.arange(self.window_size[0])
coords_w = torch.arange(self.window_size[1])
coords = torch.stack(torch.meshgrid([coords_h, coords_w])) # (2, M, M) or (2, Wh, Ww)
coords_flatten = torch.flatten(coords, 1) # (2, M^2)

# None is dummy dimension
# coords_flatten[:, :, None] = (2, M^2, 1)
# coords_flatten[:, None, :] = (2, 1, M^2)
# relative_coords = (2, M^2, M^2)
relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]

# (2, M^2, M^2) -> (M^2, M^2, 2)
relative_coords = relative_coords.permute(1, 2, 0).contiguous()
relative_coords[:, :, 0] += self.window_size[0] - 1 # make it start from 0 index
relative_coords[:, :, 1] += self.window_size[1] - 1
relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1 # w.r.t x-axis
```

```
        # x-axis + y-axis
        relative_position_index = relative_coords.sum(-1) # (M^2, M^2)

        self.register_buffer('relative_position_index', relative_position_index)

        # Attention
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias) # W_Q, W_K, W_V
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

        trunc_normal_(self.relative_position_bias_table, std=.02)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x, mask=None):
        """
        Args:
            x: input features with shape of (num_windows*B, N, C), N refers to number of patches in a window (M^2)
            mask: (0/-inf) mask with shape of (num_windows, M^2, M^2) or None
                -> 0 means applying attention, -inf means removing attention
        """
        # (batch, M^2, C)
        B_, N, C = x.shape

        # (num_windows*B, N, 3C)
        qkv = self.qkv(x)
```

```
        # (B, N, 3, num_heads, C // num_heads)
        qkv = qkv.reshape(B_, N, 3, self.num_heads, C // self.num_heads)

        # Permute to (3, B_, num_heads, N, C // num_heads)
        '''
        3: referring to q, k, v (total 3)
        B: batch size
        num_heads: multi-headed attention
        N:  M^2, referring to each token(patch)
        C // num_heads: Each head of each of (q,k,v) handles C // num_heads -> match exact dimension for multi-headed attention
        '''
        qkv = qkv.permute(2, 0, 3, 1, 4)

        # Decompose to query/key/vector for attention
        # each of q, k, v has dimension of (B_, num_heads, N, C // num_heads)
        q, k, v = qkv[0], qkv[1], qkv[2] # Why not tuple-unpacking?

        q = q * self.scale

        # attn becomes (B_, num_heads, N, N) shape
        # N = M^2
        attn = (q @ k.transpose(-2, -1))

        # Remember that relative_position_bias_table = ((2M-1)*(2M-1), num_heads), B_hat from the paper
        # relative_position_index's elements are in range [0, 2M-2]
        # Convert to (M^2, M^2, num_heads). This is B matrix from the paper
        relative_position_bias = self.relative_position_bias_table[self.relative_position_index.view(-1)].view(
```

```
    self.window_size[0] * self.window_size[1], self.window_size[0] * self.window_size[1],  -1
)

# Convert to (num_heads, M^2, M^2) to match the dimension for addition
relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()

# (B, num_heads, N, N) + (1, num_heads, M^2, M^2), where N=M^2
# attn becomes (B_, num_heads, N, N) or (B, num_heads, M^2, M^2)
attn = attn + relative_position_bias.unsqueeze(0)

if mask is not None:
    nW = mask.shape[0] # nW = number of windows

    # attn.view(...) = (B, nW, num_heads, N, N)
    # mask.unsqueeze(1).unsqueeze(0) = (1, num_windows, 1, M^2, M^2)
    # So masking is broadcasted along B and num_heads axis which makes sense
    attn = attn.view(B_ // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)

    # attn = (nW * B, num_heads, N, N)
    attn = attn.view(-1, self.num_heads, N, N)
    attn = self.softmax(attn)
else:
    attn = self.softmax(attn)

attn = self.attn_drop(attn)

# attn = (nW*B, num_heads, N, N)
# v = (B_, num_heads, N, C // num_heads). B_ = nW*B
# attn @ v = (nW*B, num_heads, N, C // num_heads)
```

