

```
In [ ]: """
    Deep Learning Assignment 3
    Conditional GAN Skeleton Code.
    Adopted from public sources, customized and improved for this assignment.
"""


```

```
#import necessary modules
import torch
import torch.nn as nn
from torchvision import transforms, datasets
from torch import optim as optim
# for visualization
from matplotlib import pyplot as plt
import math
import numpy as np
```

```
In [ ]: # tells PyTorch to use an NVIDIA GPU, if one is available.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# loading the dataset
training_parameters = {
    "img_size": 28,
    "n_epochs": 24, #24
    "batch_size": 64,
    "learning_rate_generator": 0.0002,
    "learning_rate_discriminator": 0.0002,
}
# define a transform to 1) scale the images and 2) convert them into tensors
transform = transforms.Compose([
    transforms.Resize(training_parameters['img_size']), # scales the smaller
    transforms.ToTensor(),
])

# load the dataset
train_loader = torch.utils.data.DataLoader(
    datasets.FashionMNIST(
        './data', # specifies the directory to download the datafiles to, re
        train = True,
        download = True,
        transform = transform),
    batch_size = training_parameters["batch_size"],
    shuffle=True
)

# Fashion MNIST has 10 classes, just like MNIST. Here's what they correspond
label_descriptions = {
    0: 'T-shirt/top',
    1 : 'Trouser',
    2 : 'Pullover',
    3 : 'Dress',
    4 : 'Coat',
    5 : 'Sandal',
    6 : 'Shirt',
    7 : 'Sneaker',
    8 : 'Bag',
    9 : 'Ankle boot'
}
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train
-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train
-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz
```

```

100%|██████████| 26421880/26421880 [00:01<00:00, 18510728.13it/s]
Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 29515/29515 [00:00<00:00, 340771.76it/s]
Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 4422102/4422102 [00:00<00:00, 5986230.28it/s]
Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 5148/5148 [00:00<00:00, 4609794.40it/s]
Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/FashionMNIST/raw

```

```

In [ ]: # Create the Generator model class, which will be used to initialize the generator
class Generator(nn.Module):
    def __init__(self, input_dim, output_dim, num_labels=10): # to initialize
        super(Generator, self).__init__() # initialize the parent class
        # TODO (5.4) Turn this Generator into a Conditional Generator by
        # 1. Adjusting the input dimension of the first hidden layer.
        # 2. Modifying the input to the first hidden layer in the forward class.

        self.hidden_layer1 = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.LeakyReLU(0.2)
        )
        self.hidden_layer2 = nn.Sequential(
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2)
        )
        self.hidden_layer3 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2)
        )
        self.hidden_layer4 = nn.Sequential(
            nn.Linear(1024, output_dim),
            nn.Tanh()
        )
    def forward(self, x, labels):

        output = self.hidden_layer1(x)
        output = self.hidden_layer2(output)
        output = self.hidden_layer3(output)
        output = self.hidden_layer4(output)
        return output.to(device)

```

```
In [ ]: class Discriminator(nn.Module):
    def __init__(self, input_dim, output_dim=1, num_labels=None):
        super(Discriminator, self).__init__()

        #self.label_embedding = nn.Embedding(10, 10)
        # TODO (5.4) Modify this discriminator to function as a conditional
        self.hidden_layer1 = nn.Sequential(
            nn.Linear(input_dim, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )

        self.hidden_layer2 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )

        self.hidden_layer3 = nn.Sequential(
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )

        self.hidden_layer4 = nn.Sequential(
            nn.Linear(256, output_dim),
            nn.Sigmoid()
        )

    def forward(self, x, labels=None): # labels to be used in 5.4.

        output = self.hidden_layer1(x)
        output = self.hidden_layer2(output)
        output = self.hidden_layer3(output)
        output = self.hidden_layer4(output)
        return output.to(device)
```

```
In [ ]: discriminator = Discriminator(784,1).to(device) # initialize both models, and
generator = Generator(100,784).to(device)

discriminator_optimizer = optim.Adam(discriminator.parameters(), lr=training_
generator_optimizer = optim.Adam(generator.parameters(), lr=training_paramet
```

```
In [ ]: # Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.
```

```
In [ ]: #Loss_D - discriminator loss calculated as the sum of losses for the all real
# and all fake images.
loss_func = nn.BCELoss() # Binary Cross Entropy Loss
def train_generator(batch_size):
    """
    Performs a training step on the generator by
    1. Generating fake images from random noise.
    2. Running the discriminator on the fake images.
    3. Computing loss on the result.
    :arg batch_size: the number of training examples in the current batch
    Returns the average generator loss over the batch.
    """

    # Start by zeroing the gradients of the optimizer
    generator_optimizer.zero_grad()
```

```

# 1. Create a new batch of fake images (since the discriminator has just
noise = torch.randn(batch_size, 100).to(device) # whenever you create new
generated_labels = torch.randint(0, 10, (batch_size,)).to(device)
generator_output = generator(noise, labels = generated_labels)
# 2. Run the discriminator on the fake images
discriminator_output = discriminator(generator_output, labels = generate
#####copied#####
real_label_vector = torch.full((batch_size,), real_label, dtype=torch.fl
real_label_vector = real_label_vector.view(-1, 1)
#####
# 3. Compute the loss
loss = loss_func(discriminator_output, real_label_vector)
loss.backward()
generator_optimizer.step()

loss = loss.mean().item()
return loss

def train_discriminator(batch_size, images, labels=None): # labels to be used
"""
    Performs a training step on the discriminator by
        1. Generating fake images from random noise.
        2. Running the discriminator on the fake images.
        3. Running the discriminator on the real images
        3. Computing loss on the results.
:arg batch_size: the number of training examples in the current batch
:arg images: the current batch of images, a tensor of size BATCH x 1 x 6
:arg labels: the labels corresponding to images, a tensor of size BATCH
Returns the average loss over the batch.
"""

discriminator_optimizer.zero_grad()
#####fake images#####
# 1. Create a new batch of fake images (since the discriminator has just
noise = torch.randn(batch_size, 100).to(device) # whenever you create new
generated_labels = torch.randint(0, 10, (batch_size,)).to(device)
generator_output = generator(noise, labels = generated_labels)
# 2. Run the discriminator on the fake images
discriminator_output = discriminator(generator_output, labels = generate
# 3. Compute the loss
fake_label_vector = torch.full((batch_size,), fake_label, dtype=torch.fl
fake_label_vector = fake_label_vector.view(-1, 1)
loss_fake = loss_func(discriminator_output, fake_label_vector)

#####real images#####
# 1. Run the discriminator on the real images
images = torch.flatten(images, start_dim=1)
discriminator_output = discriminator(images, labels = labels)
# 2. Compute the loss
real_label_vector = torch.full((batch_size,), real_label, dtype=torch.fl
real_label_vector = real_label_vector.view(-1, 1)
loss_real = loss_func(discriminator_output, real_label_vector)

#combine losses
loss = loss_real + loss_fake
loss.backward()
discriminator_optimizer.step()

loss = loss.mean().item()
return loss

```

In []: `for epoch in range(training_parameters['n_epochs']):
 G_loss = [] # for plotting the losses over time
 D_loss = []`

```

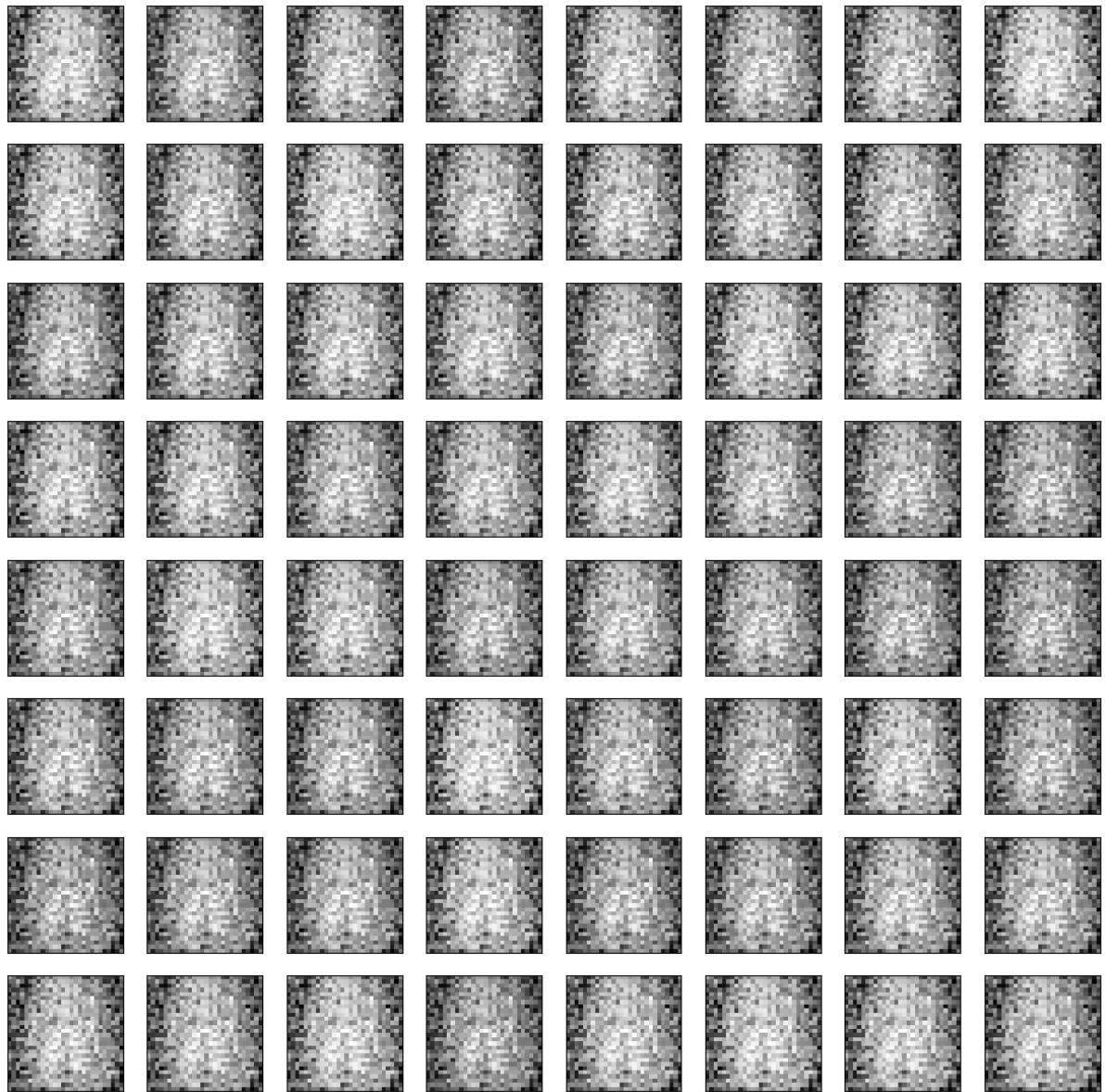
for batch, (imgs, labels) in enumerate(train_loader):
    batch_size = labels.shape[0] # if the batch size doesn't evenly divide
    #generator first training
    lossG = train_generator(batch_size)
    G_loss.append(lossG)
    #single discriminator training
    lossD = train_discriminator(batch_size, imgs, labels)
    D_loss.append(lossD)

if ((batch + 1) % 500 == 0 and (epoch + 1) % 1 == 0):
    # Display a batch of generated images and print the loss
    print("Training Steps Completed: ", batch)
    with torch.no_grad(): # disables gradient computation to speed
        noise = torch.randn(batch_size, 100).to(device)
        fake_labels = torch.randint(0, 10, (batch_size,)).to(device)
        generated_data = generator(noise, fake_labels).cpu().view(ba

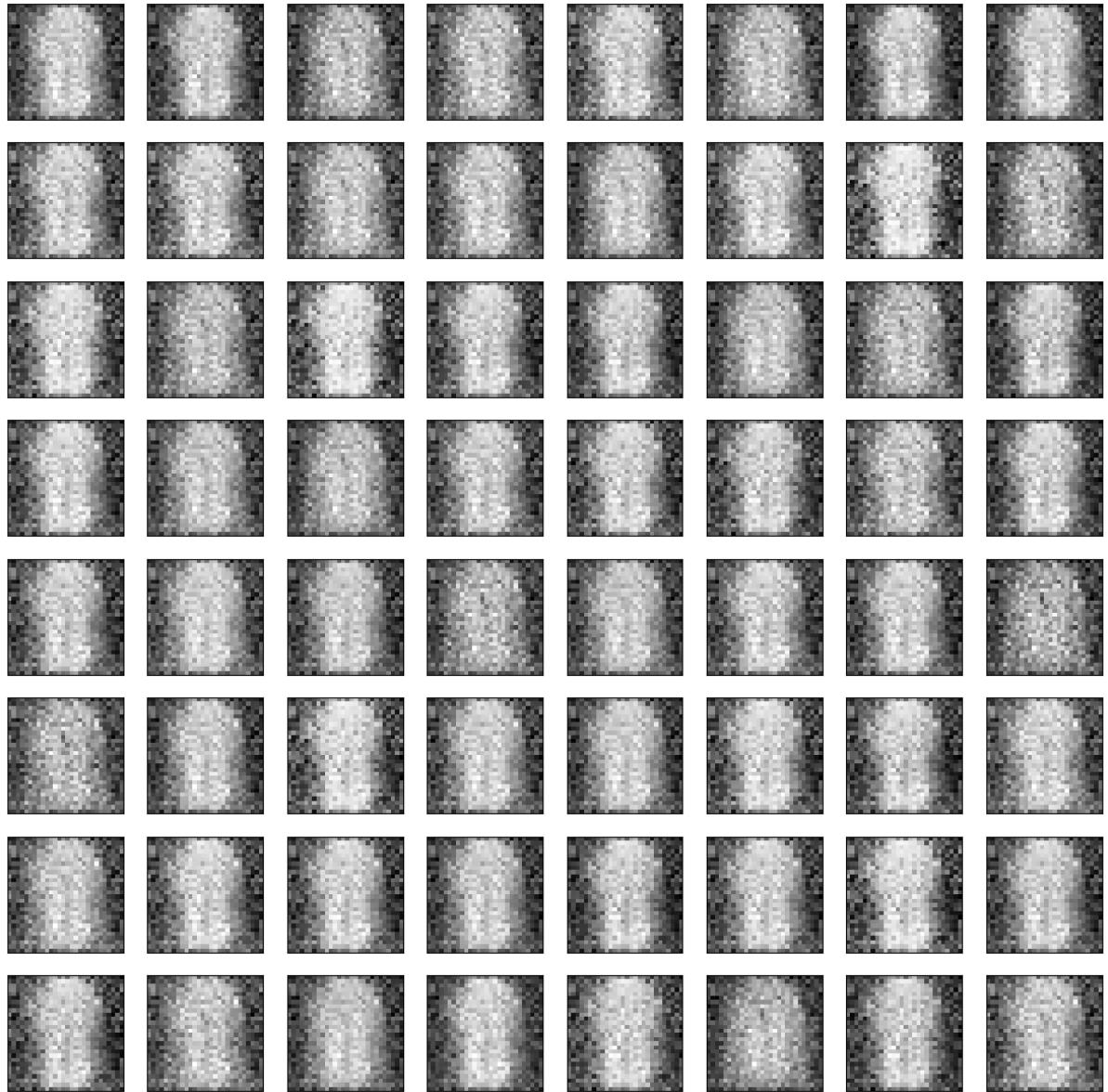
    # display generated images
    batch_sqrt = int(training_parameters['batch_size'] ** 0.5)
    fig, ax = plt.subplots(batch_sqrt, batch_sqrt, figsize=(15,
    for i, x in enumerate(generated_data):
        #ax[math.floor(i / batch_sqrt)][i % batch_sqrt].set_titl
        ax[math.floor(i / batch_sqrt)][i % batch_sqrt].imshow(x.
        ax[math.floor(i / batch_sqrt)][i % batch_sqrt].get_xaxis
        ax[math.floor(i / batch_sqrt)][i % batch_sqrt].get_yaxis
    plt.show()
    #fig.savefig(f"./results/CGAN_Generations_Epoch_{epoch}")
    #fig.savefig(f"pset/pset3/results/CGAN_Generations_Epoch_{ep
    fig.savefig(f"CGAN_Generations_Epoch_{epoch}")
    print(
        f"Epoch {epoch}: loss_d: {torch.mean(torch.FloatTensor(D

```

Training Steps Completed: 499

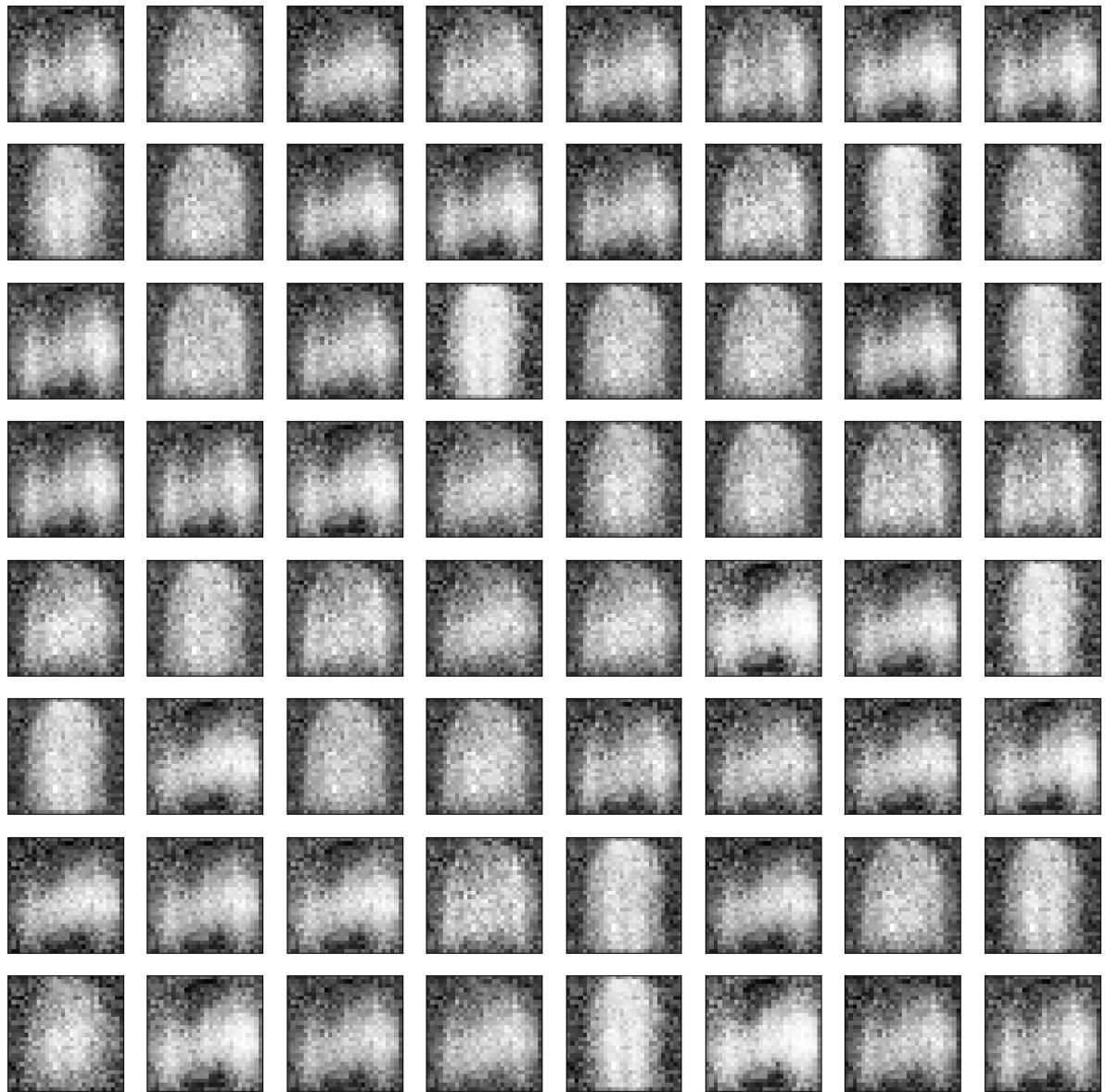


Epoch 0: loss_d: 1.2509028911590576, loss_g: 1.2509392499923706
Training Steps Completed: 499



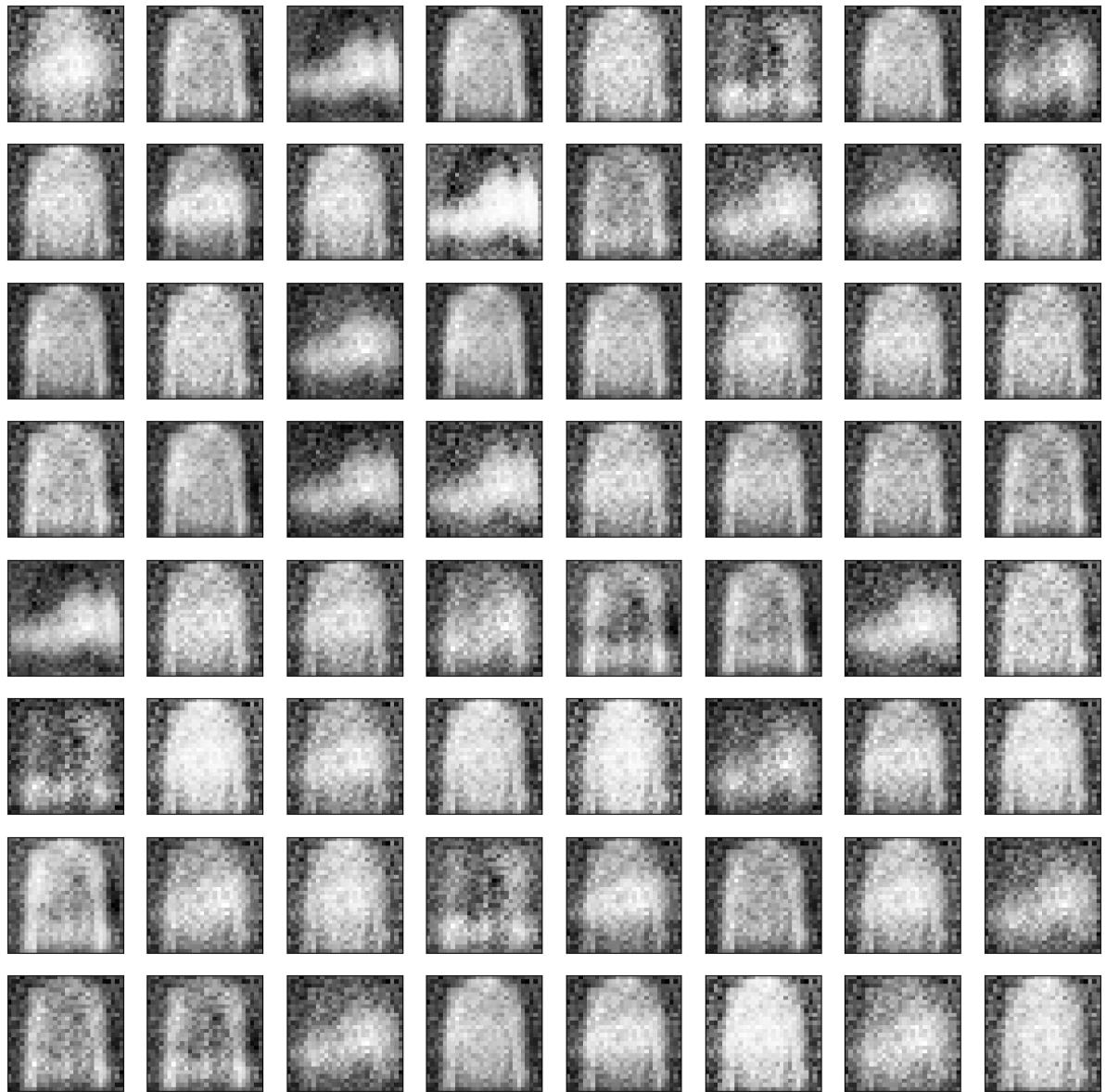
Epoch 1: loss_d: 0.937788188457489, loss_g: 1.4716280698776245

Training Steps Completed: 499

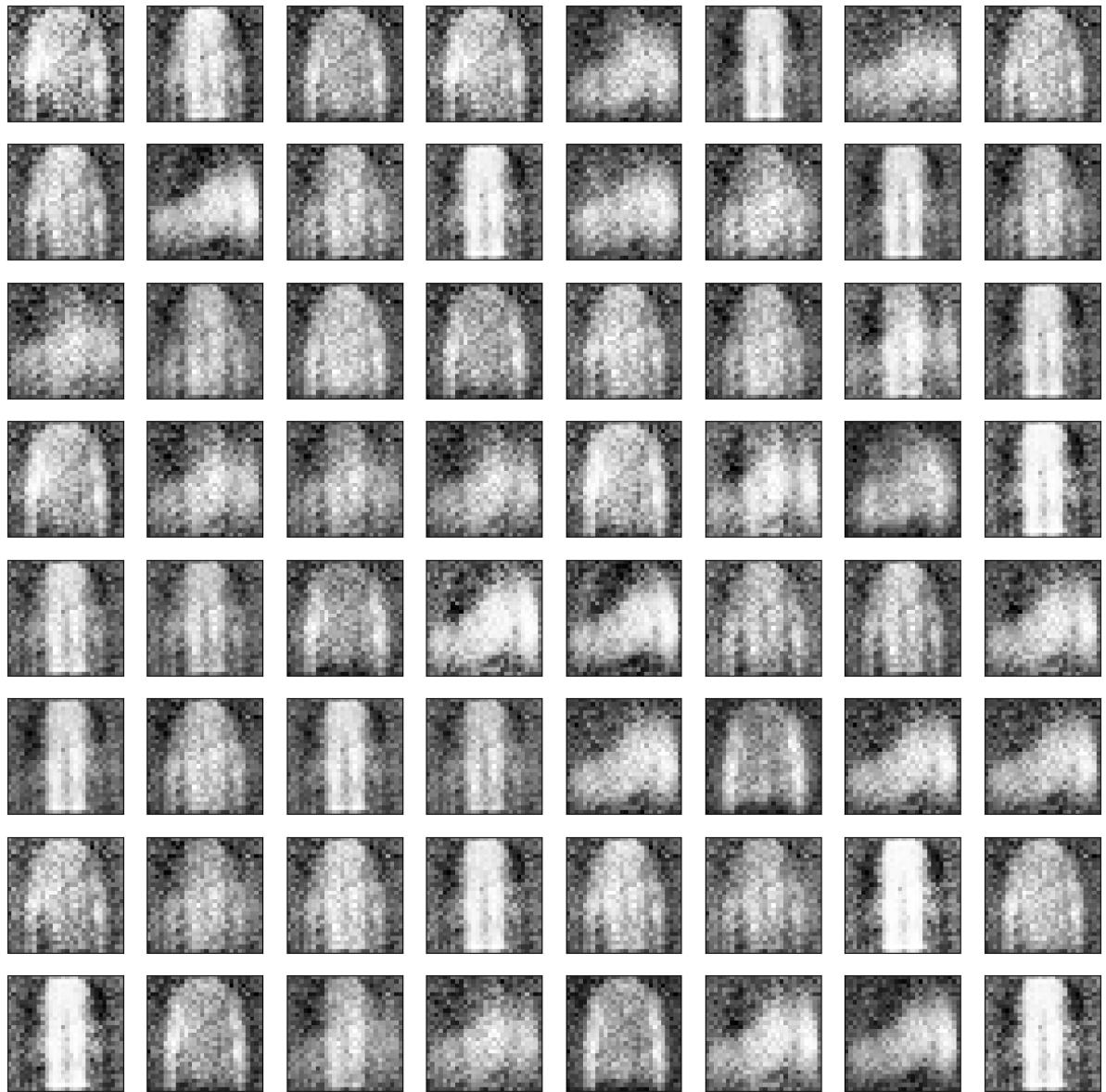


Epoch 2: loss_d: 1.086125135421753, loss_g: 1.4029086828231812

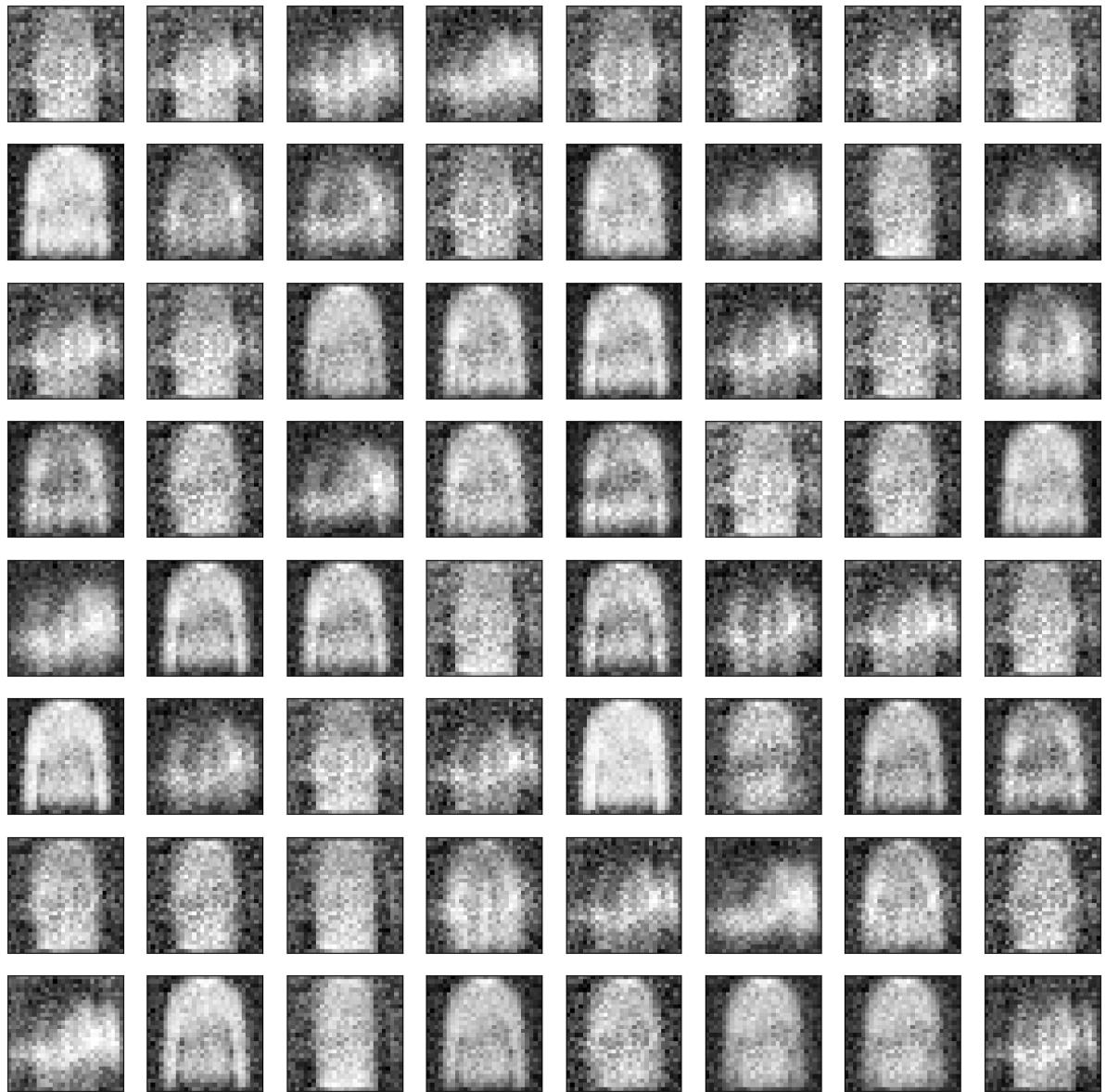
Training Steps Completed: 499



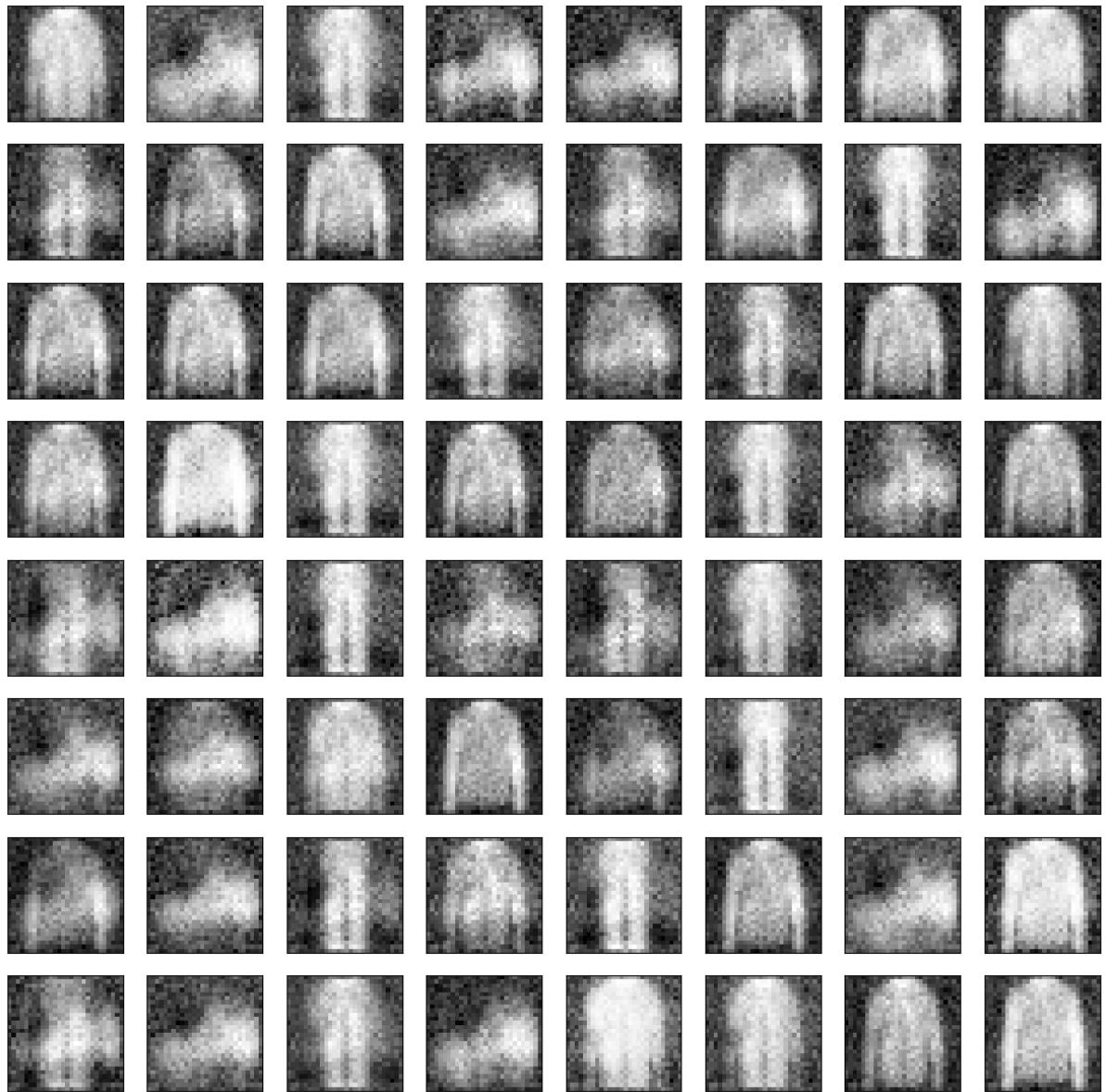
Epoch 3: loss_d: 1.0332881212234497, loss_g: 1.3870456218719482
Training Steps Completed: 499



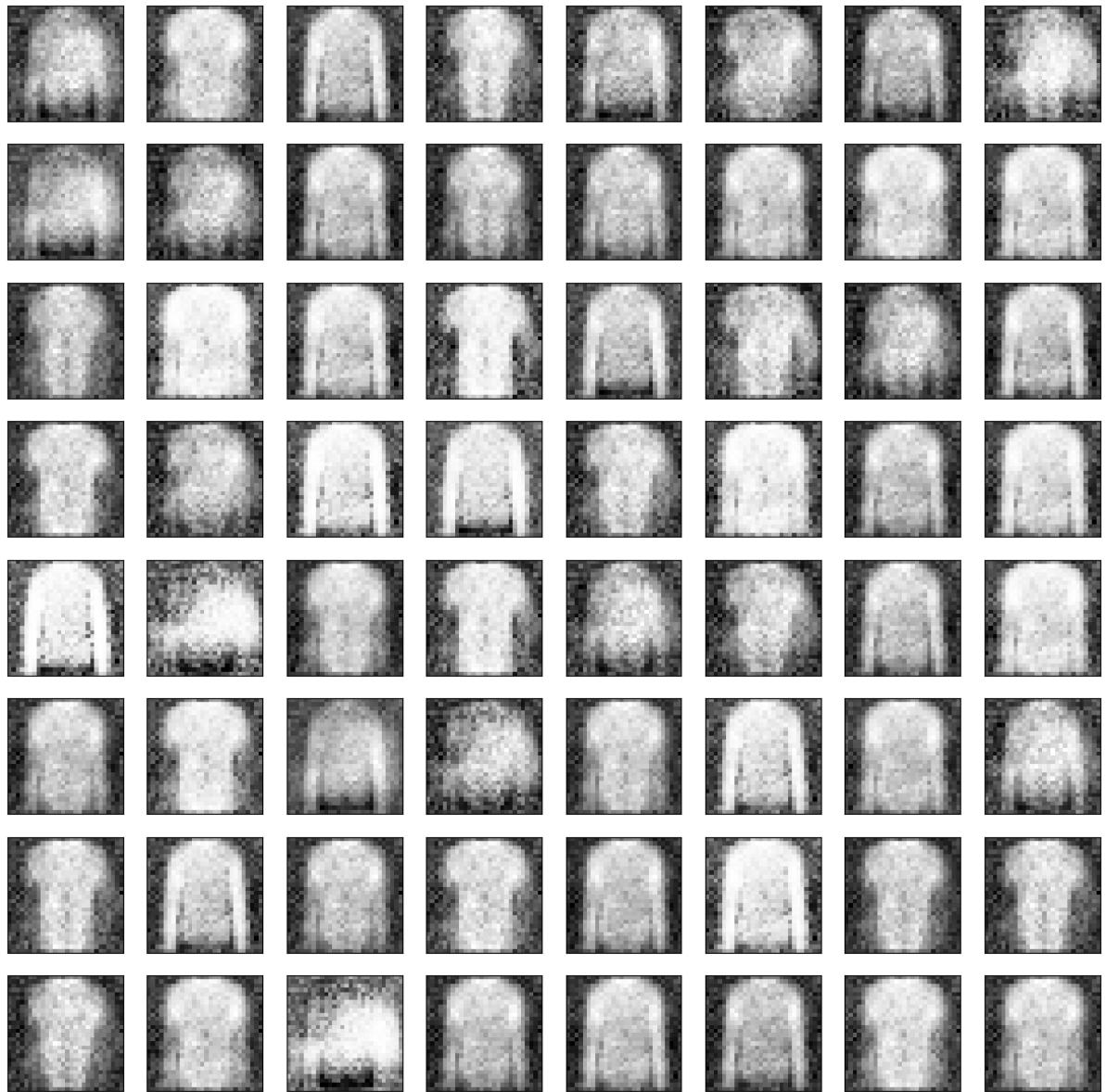
Epoch 4: loss_d: 0.948887825012207, loss_g: 1.5612504482269287
Training Steps Completed: 499



Epoch 5: loss_d: 0.9626063108444214, loss_g: 1.469051718711853
Training Steps Completed: 499



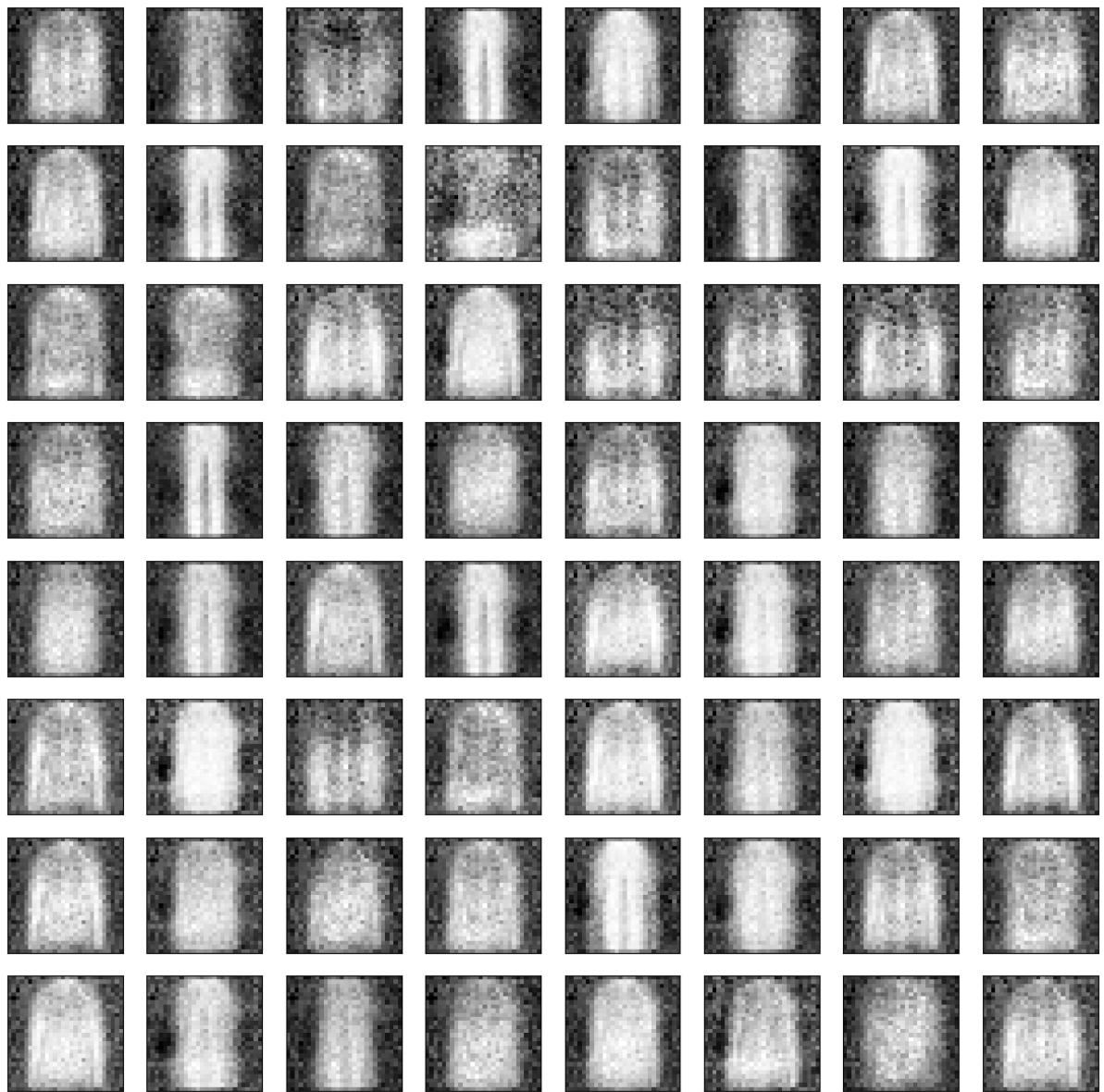
Epoch 6: loss_d: 0.8971349000930786, loss_g: 1.5725210905075073
Training Steps Completed: 499



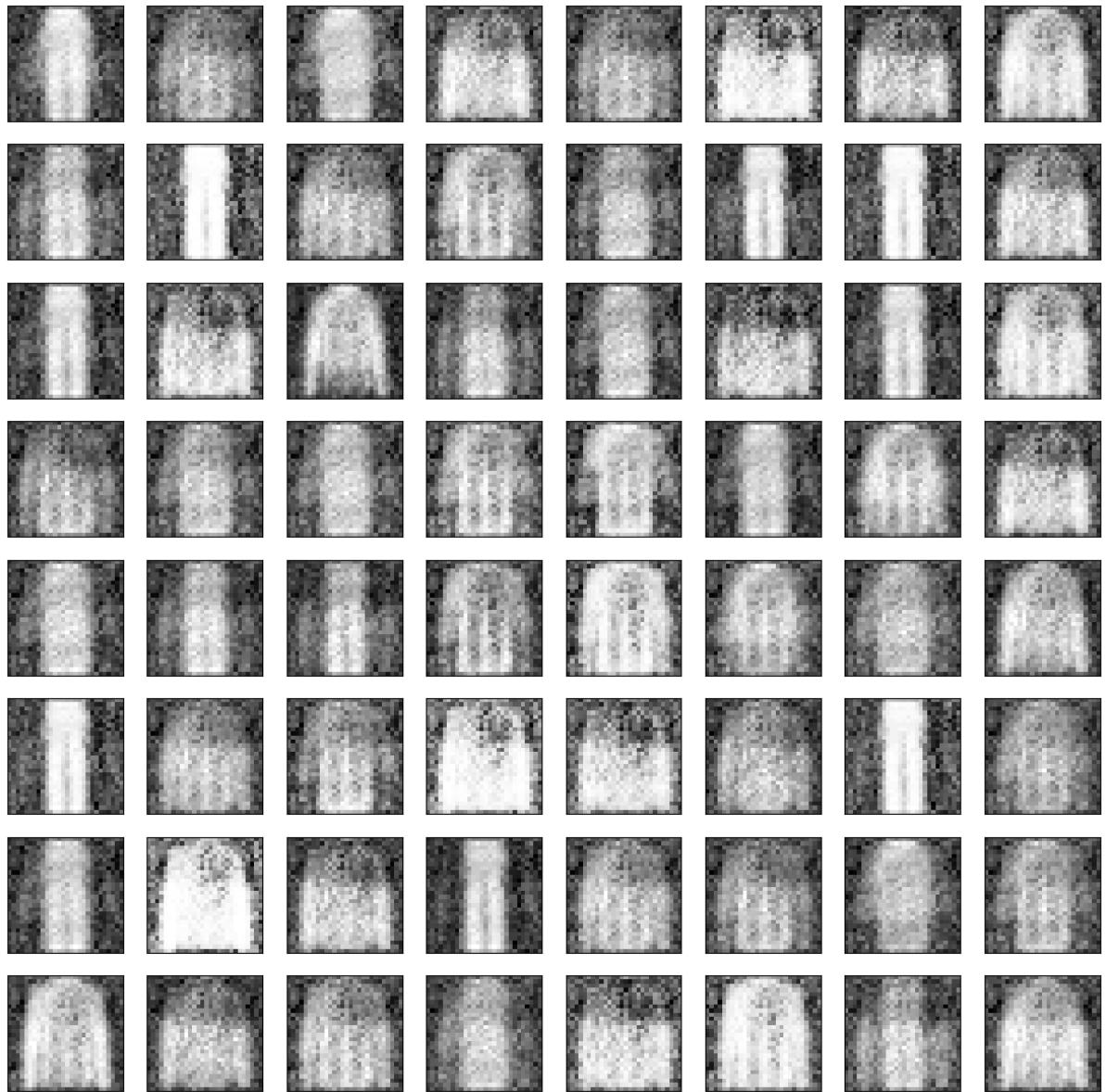
Epoch 7: loss_d: 0.9083710312843323, loss_g: 1.4896395206451416
Training Steps Completed: 499



Epoch 8: loss_d: 0.9515582919120789, loss_g: 1.3189407587051392
Training Steps Completed: 499



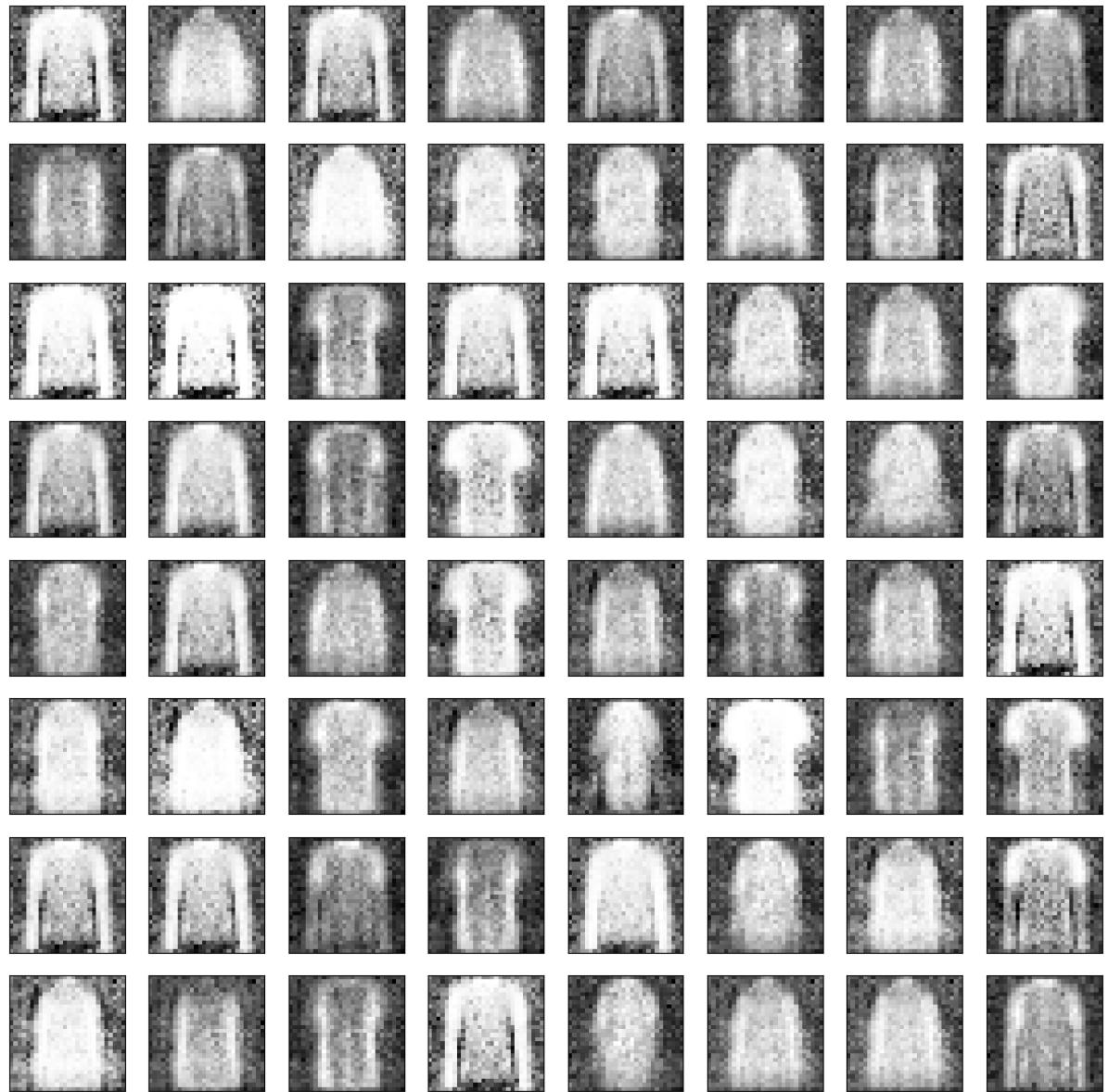
Epoch 9: loss_d: 0.9096298813819885, loss_g: 1.5300629138946533
Training Steps Completed: 499



Epoch 10: loss_d: 0.9100512266159058, loss_g: 1.5634227991104126
Training Steps Completed: 499



Epoch 11: loss_d: 0.826557993888855, loss_g: 1.6329923868179321
Training Steps Completed: 499



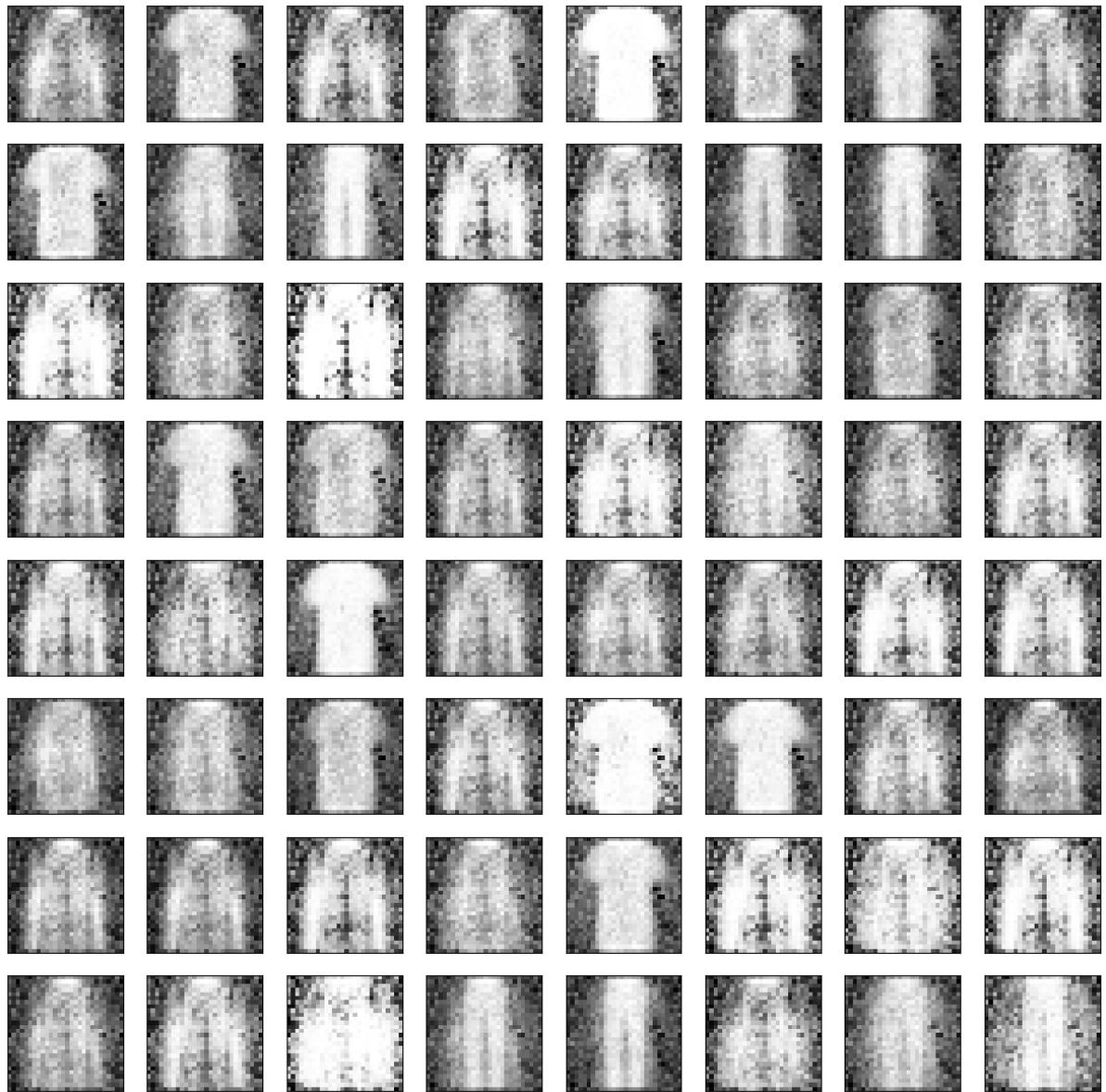
Epoch 12: loss_d: 0.8841820359230042, loss_g: 1.4438378810882568
Training Steps Completed: 499



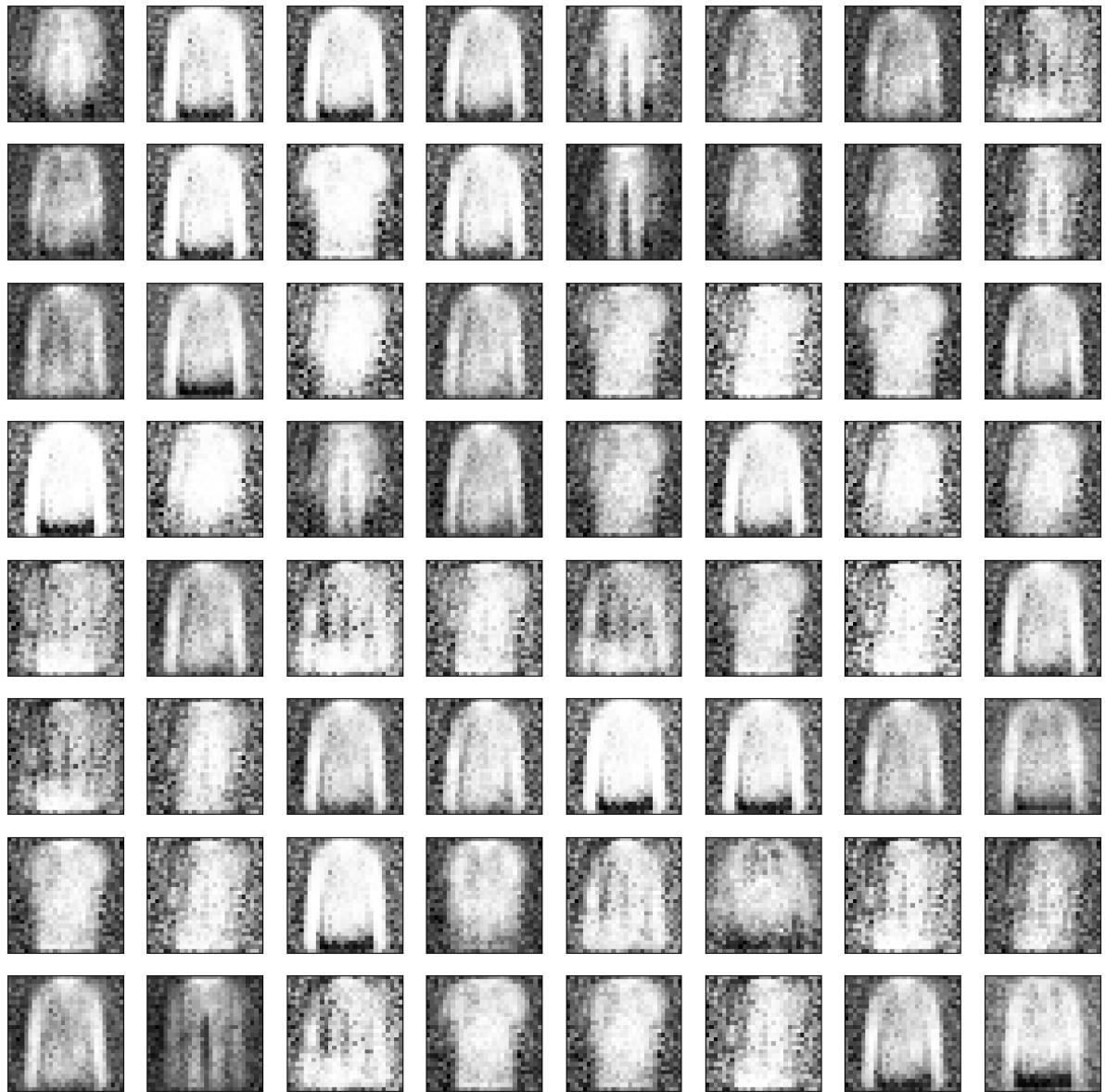
Epoch 13: loss_d: 0.8974578976631165, loss_g: 1.430528998374939
Training Steps Completed: 499



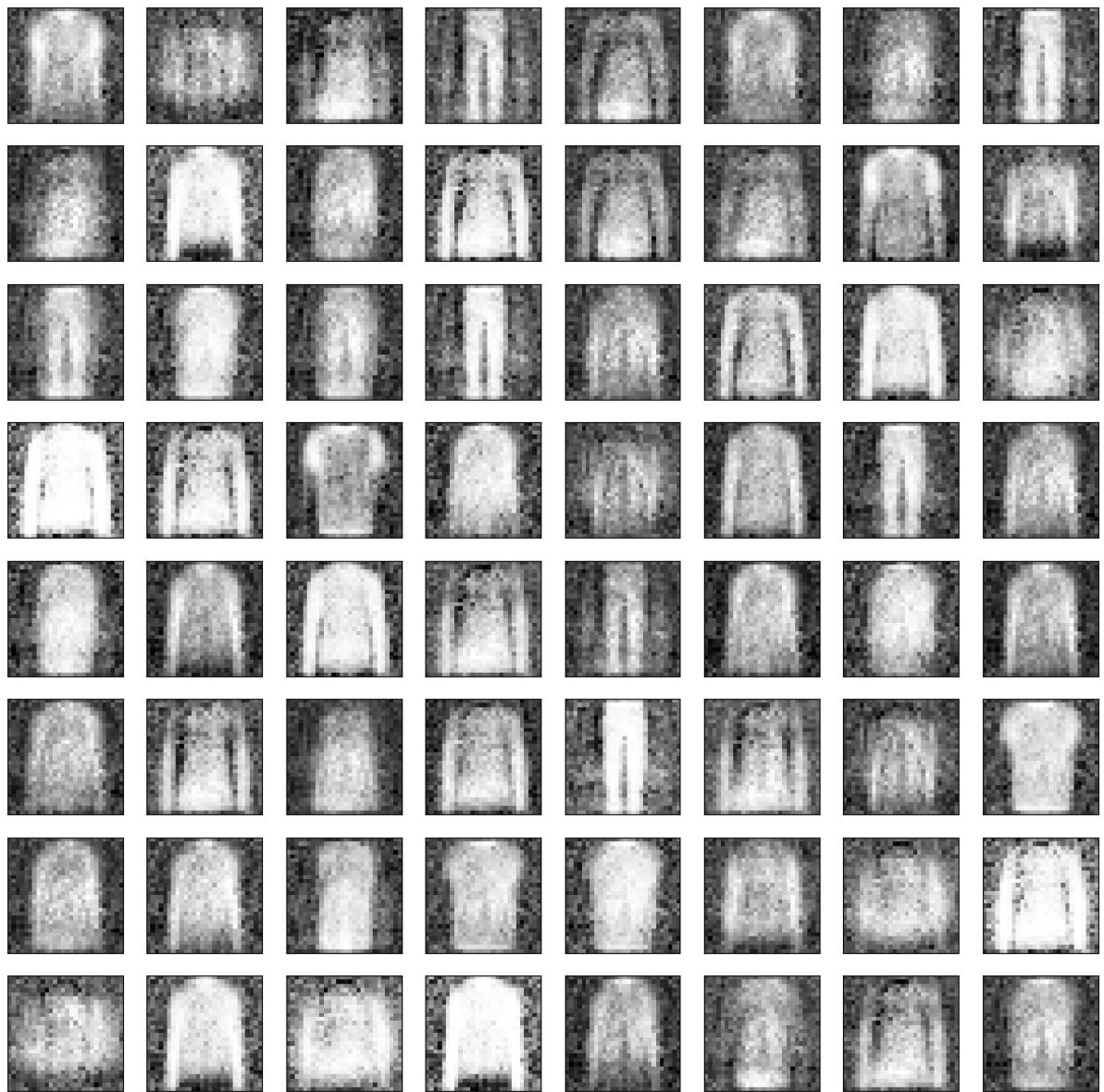
Epoch 14: loss_d: 0.9160184264183044, loss_g: 1.3691389560699463
Training Steps Completed: 499



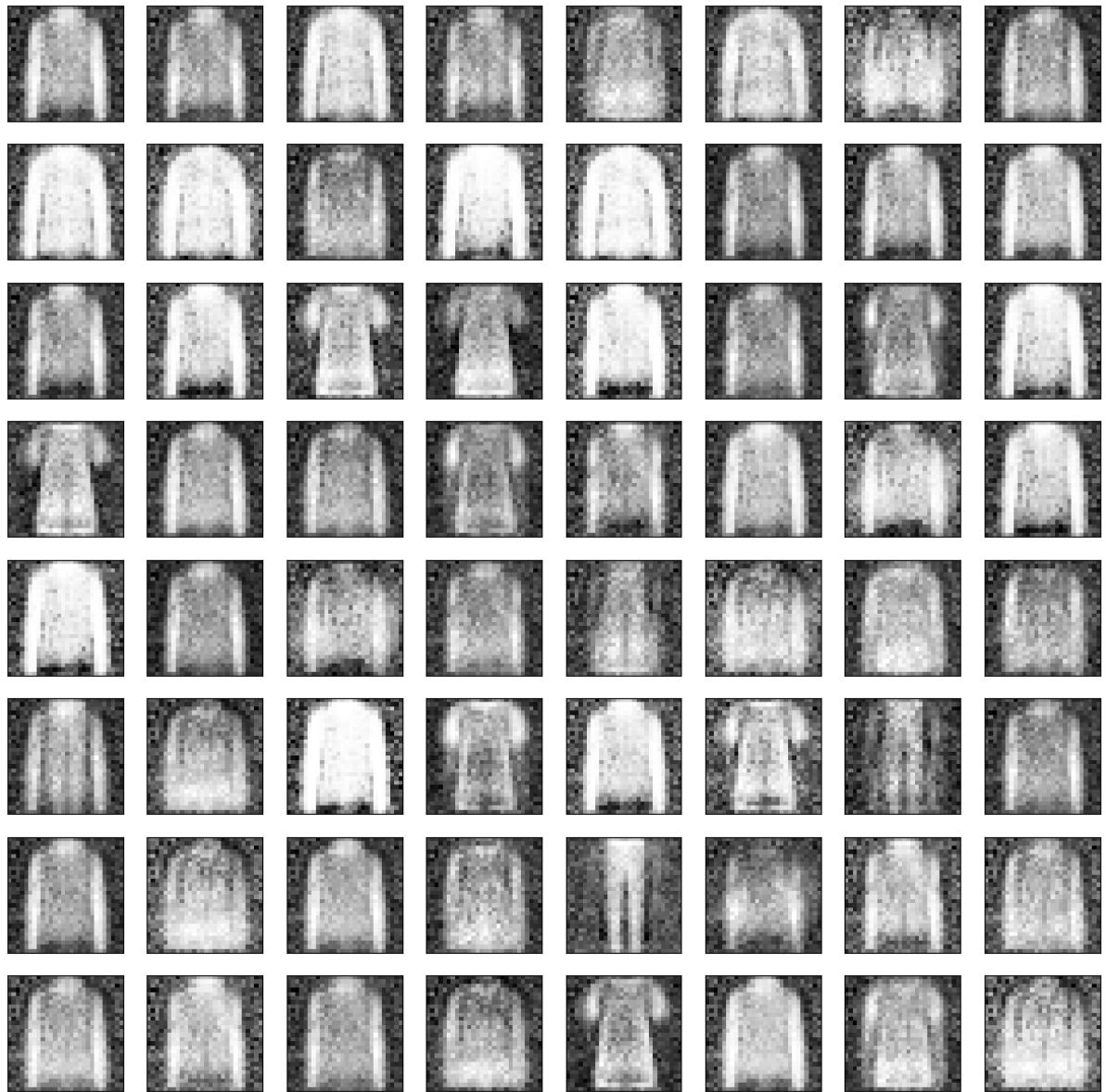
Epoch 15: loss_d: 0.9447134733200073, loss_g: 1.5155681371688843
Training Steps Completed: 499



Epoch 16: loss_d: 0.9155983328819275, loss_g: 1.4525009393692017
Training Steps Completed: 499



Epoch 17: loss_d: 0.9315450191497803, loss_g: 1.3282514810562134
Training Steps Completed: 499



Epoch 18: loss_d: 0.96403968334198, loss_g: 1.3435578346252441

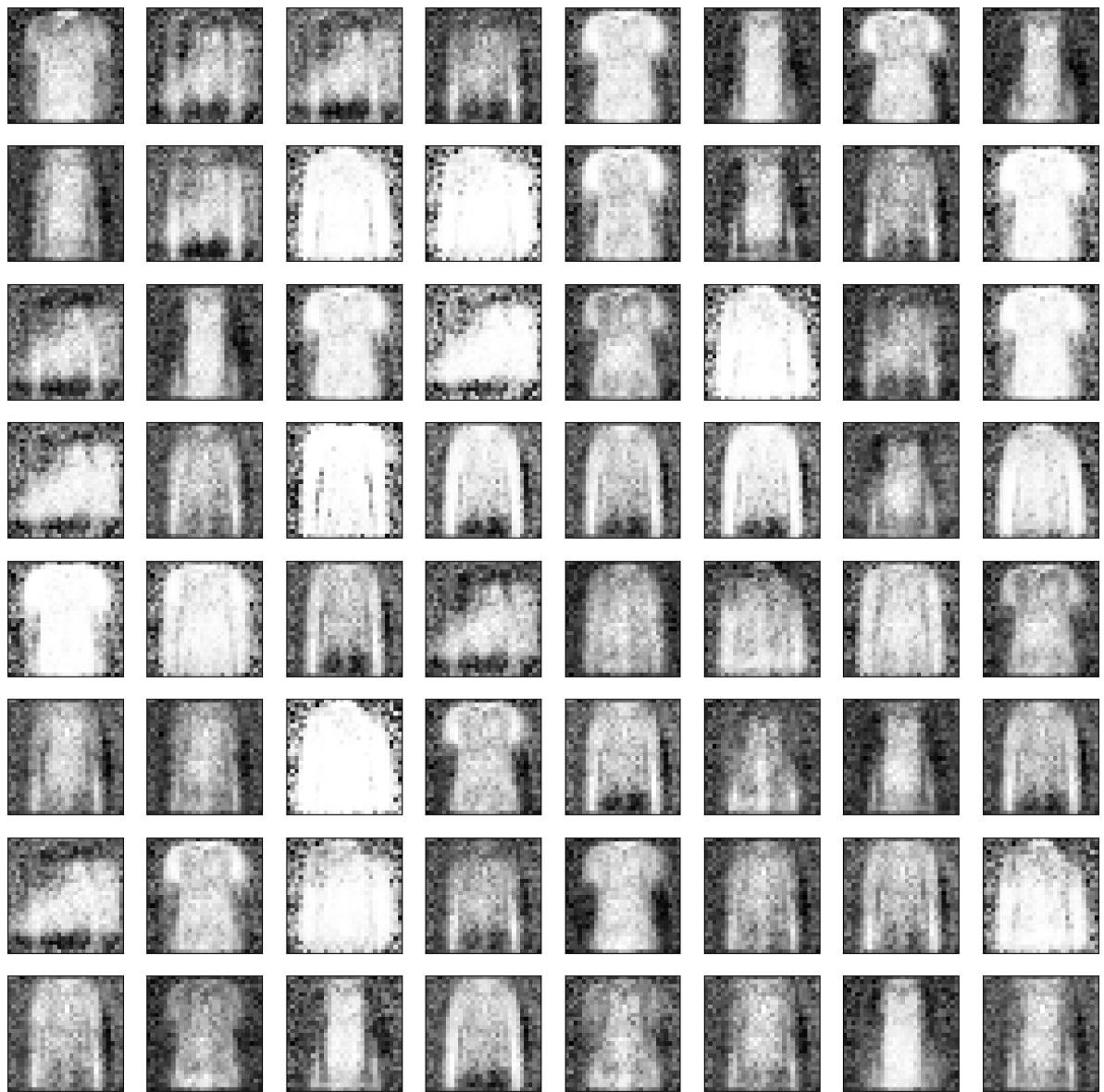
Training Steps Completed: 499



Epoch 19: loss_d: 0.9659090638160706, loss_g: 1.310976505279541
Training Steps Completed: 499



Epoch 20: loss_d: 0.8923707008361816, loss_g: 1.4438281059265137
Training Steps Completed: 499



Epoch 21: loss_d: 0.9595983028411865, loss_g: 1.3769512176513672
Training Steps Completed: 499



Epoch 22: loss_d: 0.9843803644180298, loss_g: 1.3571981191635132
Training Steps Completed: 499



Epoch 23: loss_d: 0.9943499565124512, loss_g: 1.3123371601104736

In []: *#save the model*

```
torch.save(generator.state_dict(), 'generator_cond.pth')
torch.save(discriminator.state_dict(), 'discriminator_cond.pth')
```