coursework111

February 26, 2025

1 Coursework: Masked Auto-Encoder

In this coursework, you will explore the popular self-supervised masked auto-encoder approach MAE.

The coursework is divided in the following parts:

- **Part A**: Create a dataset and a data module to handle the PneumoniaMNIST dataset.
- **Part B**: Implement MAE utility functions.
- **Part C**: Implement and train a full MAE model.
- **Part D**: Inspect the trained model.

Important: Read the text descriptions carefully and look out for hints and comments indicating a specific 'TODO'. Make sure to add sufficient documentation and comments to your code.

Submission: You are asked to submit two versions of your notebook: 1. You should submit the raw notebook in `.ipynb` format with *all outputs cleared*. Please name your file `coursework.ipynb`. 2. Additionally, you will be asked to submit an exported version of your notebook in `.pdf` format, with *all outputs included*. We will primarily use this version for marking, but we will use the raw notebook to check for correct implementations. Please name this file `coursework_export.pdf`.

1.1 Your details

Please add your details below. You can work in groups up to two.

Authors: **firstname1 lastname1** & **firstname2 lastname2**

DoC alias: **alias1** & **alias2**

1.2 Setup

[88]:
```
# On Google Colab uncomment the following line to install PyTorch Lightning and
↪the MedMNIST dataset
! pip install lightning medmnist timm
```

```
Requirement already satisfied: lightning in /usr/local/lib/python3.11/dist-
packages (2.5.0.post0)
Requirement already satisfied: medmnist in /usr/local/lib/python3.11/dist-
packages (3.0.2)
Requirement already satisfied: timm in /usr/local/lib/python3.11/dist-packages
(1.0.14)
```

```
Requirement already satisfied: PyYAML<8.0,>=5.4 in
/usr/local/lib/python3.11/dist-packages (from lightning) (6.0.2)
Requirement already satisfied: fsspec<2026.0,>=2022.5.0 in
/usr/local/lib/python3.11/dist-packages (from
fsspec[http]<2026.0,>=2022.5.0->lightning) (2024.10.0)
Requirement already satisfied: lightning-utilities<2.0,>=0.10.0 in
/usr/local/lib/python3.11/dist-packages (from lightning) (0.12.0)
Requirement already satisfied: packaging<25.0,>=20.0 in
/usr/local/lib/python3.11/dist-packages (from lightning) (24.2)
Requirement already satisfied: torch<4.0,>=2.1.0 in
/usr/local/lib/python3.11/dist-packages (from lightning) (2.5.1+cu124)
Requirement already satisfied: torchmetrics<3.0,>=0.7.0 in
/usr/local/lib/python3.11/dist-packages (from lightning) (1.6.1)
Requirement already satisfied: tqdm<6.0,>=4.57.0 in
/usr/local/lib/python3.11/dist-packages (from lightning) (4.67.1)
Requirement already satisfied: typing-extensions<6.0,>=4.4.0 in
/usr/local/lib/python3.11/dist-packages (from lightning) (4.12.2)
Requirement already satisfied: pytorch-lightning in
/usr/local/lib/python3.11/dist-packages (from lightning) (2.5.0.post0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages
(from medmnist) (1.26.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages
(from medmnist) (2.2.2)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-
packages (from medmnist) (1.6.1)
Requirement already satisfied: scikit-image in /usr/local/lib/python3.11/dist-
packages (from medmnist) (0.25.2)
Requirement already satisfied: Pillow in /usr/local/lib/python3.11/dist-packages
(from medmnist) (11.1.0)
Requirement already satisfied: fire in /usr/local/lib/python3.11/dist-packages
(from medmnist) (0.7.0)
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-
packages (from medmnist) (0.20.1+cu124)
Requirement already satisfied: huggingface_hub in
/usr/local/lib/python3.11/dist-packages (from timm) (0.28.1)
Requirement already satisfied: safetensors in /usr/local/lib/python3.11/dist-
packages (from timm) (0.5.2)
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in
/usr/local/lib/python3.11/dist-packages (from
fsspec[http]<2026.0,>=2022.5.0->lightning) (3.11.12)
Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-
packages (from lightning-utilities<2.0,>=0.10.0->lightning) (75.1.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-
packages (from torch<4.0,>=2.1.0->lightning) (3.17.0)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-
packages (from torch<4.0,>=2.1.0->lightning) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages
(from torch<4.0,>=2.1.0->lightning) (3.1.5)
```

```
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.4.127)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.4.127)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.4.127)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.4.5.8)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(11.2.1.3)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(10.3.5.147)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(11.6.1.9)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.3.1.170)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.4.127)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch<4.0,>=2.1.0->lightning)
(12.4.127)
Requirement already satisfied: triton==3.1.0 in /usr/local/lib/python3.11/dist-
packages (from torch<4.0,>=2.1.0->lightning) (3.1.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-
packages (from torch<4.0,>=2.1.0->lightning) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from
sympy==1.13.1->torch<4.0,>=2.1.0->lightning) (1.3.0)
Requirement already satisfied: termcolor in /usr/local/lib/python3.11/dist-
packages (from fire->medmnist) (2.5.0)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-
packages (from huggingface_hub->timm) (2.32.3)
Requirement already satisfied: python-dateutil>=2.8.2 in
```

```
/usr/local/lib/python3.11/dist-packages (from pandas->medmnist) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-
packages (from pandas->medmnist) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-
packages (from pandas->medmnist) (2025.1)
Requirement already satisfied: scipy>=1.11.4 in /usr/local/lib/python3.11/dist-
packages (from scikit-image->medmnist) (1.13.1)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in
/usr/local/lib/python3.11/dist-packages (from scikit-image->medmnist) (2.37.0)
Requirement already satisfied: tifffile>=2022.8.12 in
/usr/local/lib/python3.11/dist-packages (from scikit-image->medmnist)
(2025.2.18)
Requirement already satisfied: lazy-loader>=0.4 in
/usr/local/lib/python3.11/dist-packages (from scikit-image->medmnist) (0.4)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-
packages (from scikit-learn->medmnist) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in
/usr/local/lib/python3.11/dist-packages (from scikit-learn->medmnist) (3.5.0)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in
/usr/local/lib/python3.11/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (2.4.6)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.11/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (1.3.2)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.11/dist-
packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (25.1.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.11/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (1.5.0)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.11/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (6.1.0)
Requirement already satisfied: propcache>=0.2.0 in
/usr/local/lib/python3.11/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (0.2.1)
Requirement already satisfied: yarl<2.0,>=1.17.0 in
/usr/local/lib/python3.11/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (1.18.3)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-
packages (from python-dateutil>=2.8.2->pandas->medmnist) (1.17.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from
jinja2->torch<4.0,>=2.1.0->lightning) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests->huggingface_hub->timm)
(3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-
```

```
packages (from requests->huggingface_hub->timm) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests->huggingface_hub->timm)
(2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->huggingface_hub->timm)
(2025.1.31)
```

[1]:
```python
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import matplotlib.pyplot as plt

from torch.utils.data import DataLoader
from torchvision import models
from torchvision import transforms
from pytorch_lightning import LightningModule, LightningDataModule, Trainer,␣
 ↪seed_everything
from pytorch_lightning.loggers import TensorBoardLogger
from pytorch_lightning.callbacks import ModelCheckpoint, TQDMProgressBar
from torchmetrics.functional import auroc
from PIL import Image
from medmnist.info import INFO
from medmnist.dataset import MedMNIST
```

1.3 Part A: Create a dataset and a data module to handle the PneumoniaMNIST dataset.

We will be using the MedMNIST Pneumonia dataset, which is a medical imaging inspired dataset but with the characteristics of MNIST. This allows efficient experimentation due to the small image size. The dataset contains real chest X-ray images but here downsampled to **28 x 28 pixels**, with binary labels indicating the presence of Pneumonia (which is an inflammation of the lungs).

1.3.1 Task A-1: Complete the dataset implementation.

You are asked to implement a dataset class `PneumoniaMNISTDataset` suitable for training a classification model. For each sample, your dataset class should return one image and the corresponding label. We won't use the labels during training but for simplicity we will return them for model inspection purposes (part D).

To get you started, we have provided the skeleton of the dataset class in the cell below. Once you have implemented your dataset class, you are asked to run the provided visualisation code to visualise one batch of your training dataloader.

In terms of augmentation, we want to follow what has been done in the original MAE paper, that is **use random cropping (70%-100%) and horizontal flipping only** (see paragraph

Data augmentation, page 6 of the paper for further details). Hint: checkout torchvision transform `RandomResizedCrop`.

[2]:
```
class PneumoniaMNISTDataset(MedMNIST):
    def __init__(self, split = 'train', augmentation: bool = False):
        ''' Dataset class for Pneumonia MNST.
        The provided init function will automatically download the necessary
        files at the first class initialistion.

        :param split: 'train', 'val' or 'test', select subset

        '''
        self.flag = "pneumoniamnist"
        self.size = 28
        self.size_flag = ""
        self.root = './data/coursework/'
        self.info = INFO[self.flag]
        self.download()

        npz_file = np.load(os.path.join(self.root, "pneumoniamnist.npz"))

        self.split = split

        # Load all the images
        assert self.split in ['train','val','test']

        self.imgs = npz_file[f'{self.split}_images']
        self.labels = npz_file[f'{self.split}_labels']

        self.do_augment = augmentation

        # Define data augmentation pipeline
        if self.do_augment:
            self.transform = transforms.Compose([
                transforms.ToPILImage(),  # Convert numpy array to PIL image
                transforms.RandomResizedCrop(self.size, scale=(0.7, 1.0)),  #␣
↪70%-100% random crop
                transforms.RandomHorizontalFlip(),  # Random horizontal flip
                transforms.ToTensor(),  # Convert to tensor
                transforms.Normalize(mean=[0.5], std=[0.5])  # Normalize␣
↪(grayscale)
            ])
        else:
            self.transform = transforms.Compose([
                transforms.ToPILImage(),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.5], std=[0.5])
```

```
        ])

    def __len__(self):
        return self.imgs.shape[0]

    def __getitem__(self, index):
        '''
        Returns an image and its corresponding label.
        :param index:
        :return:
        '''
        img = self.imgs[index]  # Extract image (28x28 grayscale)
        label = int(self.labels[index].item())  # Extract label

        # Ensure grayscale images are handled properly
        img = np.expand_dims(img, axis=-1)  # Add channel dimension if missing

        # Apply transformation
        img = self.transform(img)

        return img, label
```

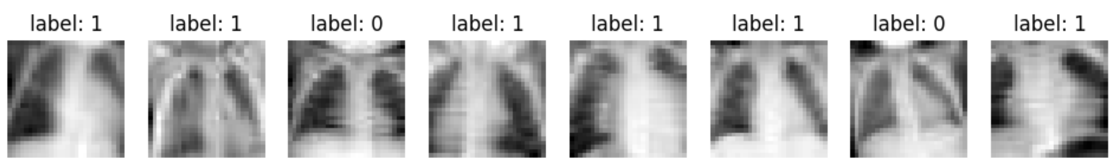
We use a LightningDataModule for handling your PneumoniaMNIST dataset. No changes needed for this part.

```
[3]: class PneumoniaMNISTDataModule(LightningDataModule):
    def __init__(self, batch_size: int = 32):
        super().__init__()
        self.batch_size = batch_size
        self.train_set = PneumoniaMNISTDataset(split='train', augmentation=True)
        self.val_set = PneumoniaMNISTDataset(split='val', augmentation=False)
        self.test_set = PneumoniaMNISTDataset(split='test', augmentation=False)

    def train_dataloader(self):
        return DataLoader(dataset=self.train_set, batch_size=self.batch_size,␣
↪shuffle=True)

    def val_dataloader(self):
        return DataLoader(dataset=self.val_set, batch_size=self.batch_size,␣
↪shuffle=False)

    def test_dataloader(self):
        return DataLoader(dataset=self.test_set, batch_size=self.batch_size,␣
↪shuffle=False)
```

Check dataset implementation. Run the below cell to visualise a batch of your training dataloader.

[92]:
```python
# DO NOT MODIFY THIS CELL! IT IS FOR CHECKING THE IMPLEMENTATION ONLY.

# Initialise data module
datamodule = PneumoniaMNISTDataModule()
# Get train dataloader
train_dataloader = datamodule.train_dataloader()
# Get first batch
batch = next(iter(train_dataloader))
# Visualise the images
images, labels = batch
f, ax = plt.subplots(1, 8, figsize=(12,4))
for i in range(8):
  ax[i].imshow(images[i, 0], cmap='gray')
  ax[i].set_title('label: ' + str(labels[i].item()))
  ax[i].axis("off")
```

```
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
```



1.4 Part B: Implement MAE utility functions.

As we saw in the lecture, Masked Auto-Encoders are based on a Vision Transformer (ViT) architecture. Importantly, the ViT architecture operates on a patch-level, not on the image-level. Hence, to feed the image into the ViT based encoder first we need to divide the images in small patches (typically 16x16 pixels).

In this part, we ask you to write three utility functions:

- `patchify`: takes in a batch of images (N, C, H, W) where N is the batch size, and returns a batch of patches of size (N, L, D) where L is the number of patches fitting in one image and D = patch_size** 2*C.
- `unpatchify`: inverts the above operation, takes in a batch of patches of size (N, L, D) and returns the corresponding a batch of images (N, C, H, W).
- `random_masking`: Randomly masks out patches during training to create a self-supervised training task of patch prediction.

1.4.1 Task B-1: Implement patchify

[4]:
```
def patchify(imgs, patch_size):
    """
    ### TODO
    ### Write a function that takes the batch of images (N, C, H, W)
    ### and returns a batch of patches (N, L, D) where
    ### L is the number of patches and D = patch_size**2*C.

    ### This function should throw an error if the H and W of the original
    image are not divisible by the patch size.

    patch_size: (patch_h, patch_w)
    """
    N, C, H, W = imgs.shape
    patch_h, patch_w = patch_size

    # Ensure H and W are divisible by patch_size
    assert H % patch_h == 0 and W % patch_w == 0, "Image dimensions must be␣
 ↪divisible by patch size"

    # Number of patches in each dimension
    num_patches_h = H // patch_h
    num_patches_w = W // patch_w
    L = num_patches_h * num_patches_w  # Total patches per image
    D = patch_h * patch_w * C  # Flattened patch dimension

    # Reshape to (N, C, num_patches_h, patch_h, num_patches_w, patch_w)
    patches = imgs.reshape(N, C, num_patches_h, patch_h, num_patches_w, patch_w)

    # Rearrange to (N, L, D)
    patches = patches.permute(0, 2, 4, 1, 3, 5).reshape(N, L, D)
    return patches
```

Let's test our implementation on the first batch of the validation set.

[94]:
```
# Load a batch of validation images
datamodule = PneumoniaMNISTDataModule()
dataloader = datamodule.val_dataloader()
batch = next(iter(dataloader))
images, labels = batch
```

```
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
```

[95]:
```
images.shape
```

[95]: torch.Size([32, 1, 28, 28])

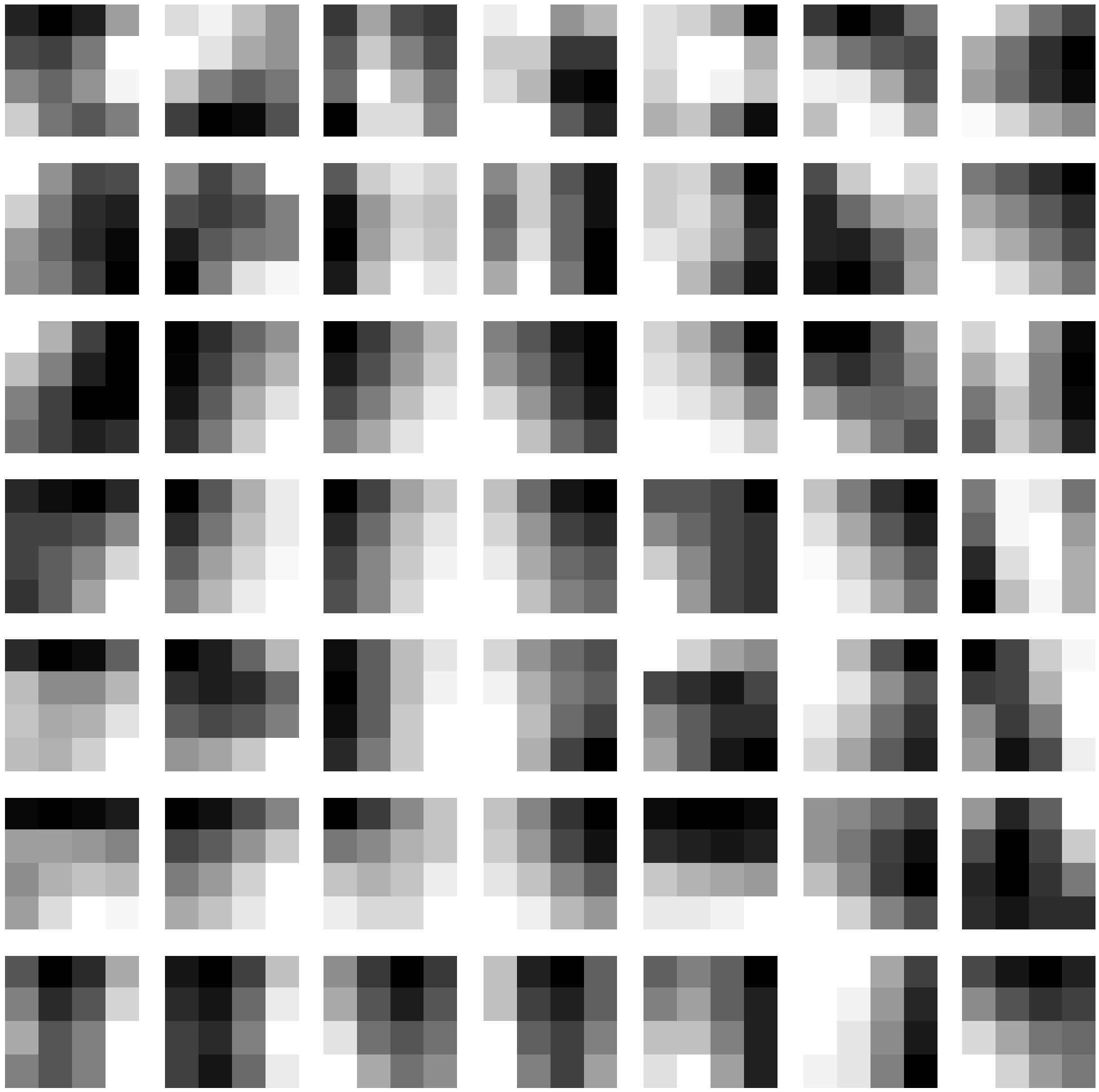
[96]:
```
# Assuming a patch size of (4,4) test your patchify function
# and test that the shape of the outputs corresponds at what is expected
patch_size = (4,4)
patches = patchify(images, (4, 4))
```

[97]:
```
patches.shape
```

[97]: torch.Size([32, 49, 16])

Visualisation of patchify output Next, we want to check our output visually. In the next cell, plot all the patches of the first image in the batch as a grid of subplots where subplot(i,j) shows patch(i,j) at the right position in the original image. You should be able to recognise the original image.

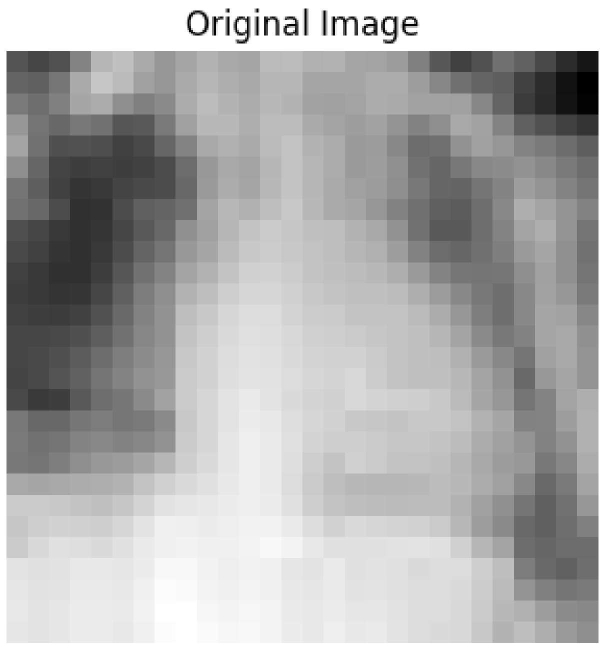
[98]:
```
def visualize_patches(img, patch_size):
    """
    Plot all patches from the first image in a grid.
    """
    # img = img.squeeze(0)  # Remove batch dimension (C, H, W)
    patches = patchify(img.unsqueeze(0), patch_size)  # (1, L, D)

    num_patches_h = img.shape[1] // patch_size[0]
    num_patches_w = img.shape[2] // patch_size[1]

    fig, axes = plt.subplots(num_patches_h, num_patches_w, figsize=(8, 8))

    for i in range(num_patches_h):
        for j in range(num_patches_w):
            patch_idx = i * num_patches_w + j
            patch = patches[0, patch_idx].reshape(patch_size[0], patch_size[1])␣
 ↪ # Reshape to 2D

            axes[i, j].imshow(patch, cmap='gray')
            axes[i, j].axis('off')

    plt.show()

# Sample visualization
sample_image = images[0]
visualize_patches(sample_image, patch_size=(4, 4))
```

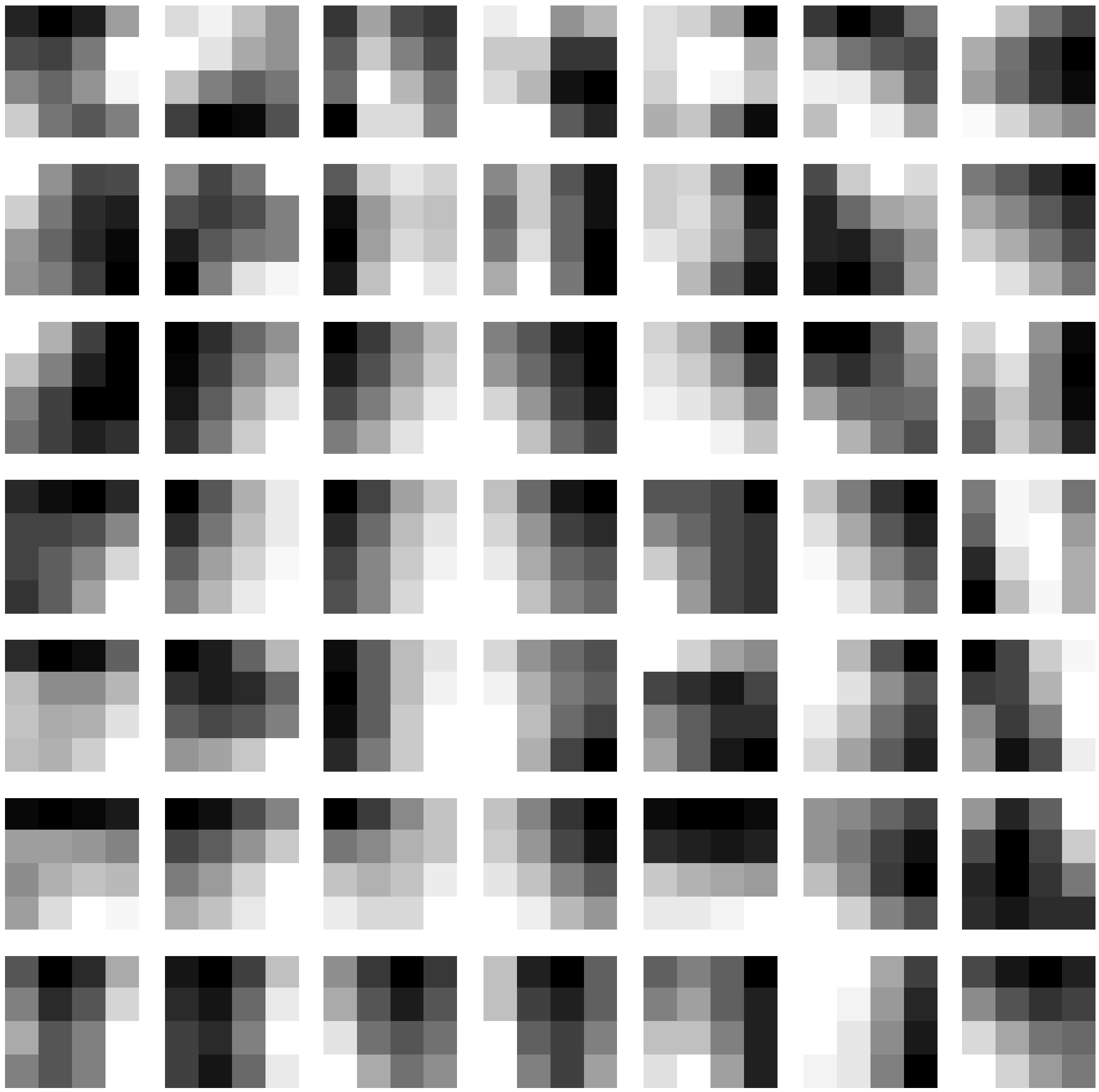


Compare the ouput with the original image

[99]:
```
def compare_patchify(img, patch_size):
    """
    Compare original image with its patchified version.
    """
    plt.figure(figsize=(4, 4))
    plt.imshow(img.squeeze(0), cmap='gray')
    plt.title("Original Image")
    plt.axis("off")
    plt.show()

    visualize_patches(img, patch_size)
```

```
sample_image = images[0]  # Get the first real image from dataset
compare_patchify(sample_image, patch_size=(4, 4))
```





1.4.2 Task B-2: Implement unpatchify

Next, you are asked to create the reverse function able to take in a batch of patches and return the corresponding batch of images.

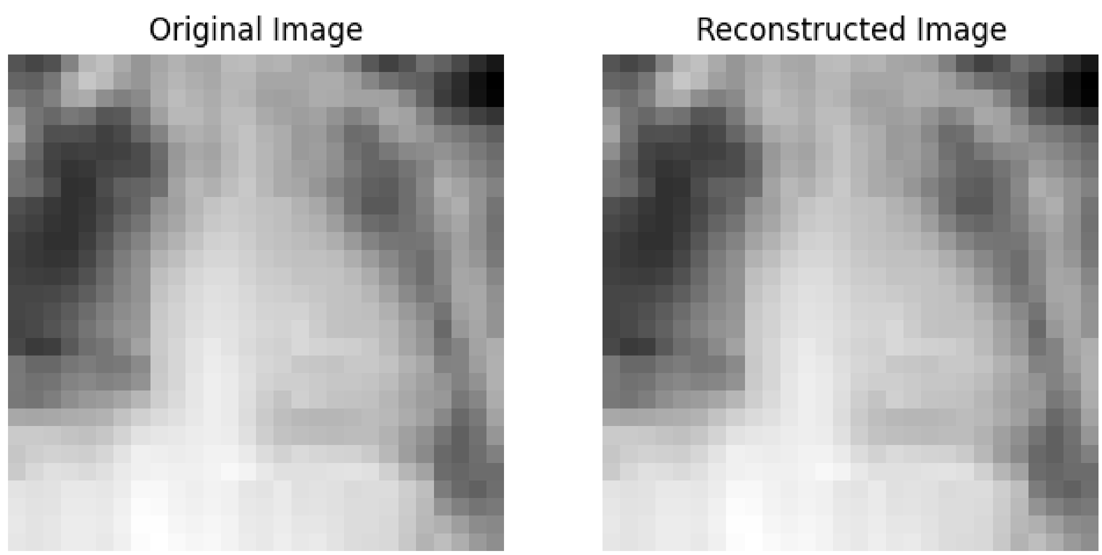
[5]:
```
def unpatchify(patches, patch_size, image_size, number_of_channels=1):
    ### Write a function that takes a batch of patches (N, L, D) where D =␣
↪patch_size**2*C
    ### and returns the batch of images (N, C, H, W)
    """
    Reconstructs images from patches.

    :param patches: Tensor of shape (N, L, D)
```

```
    :param patch_size: Tuple (patch_h, patch_w)
    :param image_size: Tuple (H, W) of original image dimensions
    :param number_of_channels: Number of channels (1 for grayscale, 3 for RGB)
    :return: Tensor of shape (N, C, H, W)
    """
    N, L, D = patches.shape
    patch_h, patch_w = patch_size
    H, W = image_size

    # Compute number of patches in each dimension
    num_patches_h = H // patch_h
    num_patches_w = W // patch_w

    # Ensure L matches the expected number of patches
    assert L == num_patches_h * num_patches_w, "Mismatch between L and expected␣
↪patches"

    # Compute channels
    C = number_of_channels if D == patch_h * patch_w * number_of_channels else 1
    assert D == patch_h * patch_w * C, f"Incorrect patch size! Expected␣
↪{patch_h * patch_w * C}, got {D}"

    # Reshape patches to (N, num_patches_h, num_patches_w, C, patch_h, patch_w)
    imgs = patches.reshape(N, num_patches_h, num_patches_w, C, patch_h, patch_w)

    # Rearrange to (N, C, num_patches_h, patch_h, num_patches_w, patch_w)
    imgs = imgs.permute(0, 3, 1, 4, 2, 5).reshape(N, C, H, W)

    return imgs
```

Check that after unpatchifying the patches obtained in the last cells, we get back to the original image batch.

[101]:
```
assert (unpatchify(patches, (4,4), (28,28)) == images).all()

# Patchify the images
def plot_patchify_unpatchify(img, patch_size, image_size):
    """
    Plot original image and reconstructed image after patchify & unpatchify.
    """
    patches = patchify(img.unsqueeze(0), patch_size)  # Patchify single image
    reconstructed = unpatchify(patches, patch_size, image_size).squeeze(0)  #␣
↪Unpatchify

    fig, ax = plt.subplots(1, 2, figsize=(8, 4))

    ax[0].imshow(img.squeeze(0), cmap='gray')
```

```
    ax[0].set_title("Original Image")
    ax[0].axis("off")

    ax[1].imshow(reconstructed.squeeze(0), cmap='gray')
    ax[1].set_title("Reconstructed Image")
    ax[1].axis("off")

    plt.show()

# **Test with the first image from dataset**
sample_image = images[0]  # Extract one image (C, H, W)
plot_patchify_unpatchify(sample_image, patch_size=(4, 4), image_size=(28, 28))
```



1.4.3 Task B-3: Implement `random_masking`

Next we need to write the function that will randomly mask out some of the patches for the encoder. We want to follow the approach described in the paper:

Your turn: follow the textual description of the algorithm above as well as the instructions in the following docstring to implement the `random_masking` function.

This function takes the original patched batch of size (N, L, D) as input and returns:

- (a) `patches_kept`: the sequence of non-masked tokens
- (b) `mask`: a binary mask indicating which grid position are masked for every image in the batch

- (c) `ids_restore`: list of indices indicating how to revert the patch shuffling operation used to create the mask.

Hint: the `gather` function in PyTorch could prove handy for this task.

[6]:
```
def random_masking(patches, mask_ratio):
        """
        ### TODO ####
        This function performs the random_masking operation as described in the␣
↪MAE paper



        Args:
            patches: original patched batch of size (N, L, D)
            mask_ratio: float between 0 and 1, the proportion of patches to␣
↪mask in each image.

        Returns:
            patches_kept: tensor (N, L_kept, D) the sequence of non-masked␣
↪patches (shuffled)
            mask: tensor (N, L) binary mask indicating which positions are␣
↪masked (in the original patch grid)
            ids_restore: tensor (N, L) list of indices indicating how to␣
↪un-shuffle the list of tokens.
        """

        N, L, D = patches.shape  # batch, length, dim
        device = patches.device

        # Step 1: create noise in [0, 1] for each patch
        noise = torch.rand(N, L, device=device)  # Shape: (N, L)

        # Step 2: sort noise for each sample
        ids_shuffle = torch.argsort(noise, dim=1)  # Indices of sorted patches␣
↪(low to high noise)
        num_keep = int(L * (1 - mask_ratio))  # Number of patches to keep

        # Step 3: store list of indices to revert shuffling operation later
        ids_restore = torch.argsort(ids_shuffle, dim=1)  # Used to unshuffle␣
↪later

        # Step 4: used shuffled list to keep only a subset of patches
        patches_shuffled = torch.gather(patches, dim=1, index=ids_shuffle.
↪unsqueeze(-1).expand(-1, -1, D))
        patches_kept = patches_shuffled[:, :num_keep, :]  # Keep a subset of␣
↪shuffled patches
```

```
        # Step 5 : generate the binary mask
        mask = torch.ones(N, L, dtype=torch.float32, device=device)  #␣
↪Initialize all as masked (1)
        mask[:, :num_keep] = 0  # Set first num_keep patches as kept (0)

        # Unshuffle the mask to align with original patch order
        mask = torch.gather(mask, dim=1, index=ids_restore)
        return patches_kept, mask, ids_restore
```

[103]:
```
patches_kept, mask, ids_restore = random_masking(patches, 0.75)
```

Check the shapes of our outputs. Are there as expected?

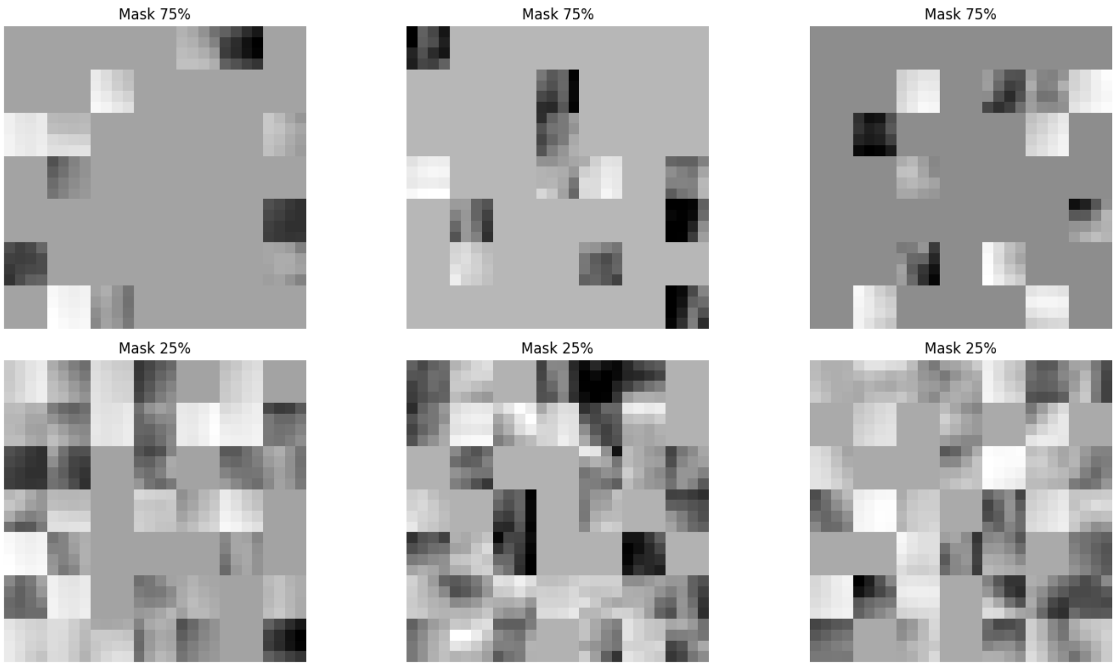
[104]:
```
# (N, L_keep, D)      (N, L)        (N, L)
patches_kept.shape, mask.shape, ids_restore.shape
```

[104]:
```
(torch.Size([32, 12, 16]), torch.Size([32, 49]), torch.Size([32, 49]))
```

Visualisation of random masking In this cell, we ask you to use the previously implemented functions `patchify`, `unpatchify` and `random_masking` to visualise the first three images in the validation batch at a masking ratio of 75% and 25%. Create a 2 x 3 subplots grids, the first row should be masked at 75%, the second one at 25%

[105]:
```
patch_size = (4,4)
images, _ = next(iter(datamodule.val_dataloader()))
f, ax = plt.subplots(2, 3, figsize=(15, 8))

# Select the first three images
images = images[:3]  # Shape: (3, 1, 28, 28)

# Define masking ratios
mask_ratios = [0.75, 0.25]
for row, mask_ratio in enumerate(mask_ratios):
    for col in range(3):  # First three images
        # 1. Patchify the image
        patches = patchify(images[col].unsqueeze(0), patch_size)  # Shape (1,␣
↪L, D)

        # 2. Apply random masking
        patches_kept, mask, ids_restore = random_masking(patches, mask_ratio)

        # 3. Reconstruct the masked image
        masked_patches = torch.zeros_like(patches)  # Create empty patches
        masked_patches[:, ids_restore[:, :patches_kept.shape[1]], :] =␣
↪patches_kept  # Restore only kept patches
        reconstructed_img = unpatchify(masked_patches, patch_size, (28, 28)).
↪squeeze(0)
```

```
        # Display the image
        ax[row, col].imshow(reconstructed_img.squeeze(0), cmap="gray")
        ax[row, col].set_title(f"Mask {mask_ratio * 100:.0f}%")
        ax[row, col].axis("off")

plt.tight_layout()
plt.show()
```



1.5 Part C: Implement and train a full MAE model.

In this part, you will use the previously defined utility functions along with some helper code that we provide to implement the full training pipeline of Masked Auto-Encoder.

We here provide you with all helper functions for defining positional embeddings and for defining the ViT forward passes. You are asked to link all these pieces together by implementing the MAE forward pass and the loss function computation, along with some visualisation function.

In the following, we provide code for creating the positional embeddings for the ViT. You do not need to implement anything here, just run this cell.

[7]:
```
from functools import partial

import torch
import torch.nn as nn
```

```python
from timm.models.vision_transformer import PatchEmbed, Block

import numpy as np

def get_2d_sincos_pos_embed(embed_dim, grid_size, cls_token=False):
    """
    grid_size: int of the grid height and width
    return:
    pos_embed: [grid_size*grid_size, embed_dim] or [1+grid_size*grid_size,␣
 ↪embed_dim] (w/ or w/o cls_token)
    """
    if isinstance(grid_size, int):
        grid_size = (grid_size[0], grid_size[1])
    grid_h = np.arange(grid_size[0], dtype=np.float32)
    grid_w = np.arange(grid_size[1], dtype=np.float32)
    grid = np.meshgrid(grid_w, grid_h)  # here w goes first
    grid = np.stack(grid, axis=0)

    grid = grid.reshape([2, 1, grid_size[0], grid_size[1]])

    # use half of dimensions to encode grid_h
    emb_h = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[0])  # (H*W,␣
 ↪D/2)
    emb_w = get_1d_sincos_pos_embed_from_grid(embed_dim // 2, grid[1])  # (H*W,␣
 ↪D/2)

    pos_embed = np.concatenate([emb_h, emb_w], axis=1)  # (H*W, D)

    if cls_token:
        pos_embed = np.concatenate([np.zeros([1, embed_dim]), pos_embed],␣
 ↪axis=0)
    return pos_embed


def get_1d_sincos_pos_embed_from_grid(embed_dim, pos):
    """
    embed_dim: output dimension for each position
    pos: a list of positions to be encoded: size (M,)
    out: (M, D)
    """
    assert embed_dim % 2 == 0
    omega = np.arange(embed_dim // 2, dtype=np.float32)
    omega /= embed_dim / 2.0
    omega = 1.0 / 10000**omega  # (D/2,)

    pos = pos.reshape(-1)  # (M,)
    out = np.einsum("m,d->md", pos, omega)  # (M, D/2), outer product
```

```python
    emb_sin = np.sin(out)  # (M, D/2)
    emb_cos = np.cos(out)  # (M, D/2)

    emb = np.concatenate([emb_sin, emb_cos], axis=1)  # (M, D)
    return emb
```

1.5.1 Task C-1: MAE model implementation

We provide you with the main skeleton for the MAE module. The init function defines the main components for you.

You are asked to fill the blanks in the following functions: * `patchify` * `configure_optimizer` * `random_masking` * `unpatchify` * `compute_loss` * `forward`

For each of these functions we give more detailed instructions in the docstring.

When you have finished implementing these functions, move on to the next cells to start training!

[8]:
```python
class MaskedAutoencoderViT(LightningModule):
    """
    Skeleton code for MAE with ViT.
    We provide most of the boiler plate code, including the ViT encoder and
    decoder forward passes. You are asked to link the pieces together
    by implementing the pieces of code marked with TODO
    """

    def __init__(
        self,
        img_size=224,
        patch_size=16,
        in_chans=3,
        embed_dim=1024,
        depth=24,
        num_heads=16,
        decoder_embed_dim=512,
        decoder_depth=8,
        decoder_num_heads=16,
        mlp_ratio=4.0,
    ):
        super().__init__()

        # MAE encoder definition
        self.patch_size = patch_size
        self.img_size = img_size
        self.in_chans = in_chans
        self.embed_dim = embed_dim
        self.in_chans = in_chans
        self.patch_embed = PatchEmbed(img_size, patch_size, in_chans, embed_dim)
```

```
        num_patches = self.patch_embed.num_patches
        print(self.patch_embed.grid_size)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(
            torch.zeros(1, num_patches + 1, embed_dim), requires_grad=False
        )

        self.blocks = nn.ModuleList(
            [
                Block(
                    embed_dim,
                    num_heads,
                    mlp_ratio,
                    qkv_bias=True,
                    norm_layer=nn.LayerNorm,
                )
                for i in range(depth)
            ]
        )
        self.norm = nn.LayerNorm(embed_dim)

        # MAE decoder definition
        self.decoder_embed = nn.Linear(embed_dim, decoder_embed_dim, bias=True)
        self.mask_token = nn.Parameter(torch.zeros(1, 1, decoder_embed_dim))
        self.decoder_pos_embed = nn.Parameter(
            torch.zeros(1, num_patches + 1, decoder_embed_dim), requires_grad=False
        )

        self.decoder_blocks = nn.ModuleList(
            [
                Block(
                    decoder_embed_dim,
                    decoder_num_heads,
                    mlp_ratio,
                    qkv_bias=True,
                    norm_layer=nn.LayerNorm,
                )
                for i in range(decoder_depth)
            ]
        )

        self.decoder_norm = nn.LayerNorm(decoder_embed_dim)
        self.decoder_pred = nn.Linear(
            decoder_embed_dim, patch_size**2 * in_chans, bias=True
        )
```

```
        # Positional embeddings
        pos_embed = get_2d_sincos_pos_embed(
            embed_dim=self.pos_embed.shape[-1],
            grid_size=self.patch_embed.grid_size,
            cls_token=True,
        )
        self.pos_embed.data.copy_(torch.from_numpy(pos_embed).float().
↪unsqueeze(0))

        decoder_pos_embed = get_2d_sincos_pos_embed(
            self.decoder_pos_embed.shape[-1],
            grid_size=self.patch_embed.grid_size,
            cls_token=True,
        )
        self.decoder_pos_embed.data.copy_(
            torch.from_numpy(decoder_pos_embed).float().unsqueeze(0)
        )


    def patchify(self, imgs):
        """
        imgs: (N, C, H, W)
        x: (N, L, D)
        """
        ### TODO: Use the previously defined function
        ### ADD YOUR CODE HERE
        return patchify(imgs, (self.patch_size, self.patch_size))

    def configure_optimizers(self):
        ### TODO: configure the optimiser to be Adam with learning rate 1e-4
        ### ADD YOUR CODE HERE
        optimizer = torch.optim.Adam(self.parameters(), lr=1e-4)
        return optimizer

    def unpatchify(self, x):
        """
        x: (N, L, D)
        imgs: (N, C, H, W)
        """
        ### TODO: Use the previously defined function
        ### ADD YOUR CODE HERE
        return unpatchify(x, (self.patch_size, self.patch_size), (self.
↪img_size, self.img_size), self.in_chans)

    def random_masking(self, x, mask_ratio):
        """
        Perform per-sample random masking by per-sample shuffling.
```

```
        Per-sample shuffling is done by argsort random noise.
        x: [N, L, D], sequence
        """
        ### TODO: Use the previously defined function
        ### ADD YOUR CODE HERE
        return random_masking(x, mask_ratio)

    def forward_encoder(self, x, mask_ratio):
        """
        Forward function for the encoding part.
        """
        # embed patches (use self.patch_embed)
        x = self.patch_embed(x)

        # add pos embed w/o cls token
        x = x + self.pos_embed[:, 1:, :]

        # masking: length -> length * mask_ratio
        x, mask, ids_restore = self.random_masking(x, mask_ratio)

        # append cls token
        cls_token = self.cls_token + self.pos_embed[:, :1, :]
        cls_tokens = cls_token.expand(x.shape[0], -1, -1)
        x = torch.cat((cls_tokens, x), dim=1)

        # apply Transformer blocks
        for blk in self.blocks:
            x = blk(x)
        x = self.norm(x)

        return x, mask, ids_restore

    def forward_decoder(self, x, ids_restore):
        """
        Forward function for the decoding part.
        """
        # embed tokens
        x = self.decoder_embed(x)

        # append mask tokens to sequence
        mask_tokens = self.mask_token.repeat(
            x.shape[0], ids_restore.shape[1] + 1 - x.shape[1], 1
        )
        x_ = torch.cat([x[:, 1:, :], mask_tokens], dim=1)  # no cls token
        x_ = torch.gather(
            x_, dim=1, index=ids_restore.unsqueeze(-1).repeat(1, 1, x.shape[2])
        )  # unshuffle
```

```
        x = torch.cat([x[:, :1, :], x_], dim=1)  # append cls token

        # add pos embed
        x = x + self.decoder_pos_embed

        # apply Transformer blocks
        for blk in self.decoder_blocks:
            x = blk(x)
        x = self.decoder_norm(x)

        # predictor projection
        x = self.decoder_pred(x)

        # remove cls token
        x = x[:, 1:, :]

        return x

    def compute_loss(self, target_patches, pred_patches, mask):
        """
        This function returns the MAE loss value for a given batch.
        Should be MSE loss over masked patches
        Args:
          target_patches: [N, C, D] ground truth patches
          pred_patches: [N, L, D] predicted patches
          mask: [N, L] binary mask indicating which patches are masked
        """
        ### TODO
        ### ADD YOUR CODE HERE

        # Compute MSE loss.
        loss = (pred_patches - target_patches).pow(2).mean(dim=-1)

        # Average loss
        loss = (loss * mask).sum() / mask.sum()
        return loss

    def forward(self, imgs, mask_ratio=0.75):
        """
        Forward function
        Args:
          imgs: batch of [N, C, H, W] images
          mask_ratio: masking ratio to use for the encoder

        Returns:
          predicted_patches [N, L, D], where D = patch_size[0]*patch_size[1]*C
          mask [N, L]
```

```
        """
        ### TODO
        ### ADD YOUR CODE HERE

        # Encoder forward
        x_encode, mask, ids_restore = self.forward_encoder(imgs, mask_ratio)
        # Predict pixel values
        x_pred = self.forward_decoder(x_encode, ids_restore)
        return x_pred, mask

    def training_step(self, batch, batch_idx):
        images = batch[0]
        predicted_patches, mask = self(images)
        target_patches = self.patchify(images)
        loss = self.compute_loss(target_patches, predicted_patches, mask)
        self.log('loss_train', loss, prog_bar=True)

        if batch_idx == 0:
            images_output = self.unpatchify(predicted_patches * mask.
↪unsqueeze(2) + target_patches * (~mask.bool()).int().unsqueeze(2))
            grid = torchvision.utils.make_grid(images[0:4], nrow=4,␣
↪normalize=True)
            self.logger.experiment.add_image('train_images_input', grid, self.
↪global_step)
            grid = torchvision.utils.make_grid(images_output[0:4], nrow=4,␣
↪normalize=True)
            self.logger.experiment.add_image('train_images_output', grid, self.
↪global_step)
            grid = torchvision.utils.make_grid(self.unpatchify(target_patches *␣
↪mask.unsqueeze(2))[0:4], nrow=4, normalize=True)
            self.logger.experiment.add_image('train_patches_target', grid, self.
↪global_step)
            grid_predicted = torchvision.utils.make_grid(self.
↪unpatchify(predicted_patches * mask.unsqueeze(2))[0:4], nrow=4,␣
↪normalize=True)
            self.logger.experiment.add_image('train_patches_predicted',␣
↪grid_predicted, self.global_step)
        return loss

    def validation_step(self, batch, batch_idx):
        images = batch[0]
        predicted_patches, mask = self(batch[0])
        target_patches = self.patchify(images)
        loss = self.compute_loss(target_patches, predicted_patches, mask)

        self.log('loss_val', loss, prog_bar=True)
```

```
    def get_class_embeddings(self, images):
        """
        Return the class embeddings extracted from the encoder
        for each image in the batch.
        This function is meant to be used at inference, we do not mask
        any patches.
        """
        embeddings, _, _ = self.forward_encoder(images, mask_ratio=0)
        return embeddings[:, 0, :]

    def predict_step(self, batch, batch_idx):
        images, labels = batch[0], batch[1]
        return {'embeddings': self.get_class_embeddings(images), 'labels':␣
↪labels}
```

Next, we define a tiny toy VIT architecture for you to use in this coursework. This is much smaller than standard ViT architectures but will allow you to train your MAE rapidly on a single GPU. Note that we use again a patch size of 4 given the small resolution of the input images.

```
[9]: def mae_vit_toy_patch4_dec256d4b():
         """
         Creates a toy ViT with patch size 4.
         """
         model = MaskedAutoencoderViT(
             in_chans=1,
             img_size=28,
             patch_size=4,
             embed_dim=384,
             depth=6,
             num_heads=6,
             decoder_embed_dim=256,
             decoder_depth=4,
             decoder_num_heads=8,
             mlp_ratio=4,
         )
         return model
```

1.5.2 Task C-2: MAE training

Tensorboard logging Load tensorboard, you should be able to monitor training and validation loss as well as your reconstructed training images.

IMPORTANT keep the output of the cell, your submitted notebook should show tensorbard as well!

```
[10]: %reload_ext tensorboard
      %tensorboard --logdir './lightning_logs/coursework/'
```

<IPython.core.display.HTML object>

We provide the training code, just run this cell and wait…

[11]:
```
seed_everything(33, workers=True)

data = PneumoniaMNISTDataModule(batch_size=32)

model = mae_vit_toy_patch4_dec256d4b()

trainer = Trainer(
    max_epochs=50,
    accelerator='auto',
    devices=1,
    logger=TensorBoardLogger(save_dir='./lightning_logs/coursework/',␣
  ↪name='mae_test'),
)
trainer.fit(model=model, datamodule=data)
```

```
Seed set to 33

Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
Using downloaded and verified file: ./data/coursework/pneumoniamnist.npz
(7, 7)

GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name            | Type       | Params | Mode
-------------------------------------------------------
0 | patch_embed     | PatchEmbed | 6.5 K  | train
1 | blocks          | ModuleList | 10.6 M | train
2 | norm            | LayerNorm  | 768    | train
3 | decoder_embed   | Linear     | 98.6 K | train
4 | decoder_blocks  | ModuleList | 3.2 M  | train
5 | decoder_norm    | LayerNorm  | 512    | train
6 | decoder_pred    | Linear     | 4.1 K  | train
  | other params    | n/a        | 32.6 K | n/a
-------------------------------------------------------
13.9 M    Trainable params
32.0 K    Non-trainable params
13.9 M    Total params
55.796    Total estimated model params size (MB)
219       Modules in train mode
0         Modules in eval mode

Sanity Checking: |          | 0/? [00:00<?, ?it/s]
```

```
D:\Anaconda\envs\DL_CW1\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:425: The
'val_dataloader' does not have many workers which may be a bottleneck. Consider
increasing the value of the `num_workers` argument` to `num_workers=11` in the
`DataLoader` to improve performance.
D:\Anaconda\envs\DL_CW1\Lib\site-
packages\pytorch_lightning\trainer\connectors\data_connector.py:425: The
'train_dataloader' does not have many workers which may be a bottleneck.
Consider increasing the value of the `num_workers` argument` to `num_workers=11`
in the `DataLoader` to improve performance.
```

```
Training: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
```

```
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
Validation: |          | 0/? [00:00<?, ?it/s]
`Trainer.fit` stopped: `max_epochs=50` reached.
```

1.6 Part D: Inspect the trained model.

In this last part, we ask you to analyse the feature embeddings (or representations) obtained from your trained model with t-SNE, similar to the tutorial on model inspection. Let's see if your model learned anything useful!

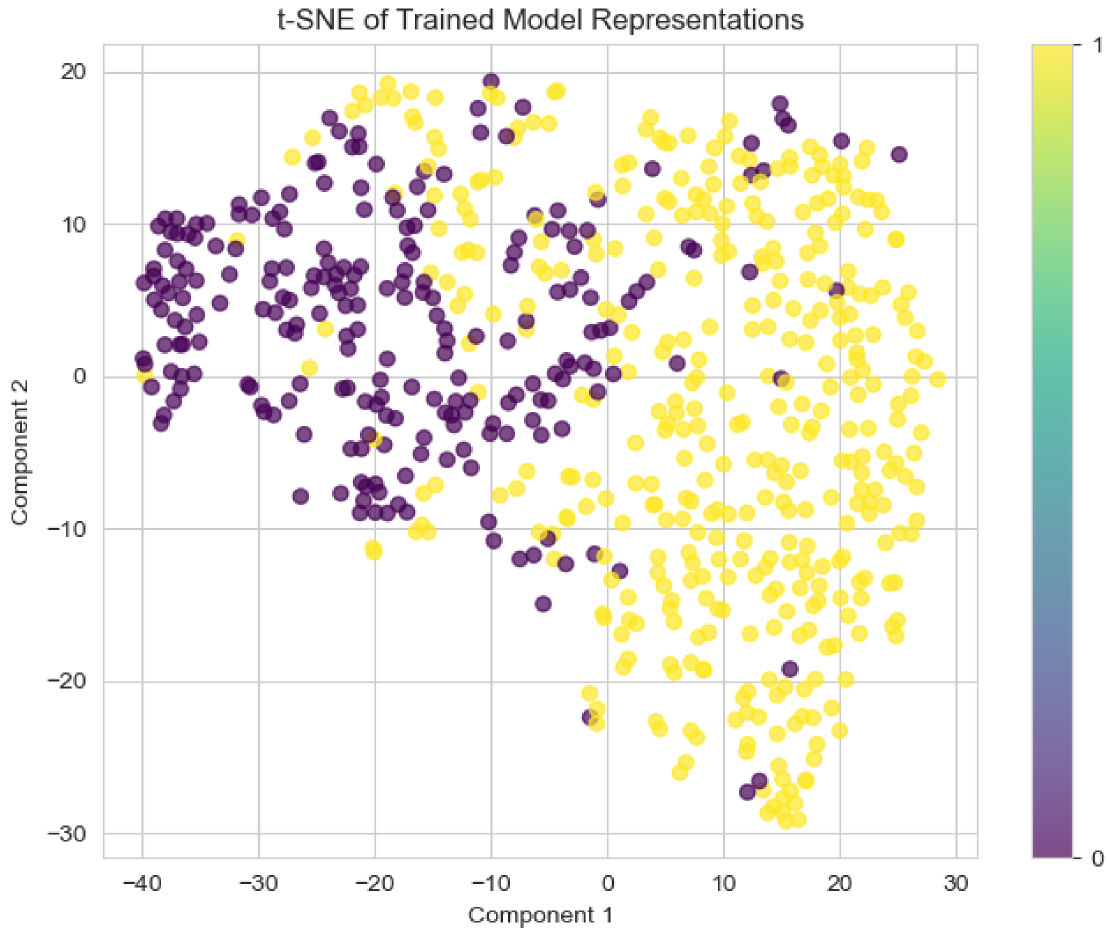
1.6.1 Task D-1: Inspect and compare the learned feature representations of your trained model.

Compare the feature embeddings of your trained model to embeddings obtained with a randomly initialised (untrained) model. Create some scatter plot visualisations and describe your findings with a few sentences.

[12]:
```
from sklearn.manifold import TSNE
import seaborn as sns
```

Let's get the representations from our trained model:

[13]:
```
def get_representations(model, dataloader):
    model.eval()
    embeddings_list = []
    labels_list = []
    with torch.no_grad():
        for imgs, labels in dataloader:
            emb = model.get_class_embeddings(imgs)
            embeddings_list.append(emb.cpu())
            labels_list.append(labels.cpu())
    embeddings = torch.cat(embeddings_list, dim=0)
    labels = torch.cat(labels_list, dim=0)
    return embeddings.numpy(), labels.numpy()

# Get the representations from trained model
trained_embeddings, trained_labels = get_representations(model, data.
 ↪test_dataloader())
# Apply t-SNE
tsne_trained = TSNE(n_components=2, random_state=42).
 ↪fit_transform(trained_embeddings)

# Plot
plt.figure(figsize=(8, 6))
scatter = plt.scatter(tsne_trained[:, 0], tsne_trained[:, 1],
                      c=trained_labels, cmap='viridis', alpha=0.7)
plt.title("t-SNE of Trained Model Representations")
plt.xlabel("Component 1")
plt.ylabel("Component 2")
cbar = plt.colorbar(scatter, ticks=[0, 1])
cbar.ax.set_yticklabels(['0', '1'])
plt.show()
```

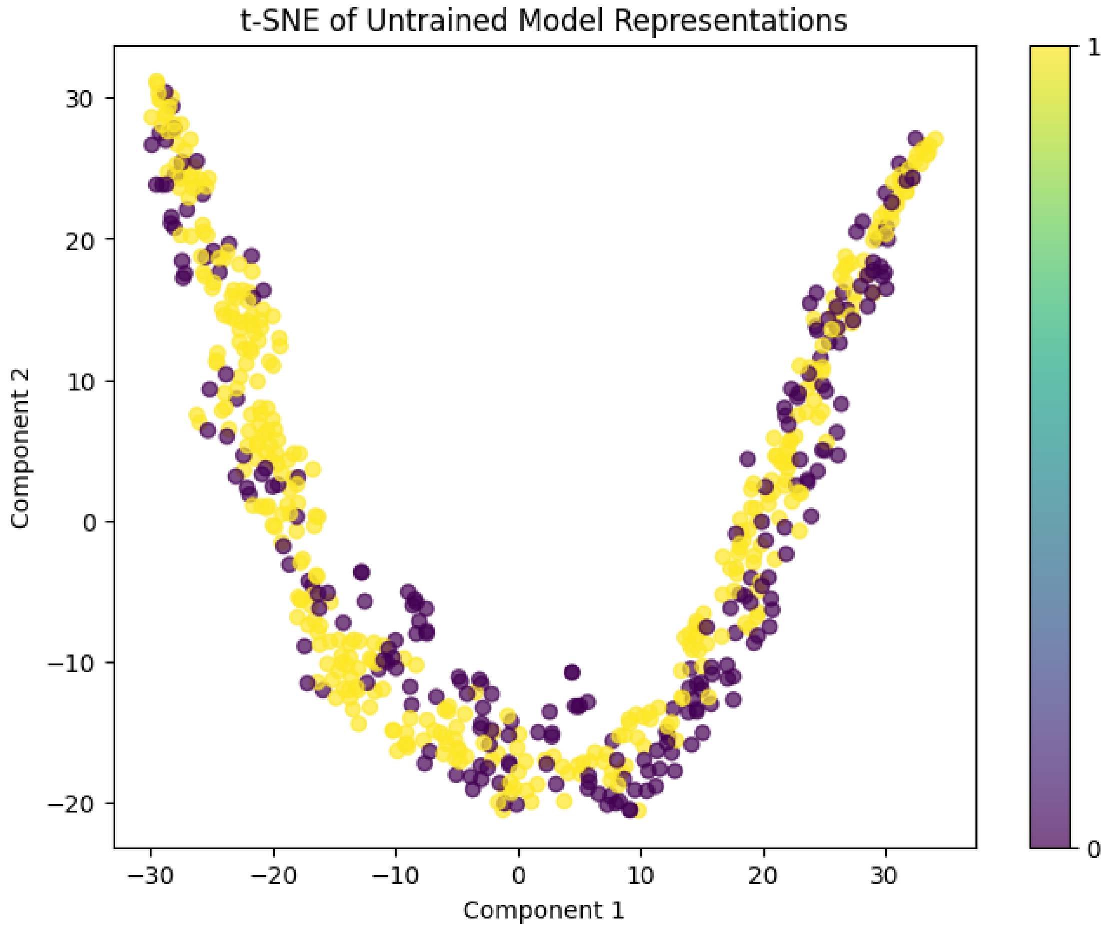


Let's compare with the representation of an untrained model

[121]:
```
# Instantiate an untrained model
untrained_model = mae_vit_toy_patch4_dec256d4b()
# Get the representations from untrained model
untrained_embeddings, untrained_labels = get_representations(untrained_model,␣
    ↪data.test_dataloader())
# Apply t-SNE
tsne_untrained = TSNE(n_components=2, random_state=33).
    ↪fit_transform(untrained_embeddings)

plt.figure(figsize=(8, 6))
scatter = plt.scatter(tsne_untrained[:, 0], tsne_untrained[:, 1],
                      c=untrained_labels, cmap='viridis', alpha=0.7)
plt.title("t-SNE of Untrained Model Representations")
plt.xlabel("Component 1")
plt.ylabel("Component 2")
cbar = plt.colorbar(scatter, ticks=[0, 1])
```

```
cbar.ax.set_yticklabels(['0', '1'])
plt.show()
```

(7, 7)



Summarise your observations…

The t-SNE scatter plot of the trained model's representations generally shows clusters where points with similar labels group together. This suggests that during training, the MAE learned representations that capture discriminative features, even though its primary task was self-supervised reconstruction.

In contrast, the t-SNE plot for the untrained model appears more random, with no clear clustering by label. This indicates that before training, the feature representations do not encode meaningful class information.