```
        # (attn @ v).transpose(1, 2) = (nW*B, N, num_heads, C // num_heads)
        # Finally, x = (nW*B, N, C), reshape(B_, N, C) performs concatenation of multi-headed attentions
        x = (attn @ v).transpose(1, 2).reshape(B_, N, C)

        # Projection Matrix (W_0). dim doesn't change since we used C // num_heads for MSA
        # x = (B_, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x




# Swin Transformer Block.

class SwinTransformerBlock(nn.Module):
    """ Swin Transformer Block. It's used as either W-MSA or SW-MSA depending on shift_size

    Args:
        dim (int): Number of input channels
        input_resolution (tuple[int]): Input resolution
        num_heads (int): Number of attention heads
        window_size (int): Window size
        shift_size (int): Shift size for SW-MSA
        mlp_ratio (float):Ratio of mlp hidden dim to embedding dim
        qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
        drop (float, optional): Dropout rate. Default: 0.0
        attn_drop (float, optional): Attention dropout rate. Default: 0.0
```

```
    drop_path (float, optional): Stochastic depth rate. Default: 0.0
    act_layer(nn.Module, optional): Activation layer. Default: nn.GELU
    norm_layer (nn.Module, optional): NOrmalization layer. Default: nn.LayerNorm
"""

def __init__(self, dim, input_resolution, num_heads, window_size=7, shift_size=0,
             mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0., drop_path=0.,
             act_layer=nn.GELU, norm_layer=nn.LayerNorm
            ):
    super().__init__()
    self.dim = dim
    self.input_resolution = input_resolution
    self.num_heads = num_heads
    self.window_size = window_size
    self.shift_size = shift_size
    self.mlp_ratio = mlp_ratio

    # If window_size > input_resolution, no partition
    if min(self.input_resolution) <= self.window_size:
        self.shift_size = 0
        self.window_size = min(self.input_resolution)
    assert 0 <= self.shift_size < self.window_size, "shift_size must in 0-window_size"

    self.norm1 = norm_layer(dim)

    # Attention
    self.attn = WindowAttention(
        dim, window_size=to_2tuple(self.window_size), num_heads=num_heads,
        qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=drop
```

```
)

self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
self.norm2 = norm_layer(dim)

# MLP
mlp_hidden_dim = int(dim * mlp_ratio)
self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop)

# Attention Mask for SW-MSA
# This handling of attention-mask is my favourite part. What a beautiful implementation.
if self.shift_size > 0:
    H, W = self.input_resolution

    # To match the dimension for window_partition function
    img_mask = torch.zeros((1, H, W, 1))

    # h_slices and w_slices divide a cyclic-shifted image to 9 regions as shown in the paper
    h_slices = (
        slice(0, -self.window_size),
        slice(-self.window_size, -self.shift_size),
        slice(-self.shift_size, None)
    )

    w_slices = (
        slice(0, -self.window_size),
        slice(-self.window_size, -self.shift_size),
        slice(-self.shift_size, None)
    )
```

```
            # Fill out number for each of 9 divided regions
            cnt = 0
            for h in h_slices:
                for w in w_slices:
                    img_mask[:, h, w, :] = cnt
                    cnt += 1

            mask_windows = window_partition(img_mask, self.window_size) # (nW, M, M, 1)
            mask_windows = mask_windows.view(-1, self.window_size * self.window_size)

            # Such a gorgeous code..
            attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2)
            attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-100.0)).masked_fill(attn_mask == 0,
float(0.0))
        else:
            attn_mask = None

        self.register_buffer('attn_mask', attn_mask)


    def forward(self, x):
        H, W = self.input_resolution
        B, L, C = x.shape
        assert L == H * W, "input feature has wrong size"

        shortcut = x # Residual
        x = self.norm1(x)
        x = x.view(B, H, W, C) # H, W refer to the number of "patches" for width and height, not "pixels"
```

```
        # Cyclic Shift
        if self.shift_size > 0:
            shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
        else:
            shifted_x = x

        # Partition Windows
        x_windows = window_partition(shifted_x, self.window_size) # (nW*B, M, M, C)
        x_windows = x_windows.view(-1, self.window_size*self.window_size, C) # (nW*B,
window_size*window_size, C)

        # W-MSA / SW-MSA
        attn_windows = self.attn(x_windows, mask=self.attn_mask) # (nW*B, window_size*window_size, C)

        # Merge Windows
        attn_windows = attn_windows.view(-1, self.window_size, self.window_size, C)
        shifted_x = window_reverse(attn_windows, self.window_size, H, W) # (B, H', W', C)

        # Reverse Cyclic Shift
        if self.shift_size > 0:
            x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
        else:
            x = shifted_x
        x = x.view(B, H*W, C)

        # FFn
        x = shortcut + self.drop_path(x)
        x = x + self.drop_path(self.mlp(self.norm2(x)))
```

```
        return x



# Patch Merging Layer from the paper (downsampling)

class PatchMerging(nn.Module):
    """ Patch Merging Layer from the paper (downsampling)
    Args:
        input_solution (tuple[int]): Resolution of input feature
        dim (int): Number of input channels. (C)
        norm_layer (nn.Module, optional): Normalization layer. (Default: nn.LayerNorm)
    """

    def __init__(self, input_resolution, dim, norm_layer=nn.LayerNorm):
        super().__init__()
        self.input_resolution = input_resolution
        self.dim = dim
        self.reduction = nn.Linear(4 * dim, 2 * dim, bias=False)
        self.norm = norm_layer(4 * dim)

    def forward(self, x):
        """
        x: (B, H*W, C)
        """

        H, W = self.input_resolution
        B, L, C = x.shape
```

```
        assert L == H * W, "input feature has wrong size"
        assert H % 2 == 0 and W % 2 == 0, f"x size ({H}*{W}) are not even."

        x = x.view(B, H, W, C)

        # Separate per patch by 2 x 2
        x0 = x[:, 0::2, 0::2, :] # (B, H/2, W/2, C) (top-left of 2x2)
        x1 = x[:, 1::2, 0::2, :] # (B, H/2, W/2, C) (bottom-left of 2x2)
        x2 = x[:, 0::2, 1::2, :] # (B, H/2, W/2, C) (top-right of 2x2)
        x3 = x[:, 1::2, 1::2, :] # (B, H/2, W/2, C) (bottom-right of 2x2)

        # Merge by channel -> (B, H/2, W/2, 4C)   # Merging 4 patches- thats why 4C
        x = torch.cat([x0, x1, x2, x3], -1)

        # Flatten H, W
        x = x.view(B, -1, 4 * C)

        x = self.norm(x)

        # Reduction Layer: 4C -> 2C
        x = self.reduction(x)

        return x


# Swin Transformer layer for one stage

class BasicLayer(nn.Module):
```

```
    """ Swin Transformer layer for one stage

    Args:
        dim (int): Number of input channels
        input_resolution (tuple[int]): Input resolution
        depth (int): Number of blocks (depending on Swin Version - T, L, ..)
        num_heads (int): Number of attention heads
        window_size (int): Local window size
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim
        qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. (Default: True)
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
        drop (float, optional): Dropout rate (Default: 0.0)
        attn_drop (float, optional): Attention dropout rate (Default: 0.0)
        drop_path (float | tuple[float], optional): Stochastic depth rate (Default: 0.0)
        norm_layer (nn.Module, optional): Normalization layer (Default: nn.LayerNorm)
        downsample (nn.Module | NOne, optional): Downsample layer at the end of the layer (Default:
None)
        use_checkpoint (bool): Whether to use checkpointing to save memory (Default: False)
    """

    def __init__(self, dim, input_resolution, depth, num_heads, window_size,
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
                 drop_path=0., norm_layer=nn.LayerNorm, downsample=None, use_checkpoint=False):
        super().__init__()
        self.dim = dim
        self.input_resolution = input_resolution
        self.depth = depth
        self.use_checkpoint = use_checkpoint
```

```
    # Build  Swin-Transformer Blocks
    self.blocks = nn.ModuleList([
        SwinTransformerBlock(dim=dim, input_resolution=input_resolution,
                             num_heads=num_heads, window_size=window_size,
                             shift_size=0 if (i % 2 == 0) else window_size // 2,
                             mlp_ratio = mlp_ratio,
                             qkv_bias=qkv_bias, qk_scale=qk_scale,
                             drop=drop, attn_drop=attn_drop,
                             drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
                             norm_layer=norm_layer
                            )
        for i in range(depth)
    ])



    # Patch Merging Layer
    if downsample is not None:
        self.downsample = downsample(input_resolution, dim=dim, norm_layer=norm_layer)
    else:
        self.downsample = None



def forward(self, x):
    for blk in self.blocks:
        # if self.use_checkpoint:
        #     x = checkpoint.checkpoint(blk, x)
        # else:
        #     x = blk(x)
        x = blk(x)
```

```
        if self.downsample is not None:
            x = self.downsample(x)

        return x


# SwinTransformer

class SwinTransformer(nn.Module):
    """ Swin Transformer

    Args:
        img_size (int | tuple(int)): Input image size (Default 224)
        patch_size (int | tuple(int)): Patch size (Default: 4)
        in_chans (int): Number of input image channels (Default: 3)
        num_classes (int): Number of classes for classification head (Default: 1000)
        embed_dim (int): Patch embedding dimension (Default: 96)
        depths (tuple(int)): Depth of each Swin-T layer
        num_heads (tuple(int)): Number of attention heads in different layers
        window_size (int): Window size (Default: 7)
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim. (Default: 4)
        qkv_bias (bool): If True, add a learnable bias to query, key, value (Default: True)
        qk_scale (float); Override default qk scale of head_dim ** -0.5 if set. (Default: None)
        drop_rate (float): Dropout rate (Default: 0)
        attn_drop_rate (float): Attention dropout rate (Default: 0)
        drop_path_rate (float); Stochastic depth rate (Default: 0.1)
        norm_layer (nn.Module): Normalization layer (Default: nn.LayerNorm)
```

```
    ape (bool): Refers to absolute position embedding. If True, add ape to the patch embedding (Default: False)
    patch_norm (bool): If True, add normalization after patch embedding (Default: True)
    use_checkpoint (bool): Whether to use checkpointing to save memory (Default: False)
    """

    def __init__(self, img_size=224, patch_size=4, in_chans=3, num_classes=1000,
                 embed_dim=96, depths=[2, 2, 6, 2], num_heads=[3, 6, 12, 24],
                 window_size=7, mlp_ratio=4., qkv_bias=True, qk_scale=None,
                 drop_rate=0., attn_drop_rate=0., drop_path_rate=0.1,
                 norm_layer=nn.LayerNorm, ape=False, patch_norm=True,
                 use_checkpoint=False, **kwargs):
        super().__init__()

        self.num_classes = num_classes
        self.num_layers = len(depths)
        self.embed_dim = embed_dim
        self.ape = ape
        self.patch_norm = patch_norm
        self.num_features = int(embed_dim * 2 ** (self.num_layers - 1))
        self.mlp_ratio = mlp_ratio

        # Split image into non-overlapping patches
        self.patch_embed = PatchEmbed(
            img_size=img_size, patch_size=patch_size, in_chans=in_chans, embed_dim=embed_dim,
            norm_layer=norm_layer if self.patch_norm else None
        )
        num_patches = self.patch_embed.num_patches
        patches_resolution = self.patch_embed.patches_resolution
```

```
        self.patches_resolution = patches_resolution

        self.pos_drop = nn.Dropout(p=drop_rate)

        # Stochastic Depth
        dpr = [x.item() for x in torch.linspace(0, drop_path_rate, sum(depths))] # stochastic depth decay rule

        # build layers
        self.layers = nn.ModuleList()
        for i_layer in range(self.num_layers):
            layer = BasicLayer(
                dim=int(embed_dim * 2 ** i_layer),
                input_resolution=(
                    patches_resolution[0] // (2 ** i_layer), # After patch-merging layer, patches_resolution(H, W)
is halved
                    patches_resolution[1] // (2 ** i_layer),
                            ),
                depth=depths[i_layer],
                num_heads=num_heads[i_layer],
                window_size=window_size,
                mlp_ratio=self.mlp_ratio,
                qkv_bias=qkv_bias, qk_scale=qk_scale,
                drop=drop_rate, attn_drop=attn_drop_rate,
                drop_path=dpr[sum(depths[:i_layer]):sum(depths[:i_layer + 1])],
                norm_layer=norm_layer,
                downsample=PatchMerging if (i_layer < self.num_layers -1) else None, # No patch merging at
the last stage
                use_checkpoint=use_checkpoint
            )
```

```
            self.layers.append(layer)

        self.norm = norm_layer(self.num_features)
        self.avgpool = nn.AdaptiveAvgPool1d(1)

        # Classification Head
        self.head = nn.Linear(self.num_features, num_classes) if num_classes > 0 else nn.Identity()

        self.apply(self._init_weights)



    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            trunc_normal_(m.weight, std=.02)
            if isinstance(m, nn.Linear) and m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.LayerNorm):
            nn.init.constant_(m.bias, 0)
            nn.init.constant_(m.weight, 1.0)



    def forward_features(self, x):
        x = self.patch_embed(x)
        # if self.ape:
        #     x = x + self.absolute_pos_embed
        x = self.pos_drop(x)

        for layer in self.layers:
```

```
            x = layer(x)

        x = self.norm(x) # (B, L, C)
        x = self.avgpool(x.transpose(1, 2)) # (B, C, 1)
        x = torch.flatten(x, 1)
        return x

    def forward(self, x):
        x = self.forward_features(x)
        x = self.head(x)
        return x
```

```
# # Swin Transformer layer for one stage

class BasicLayer(nn.Module):
    """ Swin Transformer layer for one stage

    Args:
        dim (int): Number of input channels
        input_resolution (tuple[int]): Input resolution
        depth (int): Number of blocks (depending on Swin Version - T, L, ..)
```

```
    num_heads (int): Number of attention heads
    window_size (int): Local window size
    mlp_ratio (float): Ratio of mlp hidden dim to embedding dim
    qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. (Default: True)
    qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set
    drop (float, optional): Dropout rate (Default: 0.0)
    attn_drop (float, optional): Attention dropout rate (Default: 0.0)
    drop_path (float | tuple[float], optional): Stochastic depth rate (Default: 0.0)
    norm_layer (nn.Module, optional): Normalization layer (Default: nn.LayerNorm)
    downsample (nn.Module | NOne, optional): Downsample layer at the end of the layer (Default: None)
    use_checkpoint (bool): Whether to use checkpointing to save memory (Default: False)
  """

  def __init__(self, dim, input_resolution, depth, num_heads, window_size,
               mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
               drop_path=0., norm_layer=nn.LayerNorm, downsample=None, use_checkpoint=False):
    super().__init__()
    self.dim = dim
    self.input_resolution = input_resolution
    self.depth = depth
    self.use_checkpoint = use_checkpoint

    # Build  Swin-Transformer Blocks
    self.blocks = nn.ModuleList([
      SwinTransformerBlock(dim=dim, input_resolution=input_resolution,
                           num_heads=num_heads, window_size=window_size,
                           shift_size=0 if (i % 2 == 0) else window_size // 2,
                           mlp_ratio = mlp_ratio,
```

```
                    qkv_bias=qkv_bias, qk_scale=qk_scale,
                    drop=drop, attn_drop=attn_drop,
                    drop_path=drop_path[i] if isinstance(drop_path, list) else drop_path,
                    norm_layer=norm_layer
                   )
      for i in range(depth)
   ])



   # Patch Merging Layer
   if downsample is not None:
      self.downsample = downsample(input_resolution, dim=dim, norm_layer=norm_layer)
   else:
      self.downsample = None



def forward(self, x):
   for blk in self.blocks:
      # if self.use_checkpoint:
      #     x = checkpoint.checkpoint(blk, x)
      # else:
      #     x = blk(x)
      x = blk(x)

   if self.downsample is not None:
      x = self.downsample(x)

   return x
```

Highlights

- **Patch Embedding:** Efficiently converts images into patches for the Swin Transformer model.
- **Hierarchical Structure:** The Swin Transformer processes images hierarchically, making it suitable for detailed image analysis.
- **Attention Mechanisms:** Utilizes self-attention and shifted window mechanisms to capture intricate patterns in images.
- **Training Pipeline:** A robust training pipeline using PyTorch's DataLoader, optimizer, and loss function.

Conclusion

The Swin Transformer-based image classifier effectively distinguishes between images of daisies and dandelions. The hierarchical processing and attention mechanisms of the Swin Transformer provide a strong foundation for image classification tasks. The project demonstrates the model's capability to handle complex image data and achieve accurate classification results.