Mempersiapkan

```
# For this notebook to run with updated APIs, we need torch 1.12+ and
torchvision 0.13+
try:
    import torch
    import torchvision
    assert int(torch.__version__.split(".")[1]) >= 12 or
int(torch.__version__.split(".")[0]) == 2, "torch version should be
1.12+"
    assert int(torchvision.__version__.split(".")[1]) >= 13,
"torchvision version should be 0.13+"
    print(f"torch version: {torch.__version__}")
    print(f"torchvision version: {torchvision.__version__}")
except:
    print(f"[INFO] torch/torchvision versions not as required,
installing nightly versions.")
    !pip3 install -U torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
    import torch
    import torchvision
    print(f"torch version: {torch.__version__}")
    print(f"torchvision version: {torchvision.__version__}")
```

```
torch version: 2.1.0+cu118
torchvision version: 0.16.0+cu118
```

```
# Continue with regular imports
import matplotlib.pyplot as plt
import torch
import torchvision

from torch import nn
from torchvision import transforms

# Try to get torchinfo, install it if it doesn't work
try:
    from torchinfo import summary
except:
    print("[INFO] Couldn't find torchinfo... installing it.")
    !pip install -q torchinfo
    from torchinfo import summary

# Try to import the going_modular directory, download it from GitHub
if it doesn't work
try:
    from going_modular.going_modular import data_setup, engine
    from helper_functions import download_data, set_seeds,
plot_loss_curves
except:
```

```
    # Get the going_modular scripts
    print("[INFO] Couldn't find going_modular or helper_functions
scripts... downloading them from GitHub.")
    !git clone https://github.com/mrdbourke/pytorch-deep-learning
    !mv pytorch-deep-learning/going_modular .
    !mv pytorch-deep-learning/helper_functions.py . # get the
helper_functions.py script
    !rm -rf pytorch-deep-learning
    from going_modular.going_modular import data_setup, engine
    from helper_functions import download_data, set_seeds,
plot_loss_curves
```

```
[INFO] Couldn't find torchinfo... installing it.
[INFO] Couldn't find going_modular or helper_functions scripts...
downloading them from GitHub.
Cloning into 'pytorch-deep-learning'...
remote: Enumerating objects: 4033, done.ote: Counting objects: 100%
(1224/1224), done.ote: Compressing objects: 100% (225/225), done.ote:
Total 4033 (delta 1067), reused 1097 (delta 996), pack-reused 2809
```

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

```
'cuda'
```

Dapatkan Data

```
# Download pizza, steak, sushi images from GitHub
image_path =
download_data(source="https://github.com/mrdbourke/pytorch-deep-
learning/raw/main/data/pizza_steak_sushi.zip",
                           destination="pizza_steak_sushi")
image_path
```

```
[INFO] data/pizza_steak_sushi directory exists, skipping download.
```

```
PosixPath('data/pizza_steak_sushi')
```

```
# Setup directory paths to train and test images
train_dir = image_path / "train"
test_dir = image_path / "test"
```

Buat Kumpulan Data dan Pemuat Data

Siapkan transformasi untuk gambar

```
# Create image size (from Table 3 in the ViT paper)
IMG_SIZE = 224

# Create transform pipeline manually
```

```
manual_transforms = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
])
print(f"Manually created transforms: {manual_transforms}")
```

```
Manually created transforms: Compose(
    Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)
    ToTensor()
)
```

Ubah gambar menjadi DataLoader

```
# Set the batch size
BATCH_SIZE = 32 # this is lower than the ViT paper but it's because we're starting small

# Create data loaders
train_dataloader, test_dataloader, class_names = data_setup.create_dataloaders(
    train_dir=train_dir,
    test_dir=test_dir,
    transform=manual_transforms, # use manually created transforms
    batch_size=BATCH_SIZE
)

train_dataloader, test_dataloader, class_names
```

```
(<torch.utils.data.dataloader.DataLoader at 0x7f18845ff0d0>,
 <torch.utils.data.dataloader.DataLoader at 0x7f17f3f5f520>,
 ['pizza', 'steak', 'sushi'])
```

Visualisasikan satu gambar

```
# Get a batch of images
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
image.shape, label
```

```
(torch.Size([3, 224, 224]), tensor(2))
```

```
# Plot image with matplotlib
plt.imshow(image.permute(1, 2, 0)) # rearrange image dimensions to suit matplotlib [color_channels, height, width] -> [height, width, color_channels]
```

```
plt.title(class_names[label])
plt.axis(False);
```

Mereplikasi makalah ViT: gambaran umum

Input dan output, lapisan dan blok

Lebih spesifik: ViT terbuat dari apa?

Menghitung bentuk input dan output yang menyematkan patch dengan tangan

```
# Create example values
height = 224 # H ("The training resolution is 224.")
width = 224 # W
color_channels = 3 # C
patch_size = 16 # P

# Calculate N (number of patches)
number_of_patches = int((height * width) / patch_size**2)
print(f"Number of patches (N) with image height (H={height}), width
(W={width}) and patch size (P={patch_size}): {number_of_patches}")
```

```
Number of patches (N) with image height (H=224), width (W=224) and
patch size (P=16): 196
```

```
# Input shape (this is the size of a single image)
embedding_layer_input_shape = (height, width, color_channels)
```

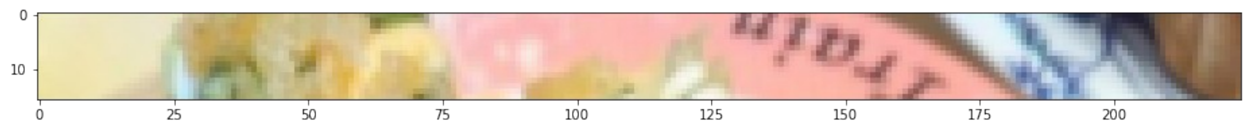
```
# Output shape
embedding_layer_output_shape = (number_of_patches, patch_size**2 * color_channels)

print(f"Input shape (single 2D image): {embedding_layer_input_shape}")
print(f"Output shape (single 2D image flattened into patches): {embedding_layer_output_shape}")
```

```
Input shape (single 2D image): (224, 224, 3)
Output shape (single 2D image flattened into patches): (196, 768)
```

Mengubah satu gambar menjadi tambalan

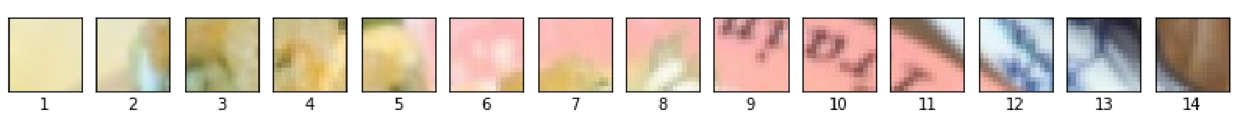
```
# View single image
plt.imshow(image.permute(1, 2, 0)) # adjust for matplotlib
plt.title(class_names[label])
plt.axis(False);
```


```
# Change image shape to be compatible with matplotlib (color_channels, height, width) -> (height, width, color_channels)
image_permuted = image.permute(1, 2, 0)

# Index to plot the top row of patched pixels
patch_size = 16
plt.figure(figsize=(patch_size, patch_size))
plt.imshow(image_permuted[:patch_size, :, :]);
```




```
# Setup hyperparameters and make sure img_size and patch_size are
compatible
img_size = 224
patch_size = 16
num_patches = img_size/patch_size
assert img_size % patch_size == 0, "Image size must be divisible by
patch size"
print(f"Number of patches per row: {num_patches}\nPatch size:
{patch_size} pixels x {patch_size} pixels")

# Create a series of subplots
fig, axs = plt.subplots(nrows=1,
                        ncols=img_size // patch_size, # one column for
each patch
                        figsize=(num_patches, num_patches),
                        sharex=True,
                        sharey=True)

# Iterate through number of patches in the top row
for i, patch in enumerate(range(0, img_size, patch_size)):
    axs[i].imshow(image_permuted[:patch_size,
patch:patch+patch_size, :]); # keep height index constant, alter the
width index
    axs[i].set_xlabel(i+1) # set the label
    axs[i].set_xticks([])
    axs[i].set_yticks([])
```

```
Number of patches per row: 14.0
Patch size: 16 pixels x 16 pixels
```

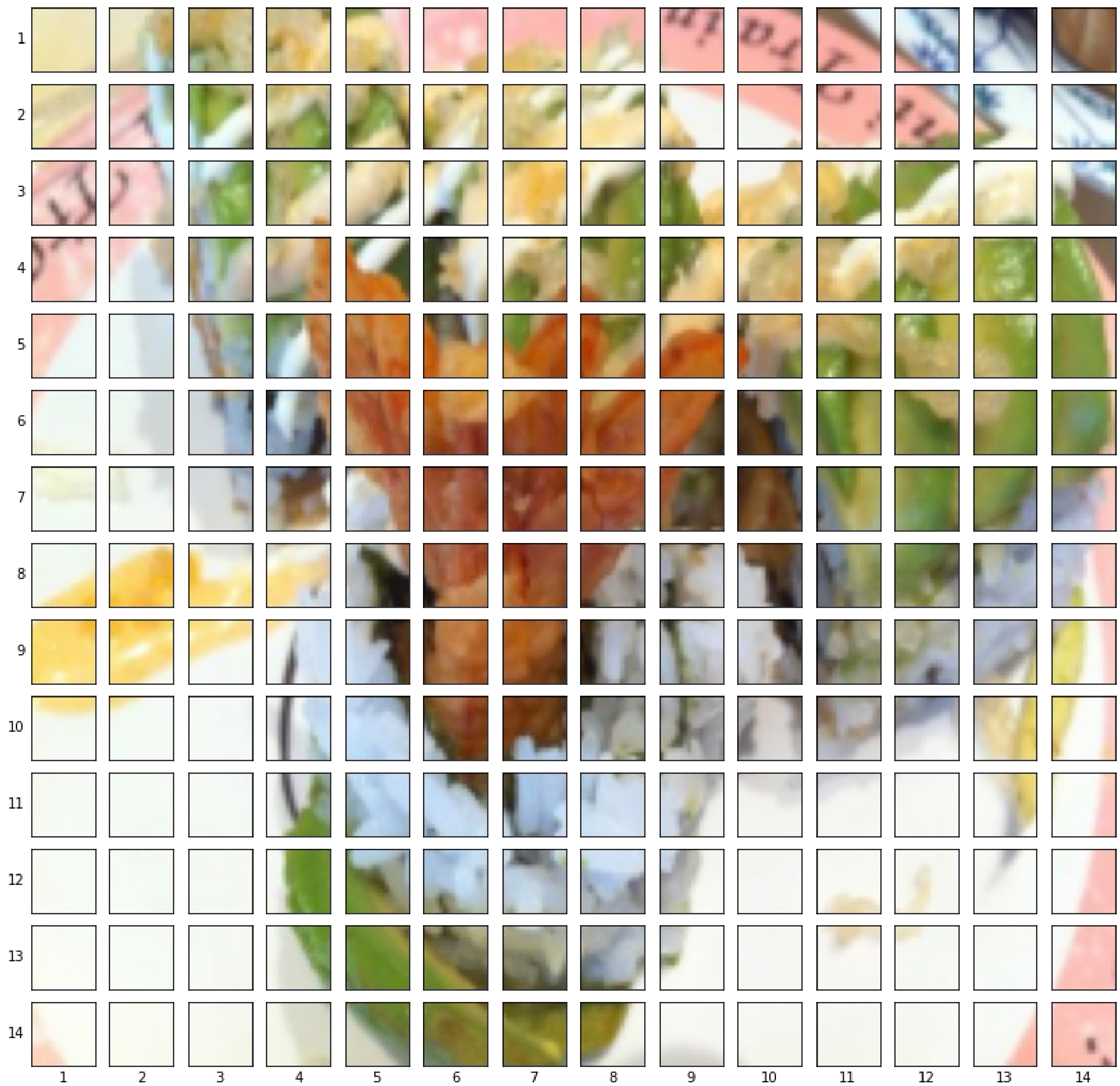



```
# Setup hyperparameters and make sure img_size and patch_size are
compatible
img_size = 224
patch_size = 16
num_patches = img_size/patch_size
assert img_size % patch_size == 0, "Image size must be divisible by
patch size"
print(f"Number of patches per row: {num_patches}\
        \nNumber of patches per column: {num_patches}\
        \nTotal patches: {num_patches*num_patches}\
```

```
        \nPatch size: {patch_size} pixels x {patch_size} pixels")

# Create a series of subplots
fig, axs = plt.subplots(nrows=img_size // patch_size, # need int not
float
                        ncols=img_size // patch_size,
                        figsize=(num_patches, num_patches),
                        sharex=True,
                        sharey=True)

# Loop through height and width of image
for i, patch_height in enumerate(range(0, img_size, patch_size)): #
iterate through height
    for j, patch_width in enumerate(range(0, img_size, patch_size)): #
iterate through width

        # Plot the permuted image patch (image_permuted -> (Height,
Width, Color Channels))
        axs[i,
j].imshow(image_permuted[patch_height:patch_height+patch_size, #
iterate through height

patch_width:patch_width+patch_size, # iterate through width
                                        :]) # get all color channels

        # Set up label information, remove the ticks for clarity and
set labels to outside
        axs[i, j].set_ylabel(i+1,
                             rotation="horizontal",
                             horizontalalignment="right",
                             verticalalignment="center")
        axs[i, j].set_xlabel(j+1)
        axs[i, j].set_xticks([])
        axs[i, j].set_yticks([])
        axs[i, j].label_outer()

# Set a super title
fig.suptitle(f"{class_names[label]} -> Patchified", fontsize=16)
plt.show()
```

```
Number of patches per row: 14.0
Number of patches per column: 14.0
Total patches: 196.0
Patch size: 16 pixels x 16 pixels
```

sushi -> Patchified



Membuat patch gambar dengan torch.nn.Conv2d()

```
from torch import nn

# Set the patch size
patch_size=16

# Create the Conv2d layer with hyperparameters from the ViT paper
conv2d = nn.Conv2d(in_channels=3, # number of color channels
```

```
                  out_channels=768, # from Table 1: Hidden size D,
this is the embedding size
                  kernel_size=patch_size, # could also use
(patch_size, patch_size)
                  stride=patch_size,
                  padding=0)

# View single image
plt.imshow(image.permute(1, 2, 0)) # adjust for matplotlib
plt.title(class_names[label])
plt.axis(False);
```

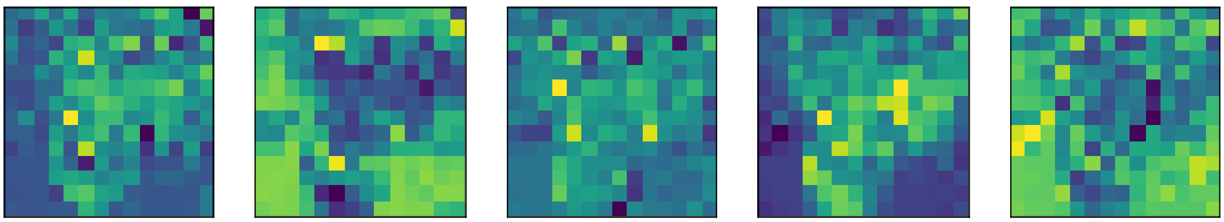

```
# Pass the image through the convolutional layer
image_out_of_conv = conv2d(image.unsqueeze(0)) # add a single batch
dimension (height, width, color_channels) -> (batch, height, width,
color_channels)
print(image_out_of_conv.shape)
```

```
torch.Size([1, 768, 14, 14])
```

```
# Plot random 5 convolutional feature maps
import random
random_indexes = random.sample(range(0, 758), k=5) # pick 5 numbers
between 0 and the embedding size
print(f"Showing random convolutional feature maps from indexes:
{random_indexes}")

# Create plot
fig, axs = plt.subplots(nrows=1, ncols=5, figsize=(12, 12))
```

```
# Plot random image feature maps
for i, idx in enumerate(random_indexes):
    image_conv_feature_map = image_out_of_conv[:, idx, :, :] # index
on the output tensor of the convolutional layer
    axs[i].imshow(image_conv_feature_map.squeeze().detach().numpy())
    axs[i].set(xticklabels=[], yticklabels=[], xticks=[], yticks=[]);
```

```
Showing random convolutional feature maps from indexes: [571, 727,
734, 380, 90]
```



```
# Get a single feature map in tensor form
single_feature_map = image_out_of_conv[:, 0, :, :]
single_feature_map, single_feature_map.requires_grad
```

```
(tensor([[[ 0.4732,  0.3567,  0.3377,  0.3736,  0.3208,  0.3913,
0.3464,
            0.3702,  0.2541,  0.3594,  0.1984,  0.3982,  0.3741,
0.1251],
          [ 0.4178,  0.4771,  0.3374,  0.3353,  0.3159,  0.4008,
0.3448,
            0.3345,  0.5850,  0.4115,  0.2969,  0.2751,  0.6150,
0.4188],
          [ 0.3209,  0.3776,  0.4970,  0.4272,  0.3301,  0.4787,
0.2754,
            0.3726,  0.3298,  0.4631,  0.3087,  0.4915,  0.4129,
0.4592],
          [ 0.4540,  0.4930,  0.5570,  0.2660,  0.2150,  0.2044,
0.2766,
            0.2076,  0.3278,  0.3727,  0.2637,  0.2493,  0.2782,
0.3664],
          [ 0.4920,  0.5671,  0.3298,  0.2992,  0.1437,  0.1701,
0.1554,
            0.1375,  0.1377,  0.3141,  0.2694,  0.2771,  0.2412,
0.3700],
          [ 0.5783,  0.5790,  0.4229,  0.5032,  0.1216,  0.1000,
0.0356,
            0.1258, -0.0023,  0.1640,  0.2809,  0.2418,  0.2606,
0.3787],
          [ 0.5334,  0.5645,  0.4781,  0.3307,  0.2391,  0.0461,
0.0095,
            0.0542,  0.1012,  0.1331,  0.2446,  0.2526,  0.3323,
```

```
0.4120],
          [ 0.5724,  0.2840,  0.5188,  0.3934,  0.1328,  0.0776,
0.0235,
            0.1366,  0.3149,  0.2200,  0.2793,  0.2351,  0.4722,
0.4785],
          [ 0.4009,  0.4570,  0.4972,  0.5785,  0.2261,  0.1447, -
0.0028,
            0.2772,  0.2697,  0.4008,  0.3606,  0.3372,  0.4535,
0.4492],
          [ 0.5678,  0.5870,  0.5824,  0.3438,  0.5113,  0.0757,
0.1772,
            0.3677,  0.3572,  0.3742,  0.3820,  0.4868,  0.3781,
0.4694],
          [ 0.5845,  0.5877,  0.5826,  0.3212,  0.5276,  0.4840,
0.4825,
            0.5523,  0.5308,  0.5085,  0.5606,  0.5720,  0.4928,
0.5581],
          [ 0.5853,  0.5849,  0.5793,  0.3410,  0.4428,  0.4044,
0.3275,
            0.4958,  0.4366,  0.5750,  0.5494,  0.5868,  0.5557,
0.5069],
          [ 0.5880,  0.5888,  0.5796,  0.3377,  0.2635,  0.2347,
0.3145,
            0.3486,  0.5158,  0.5722,  0.5347,  0.5753,  0.5816,
0.4378],
          [ 0.5692,  0.5843,  0.5721,  0.5081,  0.2694,  0.2032,
0.1589,
            0.3464,  0.5349,  0.5768,  0.5739,  0.5764,  0.5394,
0.4482]]],
        grad_fn=<SliceBackward0>),
 True)
```

Meratakan penyematan tambalan dengan torch.nn.Flatten()

```
# Current tensor shape
print(f"Current tensor shape: {image_out_of_conv.shape} -> [batch,
embedding_dim, feature_map_height, feature_map_width]")
```

```
Current tensor shape: torch.Size([1, 768, 14, 14]) -> [batch,
embedding_dim, feature_map_height, feature_map_width]
```

```
# Create flatten layer
flatten = nn.Flatten(start_dim=2, # flatten feature_map_height
(dimension 2)
                     end_dim=3) # flatten feature_map_width (dimension
3)

# 1. View single image
plt.imshow(image.permute(1, 2, 0)) # adjust for matplotlib
plt.title(class_names[label])
```

```
plt.axis(False);
print(f"Original image shape: {image.shape}")

# 2. Turn image into feature maps
image_out_of_conv = conv2d(image.unsqueeze(0)) # add batch dimension
to avoid shape errors
print(f"Image feature map shape: {image_out_of_conv.shape}")

# 3. Flatten the feature maps
image_out_of_conv_flattened = flatten(image_out_of_conv)
print(f"Flattened image feature map shape:
{image_out_of_conv_flattened.shape}")
```

