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`.

Your details

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

Authors: **Jin Wang** & **Jasper Lo**

DoC alias: **jw2824** & **jsl20**

Setup

```
from math import floor
# On Google Colab uncomment the following line to install PyTorch
Lightning and the MedMNIST dataset
#! pip install lightning medmnist timm

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
```

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).

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`.

```
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

        ### TODO: Define here your data augmentation pipeline.
        self.transform = transforms.Compose([
            transforms.RandomResizedCrop(self.size, scale=(0.7, 1.0)),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.5], std=[0.5])
        ]) if self.do_augment else transforms.Compose([
              transforms.ToTensor(),
              transforms.Normalize(mean=[0.5], std=[0.5])
        ])

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

    def __getitem__(self, index):
        ### TODO: Implement the __getitem__ function to return the
image and its class label.
        img = self.imgs[index]
        label = self.labels[index]

        img = Image.fromarray(img)
        img = self.transform(img)

        return img, label
```

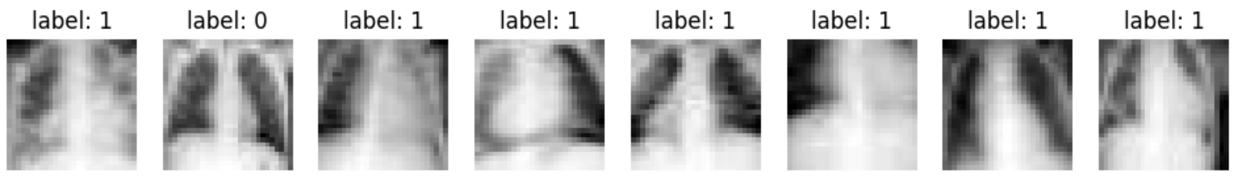
We use a LightningDataModule for handling your PneumoniaMNIST dataset. No changes needed for this part.

```
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.

```
# 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")

100%|██████████| 4.17M/4.17M [00:00<00:00, 5.78MB/s]
```



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.

Task B-1: Implement `patchify`

```
import torch
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
    device = imgs.device
    patch_h, patch_w = patch_size
    if H % patch_h != 0 or W % patch_w != 0:
        raise Exception("H and W of image are not divisible by patch
size")
    else:
        L = (H // patch_h) *  (W // patch_w)
        patches = torch.zeros(N, L, patch_h*patch_w*C, device=device)
        i = 0
        for x in range(0, H, patch_h):
            for y in range(0, W, patch_w):
                patch = imgs[:, :, x:x+patch_h, y:y+patch_w]
                patches[:, i, :] = patch.reshape(N, patch_h*patch_w*C)
                i += 1
    return patches
```

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

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

imagesize = images.shape[2:]
images.shape
```

```
torch.Size([32, 1, 28, 28])
```

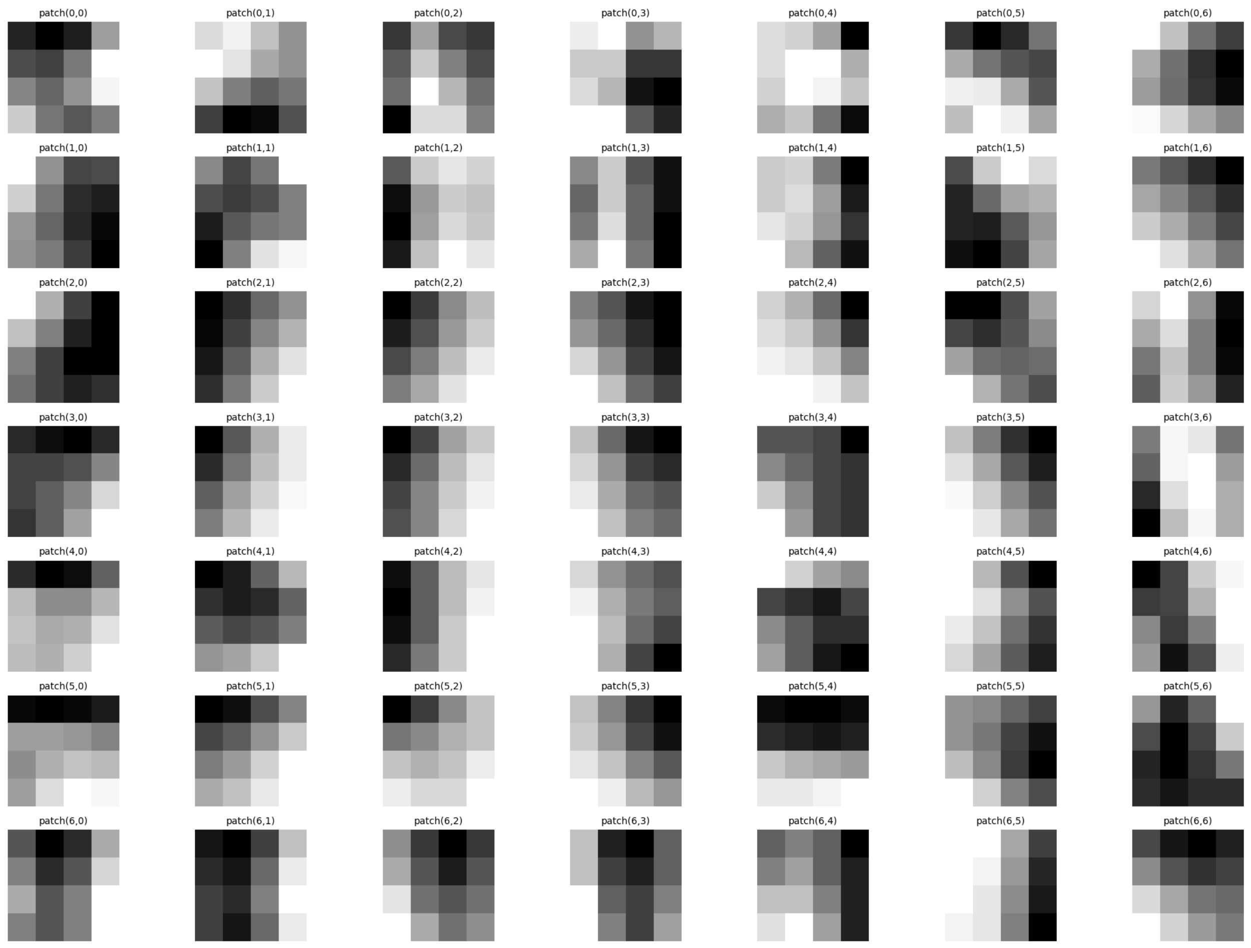
```
# Assuming a patch size of (4,4) test your patchify function
# and test that the shape of the outputs corresponds at what is
expected
patch_size = (4,4)
patches = patchify(images, patch_size)

patches.shape
```

```
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.

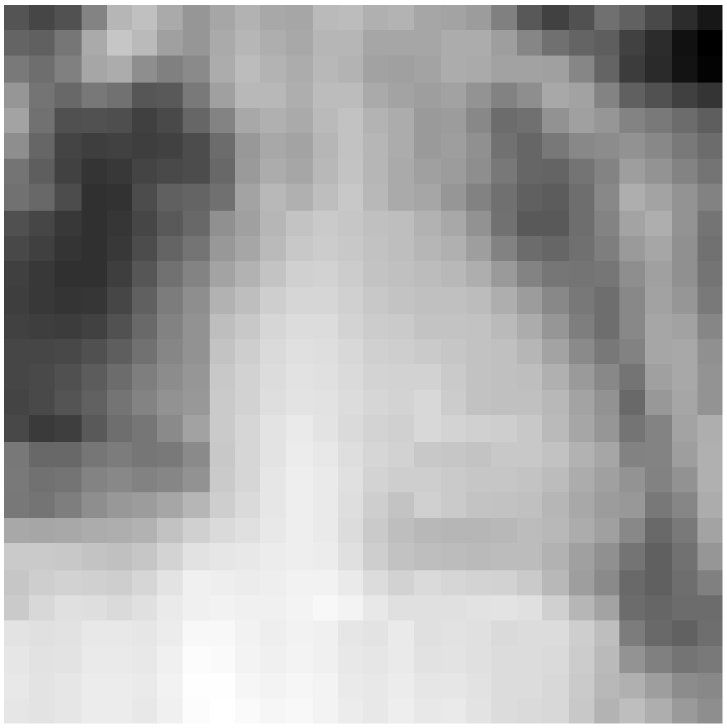
```
# TODO plot all the patches in a subplots grid (with their correct
position in the grid) for the first imge in the batch
f, ax = plt.subplots(7, 7, figsize=(20, 14))
for i in range(7):
    for j in range(7):
        ax[i, j].imshow(patches[0, i*7+j].reshape(patch_size),
cmap='gray')
        ax[i, j].set_title(f"patch({i},{j})", fontsize=10)
        ax[i, j].axis('off')
plt.tight_layout()
```



Compare the ouput with the original image

```
# TODO plot the original image for comparison
plt.imshow(images[0, 0], cmap='gray')
plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```



Task B-2: Implement `unpatchify`

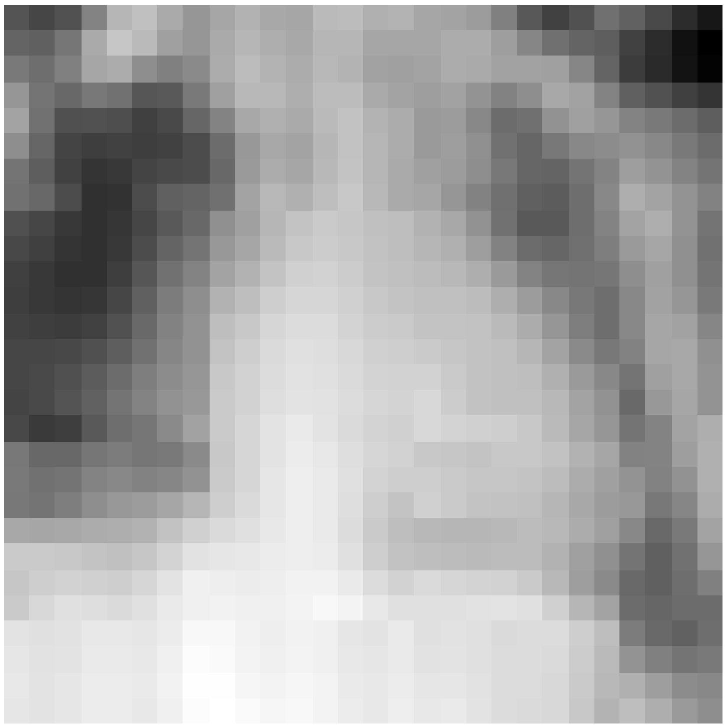
Next, you are asked to create the reverse function able to take in a batch of patches and return the corresponding batch of images.

```python
def unpatchify(patches, patch_size, image_size, number_of_channels=1):
    ### TODO
    ### 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)
    H, W = image_size
    N, L, D = patches.shape
    device = patches.device
    images = torch.zeros((N, number_of_channels, H, W), device=device)
    patch_h, patch_w = patch_size
    i = 0
    for x in range(0, H, patch_h):
        for y in range(0, W, patch_w):
            patch = patches[:, i, :].reshape(N, number_of_channels,
patch_h, patch_w)
            images[:, :, x:x+patch_h, y:y+patch_w] = patch
            i += 1
    return images
```

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

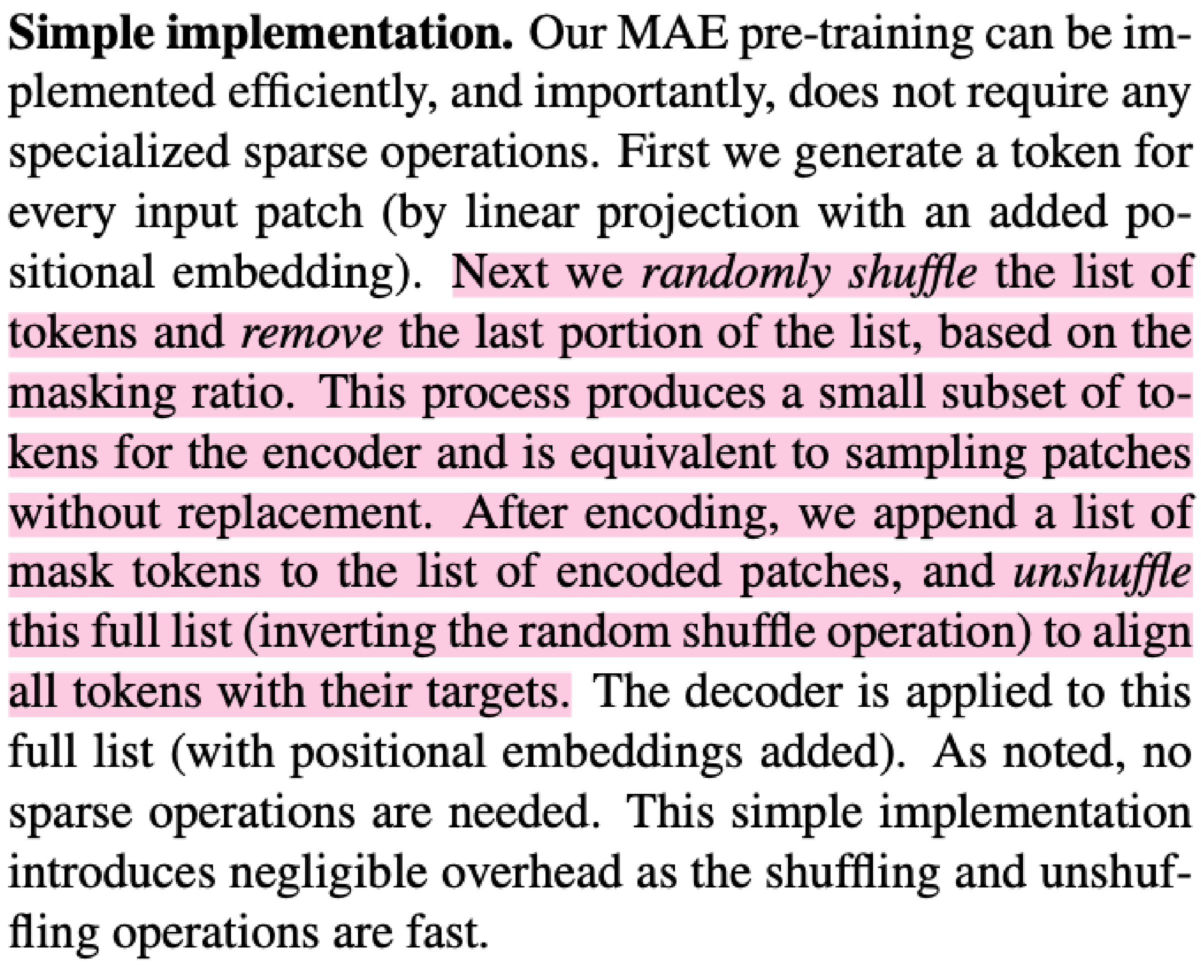
```
assert (unpatchify(patches, patch_size, imagesize) == images).all()
### TODO plot the first image after applying patchify and unpatchify
rimgs = unpatchify(patches, patch_size, imagesize)
plt.imshow(rimgs[0, 0], cmap='gray')
plt.axis('off')
```

```
(-0.5, 27.5, 27.5, -0.5)
```



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:



Simple implementation. Our MAE pre-training can be implemented efficiently, and importantly, does not require any specialized sparse operations. First we generate a token for every input patch (by linear projection with an added positional embedding). Next we *randomly shuffle* the list of tokens and *remove* the last portion of the list, based on the masking ratio. This process produces a small subset of tokens for the encoder and is equivalent to sampling patches without replacement. After encoding, we append a list of mask tokens to the list of encoded patches, and *unshuffle* this full list (inverting the random shuffle operation) to align all tokens with their targets. The decoder is applied to this full list (with positional embeddings added). As noted, no sparse operations are needed. This simple implementation introduces negligible overhead as the shuffling and unshuffling operations are fast.

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.

```
import numpy as np
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
        mask = []
        retain = int(np.floor((1 - mask_ratio) * L))
        perm = []
        for i in range(N):
            # ith cell in permutation stores the index of original
patch that will be in patches_kept ith cell
            permutation = torch.randperm(L, device = device)
            perm.append(permutation)
            m = torch.tensor([True for _ in range(L)], device =
device)
            m[permutation[:retain]] = False
            mask.append(m)
        perm = torch.stack(perm, 0).to(device=device)
        ids_restore = torch.argsort(perm, 1)
        patches_kept = torch.gather(patches, 1, perm.unsqueeze(-
1).repeat(1, 1, D))
        patches_kept = patches_kept[:, :retain, :]
        mask = torch.stack(mask, 0).to(device=device)
        return patches_kept, mask, ids_restore

patches_kept, mask, ids_restore = random_masking(patches, 0.75)
```

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

```
patches_kept.shape, mask.shape, ids_restore.shape
```

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

The function operates on a batch of size 32, providing the first number in the shapes of the 3 tensors.

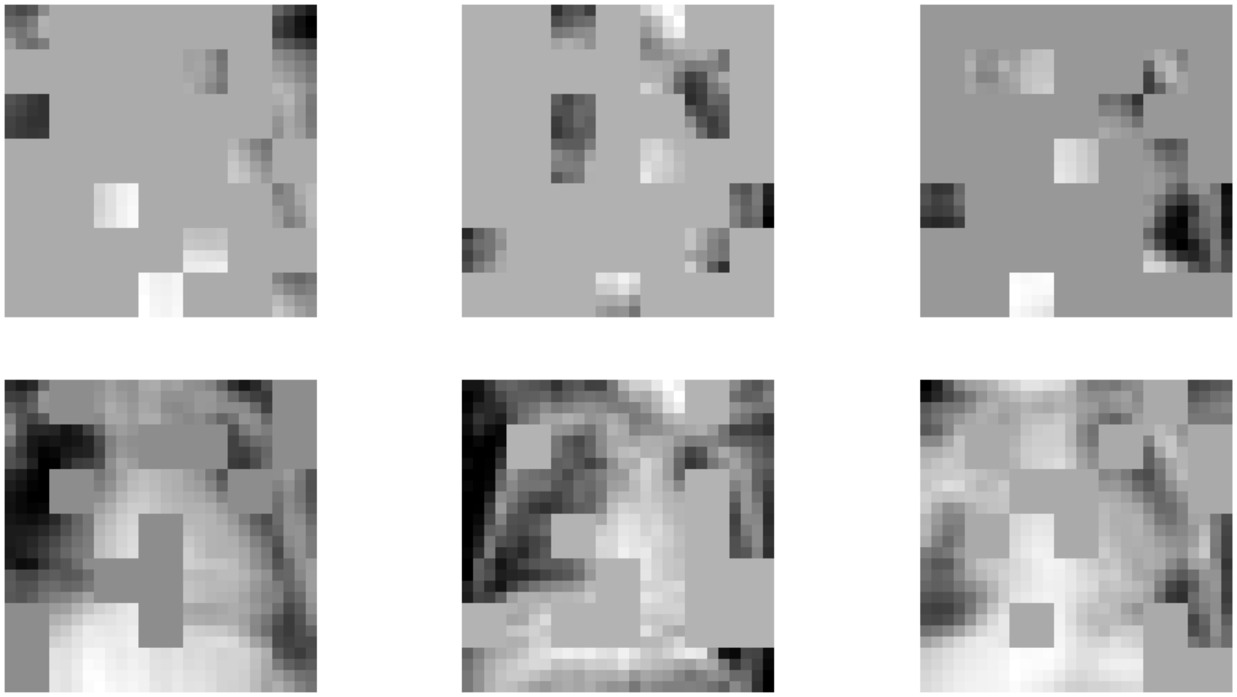
For 'patches_kept', since there are 49 patches obtained from splitting the original 28x28 image into patches of size 4x4, applying a masking ratio of 0.75 results in keeping 36 patches when rounded down to an integer, which is expected. Each patch being size 4x4 produces the dimension size 16 in 'patches_kept'.

The 'mask' and 'ids_restore' tensors contain information relating to position and order of the patches, therefore 49 is the expected length for both of those tensors since each image is divided into 49 patches.

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%

```
patch_size = (4,4)
images, _ = next(iter(datamodule.val_dataloader()))
f, ax = plt.subplots(2, 3, figsize=(15, 8))
### TODO
image_size = (28, 28)
patches = patchify(images[:3], patch_size)
N, L, D = patches.shape
patches_kept, mask, ids_restore = random_masking(patches, 0.75)
# produce empty masked patches
mask_patchs = torch.zeros(N, L-patches_kept.shape[1], D)
# append mask patches to kept patches
x_ = torch.cat([patches_kept, mask_patchs], dim=1)
patches_masked = torch.gather(x_, dim=1, index=ids_restore.unsqueeze(-
1).repeat(1, 1, D)) # unshuffle
# naive method:
# patches_masked = torch.zeros_like(patches)
# for i in range(3):
#     ids = ids_restore[i]
#     m = mask[i]
#     for x in range(4*4):
#         if not m[x]:
#             patches_masked[i, x] = patches_kept[i, ids[x]]
img = unpatchify(patches_masked, patch_size, image_size)
for i in range(3):
    ax[0,i].imshow(img[i, 0], cmap='gray')
    ax[0,i].axis('off')

patches_kept, mask, ids_restore = random_masking(patches, 0.25)
mask_patchs = torch.zeros(N, L-patches_kept.shape[1], D)
x_ = torch.cat([patches_kept, mask_patchs], dim=1)
patches_masked = torch.gather(x_, dim=1, index=ids_restore.unsqueeze(-
1).repeat(1, 1, D))
img = unpatchify(patches_masked, patch_size, image_size)
for i in range(3):
    ax[1,i].imshow(img[i, 0], cmap='gray')
    ax[1,i].axis('off')
```



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.

```python
from functools import partial

import torch
import torch.nn as nn

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, grid_size)
    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

    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
```

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!

```
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,
        # the default below is the one going to be trained
        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,
    ):
        super().__init__()

        # MAE encoder definition
        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().unsqueez
e(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
        return patchify(imgs, self.patch_embed.patch_size)

    def configure_optimizers(self):
        ### TODO: configure the optimiser to be Adam with learning
rate 1e-4
        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
        return unpatchify(x, self.patch_embed.patch_size,
self.patch_embed.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
        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, L, D] ground truth patches
          pred_patches: [N, L, D] predicted patches
          mask: [N, L] binary mask indicating which patches are masked
        """
        ### TODO
        mask_ = mask.unsqueeze(2)
        target = torch.masked_select(target_patches, mask_)
        pred = torch.masked_select(pred_patches, mask_)
        return torch.nn.functional.mse_loss(pred, target)

    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
        x, mask, ids_restore = self.forward_encoder(imgs, mask_ratio)
        pred_patches = self.forward_decoder(x, ids_restore)
        return pred_patches, 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.

```
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
```

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!

```
%reload_ext tensorboard
%tensorboard --logdir './lightning_logs/coursework/'

Launching TensorBoard...
```

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

```
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
```

```
(7, 7)
```

```
/vol/bitbucket/jw2824/DLenv/lib/python3.12/site-packages/
lightning_fabric/plugins/environments/slurm.py:204: The `srun` command
is available on your system but is not used. HINT: If your intention
is to run Lightning on SLURM, prepend your python command with `srun`
like so: srun python /vol/bitbucket/jw2824/DLenv/lib/python3.12/site-
pack ...
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
```

{"model_id":"41cea8c87fb8466ab06eb2350b837d4c","version_major":2,"vers
ion_minor":0}

```
/vol/bitbucket/jw2824/DLenv/lib/python3.12/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=5` in the `DataLoader` to improve performance.
```

```
/vol/bitbucket/jw2824/DLenv/lib/python3.12/site-packages/pytorch_light
ning/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=5`
in the `DataLoader` to improve performance.
```

{"model_id":"375a6d2547434c428b6b25574101fc8b","version_major":2,"vers
ion_minor":0}

{"model_id":"0fb2269422c04ad9b48c6e0090ad26a1","version_major":2,"vers
ion_minor":0}

{"model_id":"aac693f0900b424b88ed10b044f03204","version_major":2,"vers
ion_minor":0}

{"model_id":"1cc328afc368407cad69d1c5cfe8d577","version_major":2,"vers
ion_minor":0}

{"model_id":"537b126465ee4016bd60638d45b72fd9","version_major":2,"vers
ion_minor":0}

{"model_id":"88296a8d68ce4809919c484fb8c98bfa","version_major":2,"vers
ion_minor":0}

{"model_id":"55cec62bb18240108f0ae2f6daaba760","version_major":2,"vers
ion_minor":0}

{"model_id":"91e2dbe536cf4ade9ac8eba3ee919d58","version_major":2,"vers
ion_minor":0}

{"model_id":"e0e7a40fd0c947f8a013aa2e1c33a7bc","version_major":2,"vers
ion_minor":0}

{"model_id":"636f1e192ed74e9db8f56620d1fcd1c1","version_major":2,"vers
ion_minor":0}

{"model_id":"9714a6f7e29f42f5aaee006b27be22e1","version_major":2,"vers
ion_minor":0}

{"model_id":"718c9ec237cd4bfdb91dcb16219df19d","version_major":2,"vers
ion_minor":0}

{"model_id":"94a372e7c29d43cab24bef6a6b84314f","version_major":2,"vers
ion_minor":0}

{"model_id":"fc7e9fb21cb34bb8bf788d38dda30aee","version_major":2,"vers
ion_minor":0}

{"model_id":"5acb87377d934107a5e6b16e043b5a9a","version_major":2,"vers
ion_minor":0}

{"model_id":"71294f60a78743a59b92968aeea6e0e0","version_major":2,"vers
ion_minor":0}

```
{"model_id":"7da26cc7f9d44a35a14a6c4880dfedef","version_major":2,"version_minor":0}

{"model_id":"b3b93a277ebd43c8b35331382988231c","version_major":2,"version_minor":0}

{"model_id":"20945155b4f944cead06dac82dc3f99f","version_major":2,"version_minor":0}

{"model_id":"194be12f7b85403d9b513a5aa132a74c","version_major":2,"version_minor":0}

{"model_id":"f92668bbe07f4f3c97ee3d1dbf71dc3b","version_major":2,"version_minor":0}

{"model_id":"398a808baf37489da0fd1d947b36aef0","version_major":2,"version_minor":0}

{"model_id":"87b19c4e28e54a1cade906b35c3f0061","version_major":2,"version_minor":0}

{"model_id":"622b05d8af3a46b4a60d1a7c0aa73479","version_major":2,"version_minor":0}

{"model_id":"ec8760d1f1f64630ba8460fa38b995ee","version_major":2,"version_minor":0}

{"model_id":"5ab7702f31f84bf7a5f0eef5bbd360fc","version_major":2,"version_minor":0}

{"model_id":"0bb38e60ad2b47e080158605c12d769e","version_major":2,"version_minor":0}

{"model_id":"d193a4d46f4346c9aadede1fd6a38f0f","version_major":2,"version_minor":0}

{"model_id":"6e5dc6ae829e481c9d49860f6e79ecd4","version_major":2,"version_minor":0}

{"model_id":"e9dbb22cd011419fae6b76f3c7737c34","version_major":2,"version_minor":0}

{"model_id":"20e6b61812384668b1d58c56de69484b","version_major":2,"version_minor":0}

{"model_id":"832c199b94b640d6bd11f45cdef37da3","version_major":2,"version_minor":0}

{"model_id":"87e855c6093b4595aa854844a2711501","version_major":2,"version_minor":0}

{"model_id":"586ebacdff5742918b5ac89dcc05cc8e","version_major":2,"version_minor":0}
```

```
{"model_id":"e3d3e1ba229e47d596817bba4d479185","version_major":2,"vers
ion_minor":0}

{"model_id":"0ed55fbda74e42e584397db5b887b719","version_major":2,"vers
ion_minor":0}

{"model_id":"f8d288e7a6ac42eaa26faa5806f6a84b","version_major":2,"vers
ion_minor":0}

{"model_id":"150d0b9484864b7b8d826aaf0cd0f339","version_major":2,"vers
ion_minor":0}

{"model_id":"48f4b1d60023432890f0cfb101ac230a","version_major":2,"vers
ion_minor":0}

{"model_id":"db8cded736b146539d76fde5015ab0c1","version_major":2,"vers
ion_minor":0}

{"model_id":"2562750cb2694c66bc7272e04636acc8","version_major":2,"vers
ion_minor":0}

{"model_id":"e1899432330642d4a7909bb784e0c090","version_major":2,"vers
ion_minor":0}

{"model_id":"22ed235ce19f48038aa8b1aca1e3b3d4","version_major":2,"vers
ion_minor":0}

{"model_id":"de0d3ff3a65b481db887d5b3ce7a4e22","version_major":2,"vers
ion_minor":0}

{"model_id":"0465692be5cc4dd3a2a7135c8ab0a511","version_major":2,"vers
ion_minor":0}

{"model_id":"d7d8a52b617e4969a00c86babdbf2d7f","version_major":2,"vers
ion_minor":0}

{"model_id":"478436def396423aa5b99e46e9a9e6b0","version_major":2,"vers
ion_minor":0}

{"model_id":"23fc080fda6641208b0287e00ec12fcf","version_major":2,"vers
ion_minor":0}

{"model_id":"e5ab0ce9674448cfa73414e8db504ed6","version_major":2,"vers
ion_minor":0}

{"model_id":"ae2d0b4c3efd49afa9f12682821ab6a2","version_major":2,"vers
ion_minor":0}

{"model_id":"615c50e3146e450c816c6fd353857479","version_major":2,"vers
ion_minor":0}

`Trainer.fit` stopped: `max_epochs=50` reached.
```

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!

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.

```
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
import seaborn as sns
import pandas as pd
```

Let's get the representations from our trained model:

```
import gc
gc.collect()
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"
torch.cuda.empty_cache()
print(torch.cuda.memory_summary(device=None, abbreviated=False))
```

```
|
======================================================================
=====|
|                  PyTorch CUDA memory summary, device ID 0
|
|---------------------------------------------------------------------
------|
|            CUDA OOMs: 0              |        cudaMalloc retries: 0
|
|
======================================================================
=====|
|        Metric         | Cur Usage  | Peak Usage | Tot Alloc  | Tot
Freed  |
|---------------------------------------------------------------------
------|
| Allocated memory      |      0 B   |      0 B   |      0 B   |
0 B   |
|       from large pool |      0 B   |      0 B   |      0 B   |
0 B   |
|       from small pool |      0 B   |      0 B   |      0 B   |
0 B   |
|---------------------------------------------------------------------
```

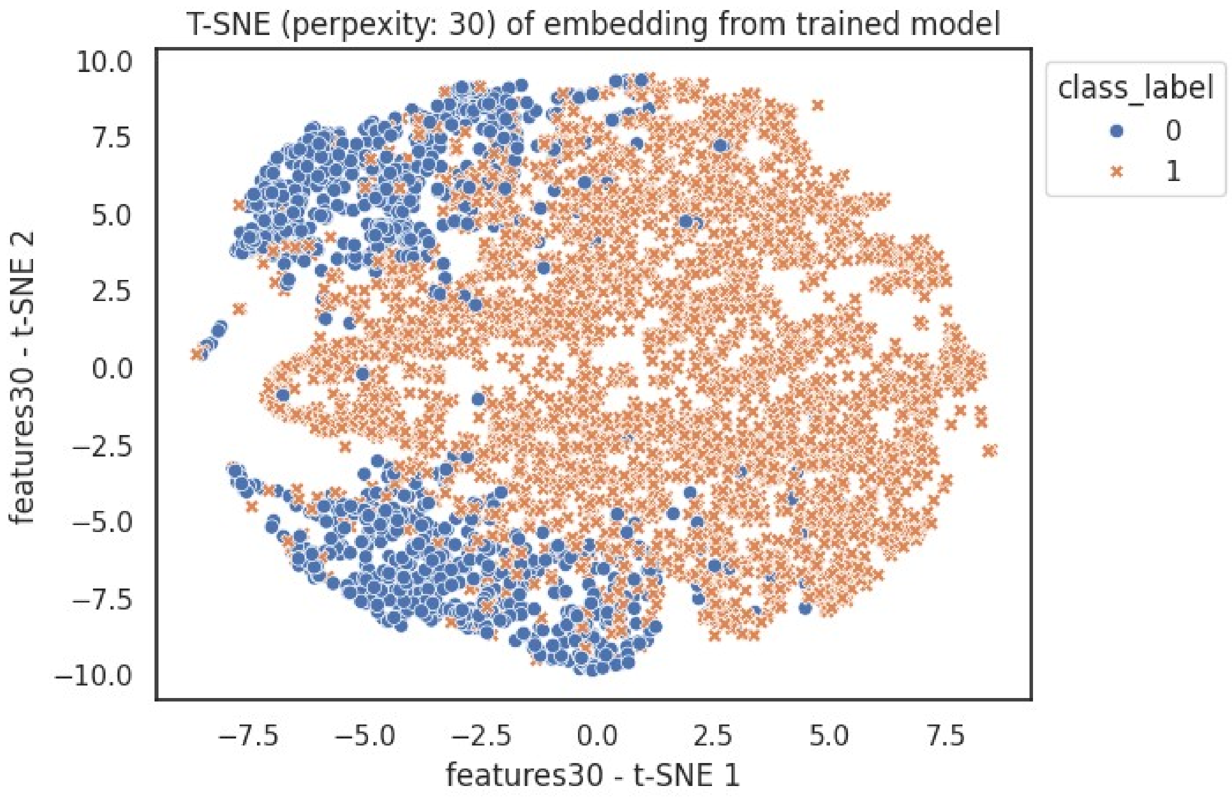
```
------|
| Active memory         |      0 B   |      0 B   |      0 B   |
0 B   |
|       from large pool |      0 B   |      0 B   |      0 B   |
0 B   |
|       from small pool |      0 B   |      0 B   |      0 B   |
0 B   |
|---------------------------------------------------------------------
------|
| Requested memory      |      0 B   |      0 B   |      0 B   |
0 B   |
|       from large pool |      0 B   |      0 B   |      0 B   |
0 B   |
|       from small pool |      0 B   |      0 B   |      0 B   |
0 B   |
|---------------------------------------------------------------------
------|
| GPU reserved memory   |      0 B   |      0 B   |      0 B   |
0 B   |
|       from large pool |      0 B   |      0 B   |      0 B   |
0 B   |
|       from small pool |      0 B   |      0 B   |      0 B   |
0 B   |
|---------------------------------------------------------------------
------|
| Non-releasable memory |      0 B   |      0 B   |      0 B   |
0 B   |
|       from large pool |      0 B   |      0 B   |      0 B   |
0 B   |
|       from small pool |      0 B   |      0 B   |      0 B   |
0 B   |
|---------------------------------------------------------------------
------|
| Allocations           |       0    |       0    |       0    |
0    |
|       from large pool |       0    |       0    |       0    |
0    |
|       from small pool |       0    |       0    |       0    |
0    |
|---------------------------------------------------------------------
------|
| Active allocs         |       0    |       0    |       0    |
0    |
|       from large pool |       0    |       0    |       0    |
0    |
|       from small pool |       0    |       0    |       0    |
0    |
|---------------------------------------------------------------------
------|
| GPU reserved segments |       0    |       0    |       0    |
0    |
|       from large pool |       0    |       0    |       0    |
0    |
|       from small pool |       0    |       0    |       0    |
0    |
|----------------------------------------------------------------------
------|
| Non-releasable allocs |       0    |       0    |       0    |
0    |
|       from large pool |       0    |       0    |       0    |
0    |
|       from small pool |       0    |       0    |       0    |
0    |
|----------------------------------------------------------------------
------|
| Oversize allocations  |       0    |       0    |       0    |
0    |
|----------------------------------------------------------------------
------|
| Oversize GPU segments |       0    |       0    |       0    |
0    |
|
======================================================================
=====|


model =
MaskedAutoencoderViT.load_from_checkpoint("./lightning_logs/coursework
/mae_test/version_0/checkpoints/epoch=49-step=7400.ckpt")
device = model.device
model.eval()

(7, 7)

MaskedAutoencoderViT(
  (patch_embed): PatchEmbed(
    (proj): Conv2d(1, 384, kernel_size=(4, 4), stride=(4, 4))
    (norm): Identity()
  )
  (blocks): ModuleList(
    (0-5): 6 x Block(
      (norm1): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=384, out_features=1152, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (proj): Linear(in_features=384, out_features=384, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (ls1): Identity()
      (drop_path1): Identity()
      (norm2): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=384, out_features=1536, bias=True)
        (act): GELU(approximate='none')
        (drop1): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (fc2): Linear(in_features=1536, out_features=384, bias=True)
        (drop2): Dropout(p=0.0, inplace=False)
      )
      (ls2): Identity()
      (drop_path2): Identity()
    )
  )
  (norm): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
  (decoder_embed): Linear(in_features=384, out_features=256,
bias=True)
  (decoder_blocks): ModuleList(
    (0-3): 4 x Block(
      (norm1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=256, out_features=768, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (proj): Linear(in_features=256, out_features=256, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (ls1): Identity()
      (drop_path1): Identity()
      (norm2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=256, out_features=1024, bias=True)
        (act): GELU(approximate='none')
        (drop1): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (fc2): Linear(in_features=1024, out_features=256, bias=True)
        (drop2): Dropout(p=0.0, inplace=False)
      )
      (ls2): Identity()
      (drop_path2): Identity()
    )
  )
  (decoder_norm): LayerNorm((256,), eps=1e-05,
elementwise_affine=True)
  (decoder_pred): Linear(in_features=256, out_features=16, bias=True)
)
```

```python
# use test data to produce representations
datamodule = PneumoniaMNISTDataModule(batch_size=16)
test_loader = datamodule.train_dataloader()
# extract embeddings from trained model
embeddings = []
labels = []
for i, l in test_loader:
  temp = i.to(device=device)
  embeddings.append(model.get_class_embeddings(temp).cpu().detach())
  del temp
  torch.cuda.empty_cache()
  labels.append(l)
embeddings = torch.cat(embeddings).numpy()
labels = torch.cat(labels).numpy()

# create dataframe to store data to be visualized
df_trained = pd.DataFrame(labels, columns=['class_label'])

# use PCA to reduce dimension from 384 to 50
pca = PCA(n_components=40)
embeddings_pca = pca.fit_transform(embeddings)
print(embeddings_pca.shape)
```

```
(4708, 40)
```

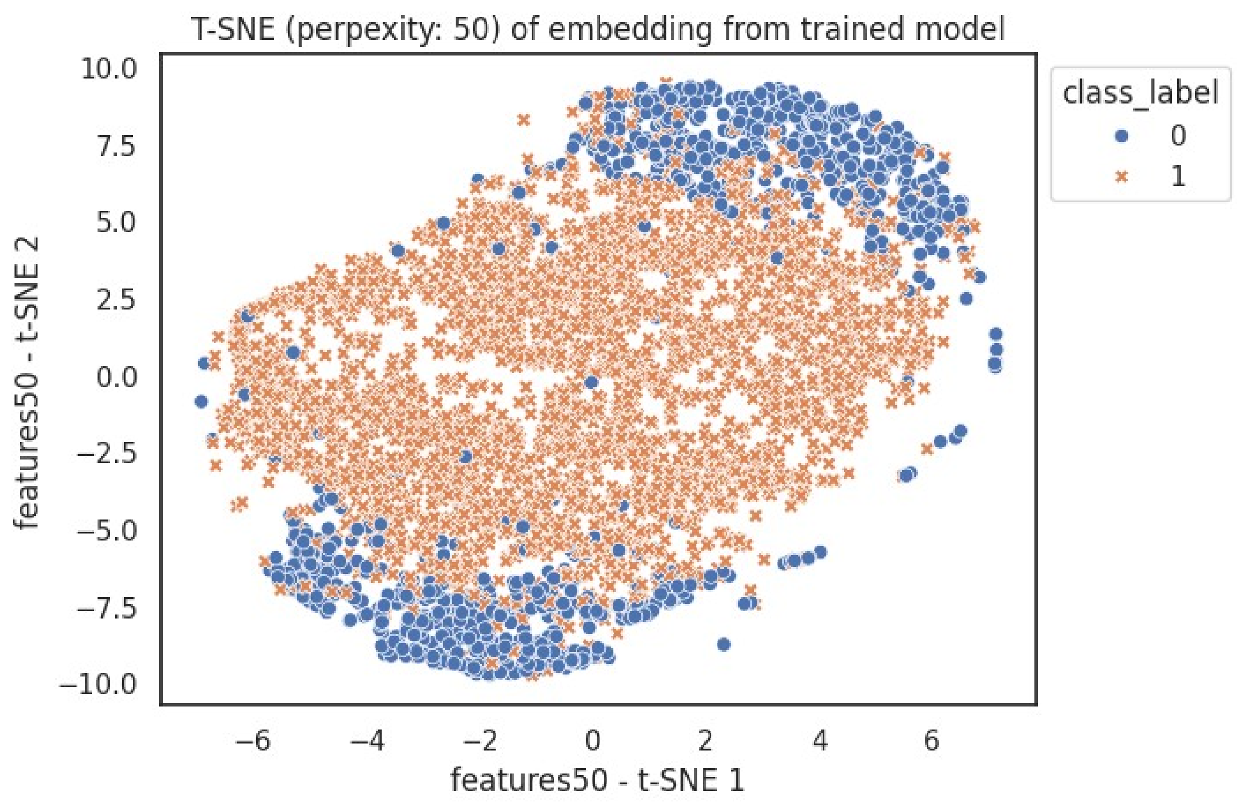
```python
# calculating t-sne data of perplexity: 30
embeddings_tsne = TSNE(perplexity=30, init="pca",
max_iter=300).fit_transform(embeddings_pca)
df_trained['features30 - t-SNE 1'] = embeddings_tsne[:,0]
df_trained['features30 - t-SNE 2'] = embeddings_tsne[:,1]

# start plotting tsne
sns.set_theme(style="white")
ax = sns.scatterplot(data=df_trained, x='features30 - t-SNE 1',
y='features30 - t-SNE 2', hue='class_label', style="class_label")
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
ax.set_title("T-SNE (perpexity: 30) of embedding from trained model")
```

```
Text(0.5, 1.0, 'T-SNE (perpexity: 30) of embedding from trained
model')
```



```
# calculating t-sne data of perplexity: 50
embeddings_tsne = TSNE(perplexity=50, init="random",
max_iter=300).fit_transform(embeddings_pca)
df_trained['features50 - t-SNE 1'] = embeddings_tsne[:,0]
df_trained['features50 - t-SNE 2'] = embeddings_tsne[:,1]

# start plotting tsne
sns.set_theme(style="white")
ax = sns.scatterplot(data=df_trained, x='features50 - t-SNE 1',
y='features50 - t-SNE 2', hue='class_label', style="class_label")
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
ax.set_title("T-SNE (perpexity: 50) of embedding from trained model")
```

```
Text(0.5, 1.0, 'T-SNE (perpexity: 50) of embedding from trained
model')
```



Let's compare with the representation of an untrained model

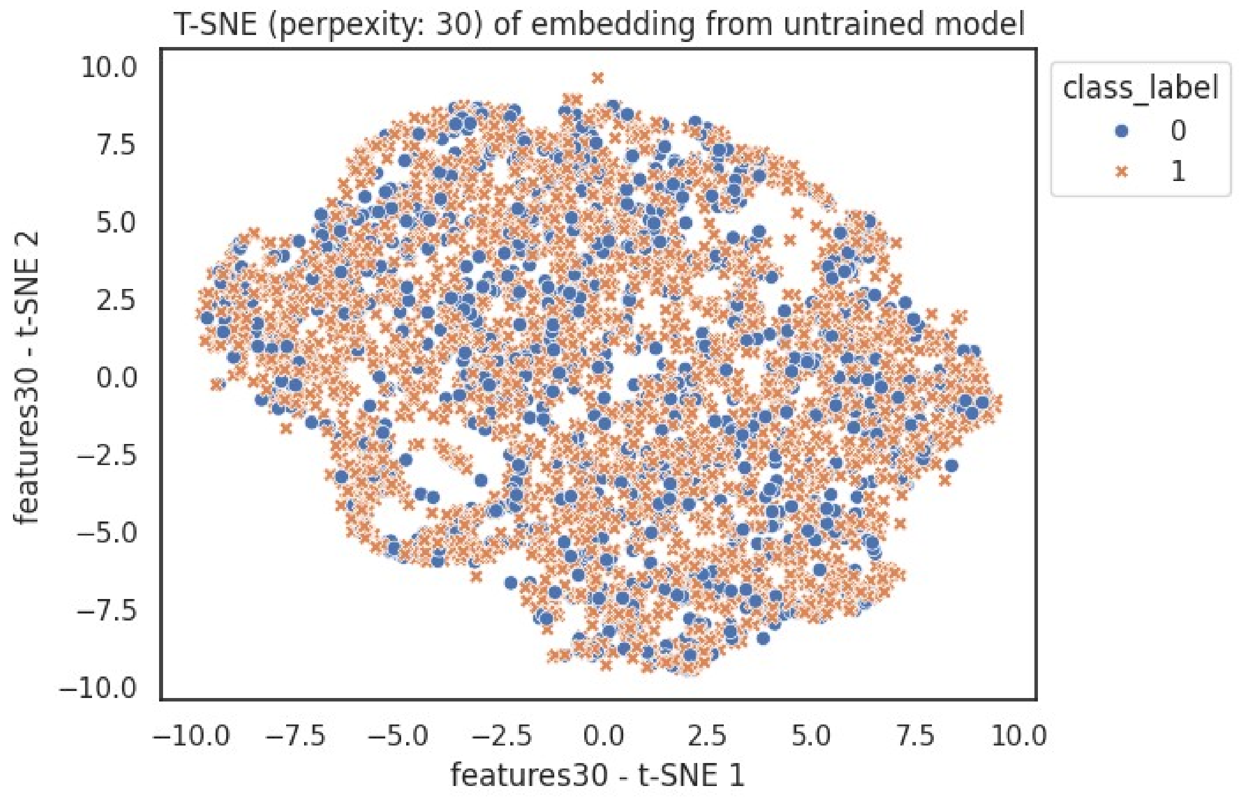
```
# generate untrained model
new_model = mae_vit_toy_patch4_dec256d4b()
new_model.eval()

(7, 7)

MaskedAutoencoderViT(
  (patch_embed): PatchEmbed(
    (proj): Conv2d(1, 384, kernel_size=(4, 4), stride=(4, 4))
    (norm): Identity()
  )
  (blocks): ModuleList(
    (0-5): 6 x Block(
      (norm1): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=384, out_features=1152, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (proj): Linear(in_features=384, out_features=384, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (ls1): Identity()
      (drop_path1): Identity()
      (norm2): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=384, out_features=1536, bias=True)
        (act): GELU(approximate='none')
        (drop1): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (fc2): Linear(in_features=1536, out_features=384, bias=True)
        (drop2): Dropout(p=0.0, inplace=False)
      )
      (ls2): Identity()
      (drop_path2): Identity()
    )
  )
  (norm): LayerNorm((384,), eps=1e-05, elementwise_affine=True)
  (decoder_embed): Linear(in_features=384, out_features=256,
bias=True)
  (decoder_blocks): ModuleList(
    (0-3): 4 x Block(
      (norm1): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (attn): Attention(
        (qkv): Linear(in_features=256, out_features=768, bias=True)
        (q_norm): Identity()
        (k_norm): Identity()
        (attn_drop): Dropout(p=0.0, inplace=False)
        (proj): Linear(in_features=256, out_features=256, bias=True)
        (proj_drop): Dropout(p=0.0, inplace=False)
      )
      (ls1): Identity()
      (drop_path1): Identity()
      (norm2): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
      (mlp): Mlp(
        (fc1): Linear(in_features=256, out_features=1024, bias=True)
        (act): GELU(approximate='none')
        (drop1): Dropout(p=0.0, inplace=False)
        (norm): Identity()
        (fc2): Linear(in_features=1024, out_features=256, bias=True)
        (drop2): Dropout(p=0.0, inplace=False)
      )
      (ls2): Identity()
      (drop_path2): Identity()
    )
  )
  (decoder_norm): LayerNorm((256,), eps=1e-05,
elementwise_affine=True)
  (decoder_pred): Linear(in_features=256, out_features=16, bias=True)
)

# extract embeddings from untrained model
embeddings = []
```

```
for i, l in test_loader:
  temp = i.to(device=device)
  embeddings.append(model.get_class_embeddings(temp).cpu().detach())
  del temp
  torch.cuda.empty_cache()
embeddings = torch.cat(embeddings).numpy()

# create dataframe to store data to be visualized
df_untrained = pd.DataFrame(labels, columns=['class_label'])

# use PCA to reduce dimension from 384 to 50
pca = PCA(n_components=50)
embeddings_pca = pca.fit_transform(embeddings)
print(embeddings_pca.shape)
```

```
(4708, 50)
```

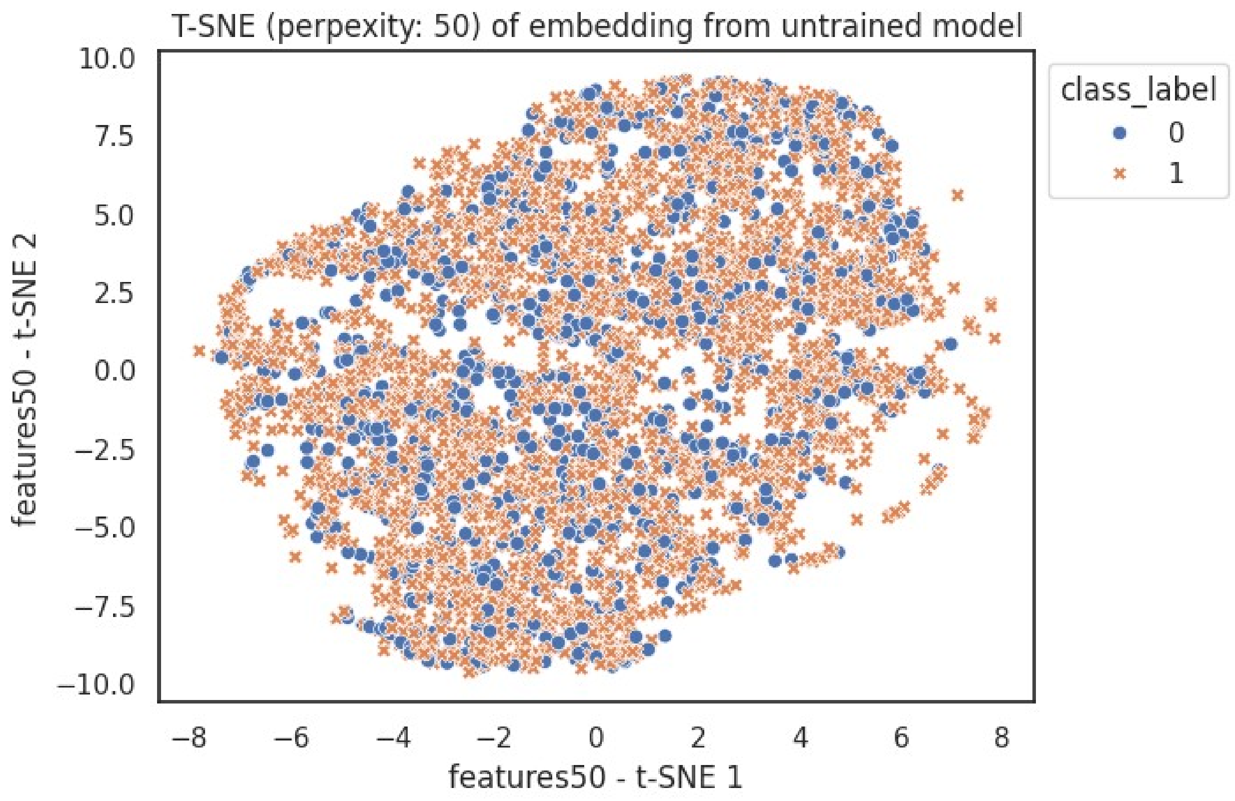
```
# gathering t-sne data
embeddings_tsne = TSNE(perplexity=30, init="pca",
max_iter=300).fit_transform(embeddings_pca)
df_untrained['features30 - t-SNE 1'] = embeddings_tsne[:,0]
df_untrained['features30 - t-SNE 2'] = embeddings_tsne[:,1]

# start plotting tsne
sns.set_theme(style="white")
ax = sns.scatterplot(data=df_untrained, x='features30 - t-SNE 1',
y='features30 - t-SNE 2', hue='class_label', style="class_label")
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
ax.set_title("T-SNE (perpexity: 30) of embedding from untrained
model")
```

```
Text(0.5, 1.0, 'T-SNE (perpexity: 30) of embedding from untrained
model')
```



```
# gathering t-sne data
embeddings_tsne = TSNE(perplexity=50, init="random",
max_iter=300).fit_transform(embeddings_pca)
df_untrained['features50 - t-SNE 1'] = embeddings_tsne[:,0]
df_untrained['features50 - t-SNE 2'] = embeddings_tsne[:,1]

# start plotting tsne
sns.set_theme(style="white")
ax = sns.scatterplot(data=df_untrained, x='features50 - t-SNE 1',
y='features50 - t-SNE 2', hue='class_label', style="class_label")
sns.move_legend(ax, "upper left", bbox_to_anchor=(1, 1))
ax.set_title("T-SNE (perpexity: 50) of embedding from untrained
model")
```

```
Text(0.5, 1.0, 'T-SNE (perpexity: 50) of embedding from untrained
model')
```



Summarise your observations...

The embedding produced by the trained model shows much clearer clustering for the 2 labels comparing to the untrained model. To be more specific, the trained model's visualization shows that most embeddings are significantly further from points of different label than those of the same class, proving that the trained model successfully extracted the (2 kinds of) special properties of images with pneumonia that do not present in normal in the latent space. This helps it to reconstruct the missing patches according to whether the few given patches shown signs of pneumonia, as while all images share similar structure, the pneumonia ones is significantly different.