Anil Poonai

Github link for files and code: https://github.com/DevonARP/DeepLearning_A3

Problem 1: Part A

CNN's are better at identifying images as each layer in it's architecture helps extract prominent features to figure out what the object in the image is, this is also helped by the feed-forward model it follows.

They are also really good at reducing the dimensions of the data, this is also caused by the layers filtering the image but it significantly reduces the computations and time needed to classify an image.

Problem 1: Part B

RNN's are typically better at dealing with temporal or sequential data, in this regard if we're dealing with text or speech, which is just sequential words at different points of time, an RNN would be preferred.

They are also great at getting context from the data, as it includes loops in it's architecture, this keeps it iterating through data sequences in order to grab the meaning of the data in regards to how it is used and what type of data is around it at that point in time.

Problem 2

Hidden state: $H_t = x_t - h_{t-1}$

Number of inputs: n

So lets start 2n inputs, this keeps the amount of the inputs even:

$h_{2n} = x_{2n} - h_{2n-1}$

$h_{2n} = x_{2n} - x_{2n-1} + h_{2n-2}$

$h_{2n} = x_{2n} - x_{2n-1} + x_{2n-2} - h_{2n-3}$

Following the alternating sign pattern:

$h_{2n} = x_{2n} - x_{2n-1} + x_{2n-2} \ldots - x_3 + x_2 - x_1 + h_0$

I'm attaching the work I manually did below this question. I used an example with 4 inputs of consecutive numbers. I also did the example above as well.

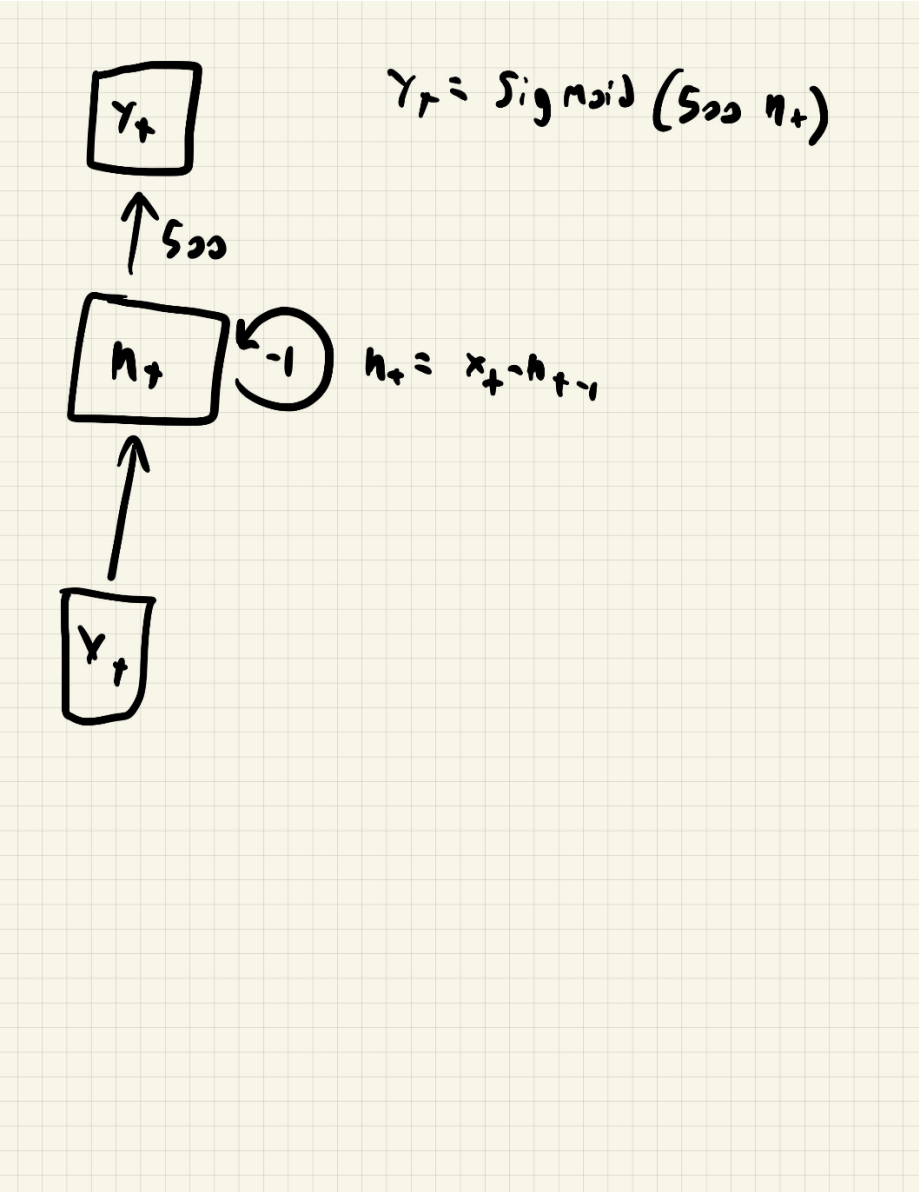
Pattern noticed:

- If the input order number is odd, the input value will end up being a negative value in the equation
- If the input order number is even, the input value will end up being a positive value in the equation

- If the values of the inputs are consecutive and the total inputs is still an even length, then the final value would be $n/2 + h_0$

Output: $y_t = \text{sigmoid}(500*h_{2n})$

$y_t = \text{sigmoid}(500*(x_{2n} - x_{2n-1} + x_{2n-2} \ldots - x_3 + x_2 - x_1 + h_0))$

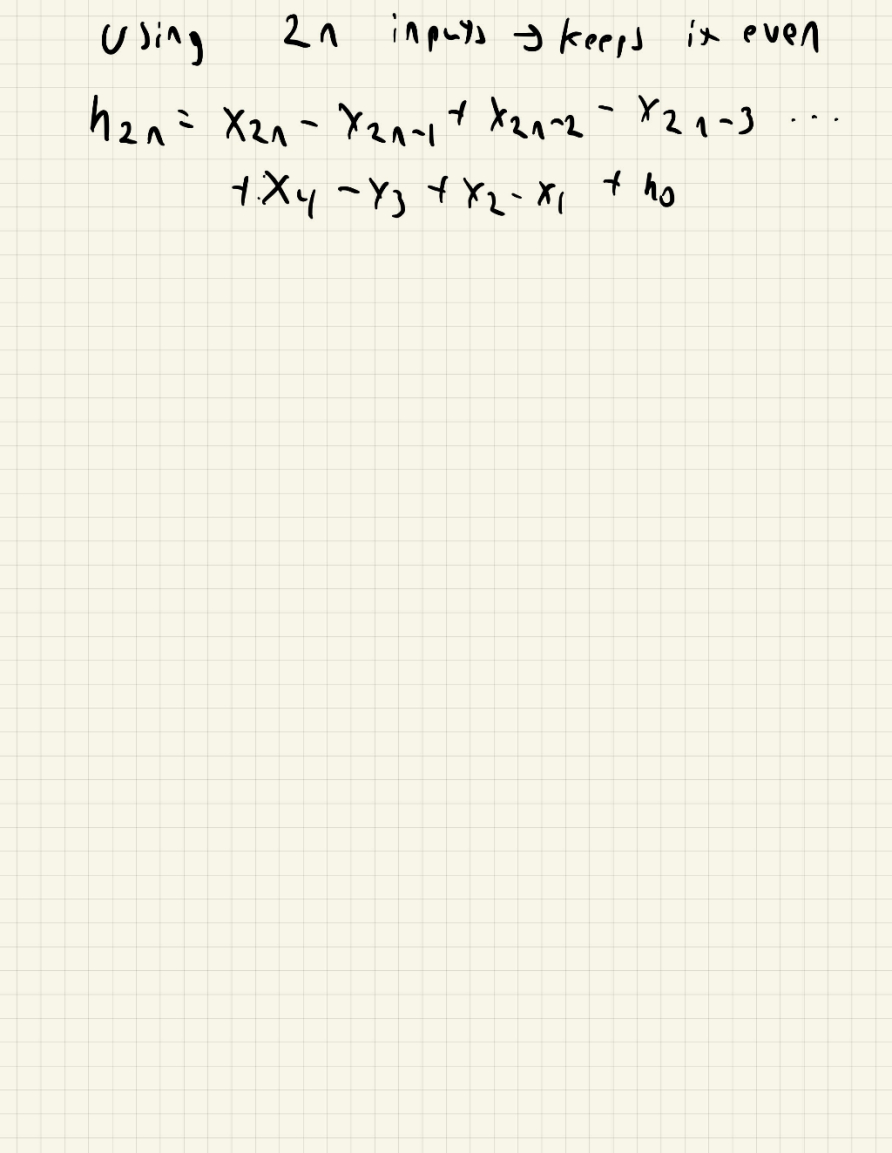


Example with 4 inputs [1,2,3,4]

$$h_4 = x_4 - h_3$$
$$= x_4 - (x_3 - h_2)$$
$$= x_4 + h_2 - x_3$$
$$= x_4 + (x_2 - h_1) - x_3$$
$$= x_4 + x_2 - (x_1 - h_0) - x_3$$
$$= x_4 + x_2 + h_0 - x_1 - x_3$$
$$= 4 + 2 + h_0 - 1 - 3$$
$$= 2 + h_0$$

- if inputs are consecutive $h_1 = \frac{1}{2} + h_0$
- all odd input orders will be subtracted & all even ones will be added

$$y_t = \text{sigmoid}\left(f_{20}\left(x_4 + x_2 + h_0 - x_1 - x_3\right)\right)$$



Problem 3

These are both in O(n) time.

We can use Longformer, which is a sparse attention, it scales linearly with the sequence length instead of quadratically, making it take up substantially less time. It uses a dilated sliding window, which lets us cover more variety of tokens but it still can lead to a loss of information. Another con is that a custom CUDA kernel is needed to make it effective as this is a sparce matrix multiplication instead of a dense one.

We can also use a matrix factorization method such as a Linformer, which makes it a linear scaling as well, it can factorize matrices into lower rank ones without losing much information. This is a great option but it does have a downside in that data is lost here, with every factorization, some data is being lost.

References:

https://towardsdatascience.com/demystifying-efficient-self-attention-b3de61b9b0fb

https://huggingface.co/docs/transformers/model_doc/longformer

https://shubhamg.in/nlp/transformer/review/longformer/2020/05/11/longformer.html

https://appliedsingularity.com/2022/05/31/nlp-tutorials-part-21-linformer-self-attention-with-linear-complexity/

https://arxiv.org/pdf/2111.14556.pdf

https://www.researchgate.net/publication/347999026_A_Transformer_Self-Attention_Model_for_Time_Series_Forecasting

https://stackoverflow.com/questions/65703260/computational-complexity-of-self-attention-in-the-transformer-model

https://www.youtube.com/watch?v=fjJOgb-E41w

Problem 4 Part A

Code is attached

Problem 4 Part B

Code is attached

Problem 4 Part C

I notice the conditional GAN seems to be a bit better at generating the images, a GAN seems to have much more noise as shown by the MNIST Gan vs the MNIST Conditional GAN at step 5000 image.

Problem 4 Part D

I did 4 different examples, 2 are on the Fashion dataset and 2 are on MNIST. They each have one Conditional GAN and one GAN being used on the data. The key difference I found is the randomness of what's generated, the Conditional GAN would be consistent and specify what type of object or number it would generate while the regular GAN would not.

hw3q4ab-fashion-gan

November 22, 2023

Fashion GAN

[ ]:
```
import torch
from torchvision import datasets
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import os
from matplotlib.pyplot import imsave
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

[ ]:
```
# A transform to convert the images to tensor and normalize their RGB values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5], std=[0.5])]
)

data = datasets.FashionMNIST(root='../data/', train=True, transform=transform,␣
 ↪download=True) #Uses FashionMNIST dataset now

batch_size = 64
data_loader = DataLoader(dataset=data, batch_size=batch_size, shuffle=True,␣
 ↪drop_last=True)
```

```
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 [00:03<00:00, 8704686.20it/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, 172394.90it/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 [00:01<00:00, 3175625.37it/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, 6623397.85it/s]

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

[ ]:
```python
def to_onehot(x, num_classes=10):
    assert isinstance(x, int) or isinstance(x, (torch.LongTensor, torch.cuda.
 ↪LongTensor))
    if isinstance(x, int):
        c = torch.zeros(1, num_classes).long()
        c[0][x] = 1
    else:
        x = x.cpu()
        c = torch.LongTensor(x.size(0), num_classes)
        c.zero_()
        c.scatter_(1, x, 1) # dim, index, src value
    return c
```

[ ]:
```python
to_onehot(3)
```

[ ]:
```
tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]])
```

[ ]:
```python
def get_sample_image(G, DEVICE, n_noise=100):
    img = np.zeros([280, 280])
    for j in range(10):
```

```
        c = torch.zeros([10, 10]).to(DEVICE)
        c[:, j] = 1
        z = torch.randn(10, n_noise).to(DEVICE)
        y_hat = G(z,c).view(10, 28, 28)
        result = y_hat.cpu().data.numpy()
        img[j*28:(j+1)*28] = np.concatenate([x for x in result], axis=-1)
    return img
```

0.1 Architecture

We now instantiate the generator and discriminator architectures. The generator takes a random noise vector and a one hot encoded label as input and produces an image. The discriminator takes an image and a one hot encoded label as input and produces a single value between 0 and 1. The discriminator is trained to output 1 for real images and 0 for fake images. The generator is trained to fool the discriminator by outputting images that look real.

I edited the number of classes from being included in the models, this will make the model a GAN instead of a Conditional GAN.

[ ]:
```
class Generator(nn.Module):
    def __init__(self, input_size=100, num_classes=10, image_size=28*28):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 128), # auxillary dimension for label; Got␣
 ↪rid of +num_classes
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = x # v: [input, label] concatenated vector; Got rid of concatenation
        y_ = self.network(v)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_
```

[ ]:
```
class Discriminator(nn.Module):
    def __init__(self, input_size=28*28, num_classes=10, num_output=1):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 512), #Got rid of +num_classes
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, num_output),
            nn.Sigmoid(),
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = x # v: [input, label] concatenated vector; Got rid of concatenation
        y_ = self.network(v)
        return y_
```

[ ]:
```
MODEL_NAME = 'GAN' #Changed the name
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

D = Discriminator().to(DEVICE) # randomly intialized
G = Generator().to(DEVICE) # randomly initialized

max_epoch = 10
step = 0
n_noise = 100 # size of noise vector

criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))

# We will denote real images as 1s and fake images as 0s
# This is why we needed to drop the last batch of the data loader
all_ones = torch.ones([batch_size, 1]).to(DEVICE) # Discriminator label: real
all_zeros = torch.zeros([batch_size, 1]).to(DEVICE) # Discriminator Label: fake
```

[ ]:
```
images, class_labels = next(iter(data_loader))
class_labels_encoded = class_labels.view(batch_size, 1)
class_labels_encoded = to_onehot(class_labels_encoded).to(DEVICE)
print(class_labels[:10])
print(class_labels_encoded[:10])
```

```
tensor([3, 7, 9, 4, 4, 8, 2, 6, 9, 8])
tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
```

```
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]], device='cuda:0')
```

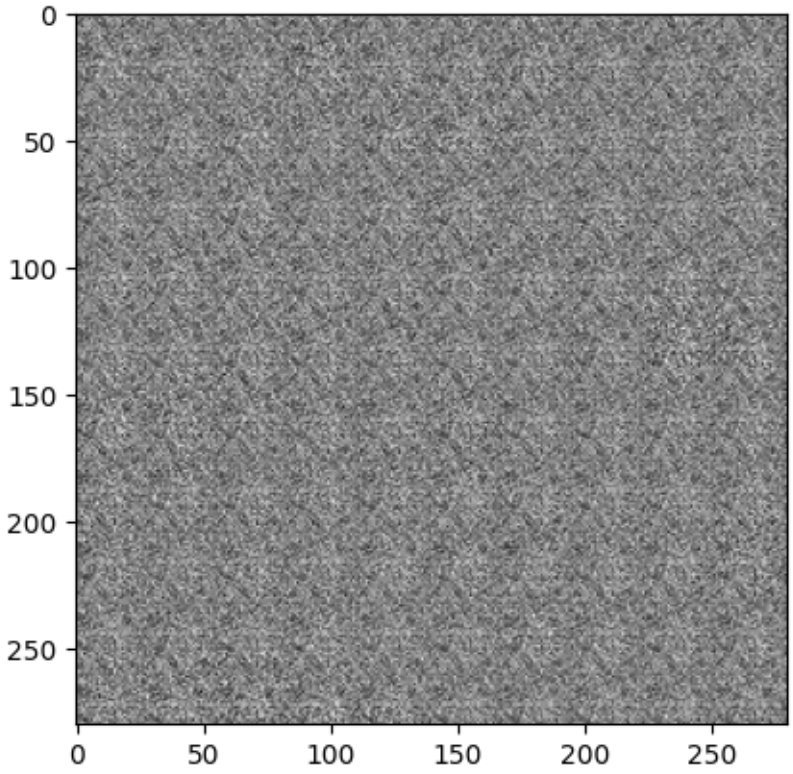
[ ]:
```
# a directory to save the generated images
if not os.path.exists('samples'):
    os.makedirs('samples')

for epoch in range(max_epoch):
    for idx, (images, class_labels) in enumerate(data_loader):
        # Training Discriminator
        x = images.to(DEVICE)
        class_labels = class_labels.view(batch_size, 1) # add singleton␣
↪dimension so batch_size x 1
        class_labels = to_onehot(class_labels).to(DEVICE)
        x_outputs = D(x, class_labels) # input includes labels
        D_x_loss = criterion(x_outputs, all_ones) # Discriminator loss for real␣
↪images

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels) # input to both␣
↪generator and discriminator includes labels
        D_z_loss = criterion(z_outputs, all_zeros) # Discriminator loss for␣
↪fake images
        D_loss = D_x_loss + D_z_loss # Total Discriminator loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        # Training Generator
        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels)
        G_loss = -1 * criterion(z_outputs, all_zeros) # Generator loss is␣
↪negative disciminator loss

        G.zero_grad()
        G_loss.backward()
        G_opt.step()

        if step % 500 == 0:
            print('Epoch: {}/{}, Step: {}, D Loss: {}, G Loss: {}'.
↪format(epoch, max_epoch, step, D_loss.item(), G_loss.item()))
```

```
        if step % 1000 == 0:
            G.eval()
            img = get_sample_image(G, DEVICE, n_noise)
            imsave('samples/{}_step{}.jpg'.format(MODEL_NAME, str(step).
↪zfill(3)), img, cmap='gray')
            G.train()
        step += 1
```

```
Epoch: 0/10, Step: 0, D Loss: 1.4167184829711914, G Loss: -0.7008953094482422
Epoch: 0/10, Step: 500, D Loss: 1.198230504989624, G Loss: -0.44864386320114136
Epoch: 1/10, Step: 1000, D Loss: 1.2353107929229736, G Loss: -0.5327345132827759
Epoch: 1/10, Step: 1500, D Loss: 1.2166428565979004, G Loss: -0.5280460119247437
Epoch: 2/10, Step: 2000, D Loss: 1.304343581199646, G Loss: -0.4828684329986572
Epoch: 2/10, Step: 2500, D Loss: 1.2642494440078735, G Loss: -0.5151442289352417
Epoch: 3/10, Step: 3000, D Loss: 1.257765531539917, G Loss: -0.5388797521591187
Epoch: 3/10, Step: 3500, D Loss: 1.4073083400726318, G Loss: -0.7102686166763306
Epoch: 4/10, Step: 4000, D Loss: 1.3161141872406006, G Loss: -0.5851697325706482
Epoch: 4/10, Step: 4500, D Loss: 1.2726058959960938, G Loss: -0.5268336534500122
Epoch: 5/10, Step: 5000, D Loss: 1.3868470191955566, G Loss: -0.5979888439178467
Epoch: 5/10, Step: 5500, D Loss: 1.3306419849395752, G Loss: -0.5476356148719788
Epoch: 6/10, Step: 6000, D Loss: 1.3461498022079468, G Loss: -0.7852540016174316
Epoch: 6/10, Step: 6500, D Loss: 1.2957665920257568, G Loss: -0.5773029327392578
Epoch: 7/10, Step: 7000, D Loss: 1.3240413665771484, G Loss: -0.658687949180603
Epoch: 8/10, Step: 7500, D Loss: 1.3452906608581543, G Loss: -0.6307313442230225
Epoch: 8/10, Step: 8000, D Loss: 1.3202133178710938, G Loss: -0.6333258152008057
Epoch: 9/10, Step: 8500, D Loss: 1.4612188339233398, G Loss: -0.6147671937942505
Epoch: 9/10, Step: 9000, D Loss: 1.390981674194336, G Loss: -0.6171368360519409
```

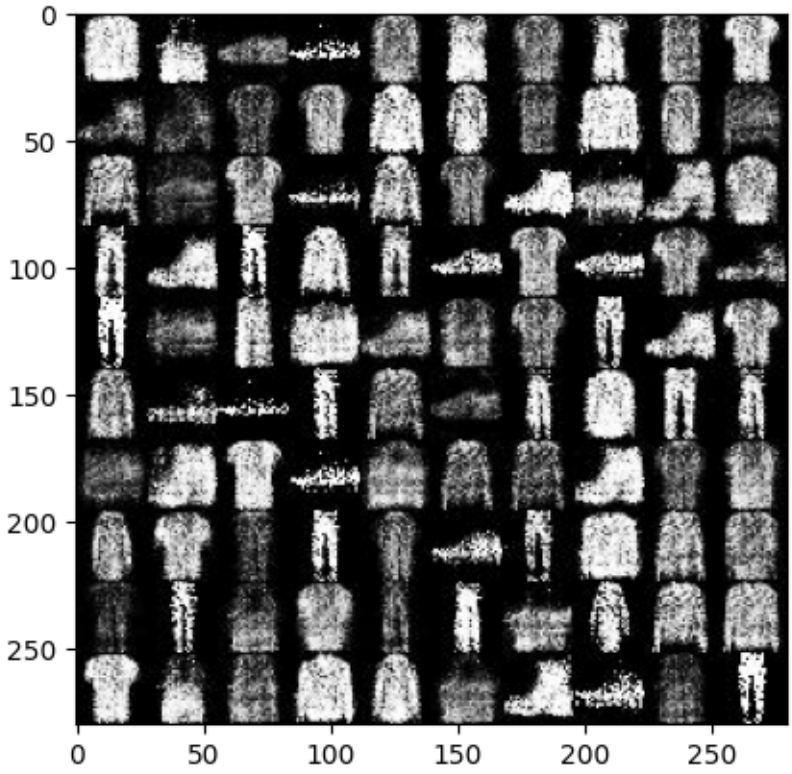
Now let's plot these images. At first, the generator just produces noise (as we expect).

[ ]:
```
img = mpimg.imread('samples/GAN_step000.jpg')
imgplot = plt.imshow(img)
plt.show()
```

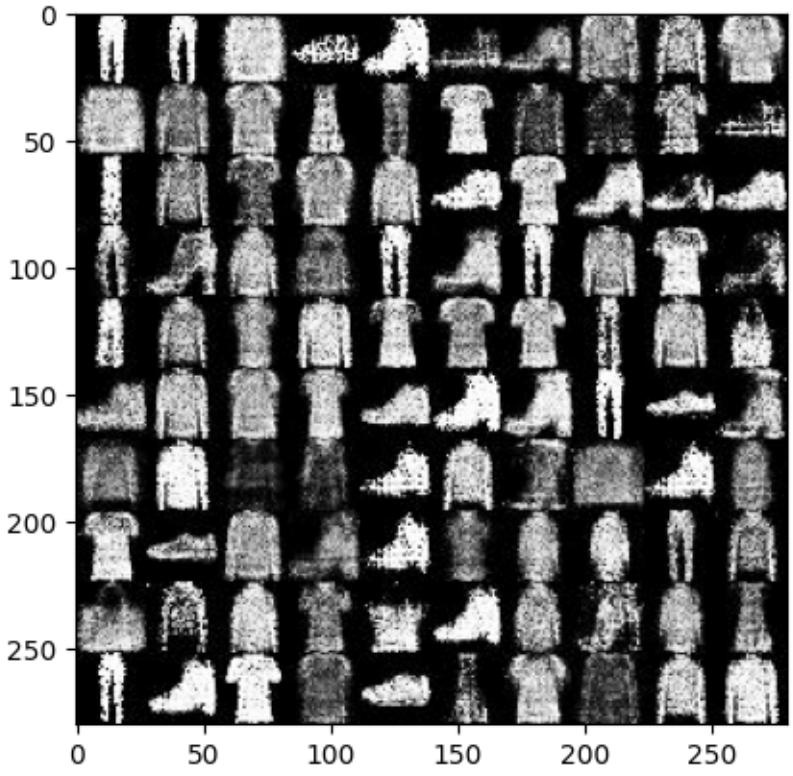


But then it gets better.

[ ]:
```
img = mpimg.imread('samples/GAN_step5000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



[ ]:
```
img = mpimg.imread('samples/GAN_step9000.jpg')
imgplot = plt.imshow(img)
plt.show()
```


0
50
100
150
200
250
0
50
100
150
200
250

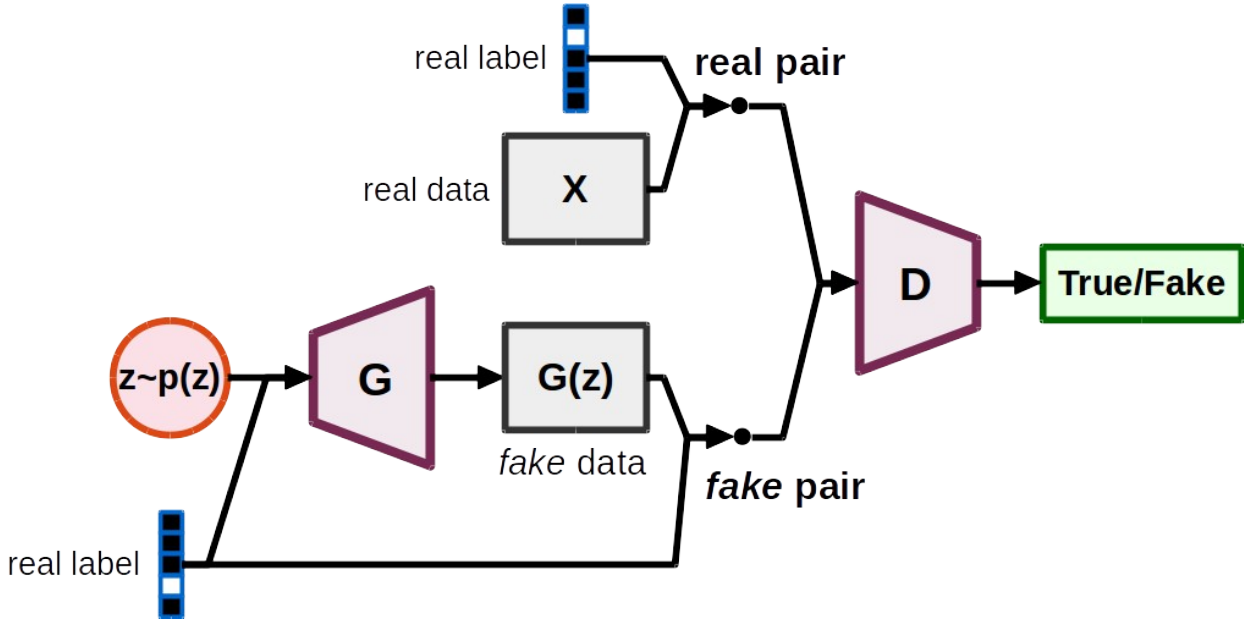
Fashion Conditional GAN

Conditional GANs

Based on the excellent tutorial here.

By formulating the process as a two-player game, Generative Adversarial Networks (GANs) can be very effective in generating realistic content. However, we may want to have more control over what is generated. Conditional GANs offer more control by letting us specify the *class* of output we want. Then we hand the generated content and the class it's supposed to be to the discriminator. The disciminator attempts to differentiate between the generated content of a certain class and the real content of a certain class.

The original paper that described conditional GANs is here.



Libaries

As always, we load lots of libraries.

```
import torch
from torchvision import datasets
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import os
from matplotlib.pyplot import imsave
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Data

For this demo, we will be using the MNIST data set. We can apply GANs to other datasets but the training process takes much longer. Our goal will be to supply random noise and a class label (e.g. a digit between 0 and 9) to the generator and produce an image of that particular digit.

```
# A transform to convert the images to tensor and normalize their RGB
values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5],
std=[0.5])]
)

data = datasets.FashionMNIST(root='../data/', train=True,
transform=transform, download=True)

batch_size = 64
data_loader = DataLoader(dataset=data, batch_size=batch_size,
shuffle=True, drop_last=True)
```

```
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 [00:00<00:00, 116392647.43it/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, 4969686.17it/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, 66057083.81it/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, 10507190.75it/s]

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

Helper Functions

We'll need several helper functions for training the conditional GAN. The first converts labels to one hot encoded vectors, we will use it to pass the desired label to the generator. The second will plot a grid of 10x10 images from the generator.

```
def to_onehot(x, num_classes=10):
    assert isinstance(x, int) or isinstance(x, (torch.LongTensor,
torch.cuda.LongTensor))
    if isinstance(x, int):
        c = torch.zeros(1, num_classes).long()
        c[0][x] = 1
    else:
        x = x.cpu()
        c = torch.LongTensor(x.size(0), num_classes)
        c.zero_()
        c.scatter_(1, x, 1) # dim, index, src value
    return c

to_onehot(3)

tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]])

def get_sample_image(G, DEVICE, n_noise=100):
    img = np.zeros([280, 280])
    for j in range(10):
        c = torch.zeros([10, 10]).to(DEVICE)
        c[:, j] = 1
        z = torch.randn(10, n_noise).to(DEVICE)
        y_hat = G(z,c).view(10, 28, 28)
        result = y_hat.cpu().data.numpy()
        img[j*28:(j+1)*28] = np.concatenate([x for x in result],
```

```
axis=-1)
    return img
```

Architecture

We now instantiate the generator and discriminator architectures. The generator takes a random noise vector and a one hot encoded label as input and produces an image. The discriminator takes an image and a one hot encoded label as input and produces a single value between 0 and 1. The discriminator is trained to output 1 for real images and 0 for fake images. The generator is trained to fool the discriminator by outputting images that look real.

```
class Generator(nn.Module):
    def __init__(self, input_size=100, num_classes=10,
image_size=28*28):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size+num_classes, 128), # auxillary
dimension for label
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = torch.cat((x, c), 1) # v: [input, label] concatenated
vector
        y_ = self.network(v)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_

class Discriminator(nn.Module):
    def __init__(self, input_size=28*28, num_classes=10,
num_output=1):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size+num_classes, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
```

```
            nn.Linear(256, num_output),
            nn.Sigmoid(),
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = torch.cat((x, c), 1) # v: [input, label] concatenated
vector
        y_ = self.network(v)
        return y_
```

Set up and Training

Now, we're ready to instantiate our models, hyperparameters, and optimizers. Since the task is so easy for MNIST, we will train for only 10 epochs. We will update the generator and discriminator in every step but often one can be trained more frequently than the other.

```
MODEL_NAME = 'ConditionalGAN'
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

D = Discriminator().to(DEVICE) # randomly intialized
G = Generator().to(DEVICE) # randomly initialized

max_epoch = 10
step = 0
n_noise = 100 # size of noise vector

criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5,
0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5,
0.999))

# We will denote real images as 1s and fake images as 0s
# This is why we needed to drop the last batch of the data loader
all_ones = torch.ones([batch_size, 1]).to(DEVICE) # Discriminator
label: real
all_zeros = torch.zeros([batch_size, 1]).to(DEVICE) # Discriminator
Label: fake

images, class_labels = next(iter(data_loader))
class_labels_encoded = class_labels.view(batch_size, 1)
class_labels_encoded = to_onehot(class_labels_encoded).to(DEVICE)
print(class_labels[:10])
print(class_labels_encoded[:10])

tensor([4, 4, 9, 2, 4, 4, 9, 8, 6, 1])
tensor([[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
```

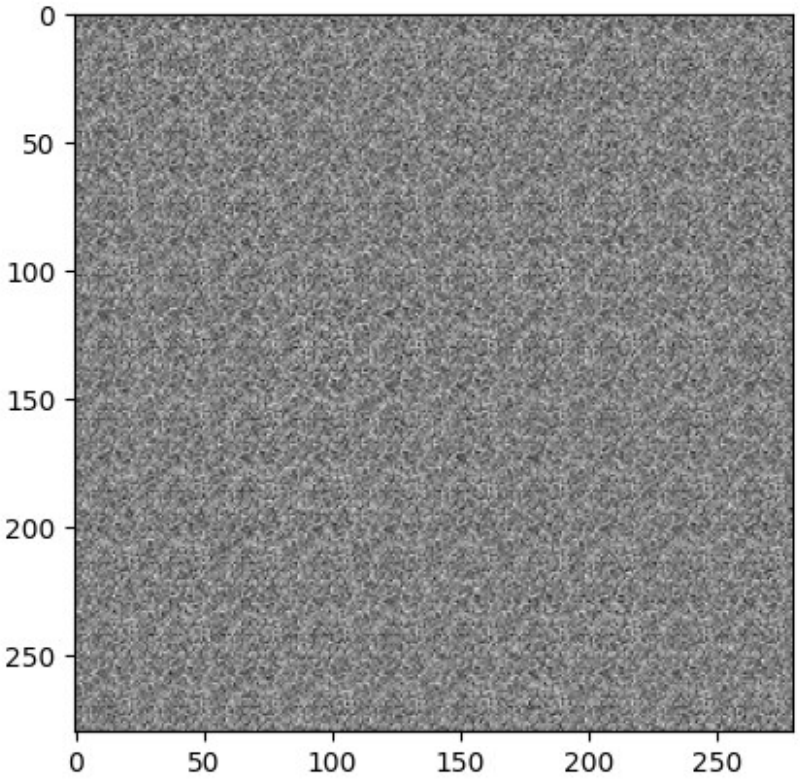
```
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]], device='cuda:0')

# a directory to save the generated images
if not os.path.exists('samples'):
    os.makedirs('samples')

for epoch in range(max_epoch):
    for idx, (images, class_labels) in enumerate(data_loader):
        # Training Discriminator
        x = images.to(DEVICE)
        class_labels = class_labels.view(batch_size, 1) # add
singleton dimension so batch_size x 1
        class_labels = to_onehot(class_labels).to(DEVICE)
        x_outputs = D(x, class_labels) # input includes labels
        D_x_loss = criterion(x_outputs, all_ones) # Discriminator loss
for real images

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels) # input to
both generator and discriminator includes labels
        D_z_loss = criterion(z_outputs, all_zeros) # Discriminator
loss for fake images
        D_loss = D_x_loss + D_z_loss # Total Discriminator loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        # Training Generator
        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels)
        G_loss = -1 * criterion(z_outputs, all_zeros) # Generator loss
is negative disciminator loss

        G.zero_grad()
        G_loss.backward()
        G_opt.step()

        if step % 500 == 0:
            print('Epoch: {}/{}, Step: {}, D Loss: {}, G Loss:
{}'.format(epoch, max_epoch, step, D_loss.item(), G_loss.item()))
```

```
        if step % 1000 == 0:
            G.eval()
            img = get_sample_image(G, DEVICE, n_noise)
            imsave('samples/{}_step{}.jpg'.format(MODEL_NAME,
str(step).zfill(3)), img, cmap='gray')
            G.train()
        step += 1
```

```
Epoch: 0/10, Step: 0, D Loss: 1.3819940090179443, G Loss: -
0.6990505456924438
Epoch: 0/10, Step: 500, D Loss: 0.9222322106361389, G Loss: -
0.36316925287246704
Epoch: 1/10, Step: 1000, D Loss: 1.306557059288025, G Loss: -
0.30022862553596497
Epoch: 1/10, Step: 1500, D Loss: 1.2047450542449951, G Loss: -
0.4338669180870056
Epoch: 2/10, Step: 2000, D Loss: 1.383216142654419, G Loss: -
0.5744942426681519
Epoch: 2/10, Step: 2500, D Loss: 1.3397421836853027, G Loss: -
0.6760445833206177
Epoch: 3/10, Step: 3000, D Loss: 1.4008440971374512, G Loss: -
0.6481665372848511
Epoch: 3/10, Step: 3500, D Loss: 1.3807786703109741, G Loss: -
0.47795313596725464
Epoch: 4/10, Step: 4000, D Loss: 1.3197503089904785, G Loss: -
0.6316052675247192
Epoch: 4/10, Step: 4500, D Loss: 1.4779423475265503, G Loss: -
0.6468305587768555
Epoch: 5/10, Step: 5000, D Loss: 1.3649685382843018, G Loss: -
0.722814679145813
Epoch: 5/10, Step: 5500, D Loss: 1.3152425289154053, G Loss: -
0.6868982315063477
Epoch: 6/10, Step: 6000, D Loss: 1.318251609802246, G Loss: -
0.7387551665306091
Epoch: 6/10, Step: 6500, D Loss: 1.2910850048065186, G Loss: -
0.5901193022727966
Epoch: 7/10, Step: 7000, D Loss: 1.3347728252410889, G Loss: -
0.6987571120262146
Epoch: 8/10, Step: 7500, D Loss: 1.4143143892288208, G Loss: -
0.6107622385025024
Epoch: 8/10, Step: 8000, D Loss: 1.3565864562988281, G Loss: -
0.6859310865402222
Epoch: 9/10, Step: 8500, D Loss: 1.3550758361816406, G Loss: -
0.640485942363739
Epoch: 9/10, Step: 9000, D Loss: 1.3872662782669067, G Loss: -
0.6289292573928833
```

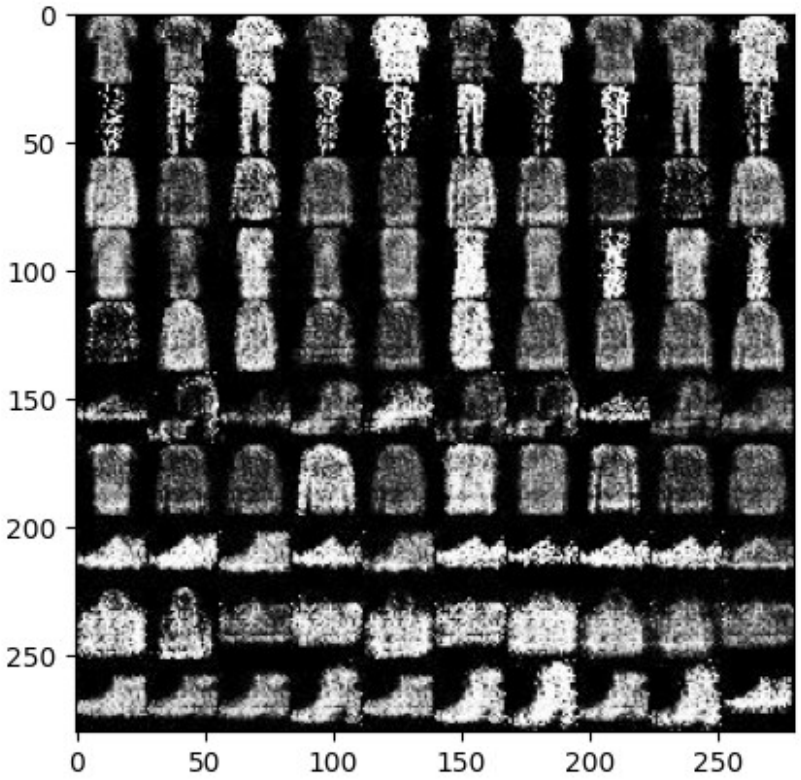
Now let's plot these images. At first, the generator just produces noise (as we expect).

```
img = mpimg.imread('samples/ConditionalGAN_step000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



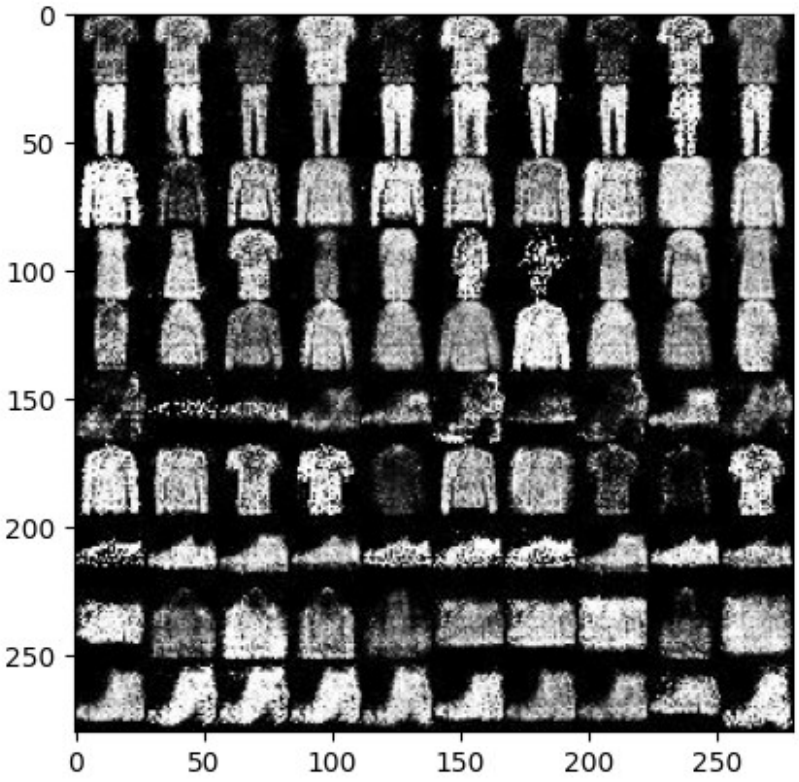
But then it gets better.

```
img = mpimg.imread('samples/ConditionalGAN_step5000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



In fact, if we don't look too closely, we can recognize the numbers it produces.

```
img = mpimg.imread('samples/ConditionalGAN_step9000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



And by the time we're done training, even the worst images look like messy handwriting!

hw3q4c-mnist-gan

November 22, 2023

MNIST GAN

[1]:
```
import torch
from torchvision import datasets
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import os
from matplotlib.pyplot import imsave
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

[2]:
```
# A transform to convert the images to tensor and normalize their RGB values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5], std=[0.5])]
)

data = datasets.MNIST(root='../data/', train=True, transform=transform,␣
 ↪download=True) #Uses FashionMNIST dataset now

batch_size = 64
data_loader = DataLoader(dataset=data, batch_size=batch_size, shuffle=True,␣
 ↪drop_last=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
../data/MNIST/raw/train-images-idx3-ubyte.gz
```

```
100%|      | 9912422/9912422 [00:00<00:00, 93556389.95it/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
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
../data/MNIST/raw/train-labels-idx1-ubyte.gz
```

```
100%|      | 28881/28881 [00:00<00:00, 87147981.17it/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
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
../data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 1648877/1648877 [00:00<00:00, 45346539.27it/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
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
../data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 4542/4542 [00:00<00:00, 21214397.29it/s]

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

[3]:
```python
def to_onehot(x, num_classes=10):
    assert isinstance(x, int) or isinstance(x, (torch.LongTensor, torch.cuda.
 ↪LongTensor))
    if isinstance(x, int):
        c = torch.zeros(1, num_classes).long()
        c[0][x] = 1
    else:
        x = x.cpu()
        c = torch.LongTensor(x.size(0), num_classes)
        c.zero_()
        c.scatter_(1, x, 1) # dim, index, src value
    return c
```

[4]:
```python
to_onehot(3)
```

[4]:
```
tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]])
```

[5]:
```python
def get_sample_image(G, DEVICE, n_noise=100):
    img = np.zeros([280, 280])
    for j in range(10):
        c = torch.zeros([10, 10]).to(DEVICE)
        c[:, j] = 1
        z = torch.randn(10, n_noise).to(DEVICE)
        y_hat = G(z,c).view(10, 28, 28)
        result = y_hat.cpu().data.numpy()
        img[j*28:(j+1)*28] = np.concatenate([x for x in result], axis=-1)
    return img
```

0.1 Architecture

We now instantiate the generator and discriminator architectures. The generator takes a random noise vector and a one hot encoded label as input and produces an image. The discriminator takes an image and a one hot encoded label as input and produces a single value between 0 and 1. The discriminator is trained to output 1 for real images and 0 for fake images. The generator is trained to fool the discriminator by outputting images that look real.

I edited the number of classes from being included in the models, this will make the model a GAN instead of a Conditional GAN.

[6]:
```
class Generator(nn.Module):
    def __init__(self, input_size=100, num_classes=10, image_size=28*28):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 128), # auxillary dimension for label; Got␣
 ↪rid of +num_classes
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = x # v: [input, label] concatenated vector; Got rid of concatenation
        y_ = self.network(v)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_
```

[7]:
```
class Discriminator(nn.Module):
    def __init__(self, input_size=28*28, num_classes=10, num_output=1):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 512), #Got rid of +num_classes
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, num_output),
            nn.Sigmoid(),
```

```
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = x # v: [input, label] concatenated vector; Got rid of concatenation
        y_ = self.network(v)
        return y_
```

[8]:
```
MODEL_NAME = 'GAN' #Changed the name
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

D = Discriminator().to(DEVICE) # randomly intialized
G = Generator().to(DEVICE) # randomly initialized

max_epoch = 10
step = 0
n_noise = 100 # size of noise vector

criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5, 0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5, 0.999))

# We will denote real images as 1s and fake images as 0s
# This is why we needed to drop the last batch of the data loader
all_ones = torch.ones([batch_size, 1]).to(DEVICE) # Discriminator label: real
all_zeros = torch.zeros([batch_size, 1]).to(DEVICE) # Discriminator Label: fake
```

[9]:
```
images, class_labels = next(iter(data_loader))
class_labels_encoded = class_labels.view(batch_size, 1)
class_labels_encoded = to_onehot(class_labels_encoded).to(DEVICE)
print(class_labels[:10])
print(class_labels_encoded[:10])
```

```
tensor([0, 7, 3, 0, 7, 4, 2, 8, 2, 8])
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]], device='cuda:0')
```

[10]:
```
# a directory to save the generated images
if not os.path.exists('samples'):
```

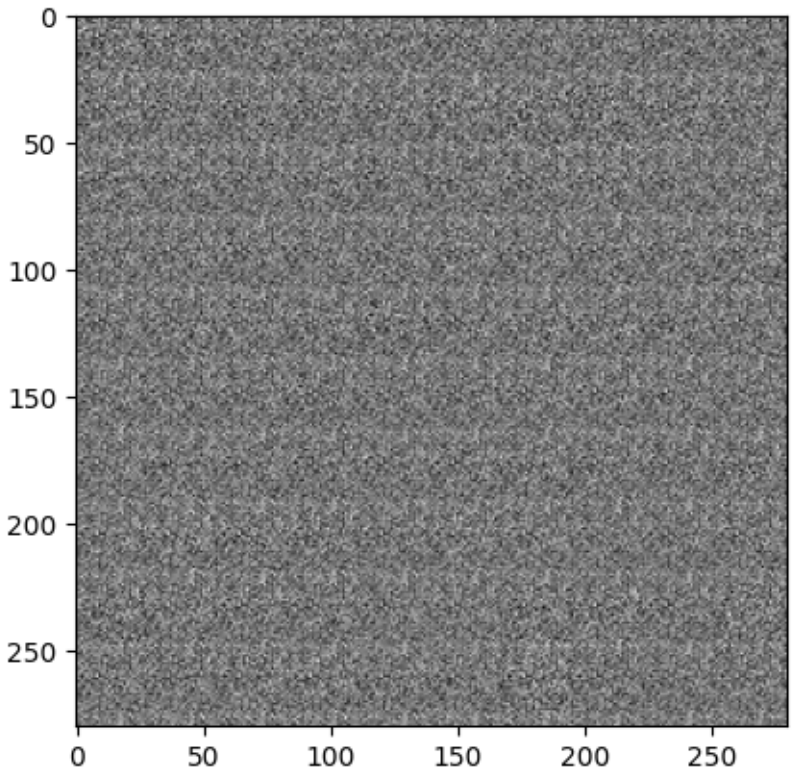
```
    os.makedirs('samples')

for epoch in range(max_epoch):
    for idx, (images, class_labels) in enumerate(data_loader):
        # Training Discriminator
        x = images.to(DEVICE)
        class_labels = class_labels.view(batch_size, 1) # add singleton␣
↪dimension so batch_size x 1
        class_labels = to_onehot(class_labels).to(DEVICE)
        x_outputs = D(x, class_labels) # input includes labels
        D_x_loss = criterion(x_outputs, all_ones) # Discriminator loss for real␣
↪images

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels) # input to both␣
↪generator and discriminator includes labels
        D_z_loss = criterion(z_outputs, all_zeros) # Discriminator loss for␣
↪fake images
        D_loss = D_x_loss + D_z_loss # Total Discriminator loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        # Training Generator
        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels)
        G_loss = -1 * criterion(z_outputs, all_zeros) # Generator loss is␣
↪negative disciminator loss

        G.zero_grad()
        G_loss.backward()
        G_opt.step()

        if step % 500 == 0:
            print('Epoch: {}/{}, Step: {}, D Loss: {}, G Loss: {}'.
↪format(epoch, max_epoch, step, D_loss.item(), G_loss.item()))

        if step % 1000 == 0:
            G.eval()
            img = get_sample_image(G, DEVICE, n_noise)
            imsave('samples/{}_step{}.jpg'.format(MODEL_NAME, str(step).
↪zfill(3)), img, cmap='gray')
            G.train()
        step += 1
```

```
Epoch: 0/10, Step: 0, D Loss: 1.4321000576019287, G Loss: -0.6797271370887756
```

```
Epoch: 0/10, Step: 500, D Loss: 1.4028527736663818, G Loss: -0.6347811222076416
Epoch: 1/10, Step: 1000, D Loss: 1.1960117816925049, G Loss: -0.5185055136680603
Epoch: 1/10, Step: 1500, D Loss: 1.1932541131973267, G Loss:
-0.39768052101135254
Epoch: 2/10, Step: 2000, D Loss: 1.2234655618667603, G Loss: -0.2393479198217392
Epoch: 2/10, Step: 2500, D Loss: 1.078147053718567, G Loss: -0.6512055397033691
Epoch: 3/10, Step: 3000, D Loss: 1.2166550159454346, G Loss: -0.6759253740310669
Epoch: 3/10, Step: 3500, D Loss: 1.2387259006500244, G Loss: -0.6251146793365479
Epoch: 4/10, Step: 4000, D Loss: 1.2493551969528198, G Loss: -0.4827510714530945
Epoch: 4/10, Step: 4500, D Loss: 1.1359643936157227, G Loss: -0.5347049236297607
Epoch: 5/10, Step: 5000, D Loss: 1.3132450580596924, G Loss: -0.3203120827674866
Epoch: 5/10, Step: 5500, D Loss: 1.1416640281677246, G Loss: -0.522498369216919
Epoch: 6/10, Step: 6000, D Loss: 1.253251552581787, G Loss: -0.7048524618148804
Epoch: 6/10, Step: 6500, D Loss: 1.1745905876159668, G Loss: -0.5737121105194092
Epoch: 7/10, Step: 7000, D Loss: 1.3402128219604492, G Loss:
-0.46168065071105957
Epoch: 8/10, Step: 7500, D Loss: 1.2298719882965088, G Loss: -0.6179988384246826
Epoch: 8/10, Step: 8000, D Loss: 1.3035839796066284, G Loss:
-0.48102065920829773
Epoch: 9/10, Step: 8500, D Loss: 1.2523696422576904, G Loss: -0.7329338788986206
Epoch: 9/10, Step: 9000, D Loss: 1.4572618007659912, G Loss: -0.9165699481964111
```

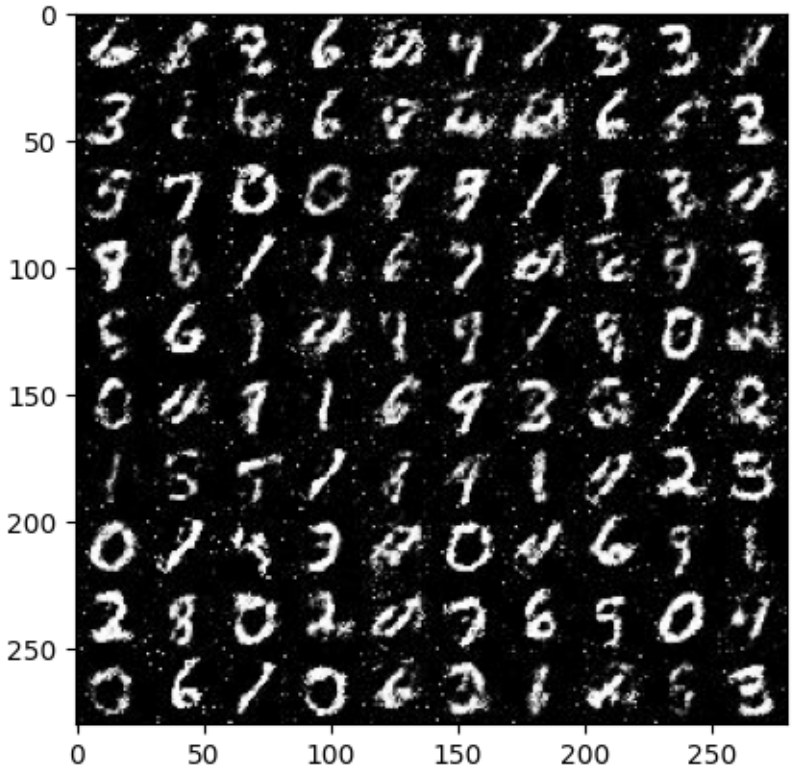
Now let's plot these images. At first, the generator just produces noise (as we expect).

[11]:
```
img = mpimg.imread('samples/GAN_step000.jpg')
imgplot = plt.imshow(img)
plt.show()
```

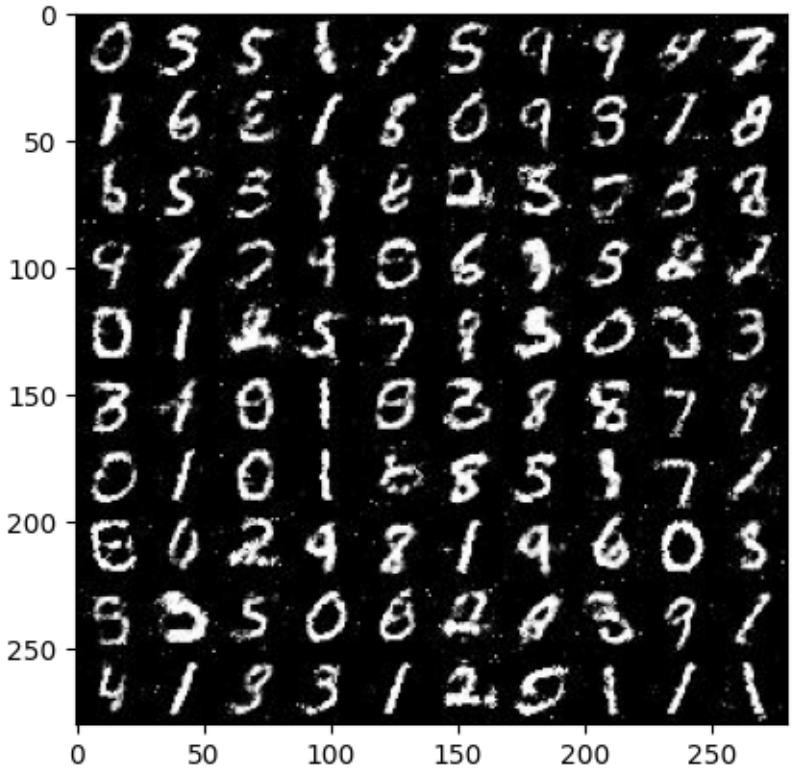


But then it gets better.

[12]:
```
img = mpimg.imread('samples/GAN_step5000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



[13]:
```
img = mpimg.imread('samples/GAN_step9000.jpg')
imgplot = plt.imshow(img)
plt.show()
```


0
50
100
150
200
250
0
50
100
150
200
250

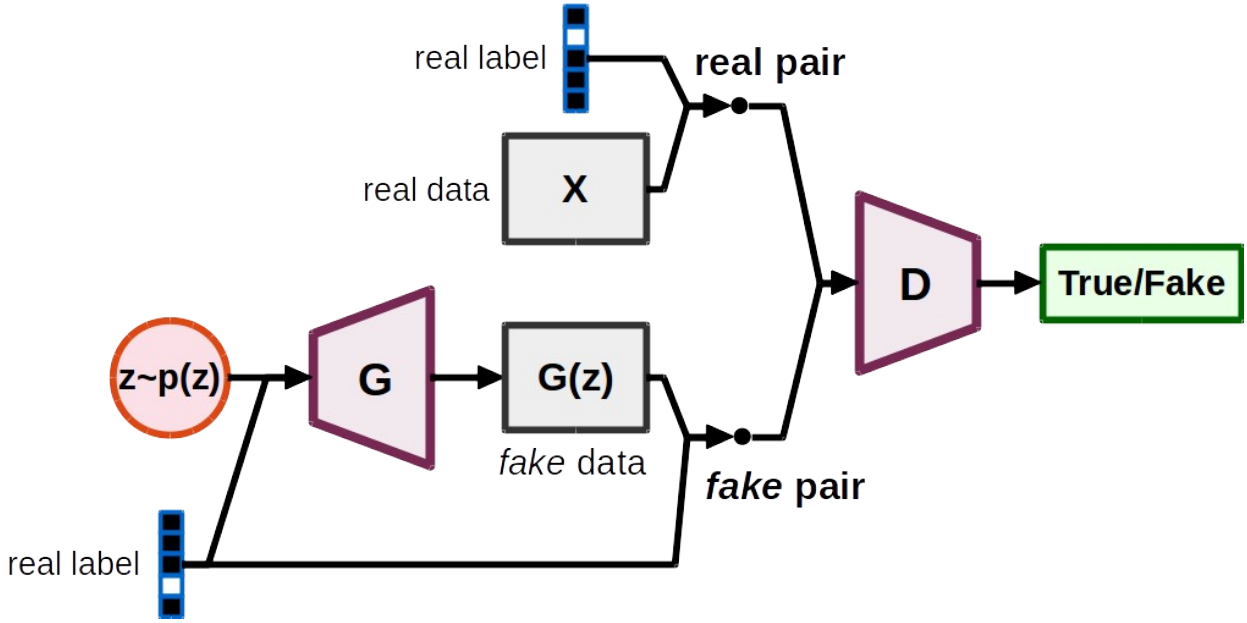
MNIST Conditional GAN

Conditional GANs

Based on the excellent tutorial here.

By formulating the process as a two-player game, Generative Adversarial Networks (GANs) can be very effective in generating realistic content. However, we may want to have more control over what is generated. Conditional GANs offer more control by letting us specify the *class* of output we want. Then we hand the generated content and the class it's supposed to be to the discriminator. The disciminator attempts to differentiate between the generated content of a certain class and the real content of a certain class.

The original paper that described conditional GANs is here.



Libaries

As always, we load lots of libraries.

```
import torch
from torchvision import datasets
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
import numpy as np
import os
from matplotlib.pyplot import imsave
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

Data

For this demo, we will be using the MNIST data set. We can apply GANs to other datasets but the training process takes much longer. Our goal will be to supply random noise and a class label (e.g. a digit between 0 and 9) to the generator and produce an image of that particular digit.

```
# A transform to convert the images to tensor and normalize their RGB
values
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=[0.5],
std=[0.5])]
)

data = datasets.MNIST(root='../data/', train=True,
transform=transform, download=True)

batch_size = 64
data_loader = DataLoader(dataset=data, batch_size=batch_size,
shuffle=True, drop_last=True)
```

Helper Functions

We'll need several helper functions for training the conditional GAN. The first converts labels to one hot encoded vectors, we will use it to pass the desired label to the generator. The second will plot a grid of 10x10 images from the generator.

```
def to_onehot(x, num_classes=10):
    assert isinstance(x, int) or isinstance(x, (torch.LongTensor,
torch.cuda.LongTensor))
    if isinstance(x, int):
        c = torch.zeros(1, num_classes).long()
        c[0][x] = 1
    else:
        x = x.cpu()
        c = torch.LongTensor(x.size(0), num_classes)
        c.zero_()
        c.scatter_(1, x, 1) # dim, index, src value
    return c

to_onehot(3)

tensor([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]])

def get_sample_image(G, DEVICE, n_noise=100):
    img = np.zeros([280, 280])
    for j in range(10):
        c = torch.zeros([10, 10]).to(DEVICE)
        c[:, j] = 1
```

```
        z = torch.randn(10, n_noise).to(DEVICE)
        y_hat = G(z,c).view(10, 28, 28)
        result = y_hat.cpu().data.numpy()
        img[j*28:(j+1)*28] = np.concatenate([x for x in result],
axis=-1)
    return img
```

Architecture

We now instantiate the generator and discriminator architectures. The generator takes a random noise vector and a one hot encoded label as input and produces an image. The discriminator takes an image and a one hot encoded label as input and produces a single value between 0 and 1. The discriminator is trained to output 1 for real images and 0 for fake images. The generator is trained to fool the discriminator by outputting images that look real.

```
class Generator(nn.Module):
    def __init__(self, input_size=100, num_classes=10,
image_size=28*28):
        super(Generator, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size+num_classes, 128), # auxillary
dimension for label
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, image_size),
            nn.Tanh()
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = torch.cat((x, c), 1) # v: [input, label] concatenated
vector
        y_ = self.network(v)
        y_ = y_.view(x.size(0), 1, 28, 28)
        return y_

class Discriminator(nn.Module):
    def __init__(self, input_size=28*28, num_classes=10,
num_output=1):
        super(Discriminator, self).__init__()
        self.network = nn.Sequential(
```

```
            nn.Linear(input_size+num_classes, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, num_output),
            nn.Sigmoid(),
        )

    def forward(self, x, c):
        x, c = x.view(x.size(0), -1), c.view(c.size(0), -1).float()
        v = torch.cat((x, c), 1) # v: [input, label] concatenated
vector
        y_ = self.network(v)
        return y_
```

Set up and Training

Now, we're ready to instantiate our models, hyperparameters, and optimizers. Since the task is so easy for MNIST, we will train for only 10 epochs. We will update the generator and discriminator in every step but often one can be trained more frequently than the other.

```
MODEL_NAME = 'ConditionalGAN'
DEVICE = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

D = Discriminator().to(DEVICE) # randomly intialized
G = Generator().to(DEVICE) # randomly initialized

max_epoch = 10
step = 0
n_noise = 100 # size of noise vector

criterion = nn.BCELoss()
D_opt = torch.optim.Adam(D.parameters(), lr=0.0002, betas=(0.5,
0.999))
G_opt = torch.optim.Adam(G.parameters(), lr=0.0002, betas=(0.5,
0.999))

# We will denote real images as 1s and fake images as 0s
# This is why we needed to drop the last batch of the data loader
all_ones = torch.ones([batch_size, 1]).to(DEVICE) # Discriminator
label: real
all_zeros = torch.zeros([batch_size, 1]).to(DEVICE) # Discriminator
Label: fake

images, class_labels = next(iter(data_loader))
class_labels_encoded = class_labels.view(batch_size, 1)
class_labels_encoded = to_onehot(class_labels_encoded).to(DEVICE)
```

```
print(class_labels[:10])
print(class_labels_encoded[:10])
```

```
tensor([4, 8, 1, 5, 0, 4, 8, 2, 5, 1])
tensor([[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]], device='cuda:0')
```

```
# a directory to save the generated images
if not os.path.exists('samples'):
    os.makedirs('samples')

for epoch in range(max_epoch):
    for idx, (images, class_labels) in enumerate(data_loader):
        # Training Discriminator
        x = images.to(DEVICE)
        class_labels = class_labels.view(batch_size, 1) # add
singleton dimension so batch_size x 1
        class_labels = to_onehot(class_labels).to(DEVICE)
        x_outputs = D(x, class_labels) # input includes labels
        D_x_loss = criterion(x_outputs, all_ones) # Discriminator loss
for real images

        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels) # input to
both generator and discriminator includes labels
        D_z_loss = criterion(z_outputs, all_zeros) # Discriminator
loss for fake images
        D_loss = D_x_loss + D_z_loss # Total Discriminator loss

        D.zero_grad()
        D_loss.backward()
        D_opt.step()

        # Training Generator
        z = torch.randn(batch_size, n_noise).to(DEVICE)
        z_outputs = D(G(z, class_labels), class_labels)
        G_loss = -1 * criterion(z_outputs, all_zeros) # Generator loss
is negative disciminator loss

        G.zero_grad()
        G_loss.backward()
        G_opt.step()
```

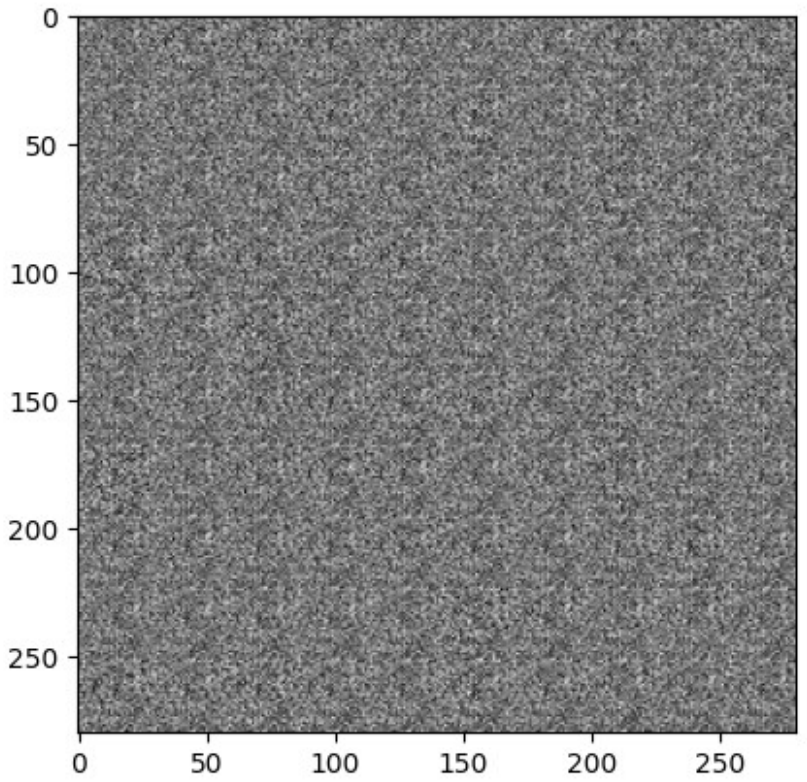
```
        if step % 500 == 0:
            print('Epoch: {}/{}, Step: {}, D Loss: {}, G Loss:
{}'.format(epoch, max_epoch, step, D_loss.item(), G_loss.item()))

        if step % 1000 == 0:
            G.eval()
            img = get_sample_image(G, DEVICE, n_noise)
            imsave('samples/{}_step{}.jpg'.format(MODEL_NAME,
str(step).zfill(3)), img, cmap='gray')
            G.train()
        step += 1
```

```
Epoch: 0/10, Step: 0, D Loss: 1.395571231842041, G Loss: -
0.691758394241333
Epoch: 0/10, Step: 500, D Loss: 1.0961824655532837, G Loss: -
0.44526758790016174
Epoch: 1/10, Step: 1000, D Loss: 1.039373755455017, G Loss: -
0.2498992681503296
Epoch: 1/10, Step: 1500, D Loss: 1.2127230167388916, G Loss: -
0.7997499108314514
Epoch: 2/10, Step: 2000, D Loss: 1.1976568698883057, G Loss: -
0.5237036943435669
Epoch: 2/10, Step: 2500, D Loss: 1.0645639896392822, G Loss: -
0.604216456413269
Epoch: 3/10, Step: 3000, D Loss: 1.2835071086883545, G Loss: -
0.6310629844665527
Epoch: 3/10, Step: 3500, D Loss: 1.5267739295959473, G Loss: -
0.2976498603820801
Epoch: 4/10, Step: 4000, D Loss: 1.3138728141784668, G Loss: -
0.5189226865768433
Epoch: 4/10, Step: 4500, D Loss: 1.425218105316162, G Loss: -
0.8295738697052002
Epoch: 5/10, Step: 5000, D Loss: 1.2697324752807617, G Loss: -
0.6327874660491943
Epoch: 5/10, Step: 5500, D Loss: 1.316536784172058, G Loss: -
0.5923196077346802
Epoch: 6/10, Step: 6000, D Loss: 1.3598530292510986, G Loss: -
0.6805770397186279
Epoch: 6/10, Step: 6500, D Loss: 1.3046804666519165, G Loss: -
0.6339365839958191
Epoch: 7/10, Step: 7000, D Loss: 1.2994040250778198, G Loss: -
0.6625391840934753
Epoch: 8/10, Step: 7500, D Loss: 1.2581830024719238, G Loss: -
0.6033541560173035
Epoch: 8/10, Step: 8000, D Loss: 1.3504083156585693, G Loss: -
0.7974995374679565
Epoch: 9/10, Step: 8500, D Loss: 1.2936216592788696, G Loss: -
0.6776021122932434
```

```
Epoch: 9/10, Step: 9000, D Loss: 1.3627541065216064, G Loss: -
0.8031416535377502
```

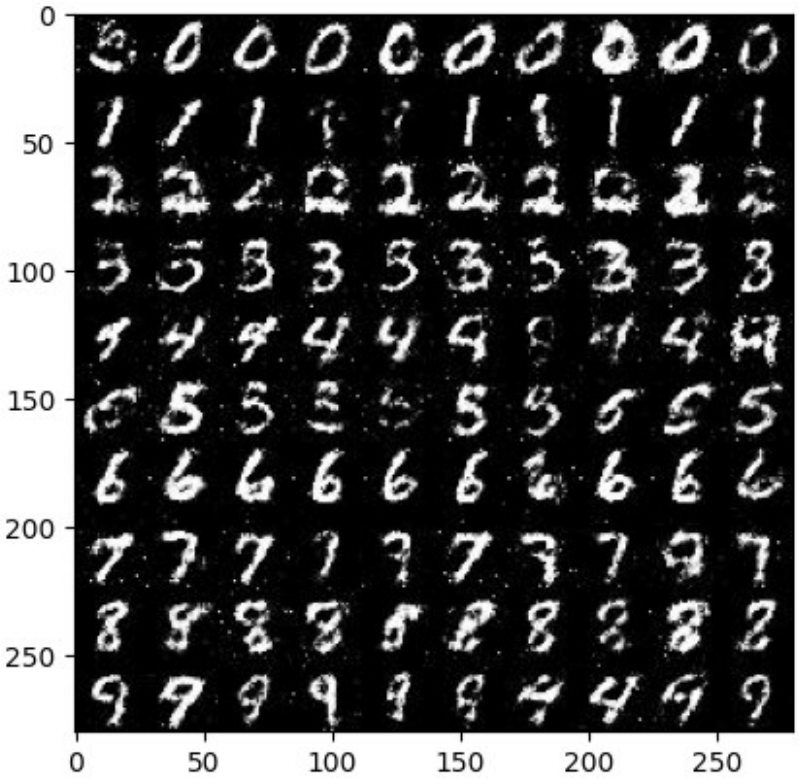
Now let's plot these images. At first, the generator just produces noise (as we expect).

```
img = mpimg.imread('samples/ConditionalGAN_step000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



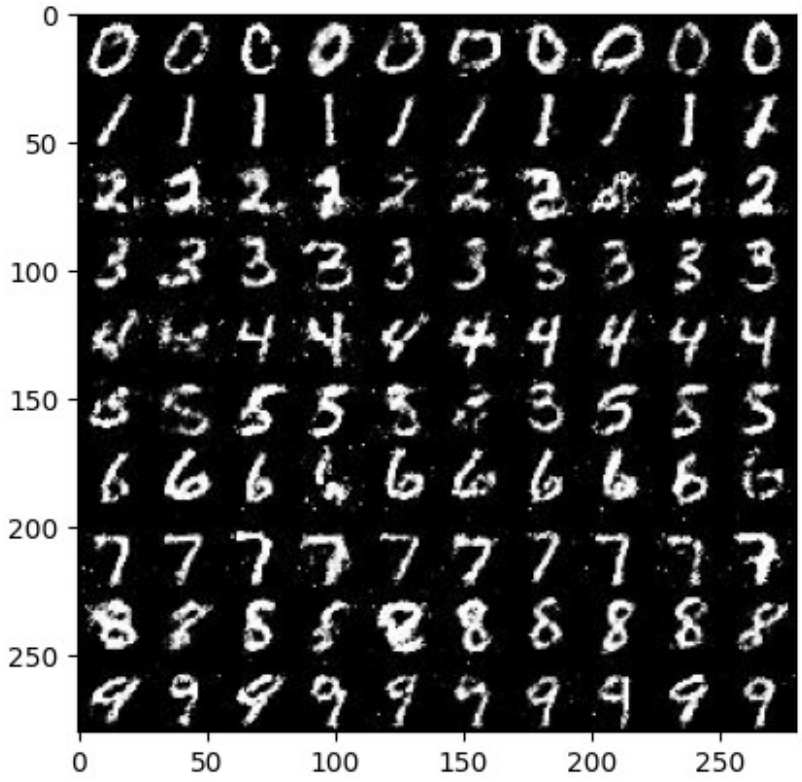
But then it gets better.

```
img = mpimg.imread('samples/ConditionalGAN_step5000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



In fact, if we don't look too closely, we can recognize the numbers it produces.

```
img = mpimg.imread('samples/ConditionalGAN_step9000.jpg')
imgplot = plt.imshow(img)
plt.show()
```



And by the time we're done training, even the worst images look like messy handwriting!