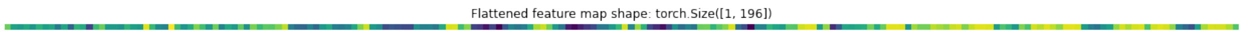
```
Original image shape: torch.Size([3, 224, 224])
Image feature map shape: torch.Size([1, 768, 14, 14])
Flattened image feature map shape: torch.Size([1, 768, 196])
```

```
# Get flattened image patch embeddings in right shape
image_out_of_conv_flattened_reshaped =
image_out_of_conv_flattened.permute(0, 2, 1) # [batch_size, P^2•C, N]
-> [batch_size, N, P^2•C]
print(f"Patch embedding sequence shape:
{image_out_of_conv_flattened_reshaped.shape} -> [batch_size,
num_patches, embedding_size]")
```

```
Patch embedding sequence shape: torch.Size([1, 196, 768]) ->
[batch_size, num_patches, embedding_size]
```

```
# Get a single flattened feature map
single_flattened_feature_map = image_out_of_conv_flattened_reshaped[:,
```

```
:, 0] # index: (batch_size, number_of_patches, embedding_dimension)

# Plot the flattened feature map visually
plt.figure(figsize=(22, 22))
plt.imshow(single_flattened_feature_map.detach().numpy())
plt.title(f"Flattened feature map shape:
{single_flattened_feature_map.shape}")
plt.axis(False);
```




```
# See the flattened feature map as a tensor
single_flattened_feature_map,
single_flattened_feature_map.requires_grad,
single_flattened_feature_map.shape
```

```
(tensor([[ 0.4732,  0.3567,  0.3377,  0.3736,  0.3208,  0.3913,
0.3464,  0.3702,
           0.2541,  0.3594,  0.1984,  0.3982,  0.3741,  0.1251,
0.4178,  0.4771,
           0.3374,  0.3353,  0.3159,  0.4008,  0.3448,  0.3345,
0.5850,  0.4115,
           0.2969,  0.2751,  0.6150,  0.4188,  0.3209,  0.3776,
0.4970,  0.4272,
           0.3301,  0.4787,  0.2754,  0.3726,  0.3298,  0.4631,
0.3087,  0.4915,
           0.4129,  0.4592,  0.4540,  0.4930,  0.5570,  0.2660,
0.2150,  0.2044,
           0.2766,  0.2076,  0.3278,  0.3727,  0.2637,  0.2493,
0.2782,  0.3664,
           0.4920,  0.5671,  0.3298,  0.2992,  0.1437,  0.1701,
0.1554,  0.1375,
           0.1377,  0.3141,  0.2694,  0.2771,  0.2412,  0.3700,
0.5783,  0.5790,
           0.4229,  0.5032,  0.1216,  0.1000,  0.0356,  0.1258, -
0.0023,  0.1640,
           0.2809,  0.2418,  0.2606,  0.3787,  0.5334,  0.5645,
0.4781,  0.3307,
           0.2391,  0.0461,  0.0095,  0.0542,  0.1012,  0.1331,
0.2446,  0.2526,
           0.3323,  0.4120,  0.5724,  0.2840,  0.5188,  0.3934,
0.1328,  0.0776,
           0.0235,  0.1366,  0.3149,  0.2200,  0.2793,  0.2351,
0.4722,  0.4785,
           0.4009,  0.4570,  0.4972,  0.5785,  0.2261,  0.1447, -
0.0028,  0.2772,
           0.2697,  0.4008,  0.3606,  0.3372,  0.4535,  0.4492,
0.5678,  0.5870,
           0.5824,  0.3438,  0.5113,  0.0757,  0.1772,  0.3677,
```

```
0.3572,  0.3742,
           0.3820,  0.4868,  0.3781,  0.4694,  0.5845,  0.5877,
0.5826,  0.3212,
           0.5276,  0.4840,  0.4825,  0.5523,  0.5308,  0.5085,
0.5606,  0.5720,
           0.4928,  0.5581,  0.5853,  0.5849,  0.5793,  0.3410,
0.4428,  0.4044,
           0.3275,  0.4958,  0.4366,  0.5750,  0.5494,  0.5868,
0.5557,  0.5069,
           0.5880,  0.5888,  0.5796,  0.3377,  0.2635,  0.2347,
0.3145,  0.3486,
           0.5158,  0.5722,  0.5347,  0.5753,  0.5816,  0.4378,
0.5692,  0.5843,
           0.5721,  0.5081,  0.2694,  0.2032,  0.1589,  0.3464,
0.5349,  0.5768,
           0.5739,  0.5764,  0.5394,  0.4482]],
grad_fn=<SelectBackward0>),
 True,
 torch.Size([1, 196]))
```

Mengubah lapisan penyematan patch ViT menjadi modul PyTorch

```
# 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 for the input
images. Defaults to 3.
        patch_size (int): Size of patches to convert input image into.
Defaults to 16.
        embedding_dim (int): Size of embedding to turn image into.
Defaults to 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 into a single vector
                                  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 patch size, image shape: {image_resolution}, patch
size: {patch_size}"

        # 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) # adjust so the embedding
is on the final dimension [batch_size, P^2•C, N] -> [batch_size, N,
P^2•C]

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 the 0th index, otherwise will error
print(f"Output patch embedding shape: {patch_embedded_image.shape}")

Input image shape: torch.Size([1, 3, 224, 224])
Output patch embedding shape: torch.Size([1, 196, 768])

# Create random input sizes
random_input_image = (1, 3, 224, 224)
random_input_image_error = (1, 3, 250, 250) # will error because image
size is incompatible with patch_size

# # Get a summary of the input and outputs of PatchEmbedding
(uncomment for full output)
# summary(PatchEmbedding(),
#         input_size=random_input_image, # try swapping this for
"random_input_image_error"
#         col_names=["input_size", "output_size", "num_params",
"trainable"],
```

```
#          col_width=20,
#          row_settings=["var_names"])
```

Membuat penyematan token kelas

```
# View the patch embedding and patch embedding shape
print(patch_embedded_image)
print(f"Patch embedding shape: {patch_embedded_image.shape} -> 
[batch_size, number_of_patches, embedding_dimension]")

tensor([[[-0.9145,  0.2454, -0.2292,  ...,  0.6768, -0.4515,  0.3496],
         [-0.7427,  0.1955, -0.3570,  ...,  0.5823, -0.3458,  0.3261],
         [-0.7589,  0.2633, -0.1695,  ...,  0.5897, -0.3980,  0.0761],
         ...,
         [-1.0072,  0.2795, -0.2804,  ...,  0.7624, -0.4584,  0.3581],
         [-0.9839,  0.1652, -0.1576,  ...,  0.7489, -0.5478,  0.3486],
         [-0.9260,  0.1383, -0.1157,  ...,  0.5847, -0.4717,  
0.3112]]],
       grad_fn=<PermuteBackward0>)
Patch embedding shape: torch.Size([1, 196, 768]) -> [batch_size, 
number_of_patches, embedding_dimension]

# Get the batch size and embedding dimension
batch_size = patch_embedded_image.shape[0]
embedding_dimension = patch_embedded_image.shape[-1]

# Create the class token embedding as a learnable parameter that 
shares the same size as the embedding dimension (D)
class_token = nn.Parameter(torch.ones(batch_size, 1, 
embedding_dimension), # [batch_size, number_of_tokens, 
embedding_dimension]
                           requires_grad=True) # make sure the 
embedding is learnable

# Show the first 10 examples of the class_token
print(class_token[:, :, :10])

# Print the class_token shape
print(f"Class token shape: {class_token.shape} -> [batch_size, 
number_of_tokens, embedding_dimension]")

tensor([[[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]]], 
grad_fn=<SliceBackward0>)
Class token shape: torch.Size([1, 1, 768]) -> [batch_size, 
number_of_tokens, embedding_dimension]

# Add the class token embedding to the front of the patch embedding
patch_embedded_image_with_class_embedding = torch.cat((class_token, 
patch_embedded_image), 
                                                      dim=1) # concat
```

```
on first dimension

# Print the sequence of patch embeddings with the prepended class
token embedding
print(patch_embedded_image_with_class_embedding)
print(f"Sequence of patch embeddings with class token prepended shape:
{patch_embedded_image_with_class_embedding.shape} -> [batch_size,
number_of_patches, embedding_dimension]")
```

```
tensor([[[ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,  1.0000],
         [-0.9145,  0.2454, -0.2292,  ...,  0.6768, -0.4515,  0.3496],
         [-0.7427,  0.1955, -0.3570,  ...,  0.5823, -0.3458,  0.3261],
         ...,
         [-1.0072,  0.2795, -0.2804,  ...,  0.7624, -0.4584,  0.3581],
         [-0.9839,  0.1652, -0.1576,  ...,  0.7489, -0.5478,  0.3486],
         [-0.9260,  0.1383, -0.1157,  ...,  0.5847, -0.4717,
0.3112]]],
       grad_fn=<CatBackward0>)
Sequence of patch embeddings with class token prepended shape:
torch.Size([1, 197, 768]) -> [batch_size, number_of_patches,
embedding_dimension]
```

Membuat posisi penyematan

```
# View the sequence of patch embeddings with the prepended class
embedding
patch_embedded_image_with_class_embedding,
patch_embedded_image_with_class_embedding.shape
```

```
(tensor([[[ 1.0000,  1.0000,  1.0000,  ...,  1.0000,  1.0000,
1.0000],
          [-0.9145,  0.2454, -0.2292,  ...,  0.6768, -0.4515,
0.3496],
          [-0.7427,  0.1955, -0.3570,  ...,  0.5823, -0.3458,
0.3261],
          ...,
          [-1.0072,  0.2795, -0.2804,  ...,  0.7624, -0.4584,
0.3581],
          [-0.9839,  0.1652, -0.1576,  ...,  0.7489, -0.5478,
0.3486],
          [-0.9260,  0.1383, -0.1157,  ...,  0.5847, -0.4717,
0.3112]]],
        grad_fn=<CatBackward0>),
 torch.Size([1, 197, 768]))
```

```
# Calculate N (number of patches)
number_of_patches = int((height * width) / patch_size**2)

# Get embedding dimension
embedding_dimension =
```

```
patch_embedded_image_with_class_embedding.shape[2]

# Create the learnable 1D position embedding
position_embedding = nn.Parameter(torch.ones(1,
                                             number_of_patches+1,
                                             embedding_dimension),
                                  requires_grad=True) # make sure it's
learnable

# Show the first 10 sequences and 10 position embedding values and
check the shape of the position embedding
print(position_embedding[:, :10, :10])
print(f"Position embeddding shape: {position_embedding.shape} ->
[batch_size, number_of_patches, embedding_dimension]")

tensor([[[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
         [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]]],
grad_fn=<SliceBackward0>)
Position embeddding shape: torch.Size([1, 197, 768]) -> [batch_size,
number_of_patches, embedding_dimension]

# Add the position embedding to the patch and class token embedding
patch_and_position_embedding =
patch_embedded_image_with_class_embedding + position_embedding
print(patch_and_position_embedding)
print(f"Patch embeddings, class token prepended and positional
embeddings added shape: {patch_and_position_embedding.shape} ->
[batch_size, number_of_patches, embedding_dimension]")

tensor([[[ 2.0000,  2.0000,  2.0000,  ...,  2.0000,  2.0000,  2.0000],
         [ 0.0855,  1.2454,  0.7708,  ...,  1.6768,  0.5485,  1.3496],
         [ 0.2573,  1.1955,  0.6430,  ...,  1.5823,  0.6542,  1.3261],
         ...,
         [-0.0072,  1.2795,  0.7196,  ...,  1.7624,  0.5416,  1.3581],
         [ 0.0161,  1.1652,  0.8424,  ...,  1.7489,  0.4522,  1.3486],
         [ 0.0740,  1.1383,  0.8843,  ...,  1.5847,  0.5283,
1.3112]]],
       grad_fn=<AddBackward0>)
Patch embeddings, class token prepended and positional embeddings
added shape: torch.Size([1, 197, 768]) -> [batch_size,
number_of_patches, embedding_dimension]
```

Menyatukan semuanya: dari gambar hingga penyematan

```
set_seeds()

# 1. Set patch size
patch_size = 16

# 2. Print shape of original image tensor and get the image dimensions
print(f"Image tensor shape: {image.shape}")
height, width = image.shape[1], image.shape[2]

# 3. Get image tensor and add batch dimension
x = image.unsqueeze(0)
print(f"Input image with batch dimension shape: {x.shape}")

# 4. Create patch embedding layer
patch_embedding_layer = PatchEmbedding(in_channels=3,
                                       patch_size=patch_size,
                                       embedding_dim=768)

# 5. Pass image through patch embedding layer
patch_embedding = patch_embedding_layer(x)
print(f"Patching embedding shape: {patch_embedding.shape}")

# 6. Create class token embedding
batch_size = patch_embedding.shape[0]
embedding_dimension = patch_embedding.shape[-1]
class_token = nn.Parameter(torch.ones(batch_size, 1,
embedding_dimension),
                           requires_grad=True) # make sure it's
learnable
print(f"Class token embedding shape: {class_token.shape}")

# 7. Prepend class token embedding to patch embedding
patch_embedding_class_token = torch.cat((class_token,
patch_embedding), dim=1)
print(f"Patch embedding with class token shape:
{patch_embedding_class_token.shape}")

# 8. Create position embedding
number_of_patches = int((height * width) / patch_size**2)
position_embedding = nn.Parameter(torch.ones(1, number_of_patches+1,
embedding_dimension),
                                  requires_grad=True) # make sure it's
learnable

# 9. Add position embedding to patch embedding with class token
patch_and_position_embedding = patch_embedding_class_token +
position_embedding
print(f"Patch and position embedding shape:
{patch_and_position_embedding.shape}")
```

```
Image tensor shape: torch.Size([3, 224, 224])
Input image with batch dimension shape: torch.Size([1, 3, 224, 224])
Patching embedding shape: torch.Size([1, 196, 768])
Class token embedding shape: torch.Size([1, 1, 768])
Patch embedding with class token shape: torch.Size([1, 197, 768])
Patch and position embedding shape: torch.Size([1, 197, 768])
```

```
# 1. Create a class that inherits from nn.Module
class MultiheadSelfAttentionBlock(nn.Module):
    """Creates a multi-head self-attention block ("MSA block" for
short).
    """
    # 2. Initialize the class with hyperparameters from Table 1
    def __init__(self,
                 embedding_dim:int=768, # Hidden size D from Table 1
for ViT-Base
                 num_heads:int=12, # Heads from Table 1 for ViT-Base
                 attn_dropout:float=0): # doesn't look like the paper
uses any dropout in MSABlocks
        super().__init__()

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

        # 4. Create the Multi-Head Attention (MSA) layer
        self.multihead_attn =
nn.MultiheadAttention(embed_dim=embedding_dim,

num_heads=num_heads,

dropout=attn_dropout,
                                                    batch_first=True)
# does our batch dimension come first?

    # 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 weights or just the layer outputs?
        return attn_output

# Create an instance of MSABlock
multihead_self_attention_block =
MultiheadSelfAttentionBlock(embedding_dim=768, # from Table 1
```

```
num_heads=12) # from Table 1

# Pass patch and position image embedding through MSABlock
patched_image_through_msa_block =
multihead_self_attention_block(patch_and_position_embedding)
print(f"Input shape of MSA block:
{patch_and_position_embedding.shape}")
print(f"Output shape MSA block:
{patched_image_through_msa_block.shape}")
```

```
Input shape of MSA block: torch.Size([1, 197, 768])
Output shape MSA block: torch.Size([1, 197, 768])
```

```
# 1. Create a class that inherits from nn.Module
class MLPBlock(nn.Module):
    """Creates a layer normalized multilayer perceptron block ("MLP
block" for short)."""
    # 2. Initialize the class with hyperparameters from Table 1 and
Table 3
    def __init__(self,
                 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
                 dropout:float=0.1): # Dropout from Table 3 for ViT-
Base
        super().__init__()

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

        # 4. Create the Multilayer perceptron (MLP) layer(s)
        self.mlp = nn.Sequential(
            nn.Linear(in_features=embedding_dim,
                      out_features=mlp_size),
            nn.GELU(), # "The MLP contains two layers with a GELU non-
linearity (section 3.1)."
            nn.Dropout(p=dropout),
            nn.Linear(in_features=mlp_size, # needs to take same
in_features as out_features of layer above
                      out_features=embedding_dim), # take back to
embedding_dim
            nn.Dropout(p=dropout) # "Dropout, when used, is applied
after every dense layer.."
        )

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

