SocratesV_HW3

April 15, 2021

Table of Contents

[10]:
```
%load_ext autoreload
%autoreload 2
```

[11]:
```
#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
```

```
from torch.nn import functional as F
from sklearn.datasets import fetch_california_housing
```

[47]:
```
from IPython.display import Image
from IPython.core.display import HTML
RESULTS_PATH = "/home/vs428/Documents/deeplearning_cpsc663/Assignment_3/results"
```

1 Problem 1

1.1 2

[3]:
```
# import sample data
housing = fetch_california_housing()
m,n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m,1)), housing.data]
X = torch.Tensor(housing['data'])
y = torch.Tensor(housing['target']).unsqueeze(1)
```

[4]:
```
# create the weight vector
w_init = torch.randn(8,1,requires_grad=True)
```

[5]:
```
# TO DO:
# a) calculate closed form gradient with respect to the weights
closed_w_grad = (2 * X.T @ X @ w_init) - (2 * X.T @ y)
print(closed_w_grad)
```

```
tensor([[-1.9356e+08],
        [-1.2805e+09],
        [-2.6385e+08],
        [-5.3455e+07],
        [-1.1534e+11],
        [-1.8394e+08],
        [-1.7675e+09],
        [ 5.9536e+09]], grad_fn=<SubBackward0>)
```

[6]:
```
y.shape
```

[6]:
```
torch.Size([20640, 1])
```

[7]:
```
# b) calculate gradient with respect to the weights w using autograd # first␣
 ↪create the activation function
X_bar = X @ w_init
loss = ((y - X_bar)**2).sum()
loss.backward()
print(w_init.grad)
```

```
tensor([[-1.9356e+08],
        [-1.2805e+09],
        [-2.6385e+08],
        [-5.3455e+07],
        [-1.1534e+11],
        [-1.8394e+08],
        [-1.7675e+09],
        [ 5.9536e+09]])
```

[8]:
```
# c) check that the two are equal
torch.abs(closed_w_grad - w_init.grad) < 10000
```

[8]:
```
tensor([[ True],
        [ True],
        [ True],
        [ True],
        [False],
        [ True],
        [ True],
        [ True]])
```

2 Problem 2

Done in Tex file.

3 Problem 3

3.1 2.

One simple example of a corruption process could be adding Gaussian noise with varying standard deviations. An equation for this is below:

$$C(\hat{x}|x) = x + \mathcal{N}(0, 0.1) \tag{1}$$

This function is implemented below on $y = x^2$

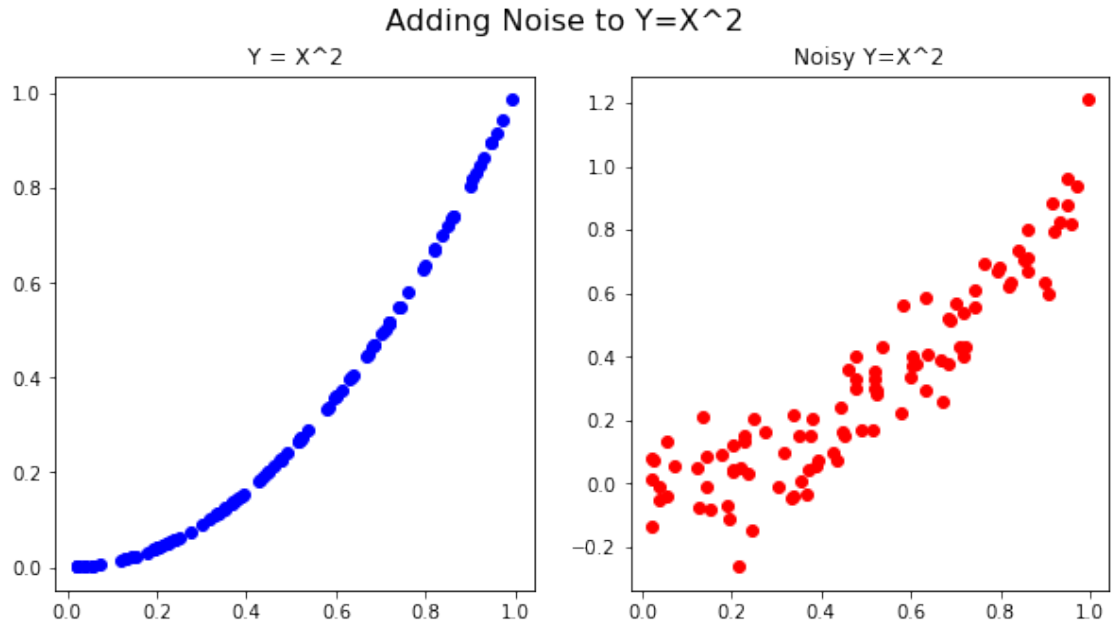
[17]:
```
import numpy as np

def addGNoise(data, mu=0,sigma=0.1):
    noise = np.random.default_rng().normal(mu, sigma, data.shape)
    return data + noise
```

