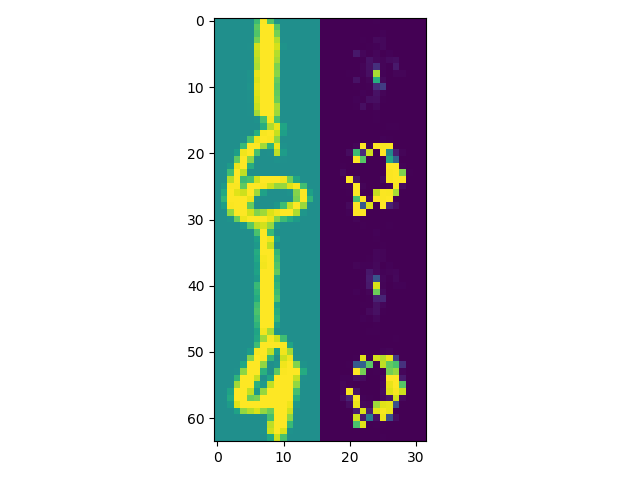
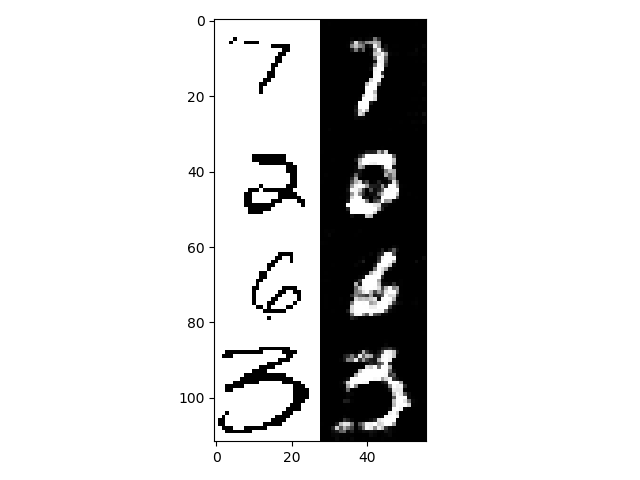
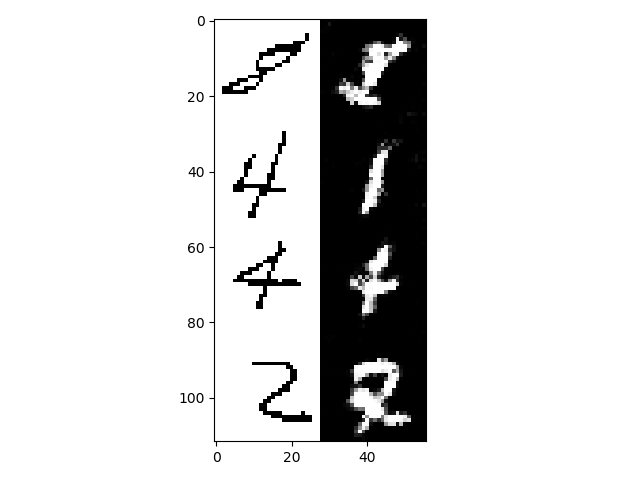
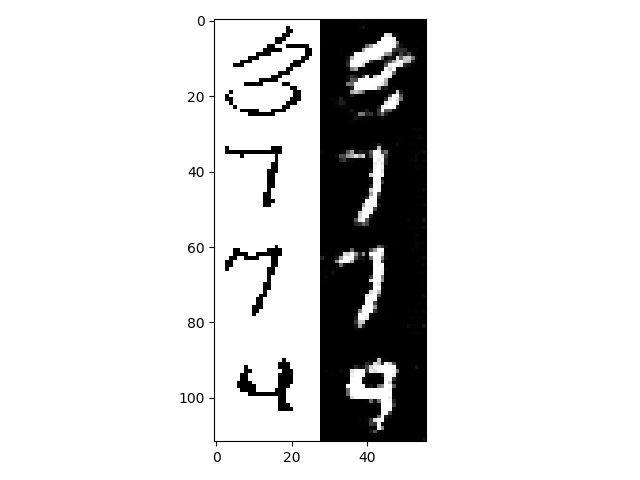
[1] 08/01: Vanilla GAN 16x16, g\_conv\_dim=64, d\_conv\_dim=64. Mode collapse to 2 outputs.



[2] 08/01: Vanilla GAN 28x28, g\_conv\_dim=64, d\_conv\_dim=64, niter=50, niter\_decay=400, batch\_size=64.

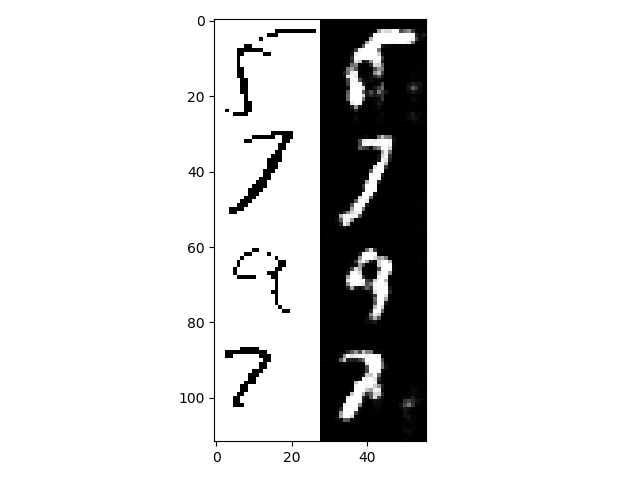
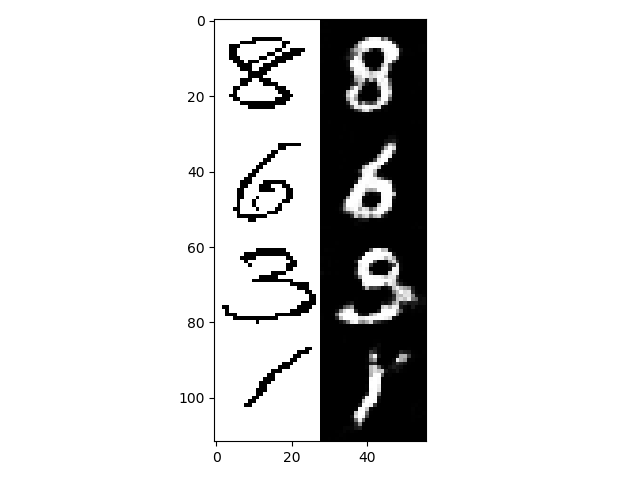
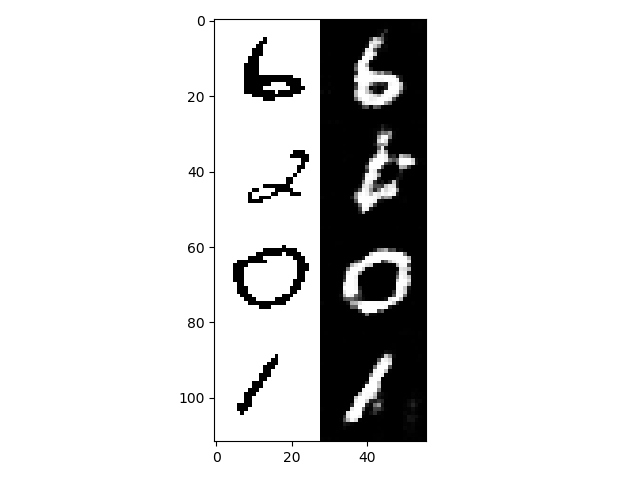
Observation: loss\_D is already close to 0.5 from the first few iterations onwards. The more iterations, the less noisy the output but eventually ends up favouring certain digits (e.g. 8 -> 3, 7 -> 1)



Maybe try tweaking g\_conv\_dim and d\_conv\_dim to find levels of sophistication of D and G that encourage fairer competition. Seeing that loss\_D ~ 0.5 early on (it already cannot tell the difference between real and fake), try making it more sophisticated by setting d\_conv\_dim > g\_conv\_dim

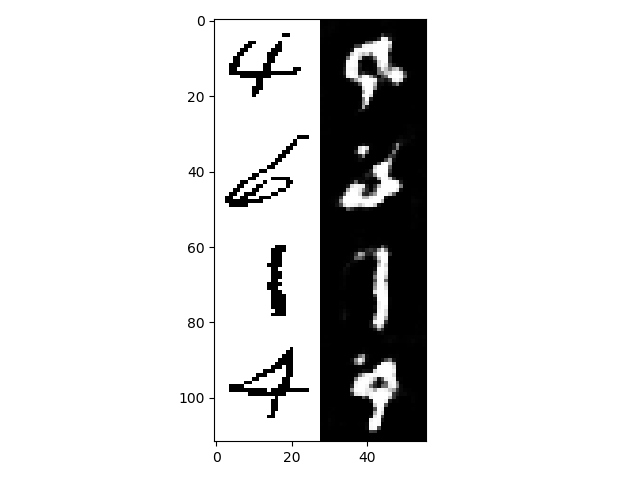
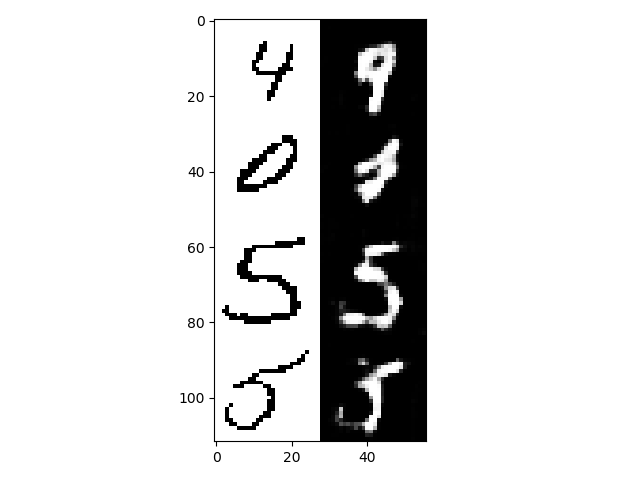
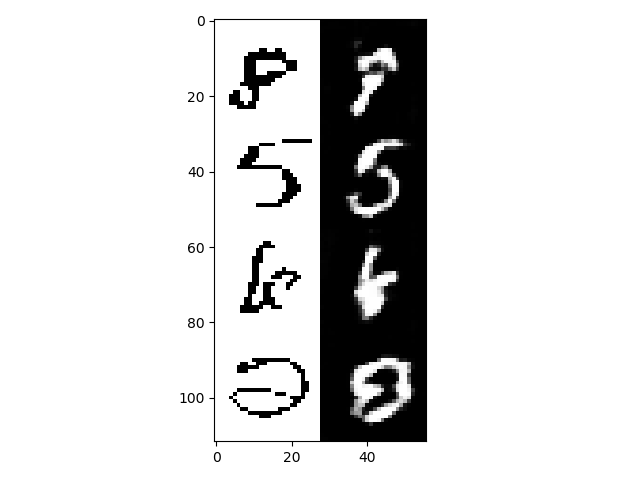
[3] 09/01: Vanilla GAN 28x28, g\_conv\_dim=32, d\_conv\_dim=64, niter=50, niter\_decay=400, batch\_size=64.

Much better



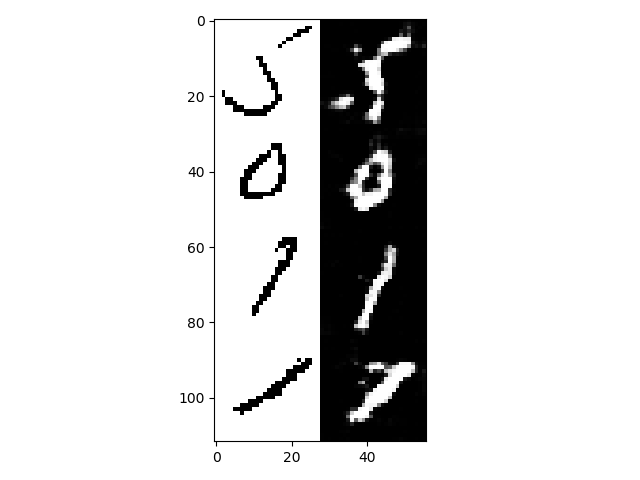
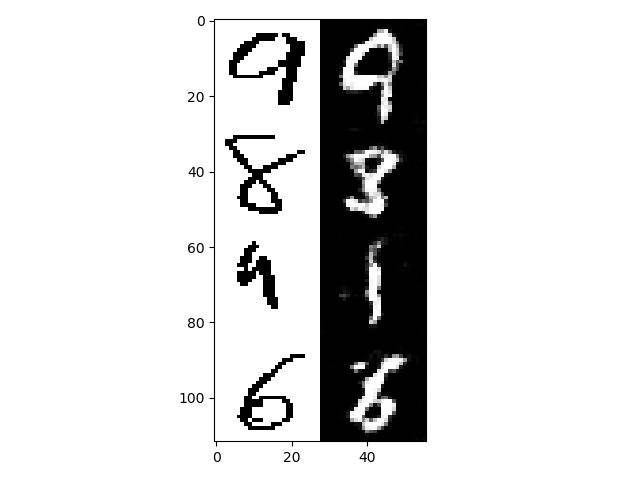
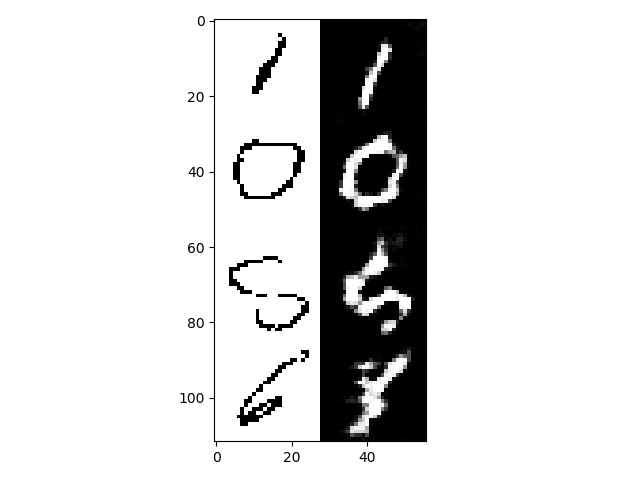
[4] 09/01: Vanilla GAN 28x28, g\_conv\_dim=20, d\_conv\_dim=64, niter=50, niter\_decay=400, batch\_size=64.

Poorer results possibly because G is not sophisticated enough to handle the variations in input



Still need d\_conv\_dim > g\_conv\_dim for the reason in [3] but concern about the network getting too large to train. Try decreasing d\_conv\_dim

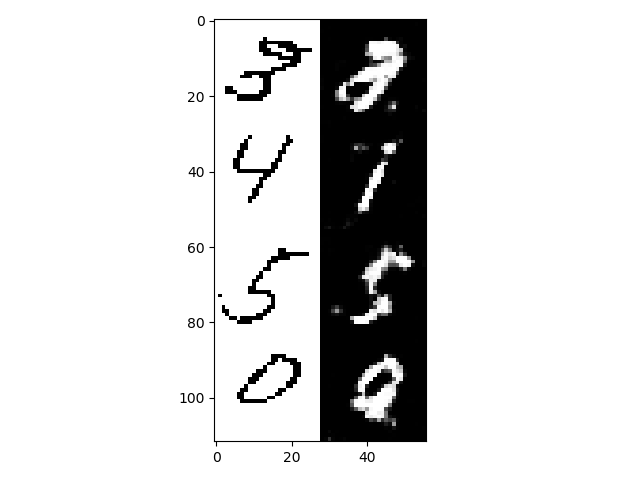
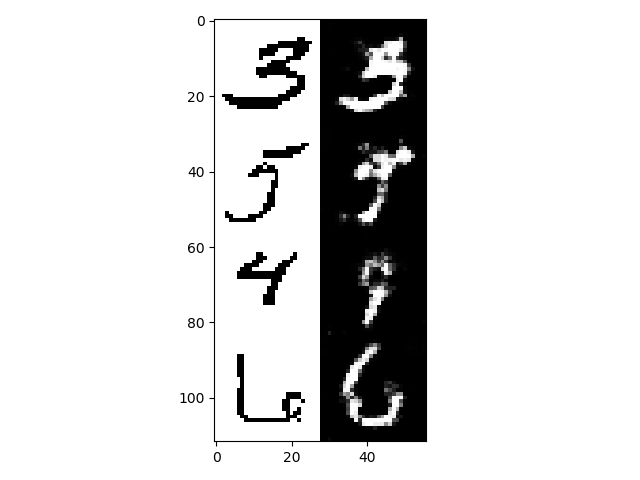
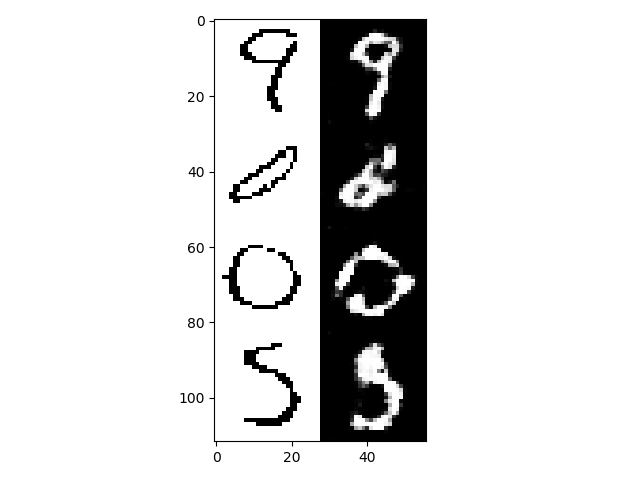
[5] 09/01: Vanilla GAN 28x28, g\_conv\_dim=32, d\_conv\_dim=48, niter=50, niter\_decay=400, batch\_size=64.



Same issues as [3] and arguably more pronounced. Will more iterations help?

[6] 09/01: Vanilla GAN 28x28, g\_conv\_dim=32, d\_conv\_dim=48, niter=70, niter\_decay=700, batch\_size=64.

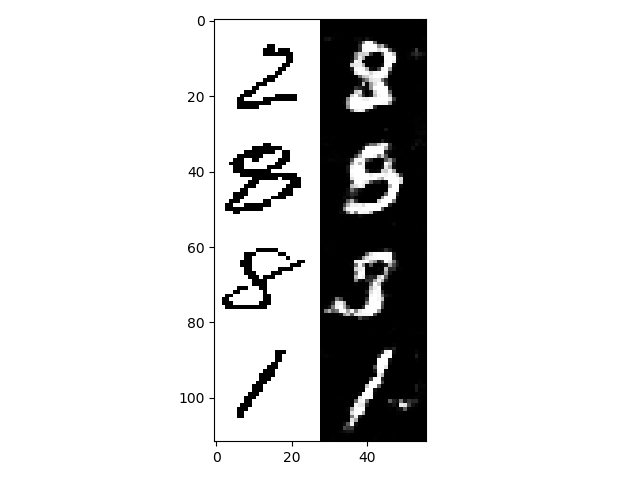
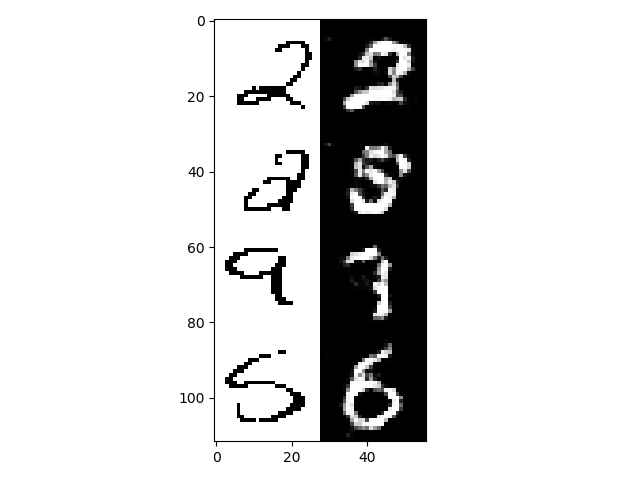
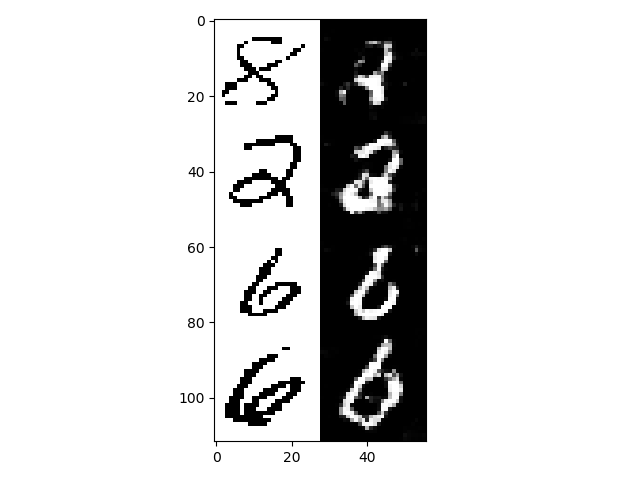
Marginal improvements only. Also the images are noisy compared to [3]. Visually, [3] is still the best result.



Let d\_conv\_dim remain at 64. Given that in [3] some numbers still fail to be rendered convincingly, try increase g\_conv\_dim to enable G to capture finer details in the image.

[7] 09/01: Vanilla GAN 28x28, g\_conv\_dim=36, d\_conv\_dim=64, niter=70, niter\_decay=700, batch\_size=64.

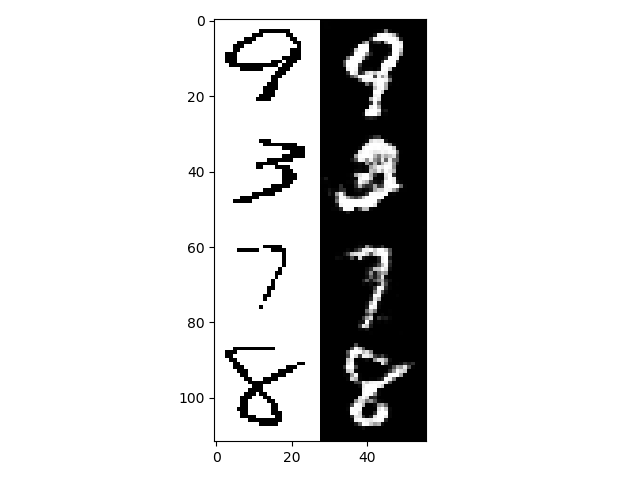
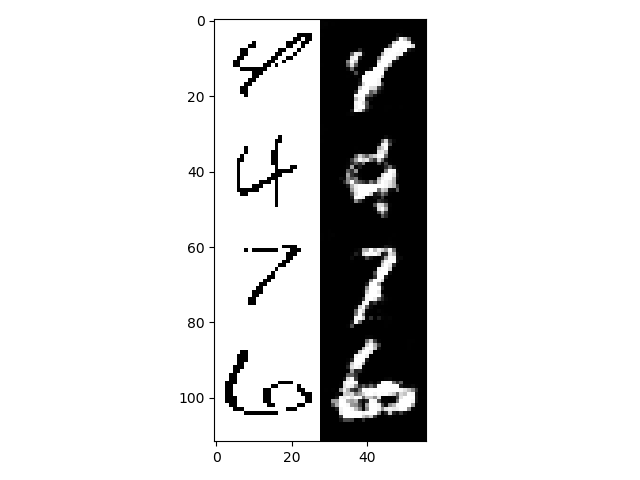
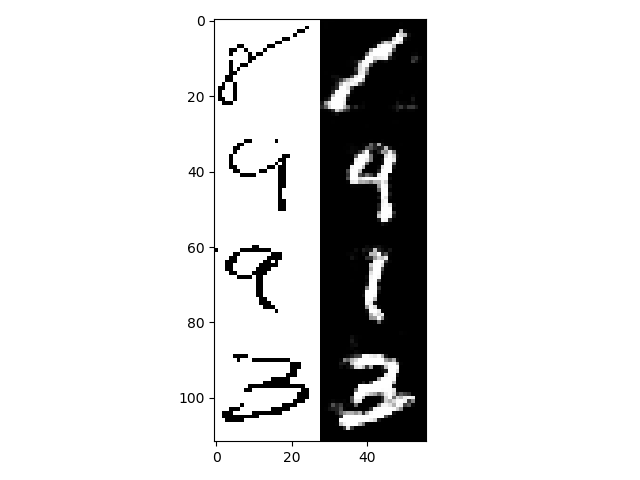
Same issues.



Even more iterations?

[8] 09/01: Vanilla GAN 28x28, g\_conv\_dim=36, d\_conv\_dim=64, niter=100, niter\_decay=1200, batch\_size=64.

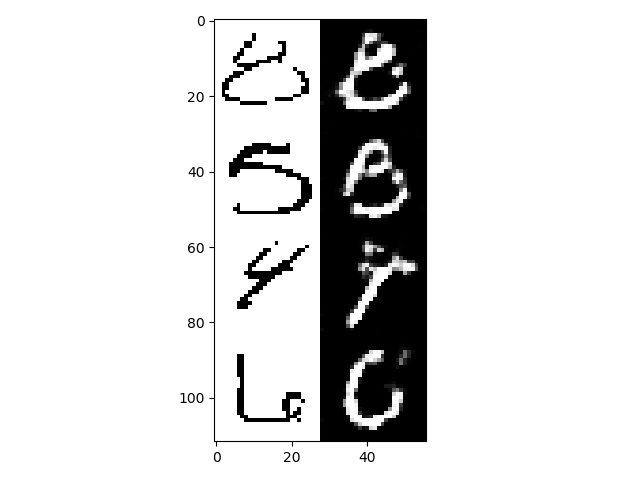
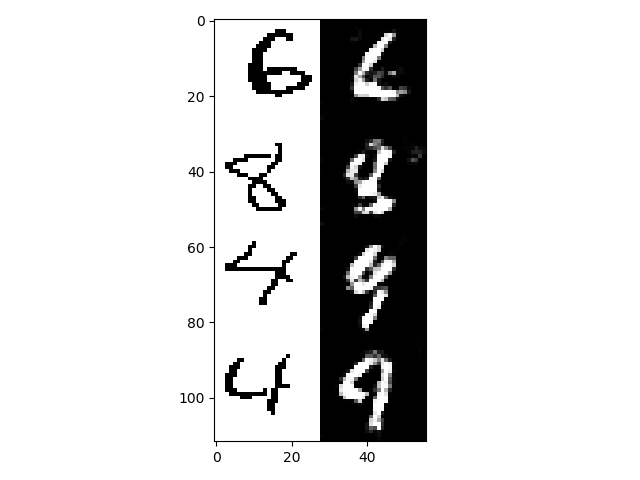
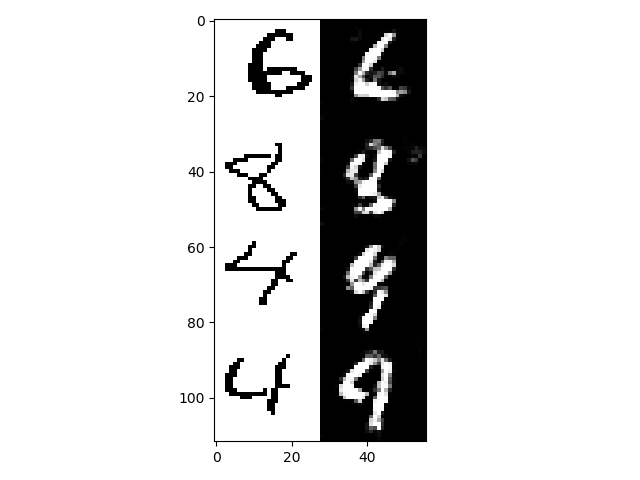
Slightly better, but still suffers the same pitfalls of generating the wrong number as well as some noisiness.



Everything above used 6 residual blocks as the network behind G. Try using which\_model\_netG=resnet\_9blocks

[9] 09/01: Vanilla GAN 28x28, g\_conv\_dim=36, d\_conv\_dim=64, niter=60, niter\_decay=600, batch\_size=64, which\_model\_netG=resnet\_9blocks.

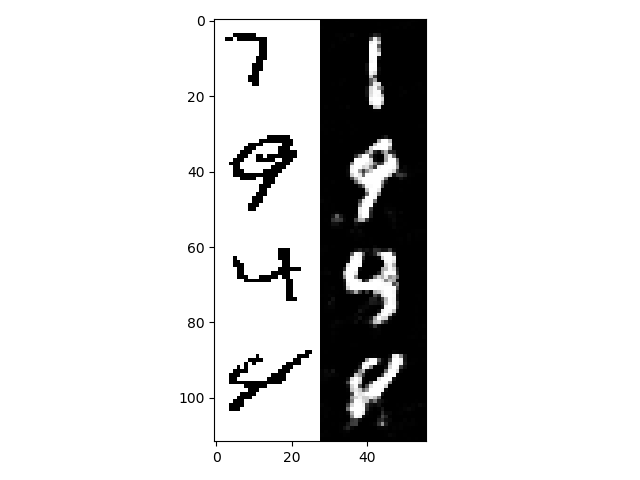
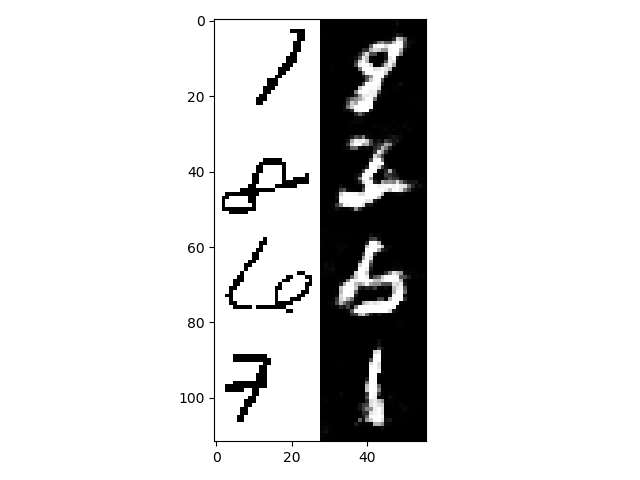
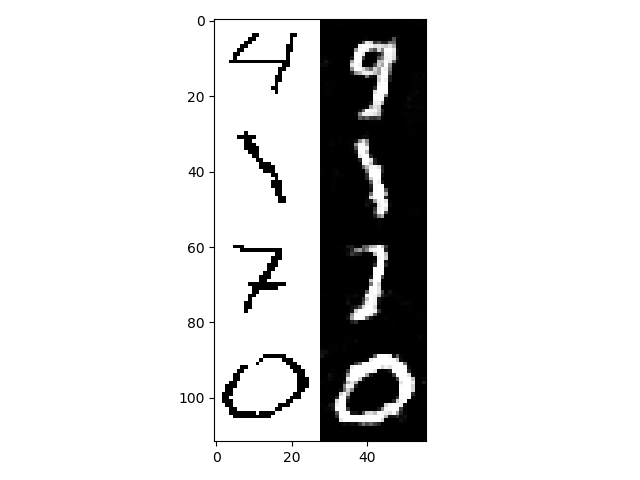
Bad.



D is still too dumb.

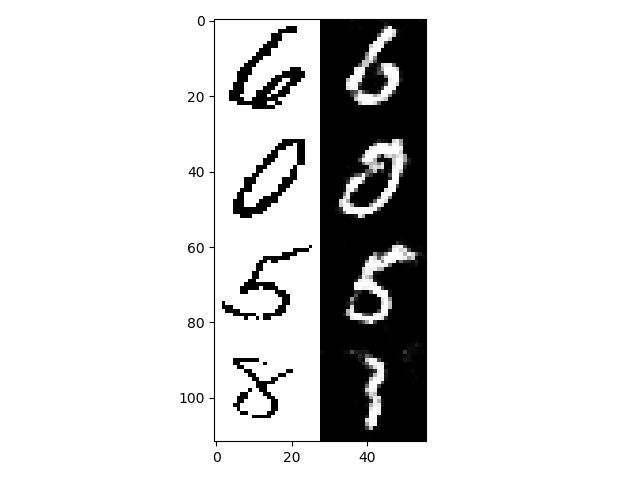
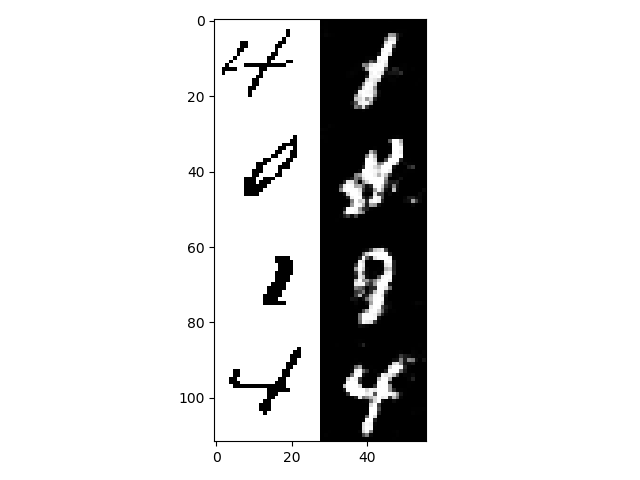
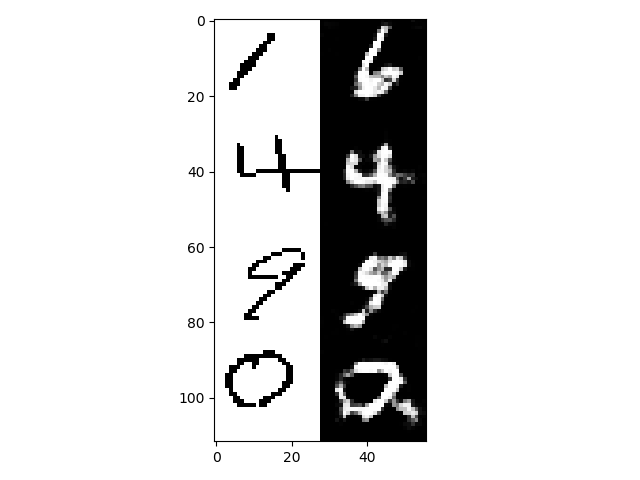
[10] 09/01: Vanilla GAN 28x28, g\_conv\_dim=36, d\_conv\_dim=64, niter=60, niter\_decay=600, batch\_size=64, which\_model\_netG=resnet\_9blocks

When the discriminator is not outsmarted by the generator, we see some rather good learning results. The output is quite convincing in some cases; however it is unstable in others.

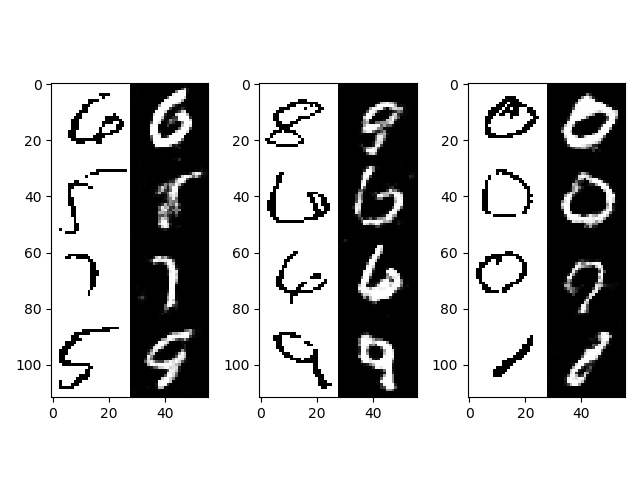


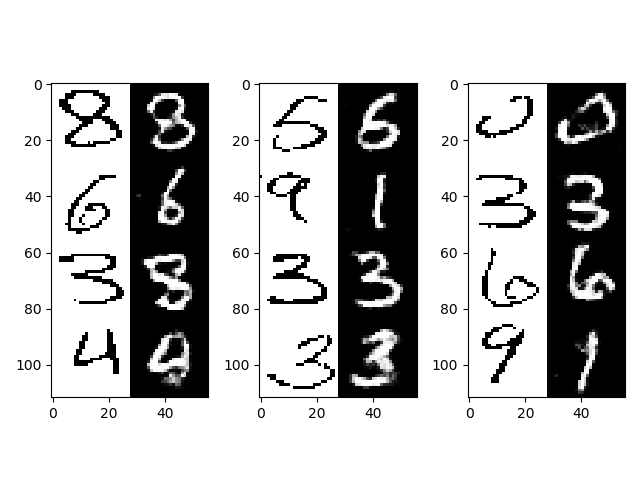
[11] 09/01: Vanilla GAN 28x28, g\_conv\_dim=64, d\_conv\_dim=64, niter=60, niter\_decay=600, batch\_size=32, which\_model\_netG=resnet\_9blocks, D.use\_sigmoid=True

Outputs are generally good but G is not mapping correct digits



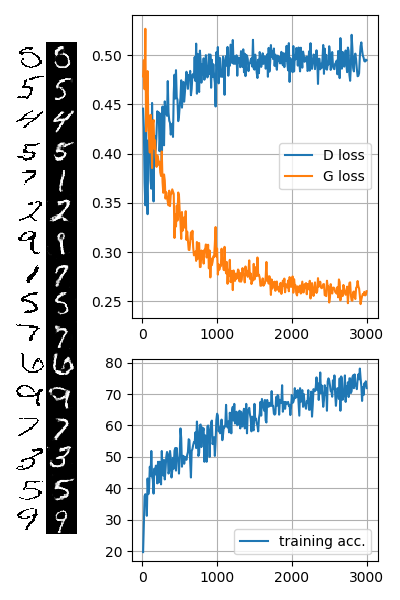
[12] 09/01: Vanilla GAN 28x28, g\_conv\_dim=64, d\_conv\_dim=64, niter=100, niter\_decay=1400, batch\_size=32, which\_model\_netG=resnet\_9blocks, D.use\_sigmoid=True, train G once every 2 iterations.





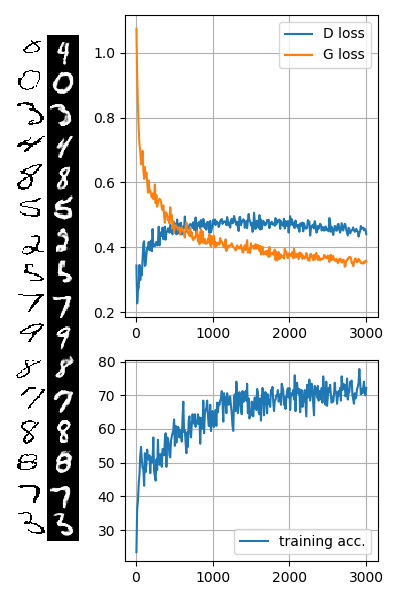
[13] 19/01: vanilla GAN. g\_conv\_dim=64, d\_conv\_dim=64, niter=1500, niter\_decay=1500, batch\_size=32, lr=0.0002, betas=(0.5, 0.999).

Result: 72.4% of test set classified correctly



[14] 19/01: GcGAN. config same as [13] but with lambda\_gc=1.0.

Result: 69.8%



[15] 19/01: GcGAN. Same as [14] but with n\_iter = n\_iter\_decay = 2000 to observe if performance plateaued

Result: 70.2%