```python
        x = self.mlp(x)
        return x

# Create an instance of MLPBlock
mlp_block = MLPBlock(embedding_dim=768, # from Table 1
                     mlp_size=3072, # from Table 1
                     dropout=0.1) # from Table 3

# Pass output of MSABlock through MLPBlock
patched_image_through_mlp_block =
mlp_block(patched_image_through_msa_block)
print(f"Input shape of MLP block:
{patched_image_through_msa_block.shape}")
print(f"Output shape MLP block:
{patched_image_through_mlp_block.shape}")
```

```
Input shape of MLP block: torch.Size([1, 197, 768])
Output shape MLP block: torch.Size([1, 197, 768])
```

Buat Encoder Transformator

Membuat Transformer Encoder dengan menggabungkan lapisan yang kami buat khusus

```python
# 1. Create a class that inherits from nn.Module
class TransformerEncoderBlock(nn.Module):
    """Creates a Transformer Encoder block."""
    # 2. Initialize the class with hyperparameters from Table 1 and
Table 3
    def __init__(self,
                 embedding_dim:int=768, # Hidden size D from Table 1
for ViT-Base
                 num_heads:int=12, # Heads from Table 1 for ViT-Base
                 mlp_size:int=3072, # MLP size from Table 1 for ViT-
Base
                 mlp_dropout:float=0.1, # Amount of dropout for dense
layers from Table 3 for ViT-Base
                 attn_dropout:float=0): # Amount of dropout for
attention layers
        super().__init__()

        # 3. Create MSA block (equation 2)
        self.msa_block =
MultiheadSelfAttentionBlock(embedding_dim=embedding_dim,

num_heads=num_heads,

attn_dropout=attn_dropout)
```

```
        # 4. Create MLP block (equation 3)
        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 (add the input
to the output)
        x =  self.msa_block(x) + x

        # 7. Create residual connection for MLP block (add the input
to the output)
        x = self.mlp_block(x) + x

        return x

# Create an instance of TransformerEncoderBlock
transformer_encoder_block = TransformerEncoderBlock()

# # Print an input and output summary of our Transformer Encoder
(uncomment for full output)
# 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=20,
#         row_settings=["var_names"])
```

Membuat Transformer Encoder dengan lapisan Transformer PyTorch

```
# Create the same as above with torch.nn.TransformerEncoderLayer()
torch_transformer_encoder_layer =
nn.TransformerEncoderLayer(d_model=768, # Hidden size D from Table 1
for ViT-Base
                                                             nhead=12,
# Heads from Table 1 for ViT-Base

dim_feedforward=3072, # MLP size from Table 1 for ViT-Base

dropout=0.1, # Amount of dropout for dense layers from Table 3 for
ViT-Base

activation="gelu", # GELU non-linear activation

batch_first=True, # Do our batches come first?

norm_first=True) # Normalize first or after MSA/MLP layers?
```

```
torch_transformer_encoder_layer

TransformerEncoderLayer(
  (self_attn): MultiheadAttention(
    (out_proj): NonDynamicallyQuantizableLinear(in_features=768, out_features=768, bias=True)
  )
  (linear1): Linear(in_features=768, out_features=3072, bias=True)
  (dropout): Dropout(p=0.1, inplace=False)
  (linear2): Linear(in_features=3072, out_features=768, bias=True)
  (norm1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (norm2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
  (dropout1): Dropout(p=0.1, inplace=False)
  (dropout2): Dropout(p=0.1, inplace=False)
)

# # Get the output of PyTorch's version of the Transformer Encoder (uncomment for full output)
# summary(model=torch_transformer_encoder_layer,
#         input_size=(1, 197, 768), # (batch_size, num_patches, embedding_dimension)
#         col_names=["input_size", "output_size", "num_params", "trainable"],
#         col_width=20,
#         row_settings=["var_names"])
```

Menggabungkan semuanya untuk menciptakan ViT

```
# 1. Create a ViT class that inherits from nn.Module
class ViT(nn.Module):
    """Creates a Vision Transformer architecture with ViT-Base hyperparameters by default."""
    # 2. Initialize the class with hyperparameters from Table 1 and Table 3
    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 embeddings
                 num_classes:int=1000): # Default for ImageNet but can
customize this
        super().__init__() # don't forget the super().__init__()!

        # 3. Make the image size is divisble by the patch size
        assert img_size % patch_size == 0, f"Image size must be
divisible by patch size, image size: {img_size}, patch size:
{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 embeddings)
        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, embedding_dim),
                                               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 (we can stack
Transformer Encoder blocks using nn.Sequential())
        # Note: The "*" means "all"
        self.transformer_encoder =
nn.Sequential(*[TransformerEncoderBlock(embedding_dim=embedding_dim,

num_heads=num_heads,

mlp_size=mlp_size,

mlp_dropout=mlp_dropout) for _ in range(num_transformer_layers)])

        # 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 (equation 1)
        class_token = self.class_embedding.expand(batch_size, -1, -1)
# "-1" means to infer the dimension (try this line on its own)

        # 14. Create patch embedding (equation 1)
        x = self.patch_embedding(x)

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

        # 16. Add position embedding to patch embedding (equation 1)
        x = self.position_embedding + x

        # 17. Run embedding dropout (Appendix B.1)
        x = self.embedding_dropout(x)

        # 18. Pass patch, position and class embedding through
transformer encoder layers (equations 2 & 3)
        x = self.transformer_encoder(x)

        # 19. Put 0 index logit through classifier (equation 4)
        x = self.classifier(x[:, 0]) # run on each sample in a batch
at 0 index

        return x

# Example of creating the class embedding and expanding over a batch
dimension
batch_size = 32
class_token_embedding_single = nn.Parameter(data=torch.randn(1, 1,
768)) # create a single learnable class token
class_token_embedding_expanded =
class_token_embedding_single.expand(batch_size, -1, -1) # expand the
single learnable class token across the batch dimension, "-1" means to
"infer the dimension"

# Print out the change in shapes
print(f"Shape of class token embedding single:
{class_token_embedding_single.shape}")
```

```
print(f"Shape of class token embedding expanded:
{class_token_embedding_expanded.shape}")
```

```
Shape of class token embedding single: torch.Size([1, 1, 768])
Shape of class token embedding expanded: torch.Size([32, 1, 768])
```

```
set_seeds()

# Create a random tensor with same shape as a single image
random_image_tensor = torch.randn(1, 3, 224, 224) # (batch_size,
color_channels, height, width)

# Create an instance of ViT with the number of classes we're working
with (pizza, steak, sushi)
vit = ViT(num_classes=len(class_names))

