

LAB 11 : Generative Adversarial Networks

Name : Vaishnavi, Joshitha

Roll Number : 180020039, 180020009

References :

1. <https://towardsdatascience.com/gans-from-scratch-8f5da17b3fb4>

▼ Problem 1 : Understanding and Training Vanilla GANs

1. Consider MNIST Digit Dataset
2. Create a Generator and a Discriminator Network and train the network. Follow this article : <https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcd8a0f>
3. Explain briefly about your understanding of GANs

▼ Write down the Objectives, Hypothesis and Experimental description for the above problem

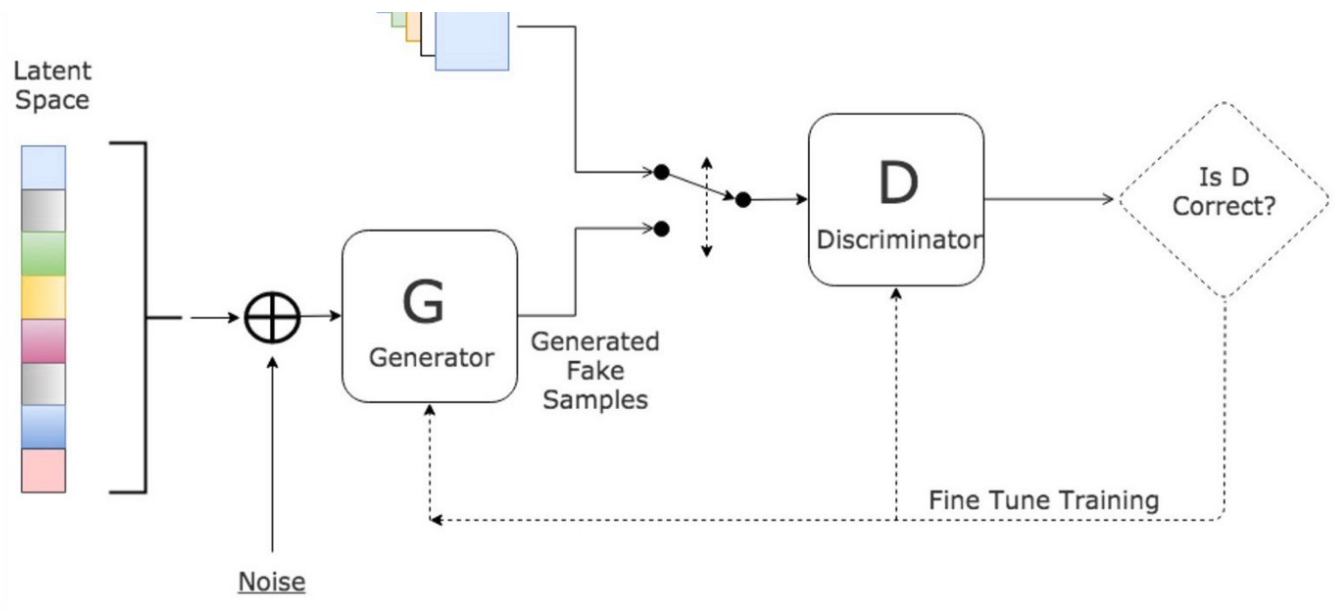
Objective : Train a GAN with a generator and discriminator on mnist digit dataset.

Experimental Description:

1. Sample a noise set and a real-data set, each with size m .
2. Train the Discriminator on this data.
3. Sample a different noise subset with size m .
4. Train the Generator on this data.
5. Repeat from Step 1.

Generative Adversarial Network





Generative Adversarial Networks are composed of two models:

1. The first model is called a Generator and it aims to generate new data similar to the expected one. The Generator could be assimilated to a human art forger, which creates fake works of art.
2. The second model is named the Discriminator. This model's goal is to recognize if an input data is 'real' — belongs to the original dataset — or if it is 'fake' — generated by a forger. In this scenario, a Discriminator is analogous to an art expert, which tries to detect artworks as truthful or fraud.

Programming :

Please write a program to demonstrate the same

```
1 ## Write your code here
2 !pip install torchvision tensorboardx jupyter matplotlib numpy
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

LeCunn's MNIST dataset, consisting of about 60,000 black and white images of handwritten

digits, each with size 28×28 pixels²

Preprocessing :

Specifically, the input values which range in between $[0, 255]$ will be normalized between -1 and 1 . This means the value 0 will be mapped to -1 , the value 255 to 1 , and similarly all values in between will get a value in the range $[-1, 1]$.

```
1 def mnist_data():
2     compose = transform = transforms.Compose([transforms.ToTensor(), transforms.
3         out_dir = './dataset'
4     return datasets.MNIST(root=out_dir, train=True, transform=compose, download=
5 # Load data
6 data = mnist_data()
7 # Create loader with data, so that we can iterate over it
8 data_loader = torch.utils.data.DataLoader(data, batch_size=100, shuffle=True)
9 # Num batches
10 num_batches = len(data_loader)
11
```

Discriminator

Discriminator will take a flattened image as its input, and return the probability of it belonging to the real dataset, or the synthetic dataset. The input size for each image will be $28 \times 28 = 784$.

Structure :

3 hidden layers, each followed by a Leaky-ReLU nonlinearity and a Dropout layer to prevent overfitting.

A Sigmoid/Logistic function is applied to the real-valued output to obtain a value in the open-range $(0, 1)$:

```
1 class DiscriminatorNet(torch.nn.Module):
2     """
3     A three hidden-layer discriminative neural network
4     """
5     def __init__(self):
6         super(DiscriminatorNet, self).__init__()
7         n_features = 784
8         n_out = 1
9
10        self.hidden0 = nn.Sequential(
11            nn.Linear(n_features, 1024),
12            nn.LeakyReLU(0.2),
13            nn.Dropout(0.3)
```

```

14         )
15         self.hidden1 = nn.Sequential(
16             nn.Linear(1024, 512),
17             nn.LeakyReLU(0.2),
18             nn.Dropout(0.3)
19         )
20         self.hidden2 = nn.Sequential(
21             nn.Linear(512, 256),
22             nn.LeakyReLU(0.2),
23             nn.Dropout(0.3)
24         )
25         self.out = nn.Sequential(
26             torch.nn.Linear(256, n_out),
27             torch.nn.Sigmoid()
28         )
29
30     def forward(self, x):
31         x = self.hidden0(x)
32         x = self.hidden1(x)
33         x = self.hidden2(x)
34         x = self.out(x)
35         return x
36
37
38 discriminator = DiscriminatorNet()
39

```

Additional functionality that allows us to convert a flattened image into its 2-dimensional representation, and another one that does the opposite.

```

1 def images_to_vectors(images):
2     return images.view(images.size(0), 784)
3
4 def vectors_to_images(vectors):
5     return vectors.view(vectors.size(0), 1, 28, 28)
6

```

Generative Network :

Takes a latent variable vector as input, and returns a 784 valued vector, which corresponds to a flattened 28x28 image. The purpose of this network is to learn how to create undistinguishable images of hand-written digits, which is why its output is itself a new image.

Structure :

1. three hidden layers, each followed by a Leaky-ReLU nonlinearity.
2. The output layer will have a TanH activation function, which maps the resulting values into

the $(-1, 1)$ range, which is the same range in which our preprocessed MNIST images is bounded.

```

1 class GeneratorNet(torch.nn.Module):
2     """
3     A three hidden-layer generative neural network
4     """
5     def __init__(self):
6         super(GeneratorNet, self).__init__()
7         n_features = 100
8         n_out = 784
9
10        self.hidden0 = nn.Sequential(
11            nn.Linear(n_features, 256),
12            nn.LeakyReLU(0.2)
13        )
14        self.hidden1 = nn.Sequential(
15            nn.Linear(256, 512),
16            nn.LeakyReLU(0.2)
17        )
18        self.hidden2 = nn.Sequential(
19            nn.Linear(512, 1024),
20            nn.LeakyReLU(0.2)
21        )
22
23        self.out = nn.Sequential(
24            nn.Linear(1024, n_out),
25            nn.Tanh()
26        )
27
28        def forward(self, x):
29            x = self.hidden0(x)
30            x = self.hidden1(x)
31            x = self.hidden2(x)
32            x = self.out(x)
33            return x
34 generator = GeneratorNet()
35

```

This function allows us to create the random noise. The random noise will be sampled from a normal distribution with mean 0 and variance 1.

```

1 def noise(size):
2     """
3     Generates a 1-d vector of gaussian sampled random values
4     """
5     # ... (code to generate noise) ...

```

```

5     n = variable(torch.randn(size, 100))
6     return n
7

```

we'll use Adam as the optimization algorithm for both neural networks, with a learning rate of 0.0002

```

1 d_optimizer = optim.Adam(discriminator.parameters(), lr=0.0002)
2 g_optimizer = optim.Adam(generator.parameters(), lr=0.0002)
3

```

Loss function : Binary Cross Entropy Loss (BCE Loss) as it resembles the log-loss for both the Generator and Discriminator

```

1 loss = nn.BCELoss()
2

```

$$L = \{l_1, \dots, l_N\}^T, l_i = -w_i [y_i \cdot \log(v_i) + (1 - y_i) \cdot \log(1 - v_i)]$$

Binary Cross Entropy Log. Mean is calculated by computing $\text{sum}(L) / N$. In this formula the values y are named targets, v are the inputs, and w are the weights. Since we don't need the weight at all, it'll be set to $w_i=1$ for all i .

Discriminator Loss:

$$\frac{1}{m} \sum_{i=1}^m \left[\log D \left(\mathbf{x}^{(i)} \right) + \log \left(1 - D \left(G \left(\mathbf{z}^{(i)} \right) \right) \right) \right]$$

If we replace $v_i = D(\mathbf{x}_i)$ and $y_i=1 \forall i$ (for all i) in the BCE-Loss definition, we obtain the loss related to the real-images. Conversely if we set $v_i = D(G(\mathbf{z}_i))$ and $y_i=0 \forall i$, we obtain the loss related to the fake-images

Since maximizing a function is equivalent to minimizing it's negative, and the BCE-Loss term has a minus sign, we don't need to worry about the sign.

We can observe that the real-images targets are always ones, while the fake-images targets are zero, so it would be helpful to define the following functions:

```

1 def ones_target(size):
2     ...

```

```

2     ...
3     Tensor containing ones, with shape = size
4     '''
5     data = Variable(torch.ones(size, 1))
6     return data
7
8 def zeros_target(size):
9     '''
10    Tensor containing zeros, with shape = size
11    '''
12    data = Variable(torch.zeros(size, 1))
13    return data
14

```

By summing up these two discriminator losses we obtain the total mini-batch loss for the Discriminator. We will calculate the gradients separately, and then update them together.

```

1 def train_discriminator(optimizer, real_data, fake_data):
2     N = real_data.size(0)
3     # Reset gradients
4     optimizer.zero_grad()
5
6     # 1.1 Train on Real Data
7     prediction_real = discriminator(real_data)
8     # Calculate error and backpropagate
9     error_real = loss(prediction_real, ones_target(N) )
10    error_real.backward()
11
12    # 1.2 Train on Fake Data
13    prediction_fake = discriminator(fake_data)
14    # Calculate error and backpropagate
15    error_fake = loss(prediction_fake, zeros_target(N))
16    error_fake.backward()
17
18    # 1.3 Update weights with gradients
19    optimizer.step()
20
21    # Return error and predictions for real and fake inputs
22    return error_real + error_fake, prediction_real, prediction_fake
23

```

Generator Loss:

$$\frac{1}{m} \sum^m \log \left(1 - D \left(G \left(z^{(i)} \right) \right) \right)$$

$$i=1 \quad \backslash \quad \backslash \quad \backslash \quad / \quad / \quad /$$

Rather than minimizing $\log(1 - D(G(z)))$, training the Generator to maximize $\log D(G(z))$ will provide much stronger gradients early in training. Both losses may be swapped interchangeably since they result in the same dynamics for the Generator and Discriminator.

Maximizing $\log D(G(z))$ is equivalent to minimizing its negative and since the BCE-Loss definition has a minus sign, we don't need to take care of the sign. Similarly to the Discriminator, if we set $v_i = D(G(z_i))$ and $y_i = 1 \forall i$, we obtain the desired loss to be minimized.

```

1 def train_generator(optimizer, fake_data):
2     N = fake_data.size(0)
3     # Reset gradients
4     optimizer.zero_grad()
5     # Sample noise and generate fake data
6     prediction = discriminator(fake_data)
7     # Calculate error and backpropagate
8     error = loss(prediction, ones_target(N))
9     error.backward()
10    # Update weights with gradients
11    optimizer.step()
12    # Return error
13    return error
14

```

```
1 !wget https://raw.githubusercontent.com/diegoalejogm/gans/master/utils.py
```

```
1 from utils1 import Logger
2
```

```
1 num_test_samples = 16
2 test_noise = noise(num_test_samples)
3
```

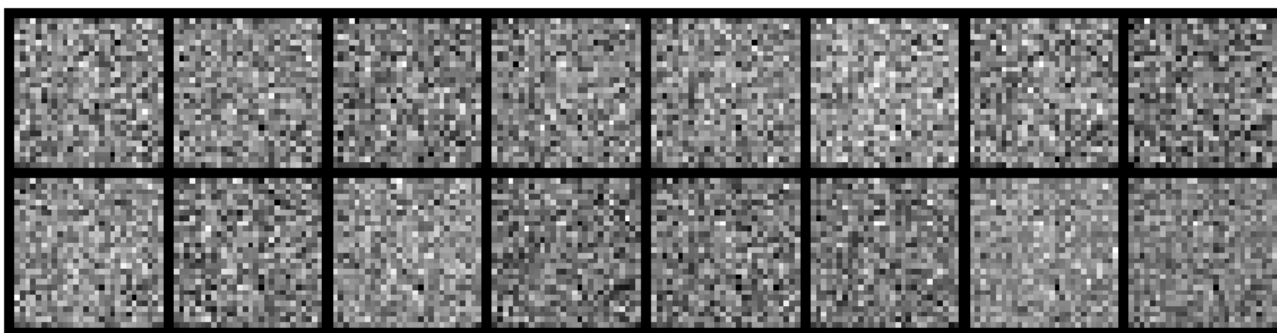
```
1 # Create logger instance
2 logger = Logger(model_name='VGAN', data_name='MNIST')
3 # Total number of epochs to train
4 num_epochs = 200
5 for epoch in range(num_epochs):
6     for n_batch, (real_batch, _) in enumerate(data_loader):
7         N = real_batch.size(0)
8         # 1. Train Discriminator
9         real_data = Variable(images_to_vectors(real_batch))
10        # Generate fake data and detach
```



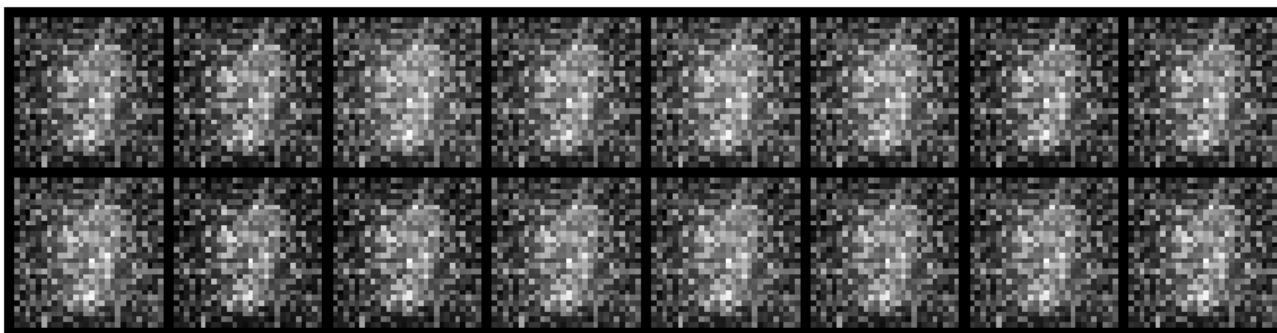
```

11     # (so gradients are not calculated for generator)
12     fake_data = generator(noise(N)).detach()
13     # Train D
14     d_error, d_pred_real, d_pred_fake = \
15         train_discriminator(d_optimizer, real_data, fake_data)
16
17     # 2. Train Generator
18     # Generate fake data
19     fake_data = generator(noise(N))
20     # Train G
21     g_error = train_generator(g_optimizer, fake_data)
22     # Log batch error
23     logger.log(d_error, g_error, epoch, n_batch, num_batches)
24     # Display Progress every few batches
25     if (n_batch) % 100 == 0:
26         test_images = vectors_to_images(generator(test_noise))
27         test_images = test_images.data
28         logger.log_images(
29             test_images, num_test_samples,
30             epoch, n_batch, num_batches
31         );
32     # Display status Logs
33     logger.display_status(
34         epoch, num_epochs, n_batch, num_batches,
35         d_error, g_error, d_pred_real, d_pred_fake
36     )
37

```

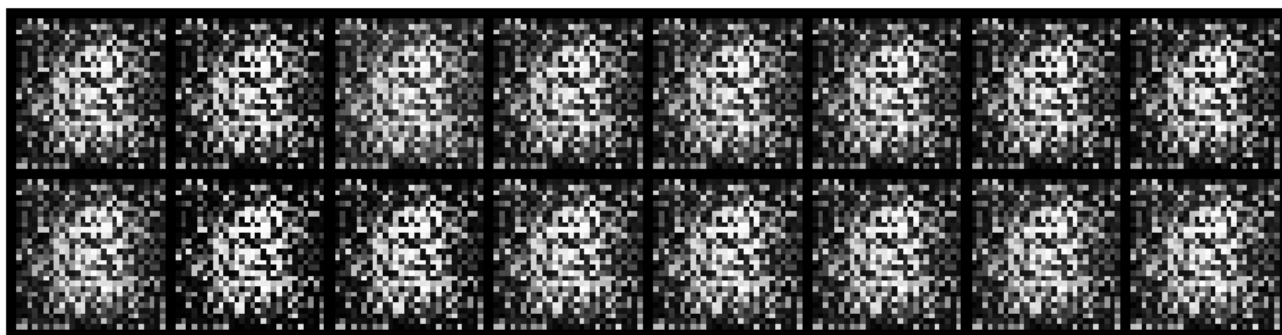


Epoch: [0/200], Batch Num: [0/600]
 Discriminator Loss: 1.3903, Generator Loss: 0.6739
 D(x): 0.5079, D(G(z)): 0.5095

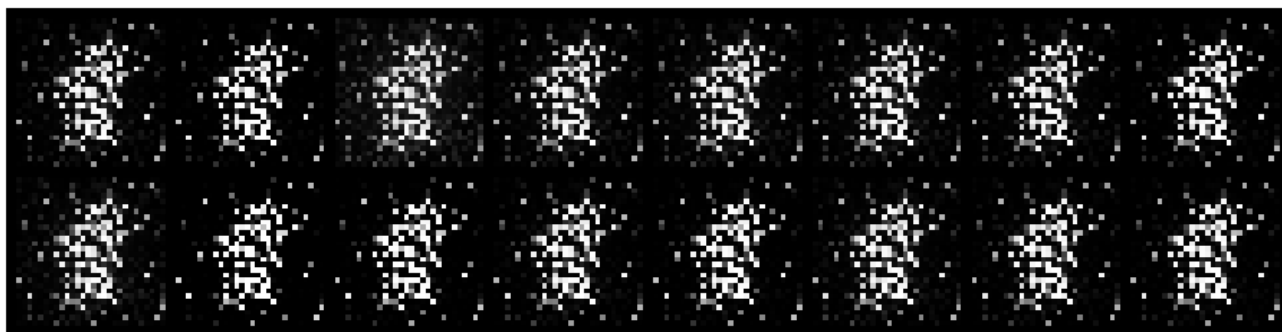


Epoch: [0/200], Batch Num: [100/600]
 Discriminator Loss: 1.3107, Generator Loss: 1.5510

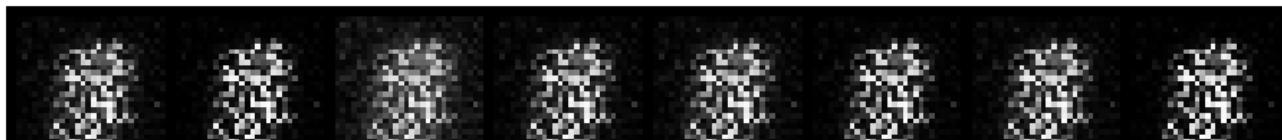
Discriminator Loss: 1.3107, Generator Loss: 1.3340
 D(x): 0.6453, D(G(z)): 0.4654



Epoch: [0/200], Batch Num: [200/600]
 Discriminator Loss: 0.4103, Generator Loss: 3.8607
 D(x): 0.8561, D(G(z)): 0.1803



Epoch: [0/200], Batch Num: [300/600]
 Discriminator Loss: 0.2172, Generator Loss: 5.2012
 D(x): 0.9144, D(G(z)): 0.0706



Inferences and Conclusion : State all the key observations and conclusion

1. The discriminator error is very high (Around 1.4 - 1.3 in the first few epochs) in the beginning, as it doesn't know how to classify correctly images as being either real or fake.
 After 1 epoch :

Discriminator Loss: 1.3903, Generator Loss: 0.6739

2. As the discriminator becomes better and its error decreases to about .5, the generator error increases, proving that the discriminator outperforms the generator and it can correctly classify the fake samples.

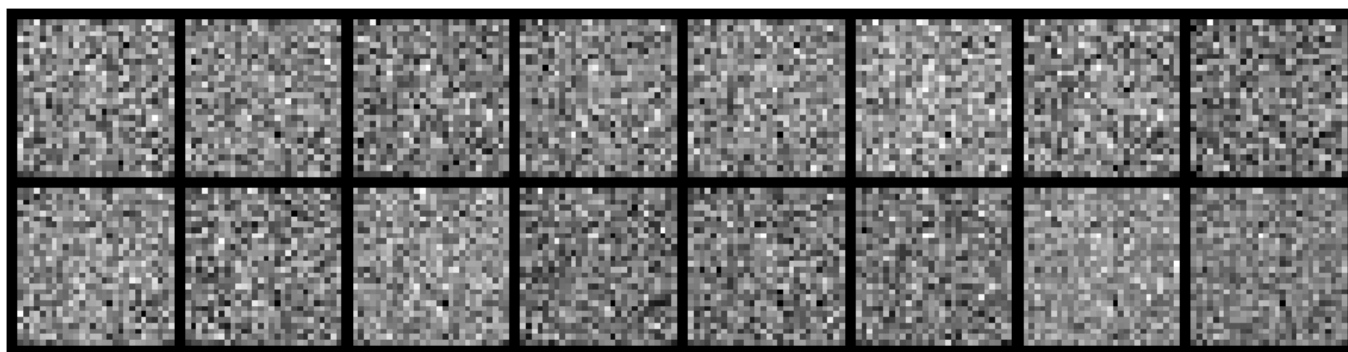
Discriminator Loss: 0.4254, Generator Loss: 5.7648

3. As time passes and training continues, the generator error lowers, implying that the images it generates are better and better. While the generator improves, the discriminator's error increases, because the synthetic images are becoming more realistic each time.

Final epoch :

Discriminator Loss: 1.3020, Generator Loss: 0.8640

Generated Images are :



After some 50 epochs :



After 200 epochs:



