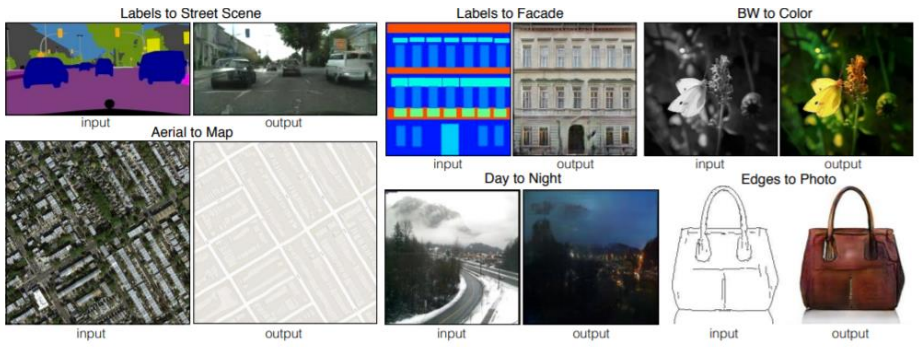
ECE 539 Project Final Report
Image to Image Translation using Generative Adversarial Networks

Shashank Verma

1. Introduction

Ever since the Generative Adversarial Networks (GANs) were proposed by Ian Goodfellow et al. as a new framework for estimating generative models, it has caught the eye of many researchers leading to rapid growth in this area. It has since been applied to various applications like computer vision, natural language synthesis, text-to-image synthesis, image captioning, semantic segmentation and many more. Among these, the application of GANs in image synthesis have produced impressive results (in terms of quality/sharpness of generated images), over and above existing generative frameworks like PixelRNN/CNN and Variational Autoencoders (VAE).

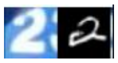
In this project, I plan to discuss and implement a solution for the problem of Image to Image translation. An example of this sort of a problem is translating a possible representation of one scene into another, such as mapping Black-and-White images into RGB images. A few other examples of such a problem is given in Figure 1.

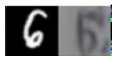


[**Figure 1**: Various examples of Image to Image Translation problems]
Image Source: [3]

2. Problem/Objective

The exact problem this project intends to solve within image-to-image translation is that of domain transfer between the common benchmark MNIST (Modified National Institute of Standards and Technology database) and SVHN (Street-View House Number) datasets. MNIST is particularly useful as its images are small (28x28 pixels) and have only one color; this makes generating images a lot more feasible. The SVHN dataset is similar to MNIST dataset where the images are of small cropped digits, but it has much more diversity and an order of magnitude more labeled data from a real-world problem. Much like MNIST, SVHN requires minimal data preprocessing and formatting. Note that we are only considering the basic SVHN dataset and not the extended one. This has been done to limit the scope of storing and managing the dataset. An example of the expected results are as follows:


[**Figure 2**: SVHN → MNIST Domain Transfer]


[**Figure 3**: MNIST → SVHN Domain Transfer]

3. Work Done/Approach

3.1. Training a Vanilla GAN to generate synthetic MNIST-like digits

Objective: To implement a simple or Vanilla GAN as an unsupervised generative model. This is trained on the MNIST dataset. It should basically generate synthetic samples/images pretending to be from the MNIST dataset.

Summary: The intention of this task was to develop a working knowledge of GANs. The simplest form of GAN that can be implemented is the "Vanilla GAN", which was the first model proposed by Ian Goodfellow et. al in [1]. While many variations and implementations are available by various contributors on GitHub, some of them still suffer with problems like non-convergence or mode-collapse oftentimes owing to unstable training dynamics. Non-convergence was an especially difficult problem in this task as GANs are known have unstable training dynamics and the most basic Vanilla GAN with MNIST takes close to 2 hours to train on an average personal computer hardware.

I read and understood the original paper [1] as part of literature review and ran various implementations of the "Vanilla GAN", finally selecting the model available in [10] which worked best with my compute environment. The "batch size" (toned down to work with the smaller mobile GPU bandwidth) and "learning rate" hyperparameters were tweaked to finally obtain convergence (meaning reasonable output images) with the MNIST dataset.

Steps to train a Vanilla GAN:

1. Pre-processing all data (MNIST) by normalizing it. The MNIST input data ranges between [0, 255] values, and we shall normalize it to range [-1, 1]. Normalizing the data is known to improve the training dynamics according to [9], which also contains other such tricks to make GANs work.

2. Drawing 'n' samples of processed data and noise (from a Gaussian distribution). The real data is the processed MNIST data and the synthetic data is generated by the Generator network (through its weights applied on noise samples.)

3. Training the Discriminator network on these 'n' samples of real and synthetic generated data (more information in step 5). This network has three hidden layers, each is followed by LeakyReLU, which are known to work good in both Discriminator and Generator networks. Sparse gradients usually reflect badly on the stability of training GANs. Additionally, "Dropout" layers are inserted after every layer to prevent overfitting. Sigmoid activation has been applied to the output of the last layer. This network takes 28x28=784 inputs where 28x28 is the dimension of each MNIST image. Both forward and backward propagation is carried out with this configuration, and "ADAM" optimization algorithm has been used for both the discriminator and the generator, since it was heuristically determined to have worked well in various implementations of GANs. The

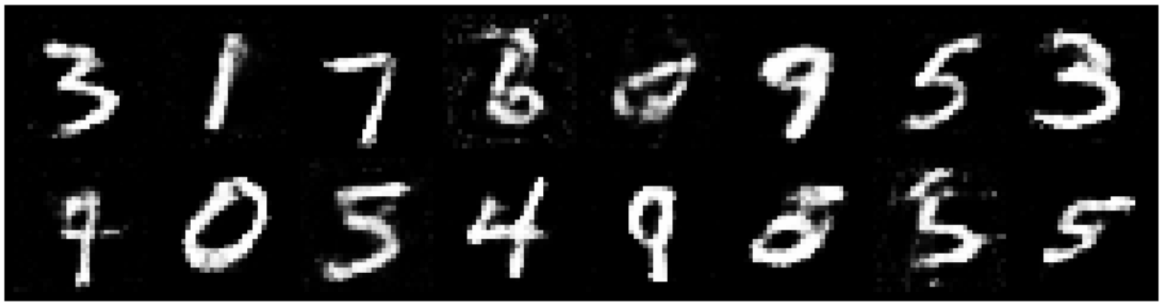
loss function used on backward propagation is the "Binary Cross Entropy (BCE) Loss" which is also unchanged from the original implementation. Following this, the gradients are updated.

4. Drawing another round of 'n' samples from the Gaussian distribution as noise input.

5. Training the Generator network on these 'n' samples. This network also has three hidden layers, each followed by LeakyReLU and having "Tanh" activation at the output. The generator is trained with the same "ADAM" optimizer and "BCE" loss like the discriminator. Due to the nature of Tanh, the output will be in the range [-1, 1], which is of the same scale as the normalized inputs, making it easier for comparison. The predictions from the "discriminator" network here are used for loss calculation, which outputs a probability that a given input is "real" or fake". In the case of the generator, we want to make this probability as close to 1 as possible since we want to fool the discriminator into thinking that the generator produced samples are "real".

6. We repeat steps 3 to 5 until either we visually see that the synthetic samples generated by the generator are satisfactorily close to the real data, and the change from one epoch to another is minimal. Ideally, we would want to stop when the discriminator probability of predicting real data from synthetic is 0.5. (It cannot tell anymore, D(x) = 0.5 and D(G(z)) = 0.5 where D and G are functions modeled by Multilayer Perceptron Networks (MLPs) in the Discriminator and Generator respectively. Also, 'x' is the real data, 'z' is the sampled noise. This is the notation followed in the original paper [1])

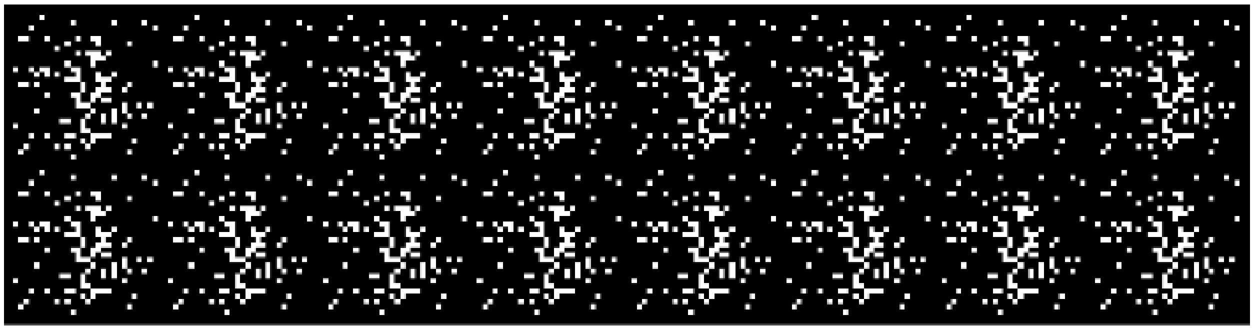
Results: Figure 4 shows "synthetic" or "fake" samples after 200 epochs. They look almost indistinguishable from real MNIST data by human eye. Time to 200 epochs on the Personal Computer hardware was 2.5 hours. Note that D(x) and D(G(x)) are very close to 0.5, but not exactly there yet. Running for more epochs could have potentially brought these model outputs closer to 0.5, but training was stopped at 200 epochs in interest of time.



```
Epoch: [199/200], Batch Num: [500/600]
Discriminator Loss: 1.3624, Generator Loss: 0.8519
D(x): 0.5140, D(G(z)): 0.4404
```

[Figure 4: MNIST "synthetic" samples generated by the Vanilla GAN trained from scratch.]

Issues faced:
Vanilla GAN was particularly difficult to train and get reasonable results. Several times the discriminator becomes too good at its job, and the generator gradient diminishes to a point that it vanishes, and the generator learns nothing between iterations. This is the problem of the **"Vanishing gradient".**



[**Figure 5:** "**Vanishing Gradient**", Generator output even after 4 epochs of training. Discriminator Loss: 0.0002, Generator Loss: 20.2470, D(x): 0.9999, D(G(z)): 0.0000. See that the discriminator gets too good D(x) ~1 and D(G(z)) = 0. It fails to recover out of this till the end of training.]

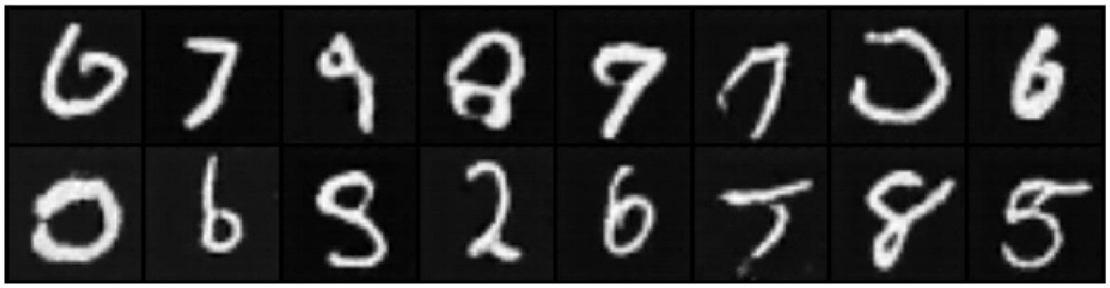
3.2. Training a Deep Convolutional GAN (DCGAN) to generate synthetic MNIST-like digits

Objective: Vanilla GANs with the Multilayer Perceptron Nets (MLPs) for Generator and Discriminator are highly unstable and take a very large time to give reasonable results. The objective is to study and implement DCGANs to learn more about convolutional GANs.

Summary: The "Deep Convolutional GAN" (DCGAN) proposed in [13] by Radford et al. leverages the success of convolutional networks (CNNs) by carefully constraining certain architectural parameters to make Convolutional GANs more stable in training. This task is requisite groundwork to understand and make use of Convolutional layers successfully in the CycleGAN and Semi-Supervised GAN discussed later in the report.

Method/Steps: Most steps like normalizing the data and the skeleton of the algorithm remains same as in the Vanilla GAN. As recommended in the DCGAN paper [13], we use the ADAM optimizer with learning rate of 0.0002 and the momentum term β_1 as 0.5. In the Generative Net, we use multiple convolutional layers with ReLu activation + Batch-Normalization, and the final output through TanH activation. The Discriminative Net also has multiple convolutional layers with "Leaky ReLu" activation + Batch-Normalization, and final layer through Sigmoid activation. This "Leaky ReLu" is in contrast to the original Vanilla GAN paper [1] in which they use the "maxout" activation. Also, all pooling layers from standard Convolutional GANs have been replaced with strided convolutions in discriminator and fractionally-strided convolutions in the generator.

Results:



[**Figure 6:** MNIST "synthetic" samples generated by the DCGAN trained from scratch. These results are after training 23 epochs using batch size 100, and 6 such batches in one epoch (600 images)]

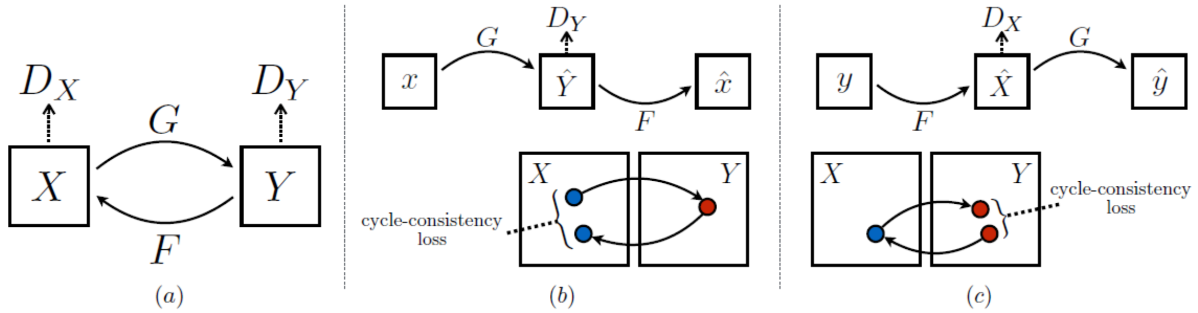
Figure 6 shows that DCGAN generates synthetic MNIST images sharper and clearer than Vanilla GAN (with lesser noise), and in a fraction of the number of trained epochs. However, since DCGANs involve Convolution Nets, each epoch takes significantly larger to train.

In comparison, DCGANs were observed to be more stable, and did not lead to pure noise outcomes on training cycles like the Vanilla GAN did on several occasions. Qualitatively, DCGAN also produced better looking images.

3.3. Understanding and training a CycleGAN for MNIST-SVHN translation

Objective: To implement CycleGAN for Image-to-Image translation between the MNIST and SVHN datasets (and vice versa).

Summary: The CycleGAN model proposed in [11] specializes in using a modified GAN framework for the task of Image to Image translation. Some results that CycleGAN can potentially generate are given in Figure 1. The approach used is to learn a mapping G: X→Y between the source and target domains X and Y respectively such that G(X) is indistinguishable from the distribution Y. In our case X is the MNIST domain and Y is the SVHN domain. As per [11], since this mapping G lacks enough constraints to optimize effectively, G is coupled with its inverse mapping F: Y→X and a cycle consistency loss F(G(X)) ≈ Y is introduced. This method proves to be the better model for image to image translation task according to the results given in the original paper.

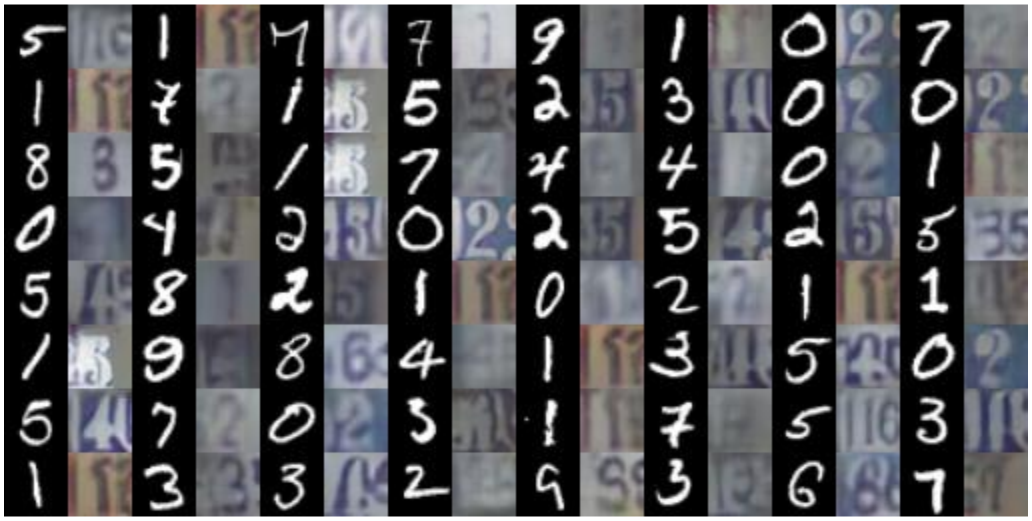


[**Figure 7:** As given above, the model contains two generator networks G:X→Y and F: Y→X, and their discriminators D_X and D_Y. (b) and (c) represent the two cycle consistency losses in both domains.]
Image Source: [11]

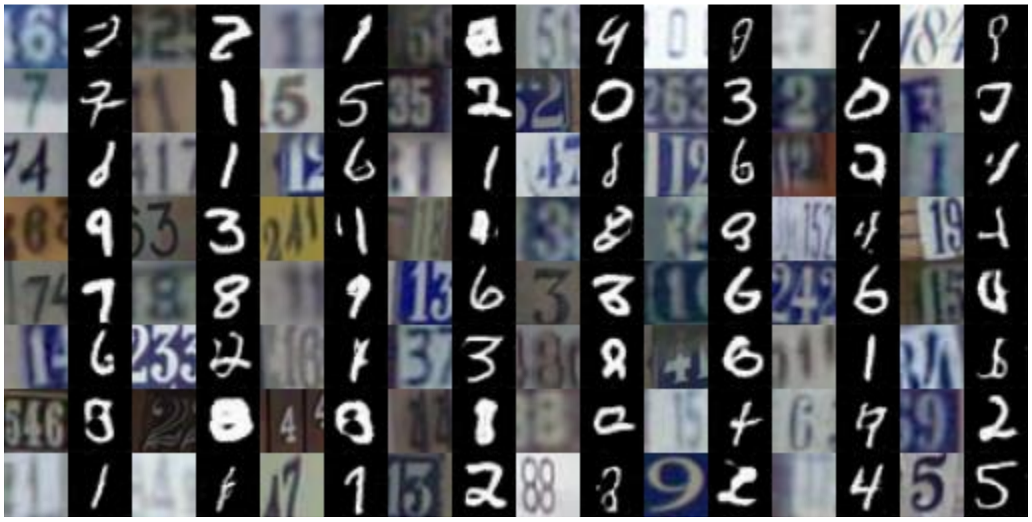
Method/Implementation: We have used a modified version (to suit our datasets) of the CycleGAN PyTorch implementations given in [14] and [15]. Both discriminators D_X and D_Y have convolution layers with "Leaky ReLU activations". The Generators G and F follow a more complicated model of "encoding blocks - residual blocks - decoding blocks" (The decoding blocks are de-convolutional layers). All the four networks have batch normalization in between layers. The learning rate has been fixed to 0.0002, and the momentum β_1 (used in ADAM) has been fixed to 0.5 for all experiments since these hyperparameters are known to be working in the original CycleGAN implementation [14]. In addition, a batch size of 64 was used even though there was a multi-GPU system at disposal as it was giving worse results for larger batch sizes.

This implementation and the details of the model can be found at the code location specified in the Appendix.

Results:


[Figure 8: MNIST→SVHN using CycleGAN after 91000 training iteration**s]**

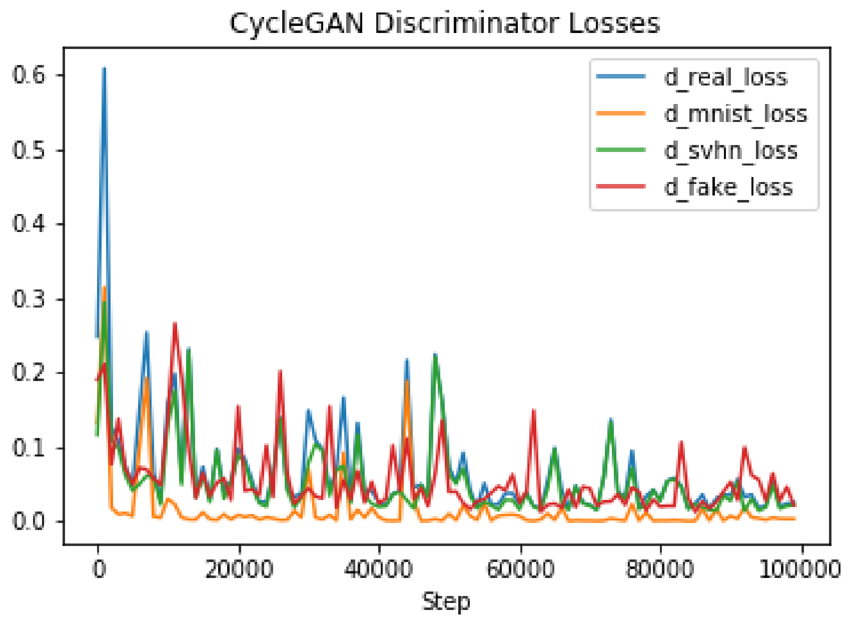
Observation/ Analyses: MNIST→SVHN translation seems to be mapping 1's correctly across domain, and some of digits are translated better than the others. On average, there are more domain translations to a different digit in SVHN. Overall, the synthetic SVHN images do look noticeably close to the real SVHN style digits.


[**Figure 9:** SVHN→MNIST using CycleGAN after 91000 training iterations]

The SVHN → MNIST conversion seems to be working better than MNIST → SVHN. Many of the translations from SVHN to MNIST maintain the digit information. But most of the generated images in the MNIST domain can be identified as handwritten digits by a human viewer.

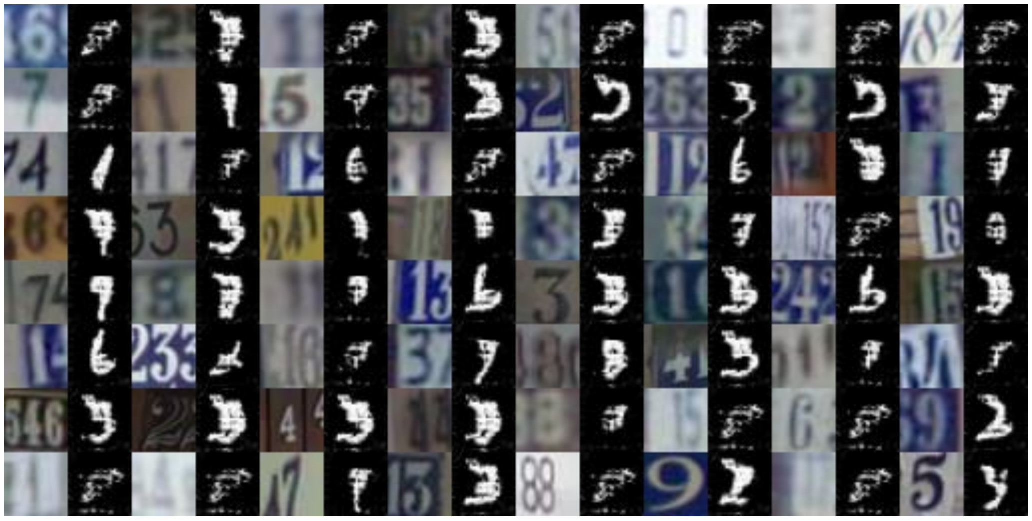
One reason SVHN → MNIST outputs look slightly better could be because the SVHN images also contain "color" information as opposed to MNIST images which are "grayscale". The generator in SVHN → MNIST can easily lose this color information and just focus on minimizing loss with respect to the digit shape information. However, the generator network for MNIST → SVHN cannot be expected to generate both the color information and the digit mapping. This is especially because we are **not using number labels for training**.

Note that the plots for discriminator and generator losses aren't particularly useful or very indicative in GANs. Only visual inspection of results can tell us about the performance of our network. But the following plot does tell us that the discriminator losses reduce over time, indicating that they are getting better at telling apart synthetic images from real.

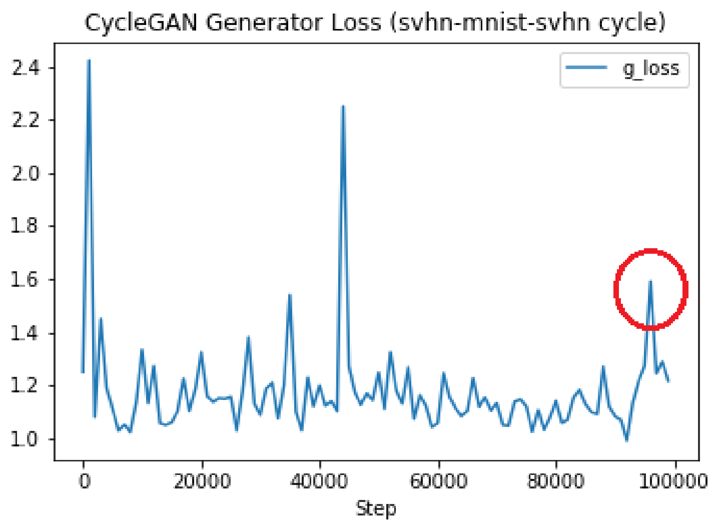


[**Figure 10:** CycleGAN Discriminator losses. 'd_real_loss' and 'd_fake_loss' are the sum of both discriminator's losses on real data and fake (synthetic/generated) data respectively. 'd_mnist_loss' and 'd_svhn_loss' are individual losses for MNIST and SVHN discriminator on real data]

Issues faced/ Failures:
Note that the results presented in figures 8 and 9 are at 91000 training iterations, while the complete training was done for 100000 iterations. This is to highlight now that the SVHN → MNIST generator network collapsed soon after at 92500 iterations and failed to recover completely till the end of training period.



[**Figure 11:** Collapse of SVHN → MNIST generator at 92500/100000 iterations]



[**Figure 12:** Collapse of the SVHN → MNIST generator coincides with a bump in the generator loss. These bumps are present in the plot before as well, but they have recovered soon after. Since the training was only done for 100000 iterations, we may not have given it enough time to recover. This, however, still highlights the unstable training dynamics of our CycleGAN implementation.]

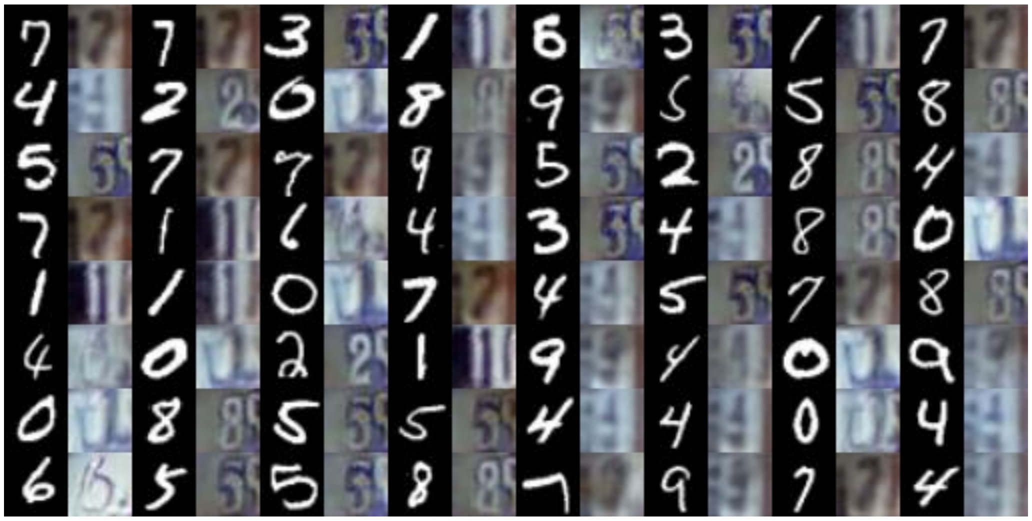
3.4. Understanding and training a Semi-supervised GAN (SGAN)

Objective: To implement SGAN for Image-to-Image translation between the MNIST and SVHN datasets (and vice versa). This model contends against the previous CycleGAN model.

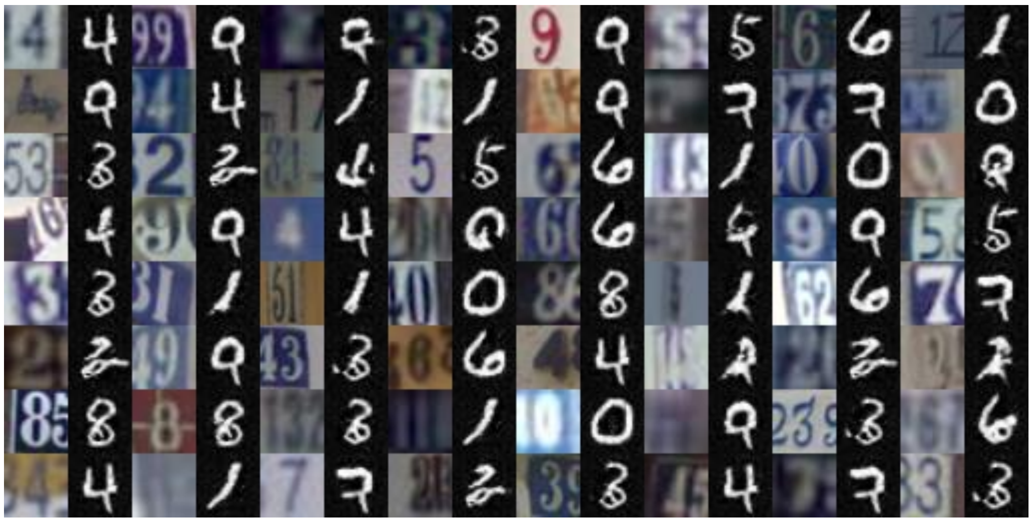
Summary: The SGAN was proposed by Odena, A. in [5] where the discriminator is forced to output class labels as well. The generative network G is trained normally as in normal GANs, but the discriminator network is trained on a dataset with inputs belonging to one of N classes. The discriminator is then made to predict that which of N+1 class labels does its input belong to where the additional class is for "synthetic" images.

We use this idea with the same exact structure, implementation and hyperparameters like the CycleGAN, but will force the discriminator to output the correct digit label instead of just a probability of realness. The discriminator losses will now be computed using a Cross Entropy Loss between predicted label and the real label. The "synthetic" image will be one of the labels, making it 11-class classification for 10 digits + 1 generated image.

Results/Analysis:

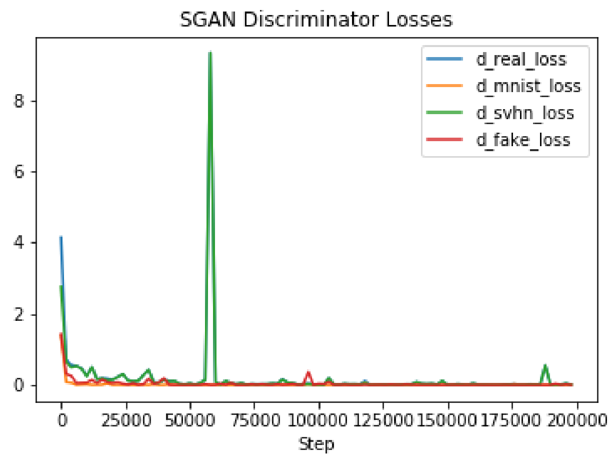

[Figure 13: MNIST → SVHN using Semi-Supervised GAN after 100000 training iterations]

Most of the digits in MNIST have transformed to the same digits in the SVHN domain. Nonetheless, even with this network, some of the digits are unclear even to a human viewer (let alone a classifier trained on SVHN), for instance some of the MNIST zeros have translated to a digit in SVHN that look like a mixture of 0 and 1. Also, some of the generated 4's are more likely to be classified as 1's by a human viewer.


[Figure 14: SVHN → MNIST using Semi-Supervised GAN after 100000 training iterations]

By visual inspection of results, we could say SVHN→MNIST translation has turned out even better than MNIST → SVHN. This could be because of the same reason as in the CycleGAN case where it might be easier for the network to ignore color information than to generate it. (Since SVHN is a colored dataset and MNIST is grayscale.) The reason we did not preprocess the SVHN images as grayscale as it defeats the purpose of "style" transfer. We would like the SGAN/CycleGAN to simulate the color information as well (or lose it in case of SVHN → MNIST).

Overall, these results are much better as compared to those of the normal CycleGAN in the previous section. Also, the SGAN model exhibits more stable training dynamics as compared to CycleGAN.



[**Figure 15:** SGAN Discriminator losses (Plot labels use same convention as in Figure 10).]

4. Conclusion

1. In this project, a Vanilla GAN and a Deep Convolutional GAN were trained from scratch to generate synthetic MNIST digits. **DCGANs were observed to be more stable in training** (they did not lead to pure noise outcomes on training cycles like the Vanilla GAN did on several occasions). **Qualitatively, DCGAN also produced better looking images.**

2. Among other models, the Vanilla GAN was particularly difficult to train. This may be because the model parameters **oscillate**, **destabilize and never converge**. There were also cases when the generator network was trained a bit too much on certain cycles and it learns that it can cheat the discriminator by specializing in generating only one if the digits well. This problem is referred to as "**mode collapse**" in several texts. Also, the **"Vanishing Gradient"** problem was faced multiple times and has been discussed in Section 3.1.

3. For the task of image to image translation, a CycleGAN was trained to transfer style between MNIST and SVHN datasets (and vice versa). Several samples the generated images did look like target domain digits, but most of them did not retain the correct digit information.

4. A modified version of CycleGAN with the semi-supervised method mentioned in [5] was trained (as SGAN), where we used label information to calculate loss of the discriminators. **The SGAN model performed better than CycleGAN, where in addition to the expected style transfer, the correct digit was retained on more occasions as compared to CycleGAN.**

5. Note that SGAN makes use of class labels for training the discriminator making it a semi-supervised method. There could be a scenario where these labels might not be present, or they may be too expensive or cumbersome to obtain. Hence, the project uses MNIST and SVHN datasets only, both of which are available freely and are labelled.

6. All but one of the tasks planned for this project were completed successfully. In addition to this, training of a DCGAN was a pop-up task which was unplanned but was imperative to gain a deeper understanding of using Convolutional layers in GAN.

7. Networks were compared by visual inspection of generated results, and it is recommended to use DCGAN over Vanilla GAN, and SGAN version over CycleGAN when dealing with similar datasets and using similar hyperparameter settings.

8. The only task that was planned but couldn't be fully completed was using pre-trained classifiers (for MNIST and SVHN) with high accuracy to classify generated style-transferred digits generated by CycleGAN and SGAN. These off-the-shelf high-accuracy classifiers will then give correct classification rate on generated digits of their respective domain, which will give us a measure on how well the digit information was retained during style transfer. This method is like the FCN score used in the CycleGAN paper [11].

9. Another task that was out of the scope of this project was doing an exhaustive hyperparameter search for comparing the above discussed models. It is probable that some models perform better than others only on certain values of these hyperparameters or on different datasets, and it is likely that better combinations exist. With the limited scope of this project, this report refrains from generalizing conclusions.

5. References

1. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).
2. Huang, H., Yu, P. S., & Wang, C. (2018). An Introduction to Image Synthesis with Generative Adversarial Nets. *arXiv preprint arXiv:1803.04469*.
3. Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. *arXiv preprint*.
4. Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved techniques for training gans. In *Advances in Neural Information Processing Systems* (pp. 2234-2242).
5. Odena, A. (2016). Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*.
6. Zhu, J. Y., Zhang, R., Pathak, D., Darrell, T., Efros, A. A., Wang, O., & Shechtman, E. (2017). Toward multimodal image-to-image translation. In *Advances in Neural Information Processing Systems* (pp. 465-476).
7. https://github.com/hindupuravinash/the-gan-zoo
8. https://jhui.github.io/2017/03/05/Generative-adversarial-models/
9. https://github.com/soumith/ganhacks
10. https://github.com/diegoalejogm/gans
11. Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv preprint*.
12. https://github.com/jcjohnson/fast-neural-style
13. Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434.
14. https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix
15. https://github.com/yunjey/mnist-svhn-transfer

6. Appendix

6. 1. Framework

The framework of choice in this project is **PyTorch** because of the following reasons:

1. It's close similarity with the NumPy framework, which makes it convenient to use.
2. PyTorch is more imperative and uses dynamic computation graphs which is preferable over TensorFlow's static-graphs. In TensorFlow, once we define the structure of the graph, it remains constant and we have no control over

it. In PyTorch, it is also possible to use control flow techniques in programming like the for loop, and if-else statements.

3. Easier debugging since PyTorch calculates graph on-the-fly.
4. Established and growing researcher and developer communities.

Note that the above comparison is between PyTorch v0.4.1 and TensorFlow v1.12.

6. 2. Hardware Used

Hardware 1:
The personal laptop was used to train the Vanilla GAN
Processor: Intel® Core (TM) i7-8550U CPU @1.80GHz, 4 Cores, 8 Logical Processors
Graphics Card: Nvidia GeForce MX-150

Hardware 2:
The Euler Supercomputing cluster was used for all other models apart from the first Vanilla GAN.
Processor: Intel® Haswell CPU (4 Cores in use)
Graphics Card: Nvidia GeForce GTX-1080 (x 2)

6. 3. Code Location

All code for this project is publicly available at this location:
https://github.com/shashank3959/GAN