# Pass the random image tensor to our ViT instance
vit(random_image_tensor)
```

```
tensor([[-0.2377,  0.7360,  1.2137]], grad_fn=<AddmmBackward0>)
```

Mendapatkan ringkasan visual model ViT kami

```
from torchinfo import summary

# # Print a summary of our custom ViT model using torchinfo (uncomment
for actual output)
# summary(model=vit,
#         input_size=(32, 3, 224, 224), # (batch_size, color_channels,
height, width)
#         # col_names=["input_size"], # uncomment for smaller output
#         col_names=["input_size", "output_size", "num_params",
"trainable"],
#         col_width=20,
#         row_settings=["var_names"]
# )
```

Membuat pengoptimal

Melatih model ViT kita

```
from going_modular.going_modular import engine

# Setup the optimizer to optimize our ViT model parameters using
hyperparameters from the ViT paper
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 but
also mentioned in ViT paper section 4.1 (Training & Fine-tuning)
```

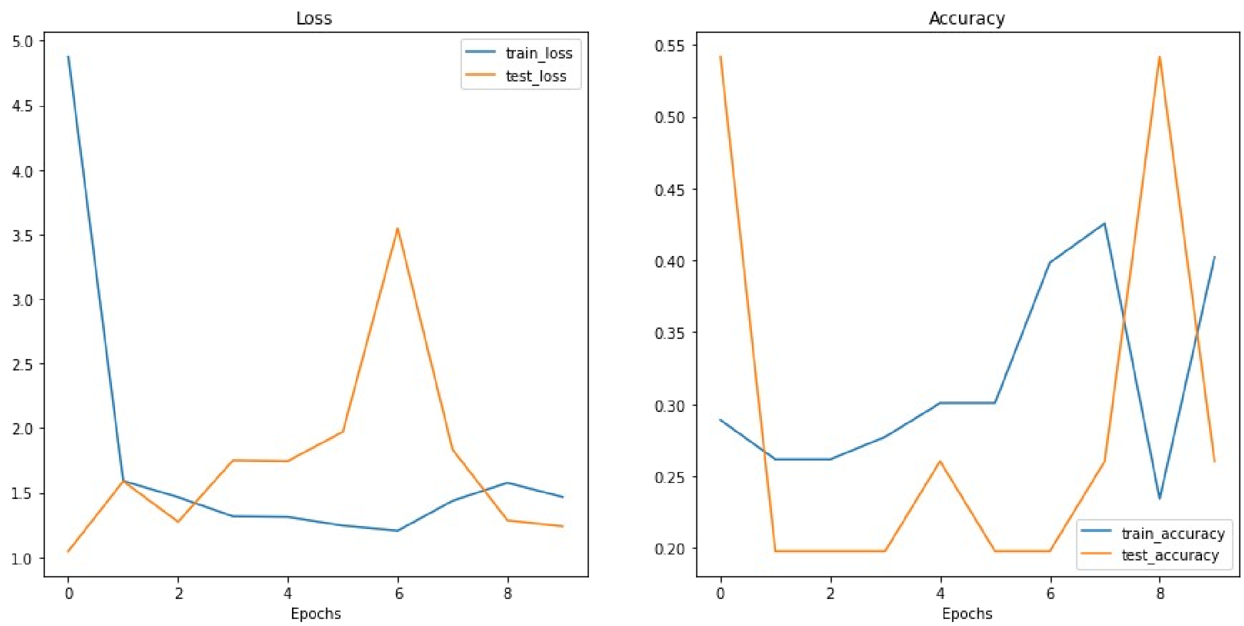
```
                              weight_decay=0.3) # from the ViT paper section 4.1 (Training & Fine-tuning) and Table 3 for ViT-* ImageNet-1k

# 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=10,
                       device=device)
```

{"model_id":"97484323a38248e98ded3df3e074655c","version_major":2,"version_minor":0}

```
Epoch: 1 | train_loss: 4.8759 | train_acc: 0.2891 | test_loss: 1.0465 | test_acc: 0.5417
Epoch: 2 | train_loss: 1.5900 | train_acc: 0.2617 | test_loss: 1.5876 | test_acc: 0.1979
Epoch: 3 | train_loss: 1.4644 | train_acc: 0.2617 | test_loss: 1.2738 | test_acc: 0.1979
Epoch: 4 | train_loss: 1.3159 | train_acc: 0.2773 | test_loss: 1.7498 | test_acc: 0.1979
Epoch: 5 | train_loss: 1.3114 | train_acc: 0.3008 | test_loss: 1.7444 | test_acc: 0.2604
Epoch: 6 | train_loss: 1.2445 | train_acc: 0.3008 | test_loss: 1.9704 | test_acc: 0.1979
Epoch: 7 | train_loss: 1.2050 | train_acc: 0.3984 | test_loss: 3.5480 | test_acc: 0.1979
Epoch: 8 | train_loss: 1.4368 | train_acc: 0.4258 | test_loss: 1.8324 | test_acc: 0.2604
Epoch: 9 | train_loss: 1.5757 | train_acc: 0.2344 | test_loss: 1.2848 | test_acc: 0.5417
Epoch: 10 | train_loss: 1.4658 | train_acc: 0.4023 | test_loss: 1.2389 | test_acc: 0.2604
```

Plot kurva kerugian model ViT kita

```
from helper_functions import plot_loss_curves

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



Mendapatkan model ViT terlatih dan membuat ekstraktor fitur

```
# The following requires torch v0.12+ and torchvision v0.13+
import torch
import torchvision
print(torch.__version__)
print(torchvision.__version__)

1.12.0+cu102
0.13.0+cu102
```

Then we'll setup device-agonistc code.

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

'cuda'

# 1. Get pretrained weights for ViT-Base
pretrained_vit_weights = torchvision.models.ViT_B_16_Weights.DEFAULT #
requires torchvision >= 0.13, "DEFAULT" means best available

# 2. Setup a ViT model instance with pretrained weights
pretrained_vit =
torchvision.models.vit_b_16(weights=pretrained_vit_weights).to(device)

# 3. Freeze the base parameters
for parameter in pretrained_vit.parameters():
    parameter.requires_grad = False

# 4. Change the classifier head (set the seeds to ensure same
```

```
initialization with linear head)
set_seeds()
pretrained_vit.heads = nn.Linear(in_features=768,
out_features=len(class_names)).to(device)
# pretrained_vit # uncomment for model output

# # Print a summary using torchinfo (uncomment for actual output)
# summary(model=pretrained_vit,
#         input_size=(32, 3, 224, 224), # (batch_size, color_channels,
height, width)
#         # col_names=["input_size"], # uncomment for smaller output
#         col_names=["input_size", "output_size", "num_params",
"trainable"],
#         col_width=20,
#         row_settings=["var_names"]
# )
```

Mempersiapkan data untuk model ViT yang telah dilatih sebelumnya

```
from helper_functions import download_data

# Download pizza, steak, sushi images from GitHub
image_path =
download_data(source="https://github.com/mrdbourke/pytorch-deep-
learning/raw/main/data/pizza_steak_sushi.zip",
                           destination="pizza_steak_sushi")
image_path
```

```
[INFO] data/pizza_steak_sushi directory exists, skipping download.
```

```
PosixPath('data/pizza_steak_sushi')
```

```
# Setup train and test directory paths
train_dir = image_path / "train"
test_dir = image_path / "test"
train_dir, test_dir
```

```
(PosixPath('data/pizza_steak_sushi/train'),
 PosixPath('data/pizza_steak_sushi/test'))
```

```
# Get automatic transforms from pretrained ViT weights
pretrained_vit_transforms = pretrained_vit_weights.transforms()
print(pretrained_vit_transforms)
```

```
ImageClassification(
    crop_size=[224]
    resize_size=[256]
    mean=[0.485, 0.456, 0.406]
    std=[0.229, 0.224, 0.225]
    interpolation=InterpolationMode.BILINEAR
)
```

```
# Setup dataloaders
train_dataloader_pretrained, test_dataloader_pretrained, class_names = 
data_setup.create_dataloaders(train_dir=train_dir,

test_dir=test_dir,

transform=pretrained_vit_transforms,

batch_size=32) # Could increase if we had more samples, such as here: 
https://arxiv.org/abs/2205.01580 (there are other improvements there 
too...)
```

Melatih model ViT ekstraktor fitur

```
from going_modular.going_modular import engine

# Create optimizer and loss function
optimizer = torch.optim.Adam(params=pretrained_vit.parameters(),
                             lr=1e-3)
loss_fn = torch.nn.CrossEntropyLoss()

# Train the classifier head of the pretrained ViT feature extractor 
model
set_seeds()
pretrained_vit_results = engine.train(model=pretrained_vit,

train_dataloader=train_dataloader_pretrained,

test_dataloader=test_dataloader_pretrained,
                                      optimizer=optimizer,
                                      loss_fn=loss_fn,
                                      epochs=10,
                                      device=device)
```

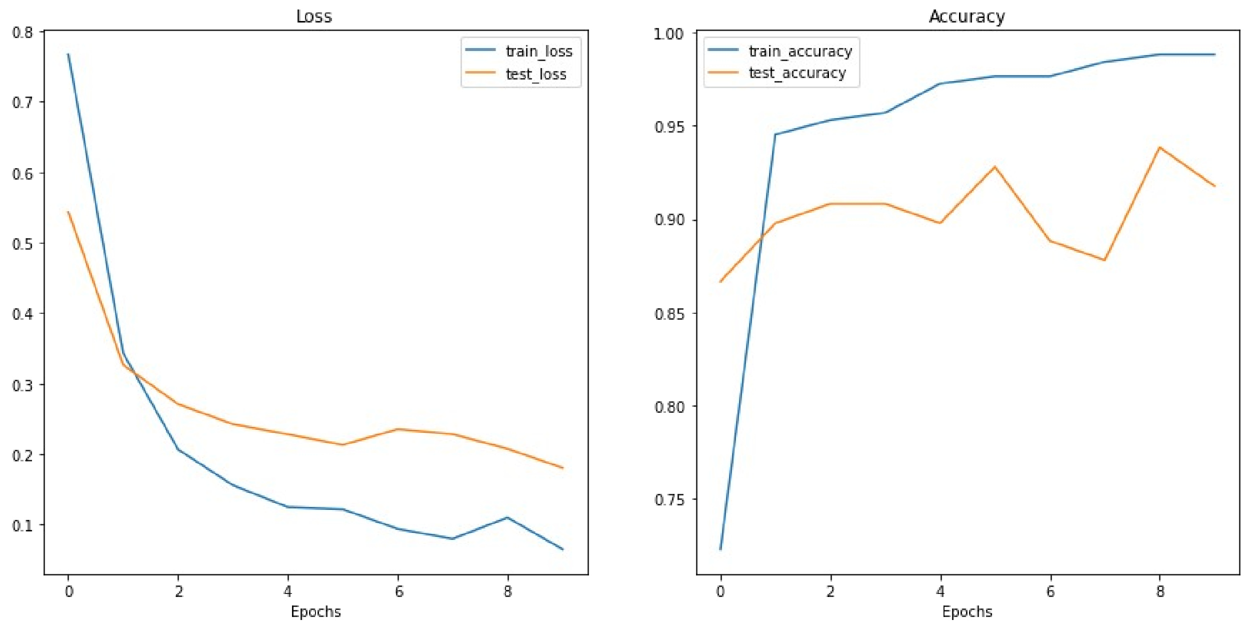
{"model_id":"e47702187773418aafc32e0078ff1895","version_major":2,"vers
ion_minor":0}

```
Epoch: 1 | train_loss: 0.7665 | train_acc: 0.7227 | test_loss: 0.5432 
| test_acc: 0.8665
Epoch: 2 | train_loss: 0.3428 | train_acc: 0.9453 | test_loss: 0.3263 
| test_acc: 0.8977
Epoch: 3 | train_loss: 0.2064 | train_acc: 0.9531 | test_loss: 0.2707 
| test_acc: 0.9081
Epoch: 4 | train_loss: 0.1556 | train_acc: 0.9570 | test_loss: 0.2422 
| test_acc: 0.9081
Epoch: 5 | train_loss: 0.1246 | train_acc: 0.9727 | test_loss: 0.2279 
| test_acc: 0.8977
Epoch: 6 | train_loss: 0.1216 | train_acc: 0.9766 | test_loss: 0.2129 
| test_acc: 0.9280
Epoch: 7 | train_loss: 0.0938 | train_acc: 0.9766 | test_loss: 0.2352
```

```
| test_acc: 0.8883
Epoch: 8 | train_loss: 0.0797 | train_acc: 0.9844 | test_loss: 0.2281
| test_acc: 0.8778
Epoch: 9 | train_loss: 0.1098 | train_acc: 0.9883 | test_loss: 0.2074
| test_acc: 0.9384
Epoch: 10 | train_loss: 0.0650 | train_acc: 0.9883 | test_loss: 0.1804
| test_acc: 0.9176
```

Plot kurva kerugian model ViT ekstraktor fitur

```python
# Plot the loss curves
from helper_functions import plot_loss_curves

plot_loss_curves(pretrained_vit_results)
```



Simpan model ViT ekstraktor fitur dan periksa ukuran file

```python
# Save the model
from going_modular.going_modular import utils

utils.save_model(model=pretrained_vit,
                 target_dir="models",

model_name="08_pretrained_vit_feature_extractor_pizza_steak_sushi.pth"
)

[INFO] Saving model to:
models/08_pretrained_vit_feature_extractor_pizza_steak_sushi.pth

from pathlib import Path
```

```python
# Get the model size in bytes then convert to megabytes
pretrained_vit_model_size =
Path("models/08_pretrained_vit_feature_extractor_pizza_steak_sushi.pth
").stat().st_size // (1024*1024) # division converts bytes to
megabytes (roughly)
print(f"Pretrained ViT feature extractor model size:
{pretrained_vit_model_size} MB")
```

```
Pretrained ViT feature extractor model size: 327 MB
```

Buat prediksi pada gambar khusus

```python
import requests

# Import function to make predictions on images and plot them
from going_modular.going_modular.predictions import
pred_and_plot_image

# Setup custom image path
custom_image_path = image_path / "04-pizza-dad.jpeg"

# Download the image if it doesn't already exist
if not custom_image_path.is_file():
    with open(custom_image_path, "wb") as f:
        # When downloading from GitHub, need to use the "raw" file
link
        request =
requests.get("https://raw.githubusercontent.com/mrdbourke/pytorch-
deep-learning/main/images/04-pizza-dad.jpeg")
        print(f"Downloading {custom_image_path}...")
        f.write(request.content)
else:
    print(f"{custom_image_path} already exists, skipping download.")

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

```
data/pizza_steak_sushi/04-pizza-dad.jpeg already exists, skipping
download.
```

Pred: pizza | Prob: 0.988