[25]:
```
x = np.random.rand(100)
y = x**2
```

```
y_noisy = addGNoise(x**2)

fig, axs = plt.subplots(1,2, figsize=(10,5))
fig.suptitle("Adding Noise to Y=X^2", fontsize=16)
axs[0].scatter(x,y, c="blue")
axs[0].set_title("Y = X^2")
axs[1].scatter(x,y_noisy, c="red")
axs[1].set_title("Noisy Y=X^2")
plt.show()
```



3.2 3.

[10]:
```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.lin1 = nn.Linear(784, 2)
        self.lin2 = nn.Linear(2,784)


    def encode(self, x):
        x = self.lin1(x)
        return x

    def decode(self, z):
        z = torch.sigmoid(self.lin2(z))
```

```
        return z

    def forward(self, x):
        z = self.encode(x)
        return self.decode(z)
```

[11]:
```
# tells PyTorch to use an NVIDIA GPU, if one is available.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

[11]:
```
device(type='cuda')
```

[61]:
```
training_parameters = {
    "img_size": 28,
    "n_epochs": 24,
    "batch_size": 64,
    "learning_rate":1e-3
}
```

[62]:
```
kwargs = {'num_workers': 1, 'pin_memory': True} if torch.cuda.is_available()␣
 ↪else {}
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=True, download=True,
                   transform=transforms.ToTensor()),
    batch_size=training_parameters['batch_size'], shuffle=True, **kwargs)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=False, transform=transforms.ToTensor()),
    batch_size=training_parameters['batch_size'], shuffle=True, **kwargs)
```

[63]:
```
def train_autoencoder(model,loss_fn, optimizer, train_loader, test_loader):
    """
    This is a standard training loop, which leaves some parts to be filled in.
    INPUT:
    :param model: an untrained pytorch model
    :param loss_fn: e.g. Cross Entropy loss of Mean Squared Error.
    :param optimizer: the model optimizer, initialized with a learning rate.
    :param training_set: The training data, in a dataloader for easy iteration.
    :param test_loader: The testing data, in a dataloader for easy iteration.
    """
    num_epochs = 100 # obviously, this is too many. I don't know what this␣
 ↪author was thinking.
    for epoch in range(num_epochs):
        # loop through each data point in the training set
        for data, targets in train_loader:
            optimizer.zero_grad()

            # run the model on the data
```

```
            data = torch.reshape(torch.squeeze(data), (-1,784))
            model_input = data.to(device)
            out = model(model_input)

            # Calculate the loss
            loss = loss_fn(out.cpu(),data)

            # Find the gradients of our loss via backpropogation
            loss.backward()

            # Adjust accordingly with the optimizer
            optimizer.step()

        # Give status reports every 100 epochs
        if epoch % 5==0:
            print(f" EPOCH {epoch}. Progress: {epoch/num_epochs*100}%. ")
            print(f" Reconstruction Loss: {loss.data}")

    # Final model acc check
    print(f" EPOCH {num_epochs}. Progress: 100%. ")
    print(f" Reconstruction Loss: {loss.data}")
```

[64]:
```
# initialize the model (adapt this to each model)
autoenc = Autoencoder().to(device)
# initialize the optimizer, and set the learning rate
adam_opt = torch.optim.Adam(autoenc.parameters(),␣
 ↪lr=training_parameters['learning_rate'])
loss_fn = torch.nn.MSELoss()
```

[65]:
```
train_autoencoder(autoenc, loss_fn, adam_opt, train_loader, test_loader)
```

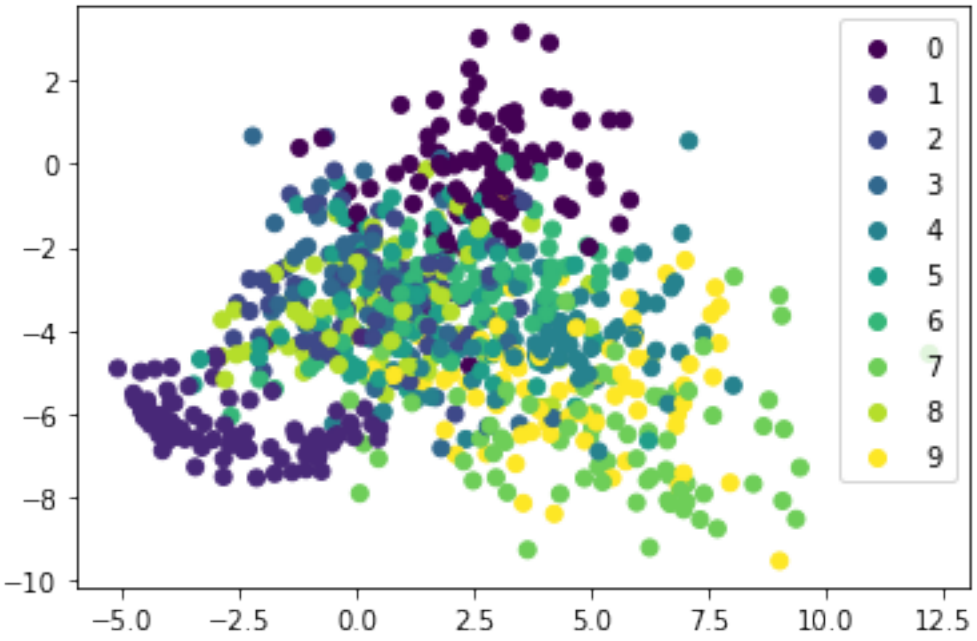
```
EPOCH 0. Progress: 0.0%.
Reconstruction Loss: 0.06711838394403458
EPOCH 5. Progress: 5.0%.
Reconstruction Loss: 0.06170640513300896
EPOCH 10. Progress: 10.0%.
Reconstruction Loss: 0.05072200670838356
EPOCH 15. Progress: 15.0%.
Reconstruction Loss: 0.05124974250793457
EPOCH 20. Progress: 20.0%.
Reconstruction Loss: 0.05451633408665657
EPOCH 25. Progress: 25.0%.
Reconstruction Loss: 0.053853053599596024
EPOCH 30. Progress: 30.0%.
Reconstruction Loss: 0.052599091082811356
EPOCH 35. Progress: 35.0%.
Reconstruction Loss: 0.05498860776424408
```

```
EPOCH 40. Progress: 40.0%.
Reconstruction Loss: 0.04880731552839279
EPOCH 45. Progress: 45.0%.
Reconstruction Loss: 0.05510849878191948
EPOCH 50. Progress: 50.0%.
Reconstruction Loss: 0.052627213299274445
EPOCH 55. Progress: 55.00000000000001%.
Reconstruction Loss: 0.05164170265197754
EPOCH 60. Progress: 60.0%.
Reconstruction Loss: 0.0596851222217083
EPOCH 65. Progress: 65.0%.
Reconstruction Loss: 0.055819448083639145
EPOCH 70. Progress: 70.0%.
Reconstruction Loss: 0.054233212023973465
EPOCH 75. Progress: 75.0%.
Reconstruction Loss: 0.054976437240839005
EPOCH 80. Progress: 80.0%.
Reconstruction Loss: 0.04565049707889557
EPOCH 85. Progress: 85.0%.
Reconstruction Loss: 0.05807165056467056
EPOCH 90. Progress: 90.0%.
Reconstruction Loss: 0.06087760627269745
EPOCH 95. Progress: 95.0%.
Reconstruction Loss: 0.050877660512924194
EPOCH 100. Progress: 100%.
Reconstruction Loss: 0.05272308737039566
```

[66]:
```
embeddings = []
for data, targets in train_loader:
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    data = data.to(device)
    targets = targets.to(device)

    model_input = data.view(data.shape[0], -1)
    embeddings.append((autoenc.encode(model_input), targets))
```

[67]:
```
i = 0
max = np.ceil(1500/128)
for gpu_data, gpu_labels in embeddings:
    data = gpu_data.cpu().detach()
    labels = gpu_labels.cpu().detach()
    scatter = plt.scatter(data[:,0], data[:,1], c=labels)
    i += 1
    if i == max:
        break
plt.legend(*scatter.legend_elements())
plt.show()
```



We can see that the AE learned some of the salient features of the dataset since we see a separation between the different digits, especially those that are very different from one another. For example, 0 and 1 are far apart, since they are very different. Same with 1 and 9, while 0 and 9 are closer since they are more similar. One of the issues we see here is that there isn't clear separation between the other points, so further optimization of the AE can be done. Furthermore, since we didn't use a VAE, we won't be able to get very reasonable examples if we sample from this latent space.

4 Problem 4

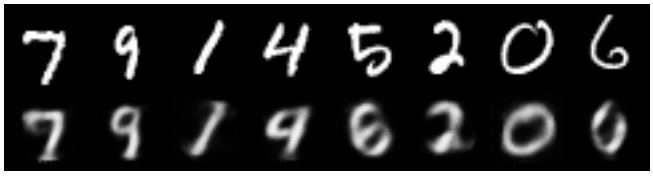
4.1 2: MNIST

[ ]:
```
!python vae.py
```

[ ]:
```
!python vae.py --learning-rate 0.0005 --epochs 30 --batch-size 128
```

[48]:
```
# first VAE reconstruction
first_VAE_recon = "/vae_results/mnist/reconstruction_1.png"
Image(filename = RESULTS_PATH + first_VAE_recon, width=500, height=500,␣
 ↪unconfined=True)
```

[48]:



[50]:
```
# last VAE reconstruction
last_VAE_recon = "/vae_results/mnist/reconstruction_10.png"
Image(filename = RESULTS_PATH + last_VAE_recon, width=500, height=500,␣
 ↪unconfined=True)
```

[50]:

We can see that the values are much clearer after 10 epochs than 1 epoch. The reconstruction loss decreased and we are more faithful to our original image after 10 epochs.

4.2 3: Fashion-MNIST

[54]:
```
# first VAE reconstruction
first_VAE_recon = "/vae_results/fashion_mnist/sample_1.png"
Image(filename = RESULTS_PATH + first_VAE_recon, width=500, height=500,␣
 ↪unconfined=True)
```

[54]:

[52]:
```
# best VAE reconstruction
best_reconstruction_VAE = "/vae_results/fashion_mnist/best_sample_30.png"
Image(filename = RESULTS_PATH + best_reconstruction_VAE, width=500, height=500,␣
 ↪unconfined=True)
```

[52]:

We performed a search over some of the hyperparameters of the VAE model, including learning rate, batch size, and number of epochs, while learning the network architecture constant. The best performing model has the attributes below:

Hyperparameter	Value
epochs	30
batch-size	128
learning-rate	0.0005

5 Problem 5

5.1 GAN: 1-3

[1]:
```
#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
```

[2]:
```
from gan import *
```

[3]:
```
# tells PyTorch to use an NVIDIA GPU, if one is available.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

[3]:
```
device(type='cuda')
```

[4]:
```
# loading the dataset
training_parameters["img_size"] = 28
training_parameters["n_epochs"] = 100
training_parameters["batch_size"] = 100

# 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␣
 ↪edge of the image to have this size
    transforms.ToTensor(),
])

# load the dataset
train_loader = torch.utils.data.DataLoader(
    datasets.FashionMNIST(
        './data', # specifies the directory to download the datafiles to,␣
 ↪relative to the location of the notebook.
        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 to:
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'
}
```

[ ]:
```
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␣
↪the dataset length, this may change on the last epoch.
        lossG = train_generator(batch_size)
        G_loss.append(lossG)
        lossD = train_discriminator(batch_size, imgs.reshape(batch_size, -1),␣
↪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␣
↪things up
                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(batch_size, 28, 28)
                generated_data = generator(noise).cpu().view(batch_size, 28, 28)

                # display generated images
                batch_sqrt = int(training_parameters['batch_size'] ** 0.5)
                fig, ax = plt.subplots(batch_sqrt, batch_sqrt, figsize=(15, 15))
                for i, x in enumerate(generated_data):
                    #ax[math.floor(i / batch_sqrt)][i % batch_sqrt].set_title(
                        # label_descriptions[int(fake_labels[i].item())]) #␣
↪TODO: In 5.4 you can uncomment this line to add labels to images.
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].imshow(x.
↪detach().numpy(), interpolation='nearest',
                                                                          ␣
↪cmap='gray')
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].get_xaxis().
↪set_visible(False)
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].get_yaxis().
↪set_visible(False)
                # plt.show()
                fig.savefig(f"./results/CGAN_Generations_Epoch_{epoch}")
                print(
                    f"Epoch {epoch}: loss_d: {torch.mean(torch.
↪FloatTensor(D_loss))}, loss_g: {torch.mean(torch.FloatTensor(G_loss))}")
```

5.1.1 Analysis of GAN Hyperparameters

We modified a number of hyperparameters and training schemes for GAN training. We recorded the first and last epoch reconstruction values for all of these in the `results/gan_results` folder in the project directory. We will describe what we changed and how it impacted our results here.

We modified the the following **5** parameters: generator/discriminator learning rates, doubling training of either generator or discriminator, β_1 for the Adam optimizer. Our baseline is just increasing the epochs with the given learning rates and other parameters.

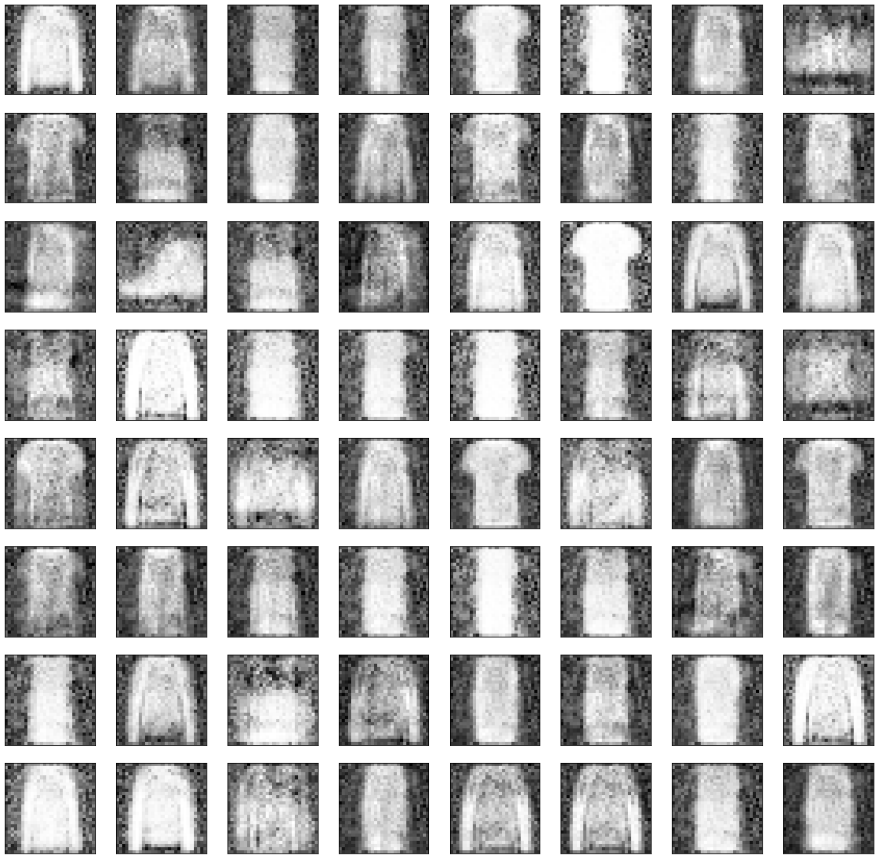
[61]:
```
gan0 = "/gan_results/genlr_00005_disclr_00005_epoch36/CGAN_Generations_Epoch_23.
↪png"
Image(filename = RESULTS_PATH + gan1, width=500, height=500, unconfined=True)
```

[61]:

Starting with changing just the learning rates, we first tried to lower just the generator learning rate, as we saw that its loss was quite low, so we wanted to slow it down to improve the minimax game. We get:

[62]:
```
gan1 = "/gan_results/genlr_0001_disclr_0002/CGAN_Generations_Epoch_23.png"
Image(filename = RESULTS_PATH + gan1, width=500, height=500, unconfined=True)
```

[62]:



It doesn't seem like there is much difference with lowering the learning rates. One thing we see is that there are fewer shoe shaped images, which might have to do with mode collapse, though I'm not sure how learning rate has to impact this. It is very possible that this is just random initialization. Next, reducing both the learning rates shows that we just needed lower learning rates. This gives us clearer generations.

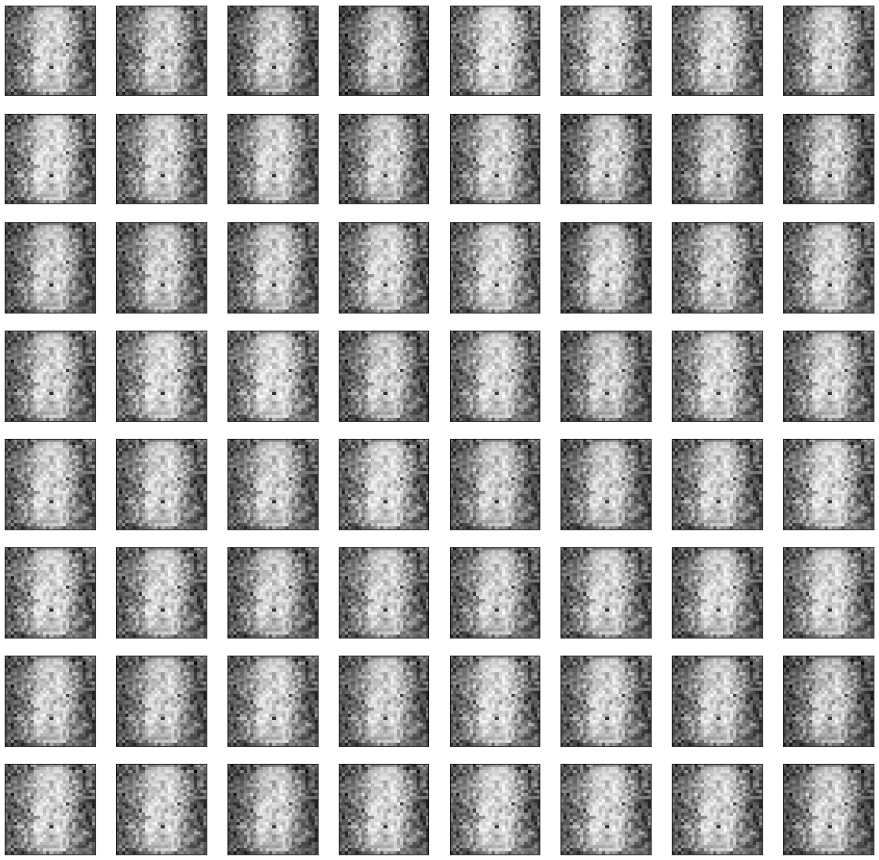
[59]:
```
gan2 = "/gan_results/genlr_0001_disclr_0001/CGAN_Generations_Epoch_23.png"
Image(filename = RESULTS_PATH + gan2, width=500, height=500, unconfined=True)
```

[59]:

Changing Training Scheme We next played with changing the training scheme. When we trained the generator twice, we receive very weird values that are difficult to explain. The losses seems to look normal, but the gradients may have blown up/collapsed. We show both the first and last epoch. In this case, the first epoch seems like it could be focusing on the correct areas of the image for shirts, but then it falls apart by the last epoch.

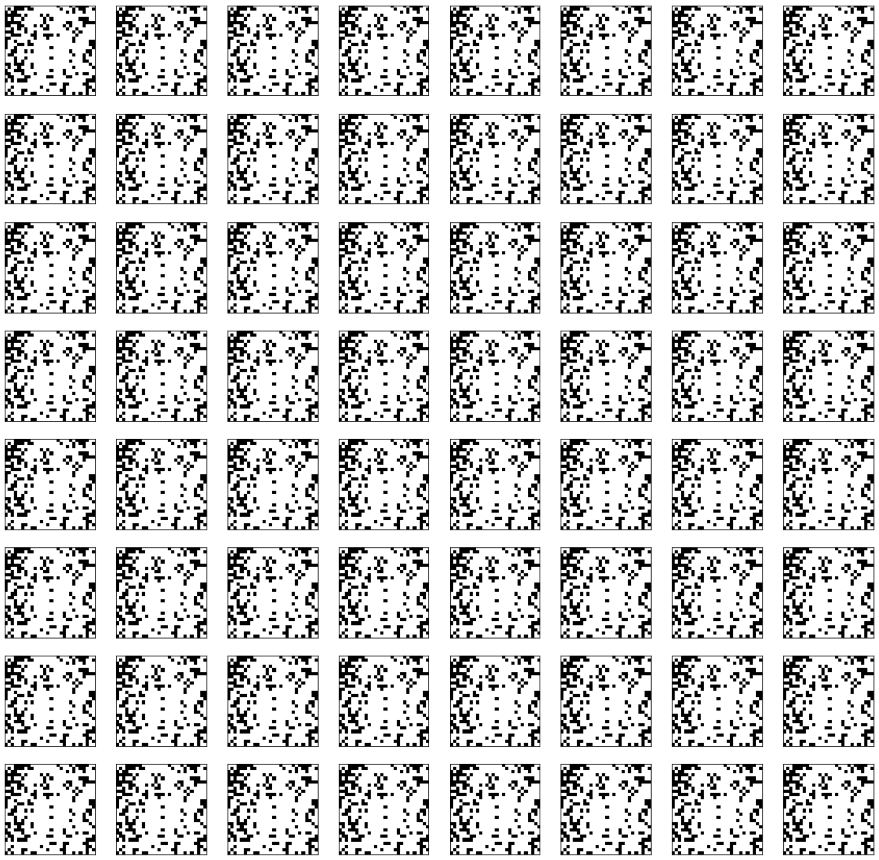
[63]:
```
gan3 = "/gan_results/double_gen/CGAN_Generations_Epoch_1.png"
Image(filename = RESULTS_PATH + gan3, width=500, height=500, unconfined=True)
```

[63]:



[66]:
```
gan4 = "/gan_results/double_gen/CGAN_Generations_Epoch_10.png"
Image(filename = RESULTS_PATH + gan4, width=500, height=500, unconfined=True)
```

[66]:



However, changing the scheme to train the discriminator twice as often as the generator seemed to provide clearer pictures than changing the learning rates.

[65]:
```
gan5 = "/gan_results/double_disc/CGAN_Generations_Epoch_23.png"
Image(filename = RESULTS_PATH + gan5, width=500, height=500, unconfined=True)
```

[65]:

Our best performing model occured when we messed with the β_1 values in the Adam optimizer

[67]:
```
gan5 = "/gan_results/betas0.5_disclr_0006_genlr_0008/CGAN_Generations_Epoch_67.
 ↪png"
Image(filename = RESULTS_PATH + gan5, width=500, height=500, unconfined=True)
```

[67]:

Instead of the fuzzy short and long sleeve shirts, we see clear separation between short and llong sleeves. However, we still have mode collapse as we don't see any shoes at all. This is resolved in the CGAN

5.2 4-5: Conditional GAN

[ ]:

[3]:
```
from gan import *
```

[4]:
```
# tells PyTorch to use an NVIDIA GPU, if one is available.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
device
```

[4]:
```
device(type='cuda')
```

[5]:
```
# loading the dataset
training_parameters["img_size"] = 28
training_parameters["n_epochs"] = 35
training_parameters["batch_size"] = 100

# 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␣
 ↪edge of the image to have this size
    transforms.ToTensor(),
])

# load the dataset
train_loader = torch.utils.data.DataLoader(
    datasets.FashionMNIST(
        './data', # specifies the directory to download the datafiles to,␣
 ↪relative to the location of the notebook.
        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 to:
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'
}
```

[ ]:
```
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␣
↪the dataset length, this may change on the last epoch.
        lossG = train_generator(batch_size)
        G_loss.append(lossG)
        lossD = train_discriminator(batch_size, imgs.reshape(batch_size, -1),␣
↪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␣
↪things up
                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(batch_size, 28, 28)
#                 generated_data = generator(noise).cpu().view(batch_size, 28,␣
↪28)

                # display generated images
                batch_sqrt = int(training_parameters['batch_size'] ** 0.5)
                fig, ax = plt.subplots(batch_sqrt, batch_sqrt, figsize=(15, 15))
                for i, x in enumerate(generated_data):
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].set_title(
                        label_descriptions[int(fake_labels[i].item())]) # TODO:␣
↪In 5.4 you can uncomment this line to add labels to images.
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].imshow(x.
↪detach().numpy(), interpolation='nearest',
                                                                          ␣
↪cmap='gray')
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].get_xaxis().
↪set_visible(False)
                    ax[math.floor(i / batch_sqrt)][i % batch_sqrt].get_yaxis().
↪set_visible(False)
                # plt.show()
                fig.savefig(f"./results/CGAN_Generations_Epoch_{epoch}")
                print(
                    f"Epoch {epoch}: loss_d: {torch.mean(torch.
↪FloatTensor(D_loss))}, loss_g: {torch.mean(torch.FloatTensor(G_loss))}")
```

5.2.1 Analysis of CGAN Hyperparameters

Using the best parameters from the GAN and changing the GAN into a conditional GAN allowed us to fix the mode collapse issue as seem below

[69]:
```
gan5 = "/cgan_results/CGAN_Generations_Epoch_34.png"
Image(filename = RESULTS_PATH + gan5, width=500, height=500, unconfined=True)
```

[69]:

In this case, we can cleary see that the pants, shirts, and shoes are all relatively clear, as compared to our best performing GAN architectures. However, unlike the GAN, we don't end up with only tops. We also see handbags and shoes. This resolved the mode collapse issue by demonstrating a better relative abundance of each class. However, compared to the best VAE, we still have a much blurrier picture. There are clearer edge separations in the CGAN than the VAE, but the images themselves sometimes are completely obfuscated, like in the third to last picture on the bottom row of the sandal. In this case, we might prefer the VAE, though empirically, it seems that GANs perform better, so maybe this is an issue of not searching the parameter space well enough.

[ ]: