Test Results

Q2.2a DDPM Schedule: Score 1/1

>>> Passed all 1 test cases!

Q1.1a Implement VAE: Score 1/1

>>> Passed all 1 test cases!

Repository Information

CID: ab6124

Repo URL: https://gitlab.doc.ic.ac.uk/lab2425_spring/DL_CW_2_ab6124

Submission SHA: 14e18f92060306597dcdd2a8561b96640180482b

Comments

In [ ]:
```
# Necessary Hyperparameters
num_epochs = 16
learning_rate = 0.01 # 0.03
batch_size = 128
latent_dim = 20 # Choose a value for the size of the latent space

# Additional Hyperparameters
beta = 5
lr_gamma = 0.1
lr_step_size = 7

# Mean and std of the training data
mean = 0.1307
std = 0.3081

# (Optionally) Modify transformations on input
transform = transforms.Compose([
    transforms.ToTensor(),
    # Don't normalise since we will use the Binary Cross-Entropy loss
    # as the data is nearly binary (see histogram plot in "Defining a Loss" section)
    # transforms.Normalize(mean, std),

    # We could add some data augmentation here (eg random rotations)
    # but to keep it simple we won't
])

# (Optionally) Modify the network's output for visualizing your images
def denorm(x):
    # transforms.Normalize(-mean/std, 1/std)
    return x
```

Data loading

In [8]:
```
train_dat = datasets.MNIST(
    data_path, train=True, download=True, transform=transform
)
test_dat = datasets.MNIST(data_path, train=False, transform=transform)

loader_train = DataLoader(train_dat, batch_size, shuffle=True)
loader_test = DataLoader(test_dat, batch_size, shuffle=False)

# Don't change
sample_inputs, _ = next(iter(loader_test))
fixed_input = sample_inputs[:32, :, :, :]
save_image(fixed_input, content_path/'CW_VAE/image_original.png')
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found
```

```
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/ra

100%|     | 9.91M/9.91M [00:00<00:00, 41.9MB/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/ra

100%|     | 28.9k/28.9k [00:00<00:00, 1.11MB/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

100%|     | 1.65M/1.65M [00:00<00:00, 10.7MB/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 404: Not Found

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

100%|     | 4.54k/4.54k [00:00<00:00, 7.39MB/s]
```

```
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

In [9]:
```
# Check input shape for VAE
# (input X , target y)
print("Input Image Shape: ", train_dat[0][0].shape)
print("Label: ", train_dat[0][1])
```

```
Input Image Shape:  torch.Size([1, 28, 28])
Label:  5
```

1 STUDENT NOTE

Please refer to the ab6124 GitLab Repository for additional Images showing the performances (image reconstructions and generations for task1 (*progress_images/VAE*) and task 2 (*progress_images/DDPM*)).

Model Definition

Fig.1 - VAE Diagram (with a Guassian prior), taken from 1.

A Variational Autoencoder (VAE) consists of several key components: * Encoder: Compresses input data into a lower dimensional latent space * Reparametrization: Samples from the latent distribution in a differentiable way * Decoder: Reconstructs the original input from the latent representation

You will need to define: * The hyperparameters - Control model capacity and training behavior * The constructor - Initialize model architecture and layers * encode - Maps input to mean and log variance of latent distribution * reparametrize - Samples latent vectors using reparametrization trick * decode - Maps latent vectors back to input space * forward - Combines all components into full forward pass

Requirements: * This model MUST NOT have more than 1M parameters

Hints: - It is common practice to encode the log of the variance, rather than the variance (This improves numerical stability during training) - You might try using BatchNorm (This can help stabilize training and reduce internal covariate shift)

In [ ]:
```
# *CODE FOR PART 1.1a IN THIS CELL*
```

```python
# Define any deep CNN/MLP network class here if needed

class VAE(nn.Module):
    # ADDED PARAMETER FOR ACTIVATION FUNCTION OF OUTPUT LAYER
    def __init__(self, channels, latent_dim, output_activation='sigmoid'):
        super(VAE, self).__init__()
        ########################################################################
        #                          ** START OF YOUR CODE **
        ########################################################################
        # encoder input channels = decoder output channels
        self.channels = channels
        self.latent_dim = latent_dim # mu/logvar output size
        self.output_activation = output_activation

        # encoder output channels = decoder input channels
        self.final_channels = 64

        # input images with shape [channels, 28, 28]
        self.encoder = nn.Sequential(
            # 28x28 -> 14x14
            nn.Conv2d(in_channels=channels, out_channels=8,
                      kernel_size=3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(inplace=True),

            # 14x14 -> 7x7
            nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3,
                      stride=2, padding=1, bias=False),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(inplace=True),

            # 7x7 -> 7x7
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=1,
                      stride=1, padding=0, bias=False),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(inplace=True),

            # 7x7 -> 4x4
            nn.Conv2d(in_channels=32, out_channels=self.final_channels,
                      kernel_size=3, stride=2, padding=1, bias=False),
            nn.BatchNorm2d(self.final_channels),
            nn.LeakyReLU(inplace=True))

        self.mu = nn.Linear(self.final_channels*4*4, self.latent_dim)
        self.logvar = nn.Linear(self.final_channels*4*4, self.latent_dim)

        # reverse steps of encoder: inputs shape [self.final_channels, 4, 4]
        self.resizer = nn.Linear(self.latent_dim, self.final_channels*4*4)
        self.decoder = nn.Sequential(
            # 4x4 -> 7x7
            nn.ConvTranspose2d(in_channels=self.final_channels, out_channels=32,
                               kernel_size=3, stride=2, padding=1,
                               output_padding=0),
            nn.BatchNorm2d(32),
```

```python
            nn.LeakyReLU(inplace=True),

            # 7x7 -> 14x14
            nn.ConvTranspose2d(in_channels=32, out_channels=16, kernel_size=3,
                               stride=2, padding=1, output_padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(inplace=True),

            # 14x14 -> 14x14
            nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=1,
                               stride=1, padding=0, output_padding=0),
            nn.BatchNorm2d(8),
            nn.LeakyReLU(inplace=True),

            # 14x14 -> 28x28
            nn.ConvTranspose2d(in_channels=8, out_channels=channels,
                               kernel_size=2, stride=2, padding=0,
                               output_padding=0))

        ########################################################################
        #                        ** END OF YOUR CODE **
        ########################################################################

    def encode(self, x):
        ########################################################################
        #                        ** START OF YOUR CODE **
        ########################################################################
        out = self.encoder(x)
        # print("Encoder output shape: ", out.shape)
        # flatten for the linear laeyr
        out = out.view(out.size(0), -1)
        # print("Flattened output shape: ", out.shape)

        mu = self.mu(out)
        # print("mu shape: ", mu.shape)
        logvar = self.logvar(out)
        # print("log variance shape: ", logvar.shape)

        return mu, logvar
        ########################################################################
        #                        ** END OF YOUR CODE **
        ########################################################################

    def reparametrize(self, mu, logvar):
        ########################################################################
        #                        ** START OF YOUR CODE **
        ########################################################################
        std = logvar.exp()
        # Gaussian Noise N(0,1)
        noise = torch.normal(mean=torch.zeros_like(mu), std=torch.ones_like(mu))
        # print("Noise shape: ", noise.shape)
        # z = mu + sigma * noise
        z = mu + (std * noise)
        # print("z shape: ", z.shape)
```

```
            return z
            ######################################################################
            #                        ** END OF YOUR CODE **
            ######################################################################

        def decode(self, z):
            ######################################################################
            #                        ** START OF YOUR CODE **
            ######################################################################
            out = self.resizer(z)
            # print("Decoder input shape: ", out.shape)
            out = out.view(-1, self.final_channels, 4, 4) # = encoder output shape
            # print("Reshaped decoder input shape: ", out.shape)
            z = self.decoder(out)
            if self.output_activation == 'sigmoid':
                z = F.sigmoid(z) # for MINST data
            else:
                z = F.tanh(z) # for HOT DOG data
            # print("Decoder output shape: ", z.shape)
            ######################################################################
            #                        ** END OF YOUR CODE **
            ######################################################################
            return z

        def forward(self, x):
            ######################################################################
            #                        ** START OF YOUR CODE **
            ######################################################################
            mu, logvar = self.encode(x)
            # print("mu shape: ", mu.shape)
            # print("logvar shape: ", logvar.shape)
            z = self.reparametrize(mu, logvar)
            # print("reparametrized shape: ", z.shape)
            out = self.decode(z)
            # print("output shape: ", out.shape)
            ######################################################################
            #                        ** END OF YOUR CODE **
            ######################################################################
            return out, mu, logvar

    model = VAE(1, latent_dim).to(device)
    params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print("Total number of parameters is: {}".format(params))
    print(model)
    # optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```
Total number of parameters is: 106281
VAE(
  (encoder): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (2): LeakyReLU(negative_slope=0.01, inplace=True)
    (3): Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.01, inplace=True)
    (6): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.01, inplace=True)
    (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): LeakyReLU(negative_slope=0.01, inplace=True)
  )
  (mu): Linear(in_features=1024, out_features=20, bias=True)
  (logvar): Linear(in_features=1024, out_features=20, bias=True)
  (resizer): Linear(in_features=20, out_features=1024, bias=True)
  (decoder): Sequential(
    (0): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): LeakyReLU(negative_slope=0.01, inplace=True)
    (3): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), output_padding=(1, 
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): LeakyReLU(negative_slope=0.01, inplace=True)
    (6): ConvTranspose2d(16, 8, kernel_size=(1, 1), stride=(1, 1))
    (7): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): LeakyReLU(negative_slope=0.01, inplace=True)
    (9): ConvTranspose2d(8, 1, kernel_size=(2, 2), stride=(2, 2))
  )
)
```

In [11]:
```python
in_size = torch.ones([1,1,28,28]).to(device)
out, mu, logvar = model(in_size)

# Shape Check
assert in_size.shape == out.shape, "Input shape != Output shape!"
print("Output shape: ", out.shape)
print("mu shape: ", mu.shape)
print("logvar shape: ", logvar.shape)
```

```
Output shape:  torch.Size([1, 1, 28, 28])
mu shape:  torch.Size([1, 20])
logvar shape:  torch.Size([1, 20])
```

In [12]:
```python
# replaces grader.check
assert sum(p.numel() for p in model.parameters() if p.requires_grad) < 1000000\
        ,"Model should have less than 1M parameters"
```

In [ ]:
```
import torch.nn.functional as F

# *CODE FOR PART 1.1b IN THIS CELL*

def loss_function_VAE(recon_x, x, mu, logvar, beta):
        ######################################################################
        #                        ** START OF YOUR CODE **
        ######################################################################
        # Assuming prior Bernoulli Distribution (see section below on
        # Data Exploration and choice of loss), we can use BCE
        NLL = F.binary_cross_entropy(input=recon_x, target=x, reduction='sum')

        # KL Divergence
        KLD = torch.mean((-0.5) * beta *
                         torch.sum(1 + logvar - logvar.exp() - mu.pow(2),
                          dim=1), dim=0)

        return NLL + KLD, NLL, KLD
        ######################################################################
        #                        ** END OF YOUR CODE **
        ######################################################################

########################### ** ADDED CODE ** ###########################
# change from pre-set hyperparameters
learning_rate = 0.01
batch_size = 128
latent_dim = 5 # reduced from 20
beta = 5 #3 #2
num_epochs = 15

# store ALL losses for plotting
train_losses = {'loss':[], 'NLL_loss': [], 'KLD_loss': []}
test_losses = {'loss':[], 'NLL_loss': [], 'KLD_loss': []}

def run_epoch(model, optimizer, dataloader, device, beta, train):
    """
    Run an epoch of training or testing for the whole dataset.
    Store the loss, NLL loss and KLD loss for each batch.
    Return the average loss, NLL loss and KLD loss for that epoch.
    """
    data_size = len(dataloader.dataset)
    total_loss, total_NLL_loss, total_KLD_loss = 0, 0, 0

    if train:
        model.train()
    else:
        model.eval()

    for data, _ in dataloader:
        data = data.to(device) # load inputs to device to match model

        recon_x, mu, logvar = model(data)
        loss, NLL_loss, KLD_loss = loss_function_VAE(recon_x,
                                                     data,
```

```
                                                     mu,
                                                     logvar,
                                                     beta)
        total_loss += loss
        total_NLL_loss += NLL_loss
        total_KLD_loss += KLD_loss

        if train:
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    losses = total_loss/data_size
    NLL_losses = total_NLL_loss/data_size
    KLD_losses = total_KLD_loss/data_size

    return losses, NLL_losses, KLD_losses

########################################################################

model = VAE(1, latent_dim).to(device)
model_path = f"./VAE_model_{latent_dim}.pth"

# optional -- do this if you want to continue training on a previous training run
load_checkpoint = False
if os.path.exists(model_path) and load_checkpoint:
    print("loading existing model...")
    model.load_state_dict(torch.load(model_path))

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                            step_size =
                                            lr_step_size,
                                            gamma = lr_gamma)

for epoch in range(num_epochs):
    ######################################################################
    #                        ** START OF YOUR CODE **
    ######################################################################
    # 1. Train Run
    losses, NLL_losses, KLD_losses = run_epoch(model=model,
                                               optimizer=optimizer,
                                               dataloader=loader_train,
                                               device=device,
                                               beta=beta, train=True)
    train_losses['loss'].append(losses)
    train_losses['NLL_loss'].append(NLL_losses)
    train_losses['KLD_loss'].append(KLD_losses)

    # 2. Evaluation Run
    with torch.no_grad():
        losses, NLL_losses, KLD_losses = run_epoch(model=model,
                                                   optimizer=optimizer,
                                                   dataloader=loader_test,
```

```
                                                   device=device,
                                                   beta=beta,
                                                   train=False)
            test_losses['loss'].append(losses)
            test_losses['NLL_loss'].append(NLL_losses)
            test_losses['KLD_loss'].append(KLD_losses)

            last_train_loss = (train_losses['loss'][-1],
                               train_losses['NLL_loss'][-1],
                               train_losses['KLD_loss'][-1])
            last_test_loss = test_losses['loss'][-1]

            lr = optimizer.param_groups[0]['lr']

            print((f'Epoch: {epoch+1}\n'
                   f'   train loss = {last_train_loss[0]:.2f}'
                   f', NLL loss = {last_train_loss[1]:.2f}'
                   f', KLD loss = {last_train_loss[2]:.2f}'))
            print(f'   test loss = {last_test_loss:.2f}, lr = {lr:.5f}')

            scheduler.step()
            ########################################################################
            #                        ** END OF YOUR CODE **
            ########################################################################
            # save the model
            if epoch % 5 == 0:
                with torch.no_grad():
                    torch.save(model.state_dict(), model_path)

        torch.save(model.state_dict(), 'VAE_model.pth')
```

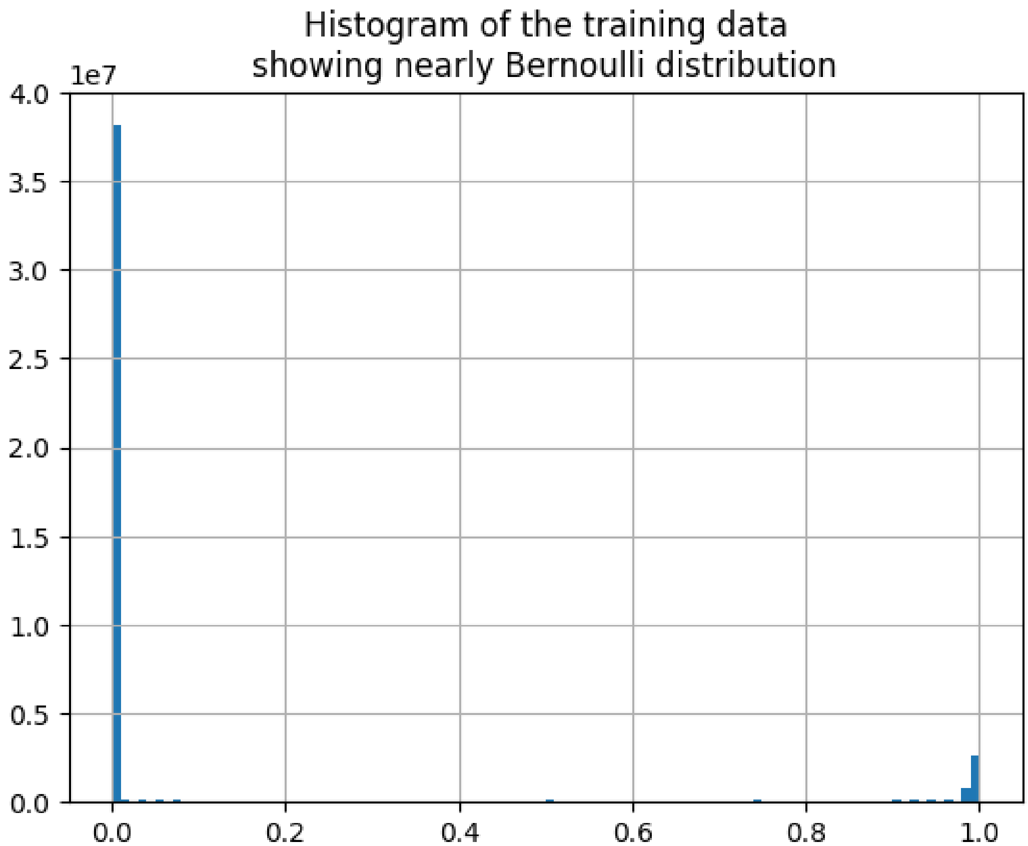
```
Epoch: 1
   train loss = 157.34, NLL loss = 156.62, KLD loss = 0.72
   test loss = 122.67, lr = 0.01000
Epoch: 2
   train loss = 118.46, NLL loss = 117.93, KLD loss = 0.53
   test loss = 114.96, lr = 0.01000
Epoch: 3
   train loss = 114.64, NLL loss = 114.13, KLD loss = 0.52
   test loss = 112.62, lr = 0.01000
Epoch: 4
   train loss = 113.00, NLL loss = 112.48, KLD loss = 0.51
   test loss = 111.63, lr = 0.01000
Epoch: 5
   train loss = 111.93, NLL loss = 111.42, KLD loss = 0.51
   test loss = 111.53, lr = 0.01000
Epoch: 6
   train loss = 111.22, NLL loss = 110.71, KLD loss = 0.51
   test loss = 110.47, lr = 0.01000
Epoch: 7
   train loss = 110.66, NLL loss = 110.15, KLD loss = 0.51
   test loss = 110.04, lr = 0.01000
Epoch: 8
```

```
   train loss = 107.98, NLL loss = 107.47, KLD loss = 0.51
   test loss = 107.46, lr = 0.00100
Epoch: 9
   train loss = 107.58, NLL loss = 107.07, KLD loss = 0.50
   test loss = 107.39, lr = 0.00100
Epoch: 10
   train loss = 107.47, NLL loss = 106.97, KLD loss = 0.50
   test loss = 107.33, lr = 0.00100
Epoch: 11
   train loss = 107.36, NLL loss = 106.87, KLD loss = 0.49
   test loss = 107.17, lr = 0.00100
Epoch: 12
   train loss = 107.26, NLL loss = 106.78, KLD loss = 0.49
   test loss = 107.13, lr = 0.00100
Epoch: 13
   train loss = 107.14, NLL loss = 106.66, KLD loss = 0.49
   test loss = 107.02, lr = 0.00100
Epoch: 14
   train loss = 107.09, NLL loss = 106.60, KLD loss = 0.48
   test loss = 107.04, lr = 0.00100
Epoch: 15
   train loss = 106.63, NLL loss = 106.15, KLD loss = 0.48
   test loss = 106.65, lr = 0.00010
```

```
In [ ]: # Any code for your explanation here

        import math
        import matplotlib.pyplot as plt

        ######################################################################
        #                        ** START OF YOUR CODE **
        ######################################################################

        # TODO
        from torch.distributions import Normal
        from torch.distributions import Bernoulli

        # Set-up
        x, _ = next(iter(loader_test))
        x = x.to(device)[0:32, :, :, :]

        recon_x, _, _ = model(x)

        # a. Reconstruction loss assuming Gaussian Distribution (with variance = 1)
        # Log probability of Normal distribution
        loss1 = -torch.mean(Normal(loc=recon_x, scale=1).log_prob(x))
        # Mean Squared Error
        loss2 = F.mse_loss(input=recon_x, target=x, reduction='mean')

        # source: https://towardsdatascience.com/mse-is-cross-entropy-at-heart-maximum-likelihood-estim
        loss3 = ((math.log(1)) +
                (0.5 * math.log(2 * math.pi)) +
                (0.5 * loss2))

        # Cross entropy
        loss4 = F.cross_entropy(recon_x, x)


        # b. Reconstruction loss assuming Bernoulli Distribution
        # Turn inputs and targets into binary (0,1)
        binary_x = (x > 0.5).float()
        # binary_recon_x = (recon_x > 0.5).float()
        binary_recon_x = torch.sigmoid(recon_x)

        # Log probability of Bernoulli distribution
        loss4 = -torch.mean(Bernoulli(binary_recon_x).log_prob(binary_x))

        # BCE
        loss5 = F.binary_cross_entropy(binary_recon_x, binary_x)


        print('Gaussian distribution')
        print(f'Normal log prob:                  {loss1:.3f}')
        print(f'MSE:                              {loss2:.3f}')
        print(f'MSE + constant terms = log prob: {loss3:.3f}')
        # print('MSE + constant terms = log prob?', (loss1==loss3).item()) # Check
        print()
        print('Multinulli distribution')
```

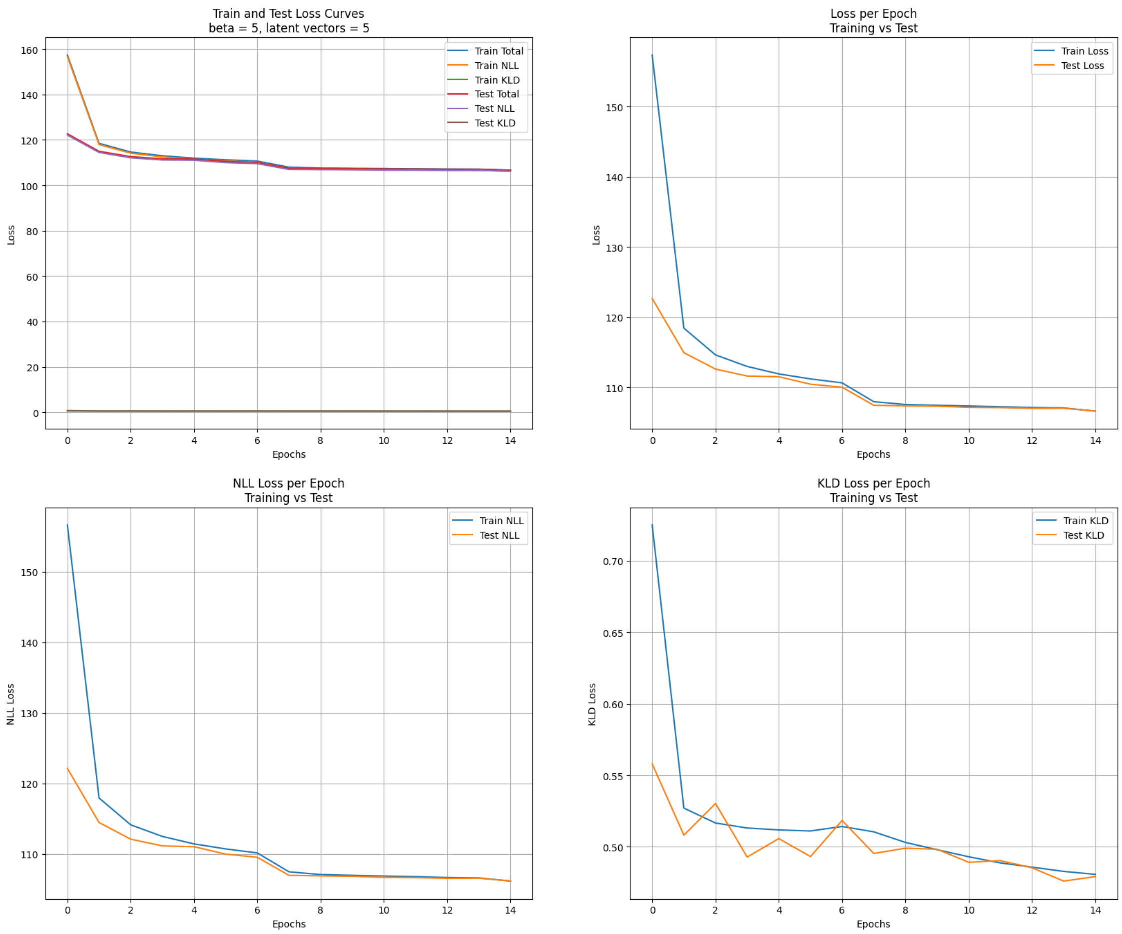
```python
print(f'Cross entropy:                    {loss4:.3f}')
print()
print('Bernoulli distribution')
print(f'Bernoulli log prob:               {loss4:.3f}')
print(f'BCE:                              {loss5:.3f}')


# c. histogram of input data domain
all_train_dataloader = DataLoader(train_dat, len(train_dat))
data = next(iter(all_train_dataloader))
data = data[0].flatten().numpy()
plt.hist(data, bins=100)
plt.grid()
plt.title('Histogram of the training data\nshowing nearly Bernoulli distribution')
plt.show()
########################################################################
#                       ** END OF YOUR CODE **
########################################################################
```

```
Gaussian distribution
Normal log prob:                 0.930
MSE:                             0.022
MSE + constant terms = log prob: 0.930

Multinulli distribution
Cross entropy:                   0.680

Bernoulli distribution
Bernoulli log prob:              0.680
BCE:                             0.680
```


Histogram of the training data
showing nearly Bernoulli distribution
1e7
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.2
0.4
0.6
0.8
1.0

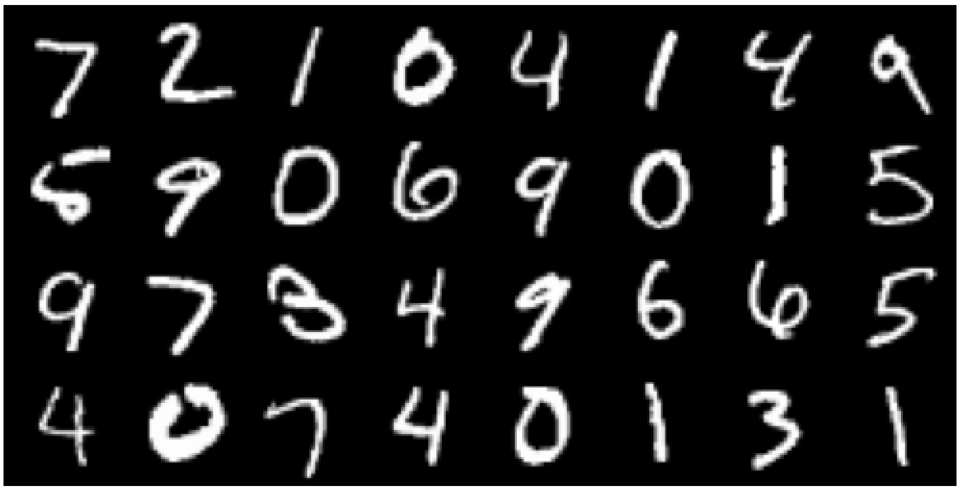
In [ ]:
```
# *CODE FOR PART 1.2a IN THIS CELL*
#######################################################################
#                        ** START OF YOUR CODE **
#######################################################################
import matplotlib.pyplot as plt

train_losses_ = {}
test_losses_ = {}

# Train losses
train_losses_['loss'] = torch.tensor(train_losses['loss'], device='cpu')
train_losses_['NLL_loss'] = torch.tensor(train_losses['NLL_loss'], device='cpu')
train_losses_['KLD_loss'] = torch.tensor(train_losses['KLD_loss'], device='cpu')
# Test Losses
test_losses_['loss'] = torch.tensor(test_losses['loss'], device='cpu')
test_losses_['NLL_loss'] = torch.tensor(test_losses['NLL_loss'], device='cpu')
test_losses_['KLD_loss'] = torch.tensor(test_losses['KLD_loss'], device='cpu')


# 1. Plot ALL losses together
plt.figure(figsize=(20,25))
plt.subplot(3,2,1)
plt.plot(train_losses_['loss'], label='Train Total')
plt.plot(train_losses_['NLL_loss'], label='Train NLL')
plt.plot(train_losses_['KLD_loss'], label='Train KLD')
plt.plot(test_losses_['loss'], label='Test Total')
plt.plot(test_losses_['NLL_loss'], label='Test NLL')
plt.plot(test_losses_['KLD_loss'], label='Test KLD')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title(f'Train and Test Loss Curves\nbeta = {beta}, latent vectors = {latent_dim}')
plt.legend()
plt.grid()

# 2. Plot train vs test loss
plt.subplot(3,2,2)
plt.plot(train_losses_['loss'], label='Train Loss')
plt.plot(test_losses_['loss'], label='Test Loss')
plt.title('Loss per Epoch\nTraining vs Test')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()

# 3. Plot train vs test NLL loss
plt.subplot(3,2,3)
plt.plot(train_losses_['NLL_loss'], label='Train NLL')
plt.plot(test_losses_['NLL_loss'], label='Test NLL')
plt.title('NLL Loss per Epoch\nTraining vs Test')
plt.xlabel('Epochs')
plt.ylabel('NLL Loss')
plt.legend()
plt.grid()
```

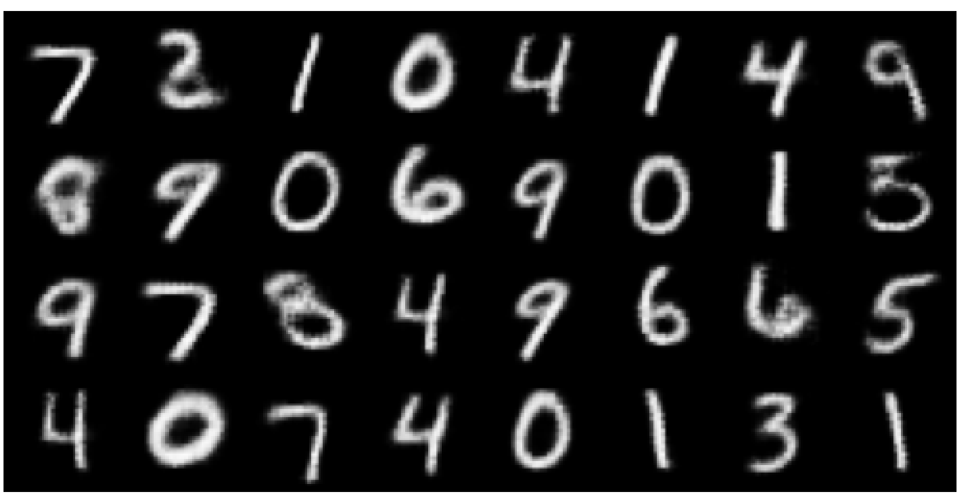
```
# 4. Plot train vs test KLD loss
plt.subplot(3,2,4)
plt.plot(train_losses_['KLD_loss'], label='Train KLD')
plt.plot(test_losses_['KLD_loss'], label='Test KLD')
plt.title('KLD Loss per Epoch\nTraining vs Test')
plt.xlabel('Epochs')
plt.ylabel('KLD Loss')
plt.legend()
plt.grid()

plt.show()
######################################################################
#                             ** END OF YOUR CODE **
######################################################################
```



In [29]:
```
# *CODE FOR PART 1.2b IN THIS CELL*

# load the model
print('Input images')
print('-'*50)

sample_inputs, _ = next(iter(loader_test))
fixed_input = sample_inputs[0:32, :, :, :]
# visualize the original images of the last batch of the test set
img = make_grid(denorm(fixed_input), nrow=8, padding=2, normalize=False,
                scale_each=False, pad_value=0)
plt.figure()
show(img)

print('Reconstructed images')
print('-'*50)
model.eval()
with torch.no_grad():
    # visualize the reconstructed images of the last batch of test set

    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################
    recon_batch, _, _ = model(fixed_input.to(device))
    #######################################################################
    #                       ** END OF YOUR CODE **
    #######################################################################

    recon_batch = recon_batch.cpu()
    recon_batch = make_grid(denorm(recon_batch), nrow=8, padding=2, normalize=False,
                            scale_each=False, pad_value=0)
    plt.figure()
    show(recon_batch)

print('Generated Images')
print('-'*50)
model.eval()
n_samples = 256
z = torch.randn(n_samples,latent_dim).to(device)
with torch.no_grad():
    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################
    samples = model.decode(z.to(device))
    #######################################################################
    #                       ** END OF YOUR CODE **
    #######################################################################

    samples = samples.cpu()
    samples = make_grid(denorm(samples), nrow=16, padding=2, normalize=False,
                            scale_each=False, pad_value=0)
    plt.figure(figsize = (8,8))
    show(samples)
```

```
Input images
--------------------------------------------------
Reconstructed images
--------------------------------------------------
Generated Images
--------------------------------------------------
```





1.0.1 Discussion

Provide a brief analysis of your loss curves and reconstructions: * What do you observe in the behaviour of the log-likelihood loss and the KL loss (increasing/decreasing)? * Can you intuitively explain if this behaviour is desirable? * What is posterior collapse and did you observe it during training (i.e. when the KL is too small during the early stages of training)? * If yes, how did you mitigate it? How did this phenomenon reflect on your output samples? * If no, why do you think that is?

1.0.2 Your Answer (3 points)

In the final VAE model, all losses (both log likelihood AND KL) descrease during training. This behaviour is positive and desireable, as it indicates that the model is effectively learning, throughout training, how to reconstruct the input data whilst better approximating the underlying data distribution.

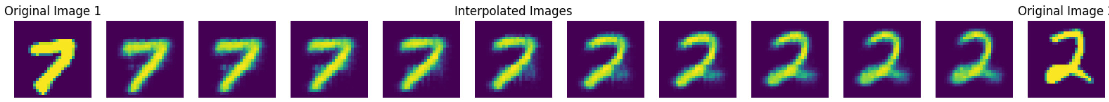
The KL divergence decreases both during training and testing, indicating that the model is correctly learning a distribution that better approximates the underlying data distribution. However, the very low values of KL loss may be and indicator of the presence of posterior collapse. This somewhat clashes with the quality of the generated images, as they don't seem to lack realism and seem to be somewhat close to the input data. However, the model does show difficulties in generating very similar digits (eg. 4 and 9, 8 and 3). Still, to mitigate this, I tried to fine-tune the value of beta, trying higher values (up to 10). However, the changes in loss values, and more importantly, in the quality of the reconstructed images, were not significant. Thus, I concluded that the low kl values could be due to the chosen smaller latent space dimension (see point below).

Through various model architechtures and training strategies (latent space dimensions, betas and learning rates experiemnted), I found that the best approach to balance out reconstruction accuracy with generation capabilities was to reduce the latent space (to 5) and keeping a relatively high beta (of 5). During the tuning process, in fact, I noticed many instances where the KL loss kept stably increasing (even significantly) during training. In these instances, as expected, the models seemed to be overfitting on the training data, giving more accurate reconstructions (even 100% equivalent to the original ones) but poorer and more noisy generations (as the model prioritized input reconstruction over learning of the feature distribution) (see examples in the `progress_images/VAE/MINST` folder).

In [32]:
```
# *CODE FOR PART 1.3a IN THIS CELL
######################################################################
#                        ** START OF YOUR CODE **
######################################################################
from sklearn.manifold import TSNE
test_dataloader = DataLoader(test_dat, 10000, shuffle=False) # 100

model.eval()
with torch.no_grad():
    for data, y in test_dataloader:
        mu, logvar = model.encode(data.to(device))
        z = model.reparametrize(mu, logvar)

# Perform TSNE
print('Performing TSNE')
z_embedded_5 = TSNE(n_components=2, perplexity = 5).fit_transform(z.cpu())
z_embedded_30 = TSNE(n_components=2, perplexity = 30).fit_transform(z.cpu())
z_embedded_50 = TSNE(n_components=2, perplexity = 50).fit_transform(z.cpu())

######################################################################
#                        ** END OF YOUR CODE **
######################################################################
```

```
Performing TSNE
```

In [34]:
```
# CODE FOR PART 1.3b IN THIS CELL
#######################################################################
#                        ** START OF YOUR CODE **
#######################################################################
import random

def get_random_image(dataset):
    """
    Get a random image from the dataset.
    Formats it to be of size [1, C, H, W]
    """
    idx = random.randint(0, len(dataset)-1)
    return dataset[idx][None,:,:,:]

num_steps = 10

test_images, _ = next(iter(loader_test))

image1 = get_random_image(test_images)
image2 = get_random_image(test_images)

interpolation_rate = torch.linspace(0, 1, num_steps)

model.eval()
with torch.no_grad():
    img_1_embedding, _ = model.encode(image1.to(device))
    img_2_embedding, _ = model.encode(image2.to(device))

    interpolated_embeddings = []
    for rate in interpolation_rate:
        interpolated_embeddings.append((img_1_embedding * (1 - rate)) + (img_2_embedding * rat
    interpolated_embeddings = torch.stack(interpolated_embeddings)

    interpolated_images = model.decode(interpolated_embeddings)

def imshow(img, ax):
    img = img.permute(1, 2, 0)
    ax.imshow(img.cpu().numpy())
    ax.axis('off')

fig, axes = plt.subplots(1, interpolated_images.size(0) + 2, figsize=(20, 5))
imshow(image1[0], axes[0])
axes[0].set_title('Original Image 1')
for i in range(interpolated_images.size(0)):
    imshow(interpolated_images[i], axes[i + 1])
axes[5].set_title('Interpolated Images')
imshow(image2[0], axes[-1])
axes[-1].set_title('Original Image 2')
plt.show()
#######################################################################
#                        ** END OF YOUR CODE **
#######################################################################
```


Original Image 1
Interpolated Images
Original Image 2

In [ ]:
```
# %%
#We need a bit more training as for MNIST but don't expect much
num_epochs = 60

def loss_function_VAE(recon_x, x, mu, logvar, beta, std=1.0):
    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################
    # Now assume Normal distribution with standard deviation = 1, i.e. use
    # MSE instead of BCE (see section Data Exploration and choice of loss)
    NLL = F.mse_loss(recon_x, x, reduction='sum')
    # KL Divergence
    KLD = torch.mean((-0.5) * beta *
                     torch.sum(1 + logvar - logvar.exp() - mu.pow(2),
                     dim=1), dim=0)
    return NLL + KLD, NLL, KLD
    #######################################################################
    #                       ** END OF YOUR CODE **
    #######################################################################

# use tanh activation for output layer to adapt to data rages of hotdog dataset
model = VAE(3, latent_dim, output_activation='tanh').to(device)
model_path = f"./VAE_hd_model_{latent_dim}.pth"

# optional -- do this if you want to continue training on a previous training run
load_checkpoint = False
if os.path.exists(model_path) and load_checkpoint:
    print("loading existing model...")
    model.load_state_dict(torch.load(model_path))

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
train_losses = {'loss':[], 'NLL_loss': [], 'KLD_loss': []}
test_losses = {'loss':[], 'NLL_loss': [], 'KLD_loss': []}
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                            step_size = lr_step_size,
                                            gamma = lr_gamma)
for epoch in range(num_epochs):
    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################
    # same helper function used for MNIST
    losses, NLL_losses, KLD_losses = run_epoch(model=model,
                                      optimizer=optimizer,
                                      dataloader=hotdogdata_loader_train,
                                      device=device, beta=beta, train=True)
    train_losses['loss'].append(losses)
    train_losses['NLL_loss'].append(NLL_losses)
    train_losses['KLD_loss'].append(KLD_losses)

    with torch.no_grad():
        # same helper function used for MNIST
        losses, NLL_losses, KLD_losses = run_epoch(model=model,
                                      optimizer=optimizer,
                                      dataloader=hotdogdata_loader_test,
```

```python
                                            device=device, beta=beta, train=False)
            test_losses['loss'].append(losses)
            test_losses['NLL_loss'].append(NLL_losses)
            test_losses['KLD_loss'].append(KLD_losses)

            last_train_loss = (train_losses['loss'][-1],
                               train_losses['NLL_loss'][-1],
                               train_losses['KLD_loss'][-1])
            last_test_loss = test_losses['loss'][-1]

            lr = optimizer.param_groups[0]['lr']

            print((f'Epoch: {epoch+1}\n'
                   f'   train loss = {last_train_loss[0]:.2f}'
                   f', NLL loss = {last_train_loss[1]:.2f}'
                   f', KLD loss = {last_train_loss[2]:.2f}'))
            print(f'   test loss = {last_test_loss:.2f}, lr = {lr:.5f}')

            scheduler.step()
            ########################################################################
            #                        ** END OF YOUR CODE **
            ########################################################################
            # save the model
            if epoch % 5 == 0:
                with torch.no_grad():
                    torch.save(model.state_dict(), model_path)

        torch.save(model.state_dict(), 'VAE_hd_model_jit.pth')
```

```
Epoch: 1
   train loss = 574.54, NLL loss = 569.02, KLD loss = 5.52
   test loss = 458.96, lr = 0.01000
Epoch: 2
   train loss = 393.77, NLL loss = 389.68, KLD loss = 4.09
   test loss = 373.09, lr = 0.01000
Epoch: 3
   train loss = 367.98, NLL loss = 364.61, KLD loss = 3.36
   test loss = 366.10, lr = 0.01000
Epoch: 4
   train loss = 355.96, NLL loss = 353.25, KLD loss = 2.71
   test loss = 352.76, lr = 0.01000
Epoch: 5
   train loss = 346.47, NLL loss = 344.24, KLD loss = 2.24
   test loss = 344.11, lr = 0.01000
Epoch: 6
   train loss = 340.25, NLL loss = 338.23, KLD loss = 2.02
   test loss = 338.16, lr = 0.01000
Epoch: 7
   train loss = 334.85, NLL loss = 332.93, KLD loss = 1.92
   test loss = 332.52, lr = 0.01000
Epoch: 8
   train loss = 327.88, NLL loss = 326.00, KLD loss = 1.88
   test loss = 324.65, lr = 0.00100
```

```
Epoch: 9
   train loss = 325.94, NLL loss = 324.12, KLD loss = 1.82
   test loss = 323.99, lr = 0.00100
Epoch: 10
   train loss = 325.23, NLL loss = 323.44, KLD loss = 1.79
   test loss = 323.88, lr = 0.00100
Epoch: 11
   train loss = 324.43, NLL loss = 322.65, KLD loss = 1.78
   test loss = 322.98, lr = 0.00100
Epoch: 12
   train loss = 323.81, NLL loss = 322.05, KLD loss = 1.75
   test loss = 322.35, lr = 0.00100
Epoch: 13
   train loss = 323.50, NLL loss = 321.77, KLD loss = 1.73
   test loss = 322.01, lr = 0.00100
Epoch: 14
   train loss = 322.78, NLL loss = 321.06, KLD loss = 1.72
   test loss = 321.75, lr = 0.00100
Epoch: 15
   train loss = 322.16, NLL loss = 320.46, KLD loss = 1.70
   test loss = 321.04, lr = 0.00010
Epoch: 16
   train loss = 321.71, NLL loss = 320.01, KLD loss = 1.70
   test loss = 320.98, lr = 0.00010
Epoch: 17
   train loss = 321.89, NLL loss = 320.19, KLD loss = 1.70
   test loss = 320.91, lr = 0.00010
Epoch: 18
   train loss = 321.78, NLL loss = 320.09, KLD loss = 1.70
   test loss = 320.94, lr = 0.00010
Epoch: 19
   train loss = 321.68, NLL loss = 319.98, KLD loss = 1.70
   test loss = 320.96, lr = 0.00010
Epoch: 20
   train loss = 321.41, NLL loss = 319.72, KLD loss = 1.70
   test loss = 320.81, lr = 0.00010
Epoch: 21
   train loss = 321.63, NLL loss = 319.94, KLD loss = 1.69
   test loss = 320.70, lr = 0.00010
Epoch: 22
   train loss = 321.55, NLL loss = 319.86, KLD loss = 1.69
   test loss = 320.78, lr = 0.00001
Epoch: 23
   train loss = 321.49, NLL loss = 319.80, KLD loss = 1.69
   test loss = 320.70, lr = 0.00001
Epoch: 24
   train loss = 321.29, NLL loss = 319.60, KLD loss = 1.69
   test loss = 320.69, lr = 0.00001
Epoch: 25
   train loss = 321.44, NLL loss = 319.75, KLD loss = 1.69
   test loss = 320.62, lr = 0.00001
Epoch: 26
   train loss = 321.44, NLL loss = 319.75, KLD loss = 1.69
   test loss = 320.65, lr = 0.00001
```

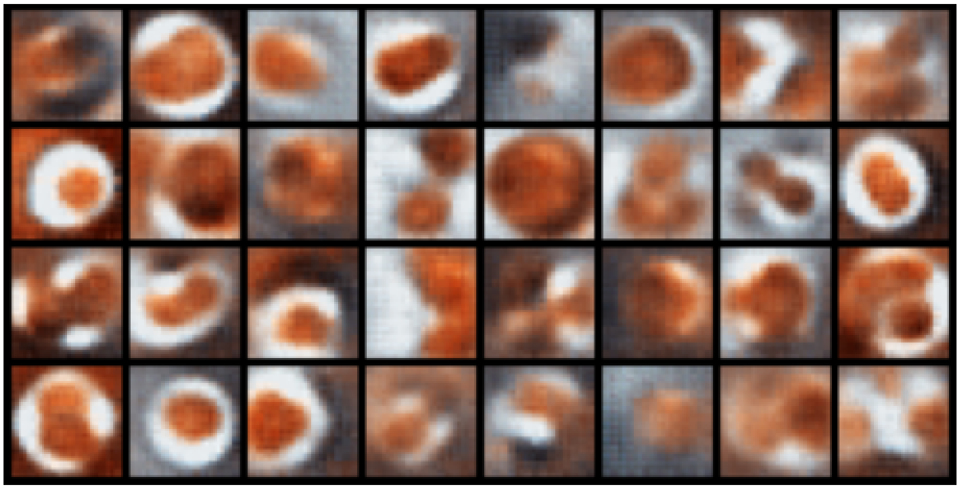
```
Epoch: 27
   train loss = 321.47, NLL loss = 319.78, KLD loss = 1.69
   test loss = 320.67, lr = 0.00001
Epoch: 28
   train loss = 321.26, NLL loss = 319.57, KLD loss = 1.69
   test loss = 320.65, lr = 0.00001
Epoch: 29
   train loss = 321.29, NLL loss = 319.60, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 30
   train loss = 321.36, NLL loss = 319.67, KLD loss = 1.69
   test loss = 320.62, lr = 0.00000
Epoch: 31
   train loss = 321.26, NLL loss = 319.57, KLD loss = 1.69
   test loss = 320.61, lr = 0.00000
Epoch: 32
   train loss = 321.24, NLL loss = 319.55, KLD loss = 1.69
   test loss = 320.67, lr = 0.00000
Epoch: 33
   train loss = 321.56, NLL loss = 319.87, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 34
   train loss = 321.38, NLL loss = 319.69, KLD loss = 1.69
   test loss = 320.69, lr = 0.00000
Epoch: 35
   train loss = 321.35, NLL loss = 319.66, KLD loss = 1.69
   test loss = 320.64, lr = 0.00000
Epoch: 36
   train loss = 321.26, NLL loss = 319.57, KLD loss = 1.69
   test loss = 320.68, lr = 0.00000
Epoch: 37
   train loss = 321.34, NLL loss = 319.65, KLD loss = 1.69
   test loss = 320.68, lr = 0.00000
Epoch: 38
   train loss = 321.15, NLL loss = 319.46, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 39
   train loss = 321.38, NLL loss = 319.69, KLD loss = 1.69
   test loss = 320.70, lr = 0.00000
Epoch: 40
   train loss = 321.41, NLL loss = 319.72, KLD loss = 1.69
   test loss = 320.64, lr = 0.00000
Epoch: 41
   train loss = 321.33, NLL loss = 319.64, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 42
   train loss = 321.35, NLL loss = 319.66, KLD loss = 1.69
   test loss = 320.59, lr = 0.00000
Epoch: 43
   train loss = 321.43, NLL loss = 319.74, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 44
   train loss = 321.21, NLL loss = 319.52, KLD loss = 1.69
   test loss = 320.66, lr = 0.00000
```

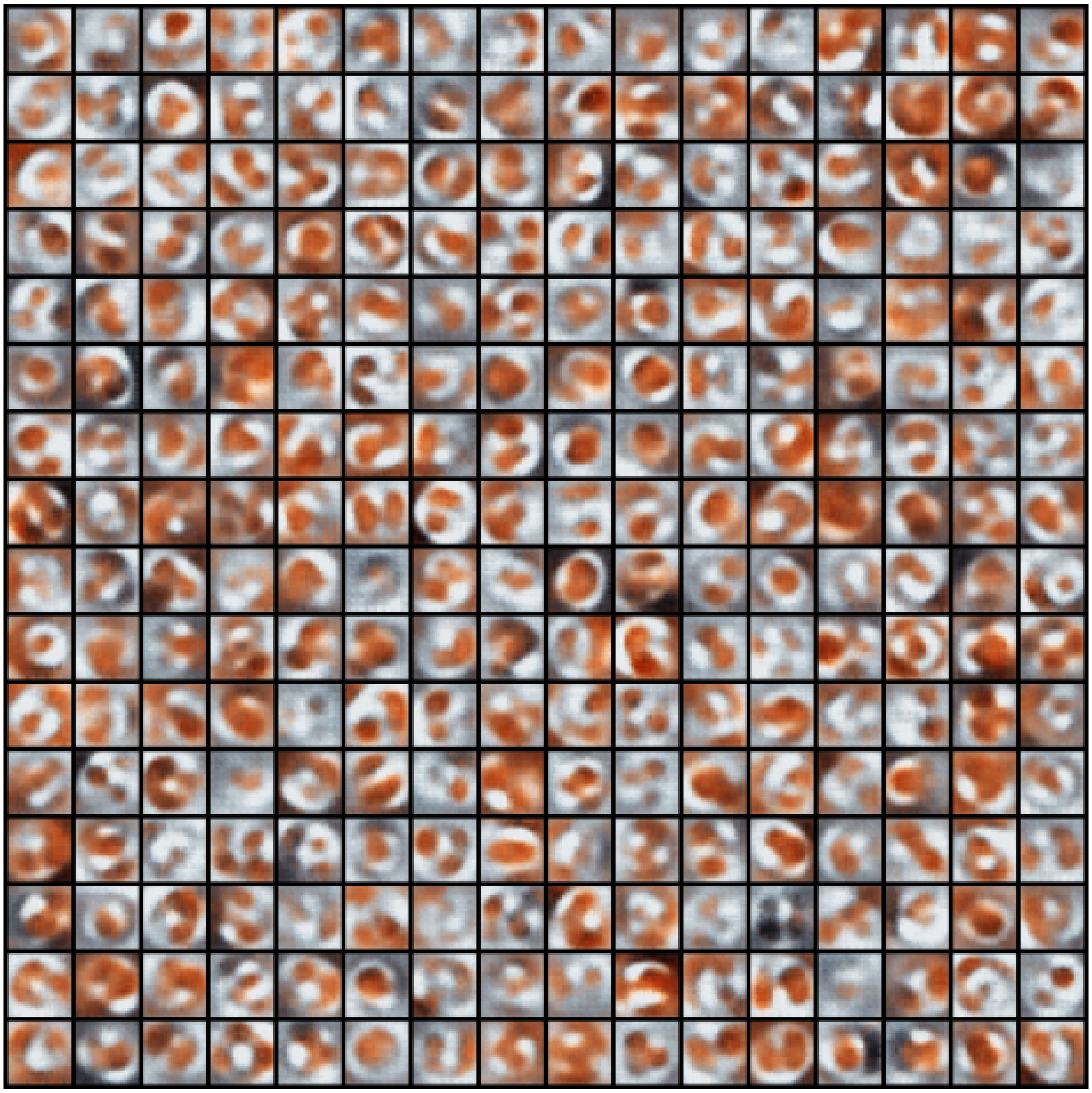
```
Epoch: 45
   train loss = 321.37, NLL loss = 319.68, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 46
   train loss = 321.34, NLL loss = 319.65, KLD loss = 1.69
   test loss = 320.66, lr = 0.00000
Epoch: 47
   train loss = 321.33, NLL loss = 319.64, KLD loss = 1.69
   test loss = 320.62, lr = 0.00000
Epoch: 48
   train loss = 321.44, NLL loss = 319.75, KLD loss = 1.69
   test loss = 320.65, lr = 0.00000
Epoch: 49
   train loss = 321.45, NLL loss = 319.76, KLD loss = 1.69
   test loss = 320.68, lr = 0.00000
Epoch: 50
   train loss = 321.48, NLL loss = 319.79, KLD loss = 1.69
   test loss = 320.65, lr = 0.00000
Epoch: 51
   train loss = 321.35, NLL loss = 319.65, KLD loss = 1.69
   test loss = 320.61, lr = 0.00000
Epoch: 52
   train loss = 321.29, NLL loss = 319.60, KLD loss = 1.69
   test loss = 320.67, lr = 0.00000
Epoch: 53
   train loss = 321.37, NLL loss = 319.68, KLD loss = 1.69
   test loss = 320.67, lr = 0.00000
Epoch: 54
   train loss = 321.36, NLL loss = 319.67, KLD loss = 1.69
   test loss = 320.59, lr = 0.00000
Epoch: 55
   train loss = 321.36, NLL loss = 319.67, KLD loss = 1.69
   test loss = 320.61, lr = 0.00000
Epoch: 56
   train loss = 321.25, NLL loss = 319.56, KLD loss = 1.69
   test loss = 320.60, lr = 0.00000
Epoch: 57
   train loss = 321.38, NLL loss = 319.69, KLD loss = 1.69
   test loss = 320.61, lr = 0.00000
Epoch: 58
   train loss = 321.32, NLL loss = 319.64, KLD loss = 1.69
   test loss = 320.60, lr = 0.00000
Epoch: 59
   train loss = 321.52, NLL loss = 319.83, KLD loss = 1.69
   test loss = 320.63, lr = 0.00000
Epoch: 60
   train loss = 321.42, NLL loss = 319.73, KLD loss = 1.69
   test loss = 320.61, lr = 0.00000
```

In [64]:
```
# %%
# *CODE FOR PART 1.2b IN THIS CELL*

########################## ** ADDED CODE ** ###########################
def denorm(x, mean=0.5, std=0.5):
    t = transforms.Normalize((-mean/std, -mean/std, -mean/std),
                             (1/std, 1/std, 1/std))
    return t(x)
#######################################################################

# load the model
print('Input images')
print('-'*50)

sample_inputs, _ = next(iter(hotdogdata_loader_test))
fixed_input = sample_inputs[0:32, :, :, :]
# visualize the original images of the last batch of the test set
img = make_grid(denorm(fixed_input), nrow=8, padding=2, normalize=True,
                value_range=None, scale_each=False, pad_value=0)
plt.figure()
show(img)

print('Reconstructed images')
print('-'*50)
model.eval()
with torch.no_grad():
    # visualize the reconstructed images of the last batch of test set

    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################
    recon_batch, _, _ = model(fixed_input.to(device))
    #######################################################################
    #                       ** END OF YOUR CODE **
    #######################################################################

    recon_batch = recon_batch.cpu()
    recon_batch = make_grid(denorm(recon_batch), nrow=8, padding=2, normalize=True,
                            value_range=None, scale_each=False, pad_value=0)
    plt.figure()
    show(recon_batch)

print('Generated Images')
print('-'*50)
model.eval()
n_samples = 256
z = torch.randn(n_samples,latent_dim).to(device)
with torch.no_grad():
    #######################################################################
    #                       ** START OF YOUR CODE **
    #######################################################################
    samples = model.decode(z.to(device))
    #######################################################################
    #                       ** END OF YOUR CODE **
```

```
##########################################################################

samples = samples.cpu()
samples = make_grid(denorm(samples), nrow=16, padding=2, normalize=True,
                    value_range=None, scale_each=False, pad_value=0)
plt.figure(figsize = (8,8))
show(samples)
```

```
Input images
--------------------------------------------------
Reconstructed images
--------------------------------------------------
Generated Images
--------------------------------------------------
```





In [67]:
```
# %%
# CODE FOR PART 1.3b IN THIS CELL
######################################################################
#                         ** START OF YOUR CODE **
######################################################################
def get_random_image(dataset):
    """
    Get a random image from the dataset.
    Formats it to be of size [1, C, H, W]
    """
    idx = random.randint(0, len(dataset)-1)
    return dataset[idx][None,:,:,:]

num_steps = 10

test_images, _ = next(iter(hotdogdata_loader_test))

image1 = get_random_image(test_images)
image2 = get_random_image(test_images)

interpolation_rate = torch.linspace(0, 1, num_steps)

model.eval()
with torch.no_grad():
    img_1_embedding, _ = model.encode(image1.to(device))
    img_2_embedding, _ = model.encode(image2.to(device))

    interpolated_embeddings = []
    for rate in interpolation_rate:
        interpolated_embeddings.append((img_1_embedding*(1-rate)) + (img_2_embedding*rate))
    interpolated_embeddings = torch.stack(interpolated_embeddings)

    interpolated_images = model.decode(interpolated_embeddings)


def imshow(img, ax):
    img = img.permute(1, 2, 0)
    ax.imshow(img.cpu().numpy())
    ax.axis('off')

fig, axes = plt.subplots(1, interpolated_images.size(0) + 2, figsize=(20, 5))
imshow(denorm(image1[0]), axes[0])
axes[0].set_title('Original Image 1')
for i in range(interpolated_images.size(0)):
    imshow(denorm(interpolated_images[i]), axes[i + 1])
axes[5].set_title('Interpolated Images')
imshow(denorm(image2[0]), axes[-1])
axes[-1].set_title('Original Image 2')
plt.show()
######################################################################
#                         ** END OF YOUR CODE **
######################################################################
```


Original Image 1
Interpolated Images
Original Image 2

In [73]:
```
# CODE FOR PART 2.1a - Linear Interpolation in Latent Space
# This code demonstrates how to:
# 1. Take two images from the dataset
# 2. Encode them into the latent space
# 3. Create interpolated points between their latent representations
# 4. Decode the interpolated points back to images

#######################################################################
#                        ** START OF YOUR CODE **
#######################################################################
import random

def get_random_image(dataset):
    """
    Get a random image from the dataset.
    Formats it to be of size [1, C, H, W]
    """
    idx = random.randint(0, len(dataset)-1)
    return dataset[idx][None,:,:,:]

# Number of interpolation steps between the two images
num_steps = 10

# Get a batch of training images
data_iter = iter(hotdogdata_loader_train_112)
images, _ = next(data_iter)
print(f"Loaded batch of images with shape: {images.shape}")

# Select first two images and move to device
image1 = get_random_image(images)
image2 = get_random_image(images)
print(f"Individual image shape: {image1.shape}")

interpolation_rate = torch.linspace(0, 1, num_steps)

# Create interpolated points in latent space
interpolated_embeddings = []
with torch.no_grad():
    img_1_embedding = sd_vae.encode(image1.to(device))
    img_2_embedding = sd_vae.encode(image2.to(device))

    for rate in interpolation_rate:
        interpolated_embeddings.append((img_1_embedding*(1-rate)) + (img_2_embedding*rate))
    interpolated_embeddings = torch.stack(interpolated_embeddings, dim=1).squeeze()

    print("Shape of each interpolation:", interpolated_embeddings[-1].shape)
    print("Shape of stacked interpolations:", interpolated_embeddings.shape)

    interpolated_images = sd_vae.decode(interpolated_embeddings)

    print(f"Shape of interpolated embeddings: {interpolated_embeddings.shape}")

def imshow(img, ax):
    """Display an image on the given matplotlib axis"""
```

```
        img = img.squeeze().permute(1, 2, 0)
        ax.imshow(img.cpu().numpy())
        ax.axis('off')

    # Create visualization of original images and interpolations
    fig, axes = plt.subplots(1, interpolated_images.size(0) + 2, figsize=(20, 5))

    # Show first original image
    imshow(image1, axes[0])
    axes[0].set_title('Original Image 1')

    # Show interpolated images
    for i in range(interpolated_images.size(0)):
        imshow(interpolated_images[i], axes[i + 1])
        axes[i + 1].set_title(f'{i*10}% of Image 2')

    # Show second original image
    imshow(image2, axes[-1])
    axes[-1].set_title('Original Image 2')

    plt.suptitle('Linear Interpolation Between Two Images in Latent Space', y=0.75)
    plt.tight_layout()
    plt.show()

    ########################################################################
    #                          ** END OF YOUR CODE **
    ########################################################################
```

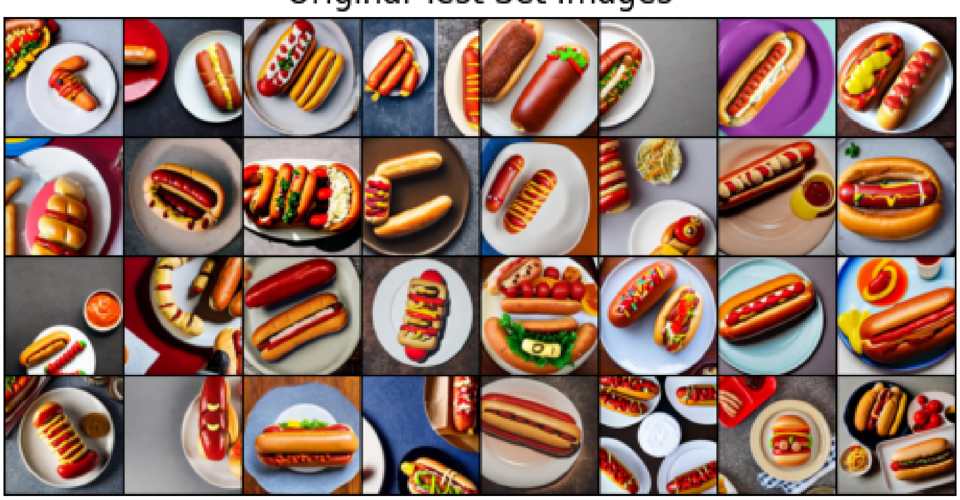
```
Loaded batch of images with shape: torch.Size([32, 3, 112, 112])
Individual image shape: torch.Size([1, 3, 112, 112])
Shape of each interpolation: torch.Size([4, 14, 14])
Shape of stacked interpolations: torch.Size([10, 4, 14, 14])
Shape of interpolated embeddings: torch.Size([10, 4, 14, 14])
```

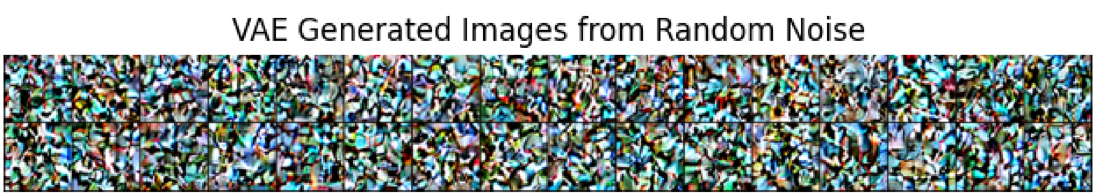
In [74]:
```python
# CODE FOR PART 2.1b - Visualization of VAE Results
# This section demonstrates reconstruction and generation capabilities of the trained VAE

from torchvision.utils import save_image, make_grid

# Helper function to denormalize images if needed
def denorm(x):
    return x

# 1. Visualize Input Images
print('\n1. Displaying Original Input Images from Test Set')
print('-'*70)
sample_inputs, _ = next(iter(hotdogdata_loader_test_112))
fixed_input = sample_inputs[0:32, :, :, :]
input_grid = make_grid(denorm(fixed_input), nrow=8, padding=2, normalize=False,
                       scale_each=False, pad_value=0)
plt.figure()
plt.title('Original Test Set Images')
show(input_grid)

# 2. Visualize Reconstructed Images
print('\n2. Displaying VAE Reconstructions')
print('-'*70)
print('The VAE should learn to accurately reconstruct the input images')

with torch.no_grad():
    #######################################################################
    #                       ** START OF YOUR CODE **
    # CODE: Implement the reconstruction process:
    # 1. Move input images to device
    # 2. Encode images to get latent embeddings
    # 3. Decode latent embeddings back to images
    #######################################################################
    recon_emb = sd_vae.encode(fixed_input.to(device))
    recon_batch = sd_vae.decode(recon_emb)
    #######################################################################
    #                       ** END OF YOUR CODE **
    #######################################################################

    recon_grid = make_grid(denorm(recon_batch.cpu()), nrow=8, padding=2, normalize=False,
                           scale_each=False, pad_value=0)
    plt.figure()
    plt.title('VAE Reconstructed Images')
    show(recon_grid)

# 3. Generate New Images from Random Noise
print('\n3. Displaying Generated Images from Random Noise')
print('-'*70)
print('The VAE should generate plausible new images from random latent vectors')

n_samples = 256
print(f'Latent embedding shape: {recon_emb.shape}')
z = torch.randn_like(recon_emb).to(device)
```

```python
with torch.no_grad():
    ########################################################################
    #                         ** START OF YOUR CODE **
    # CODE: Implement the generation process:
    # 1. Sample random noise vectors from normal distribution
    # 2. Decode noise vectors to generate new images
    ########################################################################
    samples = sd_vae.decode(z.to(device))
    ########################################################################
    #                         ** END OF YOUR CODE **
    ########################################################################

    samples_grid = make_grid(denorm(samples.cpu()), nrow=16, padding=2, normalize=False,
                             scale_each=False, pad_value=0)
    plt.figure(figsize=(8,8))
    plt.title('VAE Generated Images from Random Noise')
    show(samples_grid)
```

```
1. Displaying Original Input Images from Test Set
----------------------------------------------------------------------

2. Displaying VAE Reconstructions
----------------------------------------------------------------------
The VAE should learn to accurately reconstruct the input images

3. Displaying Generated Images from Random Noise
----------------------------------------------------------------------
The VAE should generate plausible new images from random latent vectors
Latent embedding shape: torch.Size([32, 4, 14, 14])
```

Original Test Set Images





1.0.3 Question

The quality of random samples from the VAE's latent space are poor despite using a well-trained VAE. Please detail why this is the case and why moving to latent diffusion models provides an effective solution. In your answer, discuss the limitations of random sampling and explain how latent diffusion models address these challenges.

1.0.4 Your Answer (1 point)

From the '**VAEs vs latent diffusion models*' section in Lecture 11, we know that the main reason why the quality of VAE-generated images is poor lies in the mismatch of the data distributions used during generation. During VAE training, the decoder only sees samples from the same distribution $q_\theta(z)$ and thus only learns how to use these z to recreate the original images x. During random sampling, however, the provided z may be very different from the usual samples seen from $q_\theta(z)$. This means that the VAE generation results won't be thruthful to the original data distribution, as the mismatch between usual z samples and random ones may be significant such that the model may not be able to decode them meaningfully. On the other hand, since latent diffusion models sample* approximately* from q_θ(z) directly during generation time, this distribution mismatch issue is resolved (thanks to the noising-denoising process).

In [76]:
```python
# CODE FOR PART 2.2a IN THIS CELL

def ddpm_schedules(beta1: float, beta2: float, T: int) -> Dict[str, torch.Tensor]:
    """
    Returns pre-computed schedules for DDPM sampling and training process.

    Args:
        beta1: Starting value of noise schedule (must be between 0 and 1)
        beta2: Ending value of noise schedule (must be between beta1 and 1)
        T: Number of timesteps in the diffusion process

    Returns:
        Dict containing the following tensors of shape (T+1,):
            alpha_t: The alpha schedule values
            oneover_sqrta: 1/sqrt(alpha_t) for scaling in diffusion process
            sqrt_beta_t: sqrt(beta_t) for noise scaling
            alphabar_t: The cumulative product of (1-beta)
            sqrtab: sqrt(alphabar_t) for x0 coefficient
            sqrtmab: sqrt(1-alphabar_t) for epsilon coefficient
            mab_over_sqrtmab: (1-alpha_t)/sqrt(1-alphabar_t) for posterior variance
    """
    assert 0.0 < beta1 < beta2 < 1.0, "beta1 and beta2 must be in (0, 1)"
    ##########################################################################
    #                          ** START OF YOUR CODE **
    ##########################################################################
    t = torch.tensor([i for i in range(T+1)])

    beta_t = beta1 + ((beta2 - beta1) * (t / T))
    alpha_t = 1. - beta_t
    oneover_sqrta =  1. / torch.sqrt(alpha_t)
    sqrt_beta_t = torch.sqrt(beta_t)
    alphabar_t = torch.stack([torch.prod(alpha_t[:i]) for i in range(len(alpha_t))])
    sqrtab = torch.sqrt(alphabar_t)
    sqrtmab = torch.sqrt(1. - alphabar_t)
    mab_over_sqrtmab_inv = (1. - alpha_t) / torch.sqrt(1. - alphabar_t)
    ##########################################################################
    #                          ** END OF YOUR CODE **
    ##########################################################################

    return {
        "alpha_t": alpha_t,  # \alpha_t
        "oneover_sqrta": oneover_sqrta,  # 1/\sqrt{\alpha_t}
        "sqrt_beta_t": sqrt_beta_t,  # \sqrt{\beta_t}
        "alphabar_t": alphabar_t,  # \bar{\alpha_t}
        "sqrtab": sqrtab,  # \sqrt{\bar{\alpha_t}}
        "sqrtmab": sqrtmab,  # \sqrt{1-\bar{\alpha_t}}
        "mab_over_sqrtmab": mab_over_sqrtmab_inv,  # (1-\alpha_t)/\sqrt{1-\bar{\alpha_t}}
    }
```

In [ ]:
```python
schedules = ddpm_schedules(0.0001, 0.02, 1000).items()
print("Schedules shapes == 1000 + 1?")
print([v.shape[0] == 1001 for k, v in schedules])
```

```
Schedules shapes == 1000 + 1?
[True, True, True, True, True, True, True]
```

```
In [79]: # CODE FOR PART 2.2b simple CNN IN THIS CELL (or use cell below for UNet with or without posit

         from torchvision.utils import save_image, make_grid
         import torch.nn as nn

         block = lambda ic, oc: nn.Sequential(
             nn.Conv2d(ic, oc, 7, padding=3),
             nn.BatchNorm2d(oc),
             nn.LeakyReLU(),
         )

         class SimpleEpsModel(nn.Module):
             """
             Basically, any universal R^n -> R^n model should work for denoising, so try some very simp
             """

             def __init__(self, n_channel: int) -> None:
                 super(SimpleEpsModel, self).__init__()
                 #######################################################################
                 #                       ** START OF YOUR CODE **
                 #######################################################################
                 # n_channels -> 512 channels -> n_channels
                 self.blocks = nn.Sequential(
                                     block(n_channel, 16),
                                     block(16, 64),
                                     block(64, 128),
                                     block(128, 256),
                                     block(256, 512),
                                     block(512, 256),
                                     block(256, 128),
                                     block(128, 64),
                                     block(64, 16),
                                     # no activation or batchnorm in last layer
                                     nn.Conv2d(in_channels=16,
                                               out_channels=n_channel,
                                               kernel_size=3,
                                               padding=1))
                 #######################################################################
                 #                       ** END OF YOUR CODE **
                 #######################################################################

             def forward(self, x, t) -> torch.Tensor:
                 #######################################################################
                 #                       ** START OF YOUR CODE **
                 #######################################################################
                 out = self.blocks(x)
                 # print("Output shape:", out.shape)
                 return out
                 #######################################################################
                 #                       ** END OF YOUR CODE **
                 #######################################################################

         eps_model = SimpleEpsModel(4)
         print("Num params: ", sum(p.numel() for p in eps_model.parameters()))
```

```python
if sum(p.numel() for p in model.parameters() if p.requires_grad) < 20000000:
    print("PASSED!\nNumber of parameters < 20000000")
else:
    print(f"WARNING: number of parameters {sum(p.numel() for p in model.parameters() if p.requi
```

```
Num params:  16967524
PASSED!
Number of parameters < 20000000
```

In [80]:

```python
import torch
import torch.nn as nn
import math

class Block(nn.Module):
    """
    A basic building block for the U-Net architecture that processes both spatial and temporal

    Args:
        in_ch (int): Number of input channels
        out_ch (int): Number of output channels
        time_emb_dim (int): Dimension of time embedding
        up (bool): If True, uses transposed convolution for upsampling. If False, uses regular
    """
    def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
        ########################################################################
        #                          ** START OF YOUR CODE **
        ########################################################################
        ...
        ########################################################################
        #                          ** END OF YOUR CODE **
        ########################################################################

    def forward(self, x, t):
        """
        Forward pass of the block.

        Args:
            x (torch.Tensor): Input feature maps
            t (torch.Tensor): Time embeddings

        Returns:
            torch.Tensor: Transformed feature maps
        """
        ########################################################################
        #                          ** START OF YOUR CODE **
        ########################################################################
        ...
        ########################################################################
        #                          ** END OF YOUR CODE **
        ########################################################################
```

```python
class SinusoidalPositionEmbeddings(nn.Module):
    """
    Creates sinusoidal positional embeddings for time steps.
    Uses alternating sine and cosine functions at different frequencies.

    Args:
        dim (int): Dimension of the embeddings
    """
    ########################################################################
    #                          ** START OF YOUR CODE **
    ########################################################################

    def __init__(self, dim):
        ...

    ########################################################################
    #                          ** END OF YOUR CODE **
    ########################################################################

    def forward(self, time):
        """
        Compute positional embeddings for given timesteps.

        Args:
            time (torch.Tensor): Tensor of timesteps

        Returns:
            torch.Tensor: Position embeddings of shape (batch_size, dim)
        """
        ########################################################################
        #                          ** START OF YOUR CODE **
        ########################################################################
        embeddings = ...

        return embeddings
        ########################################################################
        #                          ** END OF YOUR CODE **
        ########################################################################

class SimpleUnet(nn.Module):
    """
    A simplified variant of the Unet architecture for diffusion models.
    Includes time conditioning and skip connections.

    Args:
        in_channels (int): Number of input image channels
    """
    def __init__(self, in_channels=4):
        super().__init__()
        image_channels = in_channels
        down_channels = (128, 256, 512)  # Limited the downsampling stages
        up_channels = (512, 256, 128)
        out_dim = in_channels
```

```
        time_emb_dim = 256

        # Time embedding layers
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.ReLU()
        )

        ########################################################################
        #                        ** START OF YOUR CODE **
        ########################################################################

        # Initial projection of image
        self.conv0 = ...

        # Downsampling path
        self.downs = ...

        # Bottleneck block
        self.bottleneck = ...

        # Upsampling path
        self.ups = ...

        # Final projection to output channels
        self.output = ...

        ########################################################################
        #                        ** END OF YOUR CODE **
        ########################################################################

    def forward(self, x, timestep):
        """
        Forward pass of the U-Net.

        Args:
            x (torch.Tensor): Input tensor of shape (batch_size, channels, height, width)
            timestep (torch.Tensor): Current timestep for conditioning

        Returns:
            torch.Tensor: Output tensor of same shape as input
        """
        ########################################################################
        #                        ** START OF YOUR CODE **
        ########################################################################
        # Get time embeddings
        t = ...

        # Initial convolution
        x = ...

        # Store intermediate outputs for skip connections
        residual_inputs = []
```

```
        # Downsampling path
        ...
        # Bottleneck
        x = ...

        # Upsampling path with skip connections
        ...
        ...
        ######################################################################
        #                        ** END OF YOUR CODE **
        ######################################################################

# Test the model, your latent input shape will only be [4,14,14], which means that kernel size
input_tensor = torch.randn(1, 4, 14, 14)
timestep = torch.tensor([1.0])
UNetModel = SimpleUnet()
output = UNetModel(input_tensor, timestep)
print("Output shape:", output.shape)
print("Num params: ", sum(p.numel() for p in UNetModel.parameters()))
if sum(p.numel() for p in model.parameters() if p.requires_grad) < 20000000:
    print("PASSED!\nNumber of parameters < 1000000")
else:
    print(f"WARNING: number of parameters {sum(p.numel() for p in model.parameters() if p.requi
```

In [ ]: # CODE FOR PART 2.2c IN THIS CELL

```python
import torch
import torch.distributions as dist

class DDPM(nn.Module):
    """
    Denoising Diffusion Probabilistic Model (DDPM) implementation.

    This class implements the DDPM as described in "Denoising Diffusion Probabilistic Models"
    (Ho et al., 2020). The model learns to reverse a gradual noising process.

    Args:
        eps_model (nn.Module): The neural network that predicts noise at each timestep
        betas (Tuple[float, float]): Beta schedule parameters ( _start,  _end)
        n_T (int): Number of timesteps in the diffusion process
        criterion (nn.Module): Loss function for training, defaults to MSELoss
    """
    def __init__(
        self,
        eps_model: nn.Module,
        betas: Tuple[float, float],
        n_T: int,
        criterion: nn.Module = nn.MSELoss()
    ) -> None:
        super(DDPM, self).__init__()
        self.eps_model = eps_model

        # register_buffer allows us to freely access these tensors by name. It helps device pla
        for k, v in ddpm_schedules(betas[0], betas[1], n_T).items():
            self.register_buffer(k, v)

        self.n_T = n_T
        self.criterion = criterion

        # Initialize with dataset statistics for potentially better starting point
        mean =  0.04640255868434906
        std_dev = 0.8382343649864197
        self.normal_dist = dist.Normal(mean, std_dev)


    def forward(self, x: torch.Tensor, t: Optional[torch.Tensor] = None
                ) -> tuple[torch.Tensor,torch.Tensor,torch.Tensor]:
        """
        Performs forward diffusion and predicts the noise added at timestep t.

        This implements Algorithm 1 from the DDPM paper:
        1. Sample a random timestep t
        2. Sample random noise
        3. Create noised input x_t using the noise schedule
        4. Predict the noise using the model
        5. Return the loss between predicted and actual noise

        Args:
```

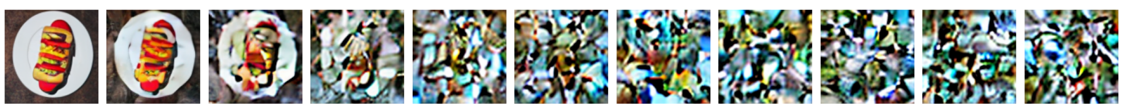
```
            x (torch.Tensor): Input images/data
            t (torch.Tensor, optional): Specific timestep. If None, randomly sampled.

        Returns:
            Tuple containing:
            - Loss between predicted and actual noise
            - The sampled noise (eps)
            - The noised input at timestep t (x_t)
        """
        ######################################################################
        #                        ** START OF YOUR CODE **
        ######################################################################
        device = x.device

        if t is None:
            # Step 1: Randomly sample timestep if not provided
            t = torch.randint(0, self.n_T, (x.shape[0],)).to(device)
        else:
            t = torch.full((x.shape[0],), t.item()).to(device)

        # Step 2: Sample noise from normal distribution
        eps = self.normal_dist.sample(x.shape).to(device)

        # Step 3: Create noised input x_t using the noise schedule
        sqrtab = self.sqrtab.to(device)[t,][:,None,None,None]
        sqrtmab = self.sqrtmab.to(device)[t,][:,None,None,None]
        # x_t = √( _bar_t) * x + √(1- _bar_t) *
        x_t = ((sqrtab * x) + (sqrtmab * eps)).to(device)

        # Step 4 & 5: Predict noise using eps_model and return loss
        # Note: timesteps are normalized to [0,1] range for the model
        predicted_noise = self.eps_model.to(device)(x_t, t)

        loss = self.criterion(predicted_noise, eps)

        return loss, eps, x_t
        ######################################################################
        #                        ** END OF YOUR CODE **
        ######################################################################

    def sample(self, n_sample: int, size, device, t = 0) -> torch.Tensor:
        """
        Samples new images using the trained diffusion model.

        Implements Algorithm 2 from the DDPM paper - the reverse diffusion process.
        Starting from pure noise, gradually denoise to generate new samples.

        Args:
            n_sample (int): Number of samples to generate
            size (tuple): Size of each sample
            device: Device to generate samples on
            t (int): Starting timestep (default=0)

        Returns:
```

```python
        torch.Tensor: Generated samples
    """
    # Start from pure noise
    x_i = torch.randn(n_sample, *size).to(device)  # x_T ~ N(0, 1)

    #########################################################################
    #                        ** START OF YOUR CODE **
    #########################################################################
    # Gradually denoise the samples
    for i in range(self.n_T, t, -1):
        # print("x_i shape:", x_i.shape)
        predicted_noise = self.eps_model(x_i, t)
        # print("predicted_noise shape:", predicted_noise.shape)

        x_i = self.oneover_sqrta[i] * (x_i - self.mab_over_sqrtmab[i] * predicted_noise)
        if i > 1: # since t in [1,T]
            # sample random noise
            noise = torch.randn(n_sample, *size).to(device)
            # print("noise shape:", noise.shape)
            x_i += self.sqrt_beta_t[i] * noise
    #########################################################################
    #                        ** END OF YOUR CODE **
    #########################################################################
    return x_i
```

In [82]:
```
# CODE FOR PART 2.2d IN THIS CELL

import torch.distributions as dist

# Helper function to display tensor images
def show_tensor_image(image):
    reverse_transforms = transforms.Compose([
        transforms.ToPILImage(),
    ])
    # Take first image of batch
    if len(image.shape) == 4:
        image = image[0, :, :, :]
    plt.imshow(reverse_transforms(image))

# Set up DDPM model parameters
print("Setting up DDPM model with 1000 timesteps...")
T = 1000

##########################################################################
#                        ** START OF YOUR CODE **
##########################################################################
ddpm = DDPM(eps_model,(0.0001, 0.02), T) # SimpleEpsModel

# Get a sample image and encode it to latent space
print("\nStep 1: Loading and encoding a sample hot dog image")
image = next(iter(hotdogdata_loader_test_112))[0]
print(f"Original image shape: {image.shape}")

image = image.to(device)

emb = sd_vae.encode(image)

latent_shape = emb.shape
print(f"Encoded latent shape: {emb.shape}")

# Visualize forward diffusion process
print("\nStep 2: Visualizing forward diffusion process over time")
print("Creating plot with 10 timesteps from t=0 to t=T...")
plt.figure(figsize=(20,2))
plt.axis('off')
num_images = 10
stepsize = int(T/num_images)

# Apply noise gradually and show results
print("\nStep 3: Applying noise gradually and decoding at each step...")
for idx in range(0, T+1, stepsize):
    plt.subplot(1, num_images+1, int(idx/stepsize) + 1)
    plt.axis('off')
    # turn timestep to tensor to comply with model.forward
    t = torch.Tensor([idx]).type(torch.int64)
    loss, eps, emb = ddpm.forward(emb, t)

    img = sd_vae.decode(emb)
    show_tensor_image(img)
```

```python
plt.tight_layout()  # Adjust spacing between subplots
print("\nVisualization complete - observe how the image becomes increasingly noisy over time")

########################################################################
#                          ** END OF YOUR CODE **
########################################################################
```

```
Setting up DDPM model with 1000 timesteps…

Step 1: Loading and encoding a sample hot dog image
Original image shape: torch.Size([32, 3, 112, 112])
Encoded latent shape: torch.Size([32, 4, 14, 14])

Step 2: Visualizing forward diffusion process over time
Creating plot with 10 timesteps from t=0 to t=T…

Step 3: Applying noise gradually and decoding at each step…

Visualization complete - observe how the image becomes increasingly noisy over time
```



In [ ]:
```
# CODE FOR PART 2.2e IN THIS CELL

#####################################################################
#                       ** START OF YOUR CODE **
#####################################################################

# ===== 1. Setup Model Architecture and Parameters =====
# Initialize shapes by processing a sample image through VAE
image = next(iter(hotdogdata_loader_test_112))[0]
image = image.to(device)

emb = sd_vae.encode(image)
img_shape = image.shape
latent_shape = emb.shape

print("Input image shape (batch_size, channels, height, width):", img_shape)
print("VAE encoded latent shape:", emb.shape)
print("Final latent shape for model input:", latent_shape)

# ===== 2. Model Selection =====
# Choose between SimpleEpsModel (faster convergence) or SimpleUnet (better quality)
rn_rn_model = None
model_choice = 0  # 0 for SimpleEpsModel, 1 for SimpleUnet

if model_choice == 0:
    # SimpleEpsModel: Faster initial convergence (~20 epochs)
    eps_model = SimpleEpsModel(latent_shape[1])
    rn_rn_model = eps_model.to(device)
    print("SimpleEpsModel total trainable parameters:", sum(p.numel() for p in eps_model.parame
    model_path = content_path/'CW_LDM/ldm_hotdogs_dummyeps.pth'
elif model_choice == 1:
    # SimpleUnet: Better final quality but slower convergence (~30 epochs)
    simple_unet = ...
    rn_rn_model = simple_unet.to(device)
    print("SimpleUnet total trainable parameters:", sum(p.numel() for p in simple_unet.paramete
    model_path = content_path/'CW_LDM/ldm_hotdogs_simple_unet_cls.pth'

# ===== 3. Initialize DDPM and Optimizer =====
print("Initializing DDPM model and optimizer...")
lr = 1e-4 * batch_size  # Learning rate scaled by batch size
ddpm = DDPM(rn_rn_model, (0.0001, 0.02), T)
ddpm = ddpm.to(device)
optim = torch.optim.Adam(ddpm.parameters(), lr=lr)
ema_decay = 0.99 # adapted from original paper (0.9999)

#####################################################################
#                       ** END OF YOUR CODE **
#####################################################################

# ===== 4. Checkpoint Loading (Optional) =====
lastepoch = 0
load_checkpoint = False
if os.path.exists(model_path) and load_checkpoint:
    print("Loading existing model checkpoint from:", model_path)
```

```python
    checkpoint = torch.load(model_path)
    ddpm.load_state_dict(checkpoint['model_state_dict'])
    optim.load_state_dict(checkpoint['optimizer_state_dict'])
    lastepoch = checkpoint['epoch']

ddpm.to(device)

# ===== 5. Training Loop =====
# Initialize tracking variables
example = None
input_sample = None
best_hd_prob = 0
best_hd_image = None

# Initialize widgets
import ipywidgets as widgets
from IPython.display import display, clear_output

n_epoch = 300  # Train for at least a few hundred epochs (several hours)

# Progress bar to track training progress across epochs
progress_bar = widgets.FloatProgress(
    value=0,
    min=0,
    max=n_epoch,
    description=f'Training (0/{n_epoch} epochs):',
    style={'description_width': 'initial'}
)
loss_text = widgets.HTML(value='Current loss: -')
image_output = widgets.Output()
display(widgets.VBox([progress_bar, loss_text, image_output]))

# Main training loop
for i in range(lastepoch, n_epoch):
    # Training phase
    ddpm.train()
    batch_pbar = tqdm(hotdogdata_loader_train_112, leave=False)
    loss_ema = None

    # Process each batch
    for x, _ in batch_pbar:
        optim.zero_grad()
        ########################################################################
        #                        ** START OF YOUR CODE **
        ########################################################################
        x_emb = sd_vae.encode(x.to(device))
        loss, eps, x_t = ddpm(x_emb)

        if loss_ema is not None:
            loss_ema = (ema_decay * loss_ema) + ((1 - ema_decay) * loss.item())
        else:
            loss_ema = loss.item()

        loss.backward()
```

```
        ########################################################################
        #                       ** END OF YOUR CODE **
        ########################################################################
        batch_pbar.set_description(f"Training loss (EMA): {loss_ema:.4f}")
        optim.step()
    # Update progress widgets
    progress_bar.value = i
    progress_bar.description = f'Training ({i+1}/{n_epoch} epochs):'
    loss_text.value = f'Current loss: {loss_ema:.4f}'

    # ===== 6. Evaluation and Sample Generation =====
    ddpm.eval()
    with torch.no_grad():
        # Generate samples
        xh = ddpm.sample(16, (latent_shape[1], latent_shape[2], latent_shape[3]), device)
        xh = sd_vae.decode(xh)

        # Save generated and input samples
        grid = make_grid(xh, nrow=4)
        save_image(grid, content_path/f'CW_LDM/ldm_sample_{i}.png')
        grid1 = make_grid(x, nrow=4)
        save_image(grid1, content_path/f'CW_LDM/input_sample_{i}.png')

        # Evaluate with hotdog classifier
        predictions, probabilities, top5_accuracy = hotdogclassifier.predict(xh.detach())

        # Track best results
        example = xh
        input_sample = x
        column_55_values = probabilities[:, 55]
        best_hd_prob_idx = torch.argmax(column_55_values)
        best_hd_prob_current = torch.max(column_55_values)

        # Update and display best hotdog image
        if best_hd_prob_current > best_hd_prob:
            best_hd_prob = best_hd_prob_current
            best_hd_image = xh[best_hd_prob_idx,:,:,:]

            with image_output:
                clear_output(wait=True)
                plt.figure(figsize=(8, 8))
                show(best_hd_image)
                plt.title(f'Best generated hot dog (confidence: {best_hd_prob:.3f})')
                display(plt.gcf())
                plt.close()

        # Save checkpoint
        state = {
            'epoch': i,
            'model_state_dict': ddpm.state_dict(),
            'optimizer_state_dict': optim.state_dict(),
            'loss': loss.item(),
        }
        torch.save(state, model_path)
```

```
Input image shape (batch_size, channels, height, width): torch.Size([32, 3, 112, 112])
VAE encoded latent shape: torch.Size([32, 4, 14, 14])
Final latent shape for model input: torch.Size([32, 4, 14, 14])
SimpleEpsModel total trainable parameters: 16967524
Initializing DDPM model and optimizer…
```

```
VBox(children=(FloatProgress(value=0.0, description='Training (0/300 epochs):', max=300.0, style=Progres
```

Please see the progress_images/DDPM folder for generatwed hot god images. In this run, the image below was the final (best) generated hotdog:

Best generated hot dog (confidence: 0.927)