NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6121 - Computer Vision

Project: Image Translation and UDA

Done by:

Lam See Hwee **G2102084K Email: seehwee001@e.ntu.edu.sg**

Liu YaoHong **G2203319C Email: liuy0220@e.ntu.edu.sg**

Patricia Njo **G2205032J Email: patricia003@e.ntu.edu.sg**

Content

1. Introduction

What is image-to-image translation? In the context of computer vision, it is defined as the class of problems associated with vision and graphics or described as a synthesis task performed on images.

The main objective of image-to-image translation is to learn and figure out how a training set of images can map between the source and an output image of a given scene or unique characteristic. The translation performed either has the presence of unpaired or paired data.

In this assignment, there will be two sections of presentation assignment:

1. The first section will describe and discuss the image translation network CycleGAN, which our team have adopted on the unpair dataset and has been guided by the paper [3]
2. The second section will discuss and practice UDA via input-space alignment.

2. Dataset

The datasets selected for the 2 tasks:

Task 1

GTA5toCityscape: This data is a commonly used dataset for training for CycleGAN. This dataset has 22080 images of GTAs and 22061 of cityscape images. This image is commonly used in CycleGAN research papers. This dataset has been split into train and test datasets. It is retrieved from the link efrosgans.eecs.berkeley.edu. This dataset consists of Train and Test data.

Task 2

Cityscapes&GTA: In Source-Only task we use the both Cityscape and GAT dataset with the labeled image to feed into CycleGAN. During UDA task, we also use the generated image by Task 1 model as training image and use real Cityscape image as target image

3. Task 1: Image-to-Image Translation

In this image-to-image (i2i) translation section 1 task, our team trained on the Cycle Generative Adversarial Model (GAN) and performed object Transfiguration on the unpaired image using GTA5toCityScape.

In the past, the traditional method of image translation of the dataset commonly used was paired or corresponding images. Although they can attain high-quality image translation, the number of datasets used in the research areas is limited and hinders advancement because they are expensive to collect. The pair translations are frequently introduced in the medical industry, especially in disease detection like MR to CT or enhanced degraded images (Pauliina Paavilainen, 2021).

The formula of Paired Images is given by: $\{x_i, y_i\}_{i=1}^{N}$ where the X and Y denote the source and target images.

In order to resolve the challenges encountered by paired image translation, unpaired image translation techniques are being designed. The unpaired image translation is the opposite of the paired translation, which indicates that they do not have a pair or corresponding direct images to be

used in the image translation. Unpair image-to-image translation aims to connect and have the ability to map the source to the target. Figure 1 illustrates the differences between paired and unpaired. One famous technique is the Cycle-Consistent GAN, which is also a part of the Generative Adversarial Models (GAN) (Bengio, 2014) family used for the unpair image-to-image translation.

The formula of Unpair Images is given by and $\{y_j\}_{i=1}^{M} (y_j \in Y)$

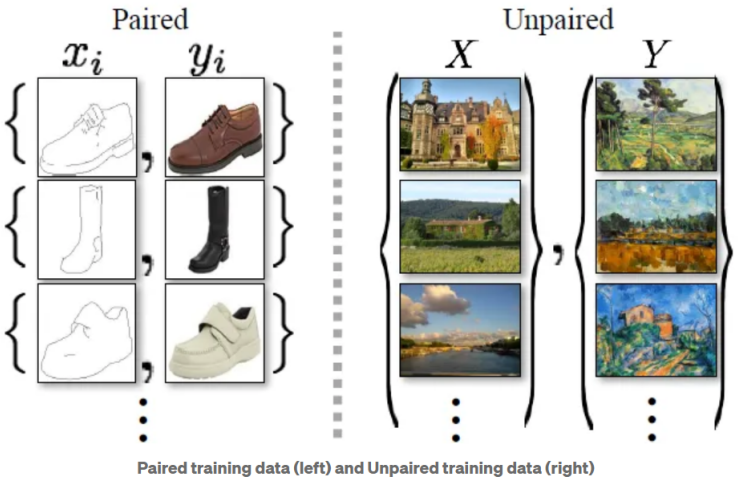


Figure 1: Example of the Paired and Unpaired

3.1 Generative Adversarial Models (GAN)

As mentioned previously, Cycle-Consistent GANs are part of the GAN family, where the architecture is built upon the GAN architecture. Therefore, it is essential to understand the fundamental structures of Generative Adversarial Models (GAN).

A Generative Adversarial Model (GAN) is a type of deep neural network architecture consisting of 2 Neural Networks. Under this neural network architecture, they have generators and a discriminator.

The Generator or generative model is a deep learning convolutional neural network (CNN). Its primary function is to generate output images, which can be mistaken as actual images. While the discriminator or discriminative model is commonly known as the deconvolutional network. Its primary function is to discriminate between actual and predicted samples. For example, the Generator can classify actual and fake images from real-life images, trying to produce fake images that cannot be easily detected. The Discriminator will take the role of detective to detect fake images and have the ability to determine the authenticity of the images. Both modules will compete to improve their methods until the images are identical to the authentic images with the help of Adversarial Loss. However, the downside is that it cannot generate a translation for each image due to the limitations of the models.

Therefore, our team will implement the Cycle Generative Adversarial Model (GAN) as our task 1 model in our assignment, where it can perform translation for each image with the help of the extra neural network.

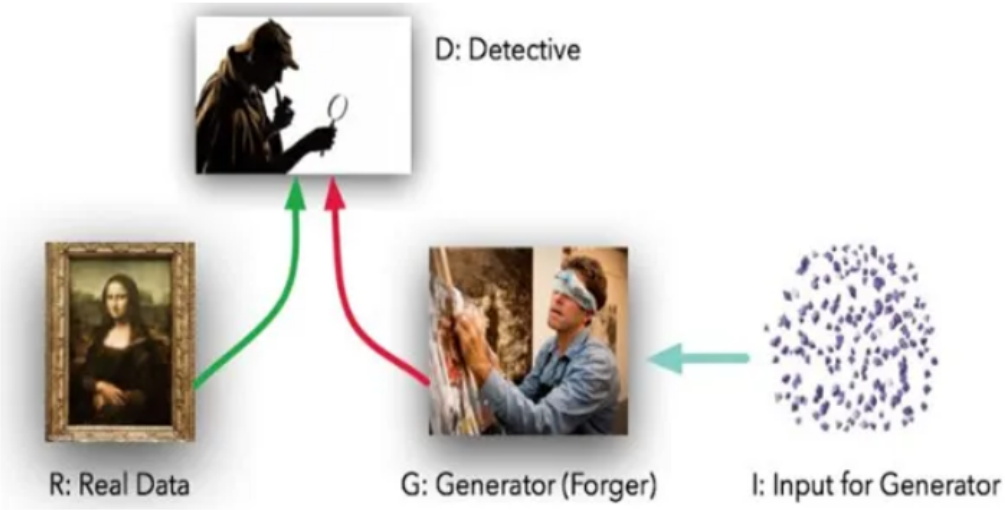


Figure 2: Illustration of GAN in our description

3.2 Cycle Consistent Generative Adversarial Models (CycleGAN)

As introduced previously, the Cycle Consistent Generative Adversarial model (GAN) structure is built upon the GAN structure, which has 2 more neural networks with an additional 1 generator and 1 discriminator each compared to the GAN, accumulating into 4 neural networks.

CycleGAN is commonly used on unpaired datasets training and performing image translation between the Source and the target image even when no corresponding images exist. It also leverages Cycle Consistency to maintain the consistency of transformations in the quality and ensure that the generated image is identical to the original image. Under the CycleGAN, the data are concurrently trained on the discriminator and Generator.

The first batch of images will be taken in as an input and output for the second set of images, which will later be taken in by the second Generator as an input and generate images for the first batch set. Using our dataset as an illustration, the first batch set is a GTA, and the second batch is a cityscape In the first Generator, and it will generate output images that look like cityscape. During the process, the first Generator takes all the features or colours of the cityscape(target) and applies them to the GTA(Source), generating the layer of cityscape image on top of the GTA. While the second Generator, did it oppositely by taking in the image of a cityscape and generating the output of a GTA during the process. Similarly, the second generator will take in the features or colour of the GTA and apply it to the cityscape to generate an output image of the cityscape.

Example: X denotes as GTA, Y denotes cityscape as in the Figure 3

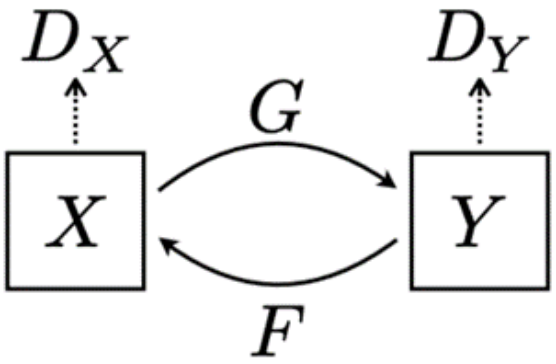


Figure 3: Illustration of the 2 mapping functions used in the model

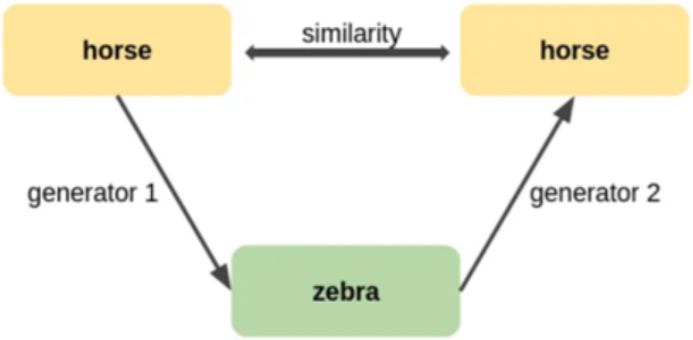


Figure 4: Illustration based on the dataset used

Although CycleGAN does not require any corresponding image sample to train the model. However, it still requires the collection of the images from each set. The translation is performed to handle non-pair images by connecting and extracting key features. Integration of the loss functions can control the mapping, which will be described in more detail under the Model Architecture Design.

3.2.1 CycleGAN Loss Function

The initial function contains two terms. The first term refers to the adversarial losses incurred when the generated images are matched to the target domain. The second term refers to the cycle consistency loss during the prevention of inconsistency in the mappings of G and F. Next, adversarial losses are implemented to both mapping functions. This is the formula for the adversarial loss of mapping G:

$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x))] \quad (1)$$

The formula for the adversarial loss of mapping F follows the same format. The mappings of G and F can produce target domains Y and X. The goal is to ensure G and F are cycle consistent with the target domains. In other words, we don't want G and F to be random.

There are two types of cycle-consistency: forward cycle-consistency and backward cycle-consistency. Firstly, forward cycle-consistency means the image translation cycle should be able to reconstruct x to its original image. This is what forward cycle-consistency looks like:

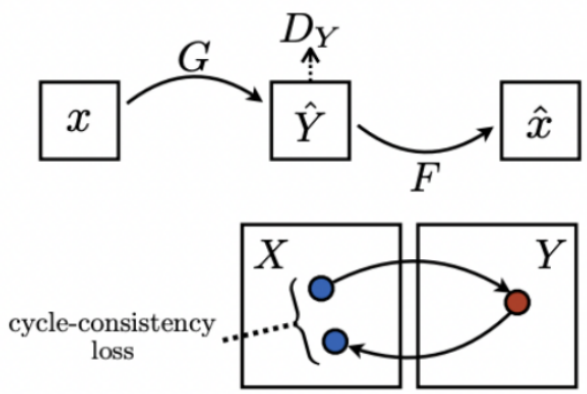


Figure 5: Forward cycle-consistency

Secondly, backward cycle-consistency means the image translation cycle should be able to use its original image to reconstruct y. This is what backward cycle-consistency looks like:

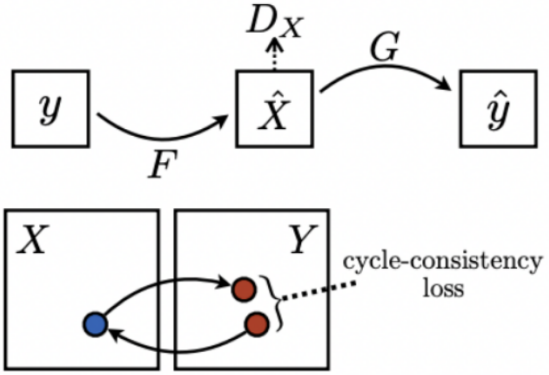


Figure 6: Backward cycle-consistency

Therefore, combining forward cycle-consistency and backward cycle-consistency together creates the cycle consistency loss function. Below is the formula of a cycle-consistency loss function:

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G,F) =& \mathbb{E}_{x\sim p_{\text{data}}(x)}[\|F(G(x))-x\|_1] \\ &+\mathbb{E}_{y\sim p_{\text{data}}(y)}[\|G(F(y))-y\|_1].\end{aligned} \tag{2}$$

This is the final formula of the CycleGAN loss function which consists of 2 GAN loss functions and a cycle-consistency loss function:

$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) =& \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ &+ \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ &+ \lambda \mathcal{L}_{\text{cyc}}(G, F),\end{aligned} \tag{3}$$

3.3 Methodology

3.3.1 Data Preprocessing

The image data are being obtained from the GTA (Richter et al) dataset and the cityscapes dataset(Cordts, Marius, et al) with annotation of labeled data in pixel level. The GTA training dataset contains 22080 images which will be reloaded and take in the size 256 * 256 of images to reduce the cost of computation time. Subsequently, the data is then loaded into the model for training.

Below are images samples which are being selected randomly to give us a deeper understanding and insight on what type of image exists and has been loaded for the image to image translation task.

Figure 8: Random 4 image from the both data set

In addition, since our team adopted the Keras package, some files, such as Keras-contribs, a high-level network, are implemented in this training. This high-level network contains Conditional Random Fields that are more advanced and related to Keras. It is a package where high level computation of the layers are being calculated.

```
pip install git+https://www.github.com/keras-team/keras-contrib.git

Collecting git+https://www.github.com/keras-team/keras-contrib.git
  Cloning https://www.github.com/keras-team/keras-contrib.git to /tmp/pip-req-build-yhtzv_9o
  Running command git clone --filter=blob:none --quiet https://www.github.com/keras-team/keras-contrib.git /tmp/pip-req-build-yhtzv_9o
  warning: redirecting to https://github.com/keras-team/keras-contrib.git/
  Resolved https://www.github.com/keras-team/keras-contrib.git to commit 3fc5ef709e061416f4bc8a92ca3750c824b5d2b0
  Preparing metadata (setup.py) ... done
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (from keras-contrib==2.0.8) (2.14.0)
Building wheels for collected packages: keras-contrib
  Building wheel for keras-contrib (setup.py) ... done
  Created wheel for keras-contrib: filename=keras_contrib-2.0.8-py3-none-any.whl size=101056 sha256=e2e6ef42e80b90da82b757f190879470c9633ddb2061ab728a99902b95ccdcab
  Stored in directory: /tmp/pip-ephem-wheel-cache-u0du92kk/wheels/74/d5/f7/0245af7ac33d5b0c2e095688649916e4bf9a8d6b3362a849f5
Successfully built keras-contrib
Installing collected packages: keras-contrib
Successfully installed keras-contrib-2.0.8
```

Figure 9: Conditional Random Fields.

3.3.2 Model

Our team had designed the CycleGAN architecture with Keras to perform the unpair image-to-image translation based on the literature paper proposed by Efros et al.

3.3.2.1 Discriminator

Under CycleGAN, the Discriminator adopts the design of PatchGAN(Phillip Isola, 2018). Under the CycleGAN, the Discriminator penalizes the image patches locally instead of penalizing the entire image. The individual image patches will then be categorized as either real or fake.

The PatchGAN uses a neural network model like the Convolutional-BatchNorm-LeakyReLU layers to provide an overall score by averaging the category. The neural network model generally consists of a few components: The convolutional layers, activation functions, normalization, strides convolutional layer, and the output layer.

In the convolutional layer, it takes in an image of a GTA(Source) while increasing the number of channels or features, which allows us to learn on the extracted features of the image. The activation function is used to identify the real and fake images and also aids in problems related to gradient diminishing. Apart from these roles, it also supports training stability and influences the learning process. The next layer, the strides convolutional layer, is used for down-sampling in order to reduce the dimension of the feature map. In the model, each output feature map ensures that each feature map is normalized.

Furthermore the discriminator will perform the training directly on both the real and generated images.

<u>Our model implementation:</u>

Under our model, there are 8 layers of Convolutional Layer with no inclusion of Resnet Layer. The Discriminator will take in tensor of shape(1,3, 256*256). Our model uses the L2 Least Square loss as the optimization as specified by Efros et al . Therefore the standard deviation's weight of 0.2 from , which allows faster convergence during training and adopts 0.5 as the weight in the model mainly to slow down the changes of the Discriminator and provide a fairer input to the Generator in training.

LeakyRelu has also been introduced to the convolutional layer and RELU in the Resnet block layer with alpha=0.2 under the activation function. The reason for using a small alpha is to allow the gradient to pass through efficiently when the unit is inactive. The update of the Discriminator does not use the latest generated result but uses the generated image history. To check if the image is real or fake, the Discriminator will check in every overlap 70*70 area iteratively.

However, in this implementation, our team modified and introduced binary classification to determine whether the image was real or fake. The Loss has also been changed to a binary classification task instead of a Mean Square Error. This is because it is easy and robust to implement into a large dataset.

3.3.2.2 Generators

Apart from the Discriminator, the Generator is the next most essential component in the CycleGAN, where it adopts encoder-decoder architecture. The generators take in the GTA(source) and generate the cityscape(Target).

Similar to the component of the Discriminator, the generator neural network consists of a few components: Encoder, bottleneck, Decoder, and output layer. The Encoder uses the convolutional layer and takes in a GTA image(Source) while increasing the number of channels or features. Key

features are extracted based on lower dimensions or representations, which are dense through image compression in the Encoder. Non-linear activation like LeakyRelu or RELU is used in the deep neural network. Figure 10 is an illustration of the CycleGAN generator neural network design . After extracting the key feature, it will pass to the bottleneck, representing the input data in compressed form. It consists of the neural network concept with skip connection, known as the ResNet Block Layer and within this block layer, it has normalization and activation functions. In CycleGAN, several Resnet layers help interpret and transform the encoded features, retaining their information.

Furthermore, adopting the skip connection can aid in training a network more deeply while easing the vanishing gradient problem. The other component is the Decoder, which will receive the image representation from the bottleneck. In this component, the target image will be reconstructed and performed oppositely to the Encoder, where it will increase the dimensions or representation while reducing the number of channels or features and increasing the dimensions or representations, which are dense through image compression. This will allow the target image to be reconstructed. Lastly, the output layer uses a convolution layer with a function like sigmoid, tahn or relu to generate the final image.

However, the generator is opposite as a discriminator where it did train on the real and generated but on the fake images. Subsequently the generators will be trained from the discriminator where they perform adversarial loss where it updates in order to minimise the generated image predicted loss by the discriminator. This will provide a better fitting result for the image which is generated to the target set of images. Furthermore, cycle loss is also performed based on the effectiveness of the source image being regenerated when it is adopted by the generator model. The identity loss as a part of cycle consistency will provide a final output images to the target set without translation being applied.

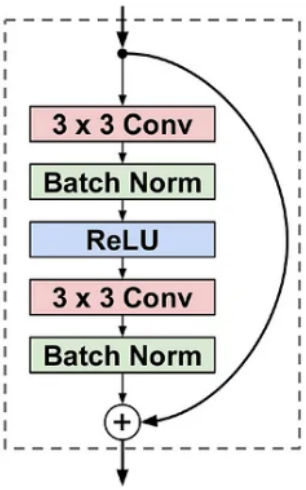


Figure 10: illustration of the CycleGAN generator neural network

Our model implementation:

In our Generator design, we have three different components: one is a Resnet model, a convolutional network, while the other is the composite model, which performs the merging of both Generators and Discriminator to update the results for the 2 losses: Adversarial and Cycle Loss.

This implementation uses the Keras package and will be in Conv2D, requiring input in the row, width and channel. The Resnet block component uses a 3*3 filter convolutional layer, which uses instance normalisation where it only normalises the instances rather than batch. It also adopts RELU activation in the 1st block, while the adoption of RELU activation in the second block is omitted.

The generator model uses 9 Resent blocks. Similar to the discriminator described above, there are 8 layers of the Convolutional Layer with no inclusion of the Resnet Layer. It takes in tensor of shape(1,3, 256*256). Based on Efros et al., the standard deviation's weight of 0.2 and our team decided to adopt it to allow faster convergence during training. In addition, we also adopt similar parameters of 0.5 as the weight in the model and also use RELU and tahn as activation as well. The discriminators differ because the generator implements a layer with Conv2Dtranspose. This allows the image's original size to be mapped back with a decreased channel size. Furthermore, the generator does not train on the real and generated image but trains on their related discriminator models.

As discussed previously, the composite model component consists of 2 inputs for the actual image from A and B sets of images. The discriminator is allocated 4 outputs for 4 loss functions. The 4 loss functions are Adversarial Loss, Identity loss, Forward and Backward Loss,explained on session 3.2.1. Their roles are different in the translation task and from the Source to the target, and vice versa. The forward loss and backward loss are a form of cycle consistency loss where its main function is to allow the different set of images to be converted back to its raw form. The model's weights remain constant and will not be trained. However, it will be updated with the sum of all losses to update the weights of the main generator performing the translation instead. In a similar adaptation based on Efros et al , the weights are assigned where cycle loss is given more weight than cycle loss and identity loss. These Loss functions are calculated during the training process, which helps to handle domain translation in the event of no pair data being available.

Below is an illustration of the flow in the A generator for translating the Target to the Source.

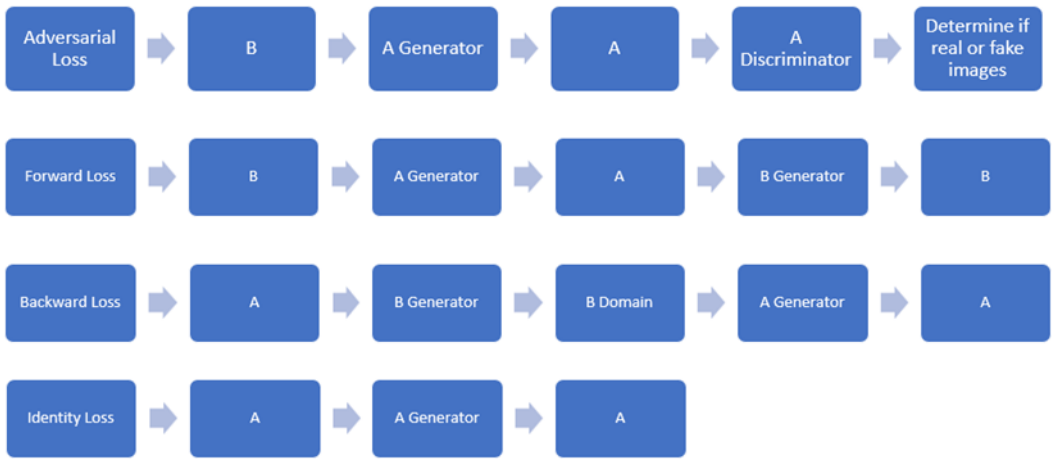


Figure 12: illustration of the flow in the A generator

Below is an Illustration of the flow in the B generator for translating the Source to the Target. It performs in different directions depending on the input sets.

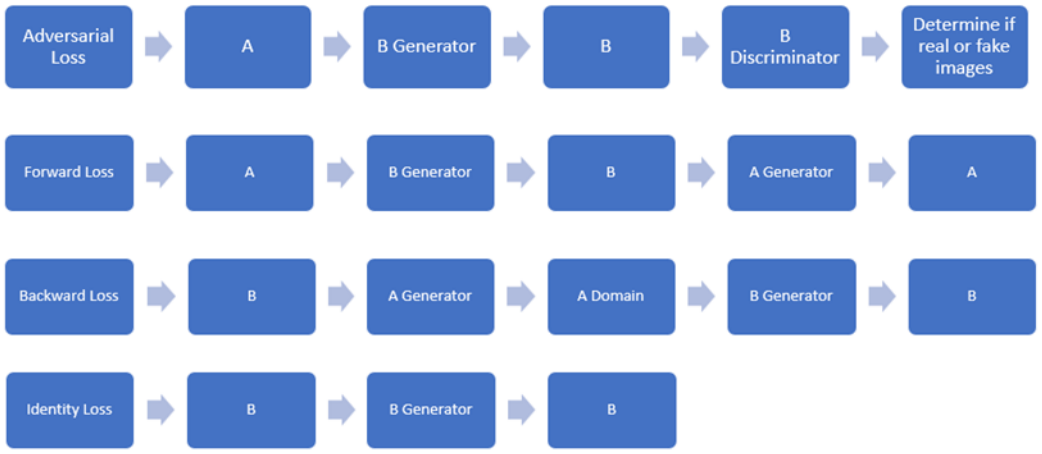


Figure 13: illustration of the flow in the B generator

3.3.3 Model Training

3.3.3.1 Training Process

During the training process, our team adopted the adversarial methodology where both the Generator and the Discriminator are concurrently training. During this training process, the Generator will do its part to generate images which are more identical while the Discriminator is to decrypt which is the fake or real image. In this process, epoch=100, bacth_size=8,the optimizer is Adam with β1=0.5 and β2=0.99, loss_weight =0.5.

Our model implementation:

In our CycleGAN model, as mentioned both Generator and Discriminator will concurrently train. The Generator will use the image pools to store the images generated and then feed them into the Discriminator to train. The image will be consistently updated and not being trained.

3.3.4 Results

The results from this training are being generated below after a few run trains by my team . Randomly selected image from to view the result .

In the beginning, our team used a small dataset, horse-to-zebra, to prove the flexibility of our model. Our team observed that if the epoch is 70 it will affect the image's result where there will be a blur layer on the image itself . This gives an indication that it is not sufficient to allow the model to know how to map and underfitting occurs. This also aligns with Efros et al where they used 100 epochs in their experiment. In addition, based on the co-author Junyanz, it stated that there is no way to give any indication on what will be the optimal epoch to train . Therefore the only way is evaluating different epoch or applying the common used and also based on the image quality and requirement .

Below Image are produced after training with 70 epochs:

Figure 14: Result of horse to zebra

After testing the flexibility,we fed the GTA-to-cityscape data to train the model. At the end, testing with cityscape, the result is showing:

Figure 15: Result of cityscape to GTA

By comparing the result with 2 models we can see, CycleGAN only changes the color and texture of the image, changing the shape of the image is still a challenging task.

3.3.5 Constraints of CycleGAN

Although it is state of the art to generate the unpair images, there are some constraints in adopting the CycleGANs, which our team have discovered in this training process.

The identified constraints are:

1. Unable to translate changes in geometric: CycleGAN's primary design is to learn and transfer style and texture while maintaining the original appearance. It is not designed to

handle and understand the geometric changes in the images. Furthermore, it is limited by the cycle loss, where the objective is to allow the image to retain its original form even if it has been transformed.

2. Requiring longer training time and computation resources is high: The number of neural networks implemented in CycleGAN increases the time the model takes to train. In addition, in CycleGAN, there is no solution that can determine the best epoch to use in training. It will be based on the application's specific results, requiring long training and computation resources to derive the best epochs to utilise.

3. Drop in Accuracy: Since CycleGAN does not require any corresponding direct image matches. It may not always generate accurate images compared with paired images if the image setting is in complex scenes because of its limited distribution in the dataset. Therefore, it requires a large diversity or distributed dataset to perform image-to-image translation.

4. Only able to perform on 2 data sets: CycleGAN allows only the translation image-to-image between 2 domains at a time. Computation and model complexity increase with the increase of the domain or set of images required.

4. Task 2: Unsupervised Domain Adaptation (UDA) via Image-to-Image Translation

4.1 Unsupervised Domain Adaptation (UDA)

This section will describe image-to-image translation using the input space Unsupervised Domain Adaptation, which is also known as UDA. The machine learning framework adopts transfer knowledge to process the source domain to the target domain. This task is getting ideas from CycleCADA(Judy Hoffman, 2017), and leveraging it to the CycleGAN. In this Domain Adaptive Semantic, the Source will use the fake image generated by the CycleGAN model, while on the other hand, the target data is the real image. In UDA, it consists of an input space and a label space. Similar to Task 1 CycleGAN, UDA is also used to tackle the image-to-image translation task.

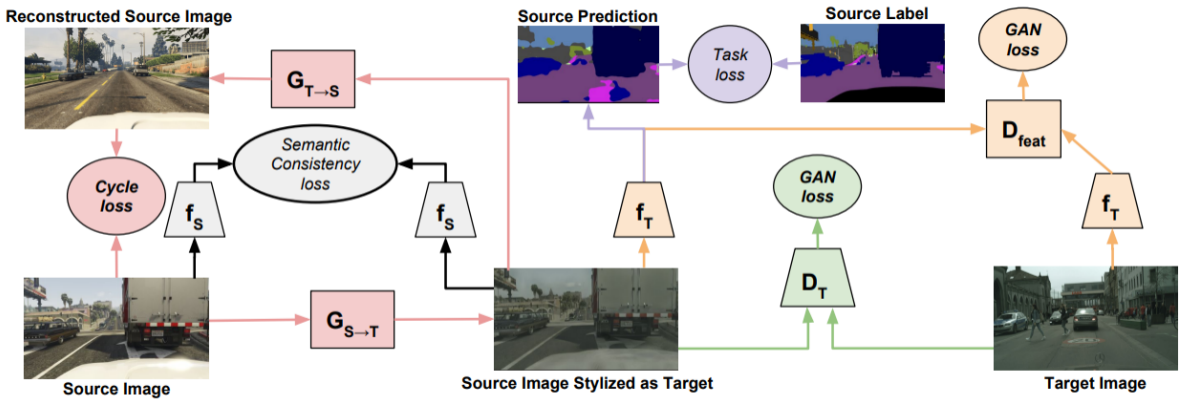


Figure 16. Process of UDA via Input-Space Alignment

In UDA, the input image is the source image stylized as target (see Figure 16) which is generated by the CycleGAN model with target image style. The target image is the real image. UDA is performing

to predict the image segmentation, it uses the source prediction and source label to compute the task loss. During this task, our team adopted the script from Anurag et al.

4.2 Source-Only Semantic Segmentation Model

Firstly, our trainA dataset includes 3000 images from GTA5 and our trainB dataset includes 3000 semantic label images from GTA5. Secondly, our testA dataset includes 100 images from Cityscapes leftimg8bit_trainvaltest.zip, which would produce 100 semantic label images. Consequently, the model was trained for 100 epochs, the batch size of 8, learning rate of 0.0002 and the optimizer is Adam with β1=0.5 and β2=0.99.

Figure 17 shows the implementation of the source-only semantic segmentation model, which involves training on CycleGAN.

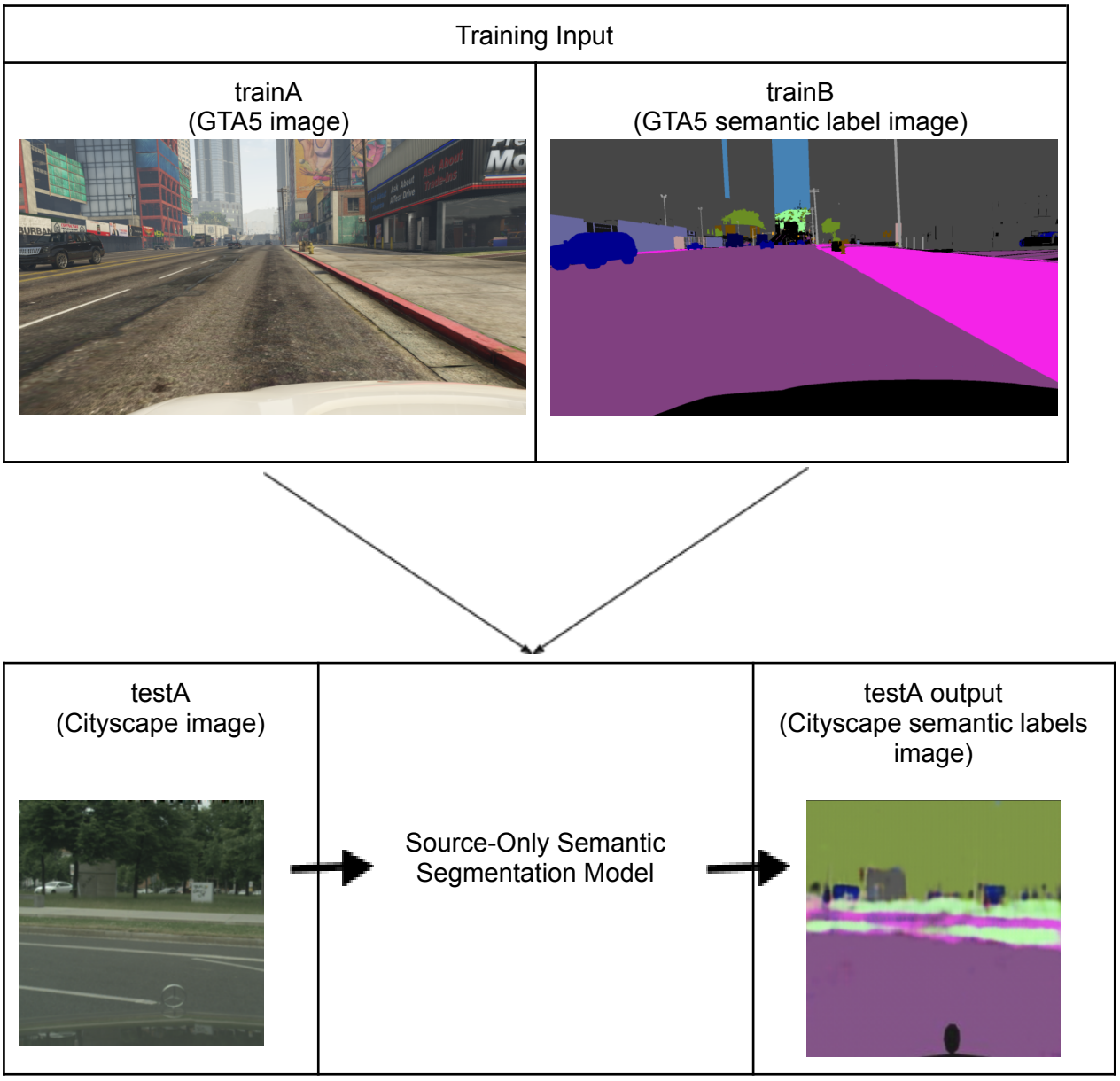


Figure 17. Implementation of Source-Only Semantic Segmentation Model

4.3 Domain Adaptive Semantic Segmentation Model

Utilizing images transformed by CycleGAN, we trained the semantic segmentation model on the source-translated data, applying identical settings to those used in the prior training of the source-only semantic segmentation model.

Firstly, our trainA dataset includes 3000 images from image2image translation model and our trainB dataset includes 3000 semantic label images from GTA5. Secondly, our testA dataset includes 100 images from Cityscapes leftimg8bit_trainvaltest.zip, which would produce 100 semantic label images. Consequently, the model was trained for 100 epochs, the batch size of 8, learning rate of 0.0002 and the optimizer is Adam with $\beta 1=0.5$ and $\beta 2=0.99$.

Figure 18 shows the implementation of the domain adaptive semantic segmentation model, which involves training on CycleGAN.

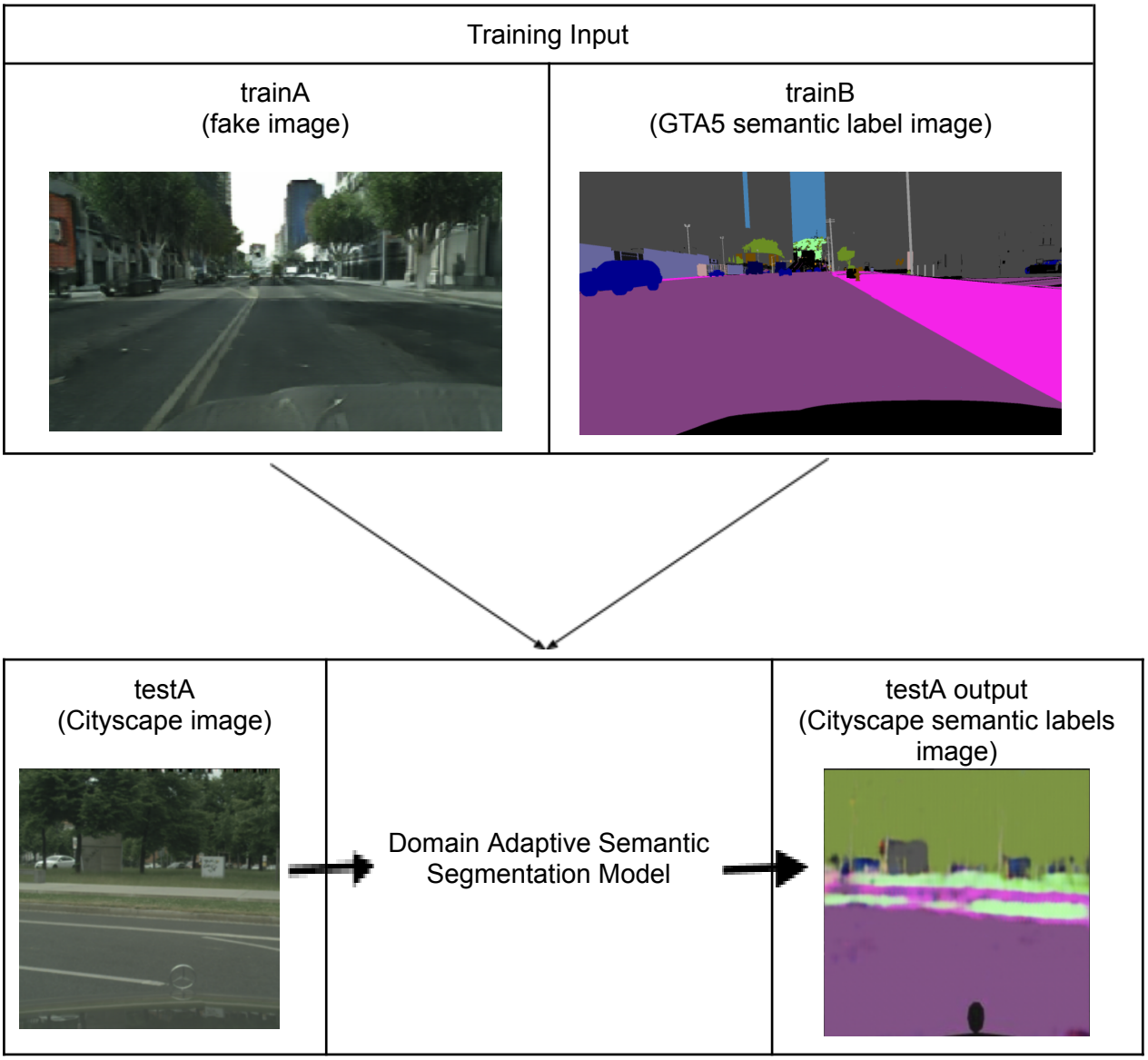


Figure 18. Implementation of Domain Adaptive Semantic Segmentation Model

4.4 UDA Results

These are the results of the two UDA models. In Figure 19, we have highlighted some positive results from the UDA models.

In the first row, we can see that the people walking on the street can be identified. On the source-only semantic segmentation model, it is clearer with less shadow.

In the second row, we can see that the cars parked by the street can be identified. On the domain adaptive semantic segmentation model, more colors and shadows can be seen.

In the third row, we can see that the pedestrian sign can be identified. On the source-only semantic segmentation model, the lines are more distinct and the pedestrian sign can be identified more easily.

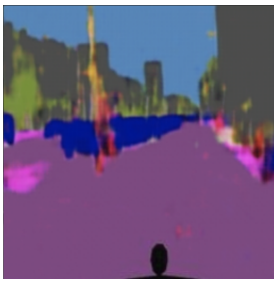
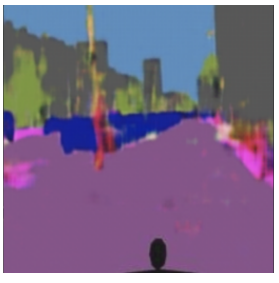
Original	Source-Only Semantic Segmentation Model	Domain Adaptive Semantic Segmentation Model
		
		
		

Figure 19. Positive Results from UDA Models

In Figure 20, we have highlighted some negative results from the UDA models.

In the first row, we can see that the street lines are all blurry and it is difficult to distinguish between the street and the sidewalk. On the domain adaptive semantic segmentation model, the lines have slightly smoother outlines.

In the second row, we can see that the street lines are not visible. Only the outline of the car can be seen. On the domain adaptive semantic segmentation model, the outline of the car looks clearer.

In the third row, we can see that the train blends in with the background. On the domain adaptive semantic segmentation model, there is at least the faint outline of the train.

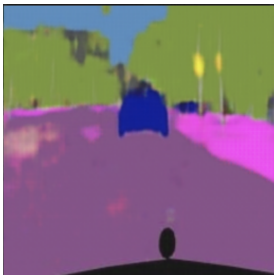
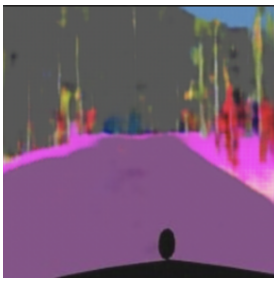
Original	Source-Only Semantic Segmentation Model	Domain Adaptive Semantic Segmentation Model
		
		
		

Figure 20. Negative Results from UDA Models

Currently, it is hard to decide which model performed better because it is a visually subjective decision. The differences between the two models are marginal. We believe that by increasing the size of the training dataset and the training epochs, the difference of the two models can be more obvious.

4.5 Constraints of UDA

1. Different Domain Class (Kuniaki, 2018): When there are no labels, there could be classes in the target domain which do not exist in the source domain. In these cases, many of today's methods of domain adaptation that are based on distribution matching often fail. They cannot match their unknown target samples with the source since they do not share common classes.
2. Domain Shift (Csurka, 2017): The domain shift between the source and target domains is one of UDA's constraints. The shift can, for example, be seen as variations of light and darkness, color of the background, or the shape of objects. This poses a challenge in building models that will work adequately in source and target domains.
3. Limited Generalizability Across Multiple Domains (Peng, 2019): Although UDA methods work quite well in converting between two domains, they are usually problematic if extended to many or more diverse domains. Due to this, they can only be applied narrowly in real world scenarios with various domain shifts.

Conclusion

In conclusion, after doing these two practices using the two techniques in performing image-to-image translation, we have realized that it is computationally expensive and time-consuming to train the data. Our team has also encountered some hiccups when using the dataset and realized the importance of preparing the dataset well before implementation.

Consequently, we have observed that CycleGAN and UDA both are good image to image techniques to be used in image-to-image translation tasks. For CycleGAN the main factors which will affect the model include data quality and parameters, such as the activation and learning rate. These factors combined will make a significant impact toward the model.

Whereas for UDA, it is able to segment images into objects without affecting its accuracy, making it robust for deployment. In addition, the training is able to be applied to different domains, which makes it more robust in comparison to CycleGAN, and thus it might be a better choice. Therefore, it is necessary to look at the task at hand to determine which technique to employ.

References

[1]Bengio, I. J.-A.-F. (2014). Generative Adversarial Networks.

[2]efrosgans.eecs.berkeley.edu. (N.A N.A, N.A). *CycleGAN\datasets.* Retrieved from efrosgans.eecs.berkeley.edu: http://efrosgans.eecs.berkeley.edu/CycleGAN/datasets/).

[3]Csurka. (2017). "Domain adaptation for visual applications: A comprehensive survey." arXiv preprint arXiv:1702.05374.

[4]Kuniaki, et al. (2018). "Open set domain adaptation by backpropagation." Proceedings of the European conference on computer vision (ECCV).

[5]Pauliina Paavilainen, S. U. (2021). Bridging the Gap Between Paired and Unpaired Medical Image Translation | SpringerLink. *arXiv*. doi:10.1007/978-3-030-88210-5_4

[6]Peng, et al. (2019). "Moment matching for multi-source domain adaptation." Proceedings of the IEEE/CVF International Conference on Computer Vision.

[6] Phillip Isola, J.-Y. Z. (2018). Image-to-Image Translation with Conditional Adversarial Networks. *arXiv*.

[7] Tsang, S.-H. (15 May, 2021). *Review — CycleGAN: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (GAN)*. Retrieved from medium.com: https://sh-tsang.medium.com/review-CycleGAN-unpaired-image-to-image-translation-using-cycle-consistent-adversarial-networks-1c2602805be2

[8]Wu, F. (2 December, 2019). *Overview of CycleGAN architecture and training*. Retrieved from Towards Data Science: https://towardsdatascience.com/overview-of-CycleGAN-architecture-and-training-afee31612a2f

[9] Judy Hoffman, Eric Tzeng (2017). CyCADA: Cycle-Consistent Adversarial Domain Adaptation. *arvix*.

[10] Efros, J.-Y. Z. (2020). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *arvix*.

[11] junyanz, (N.A) .When training the GAN model, how many epochs do we need to train #1127. (19 Aug, 2020). Retrieved from github.com: https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/112

[12] CycleGAN Code: Brownlee, J. (1 September , 2020). How to Develop a CycleGAN for Image-to-Image Translation with Keras. Retrieved from machinelearningmastery.com: https://machinelearningmastery.com/CycleGAN-tutorial-with-keras/

[13] UDA code: Anurag Paul GTA-to-Cityscapes-Domain-Adaptation : https://github.com/anurag1paul/GTA-to-Cityscapes-Domain-Adaptation/tree/master

[14] Richter, Stephan R., et al. "Playing for data: Ground truth from computer games." European conference on computer vision. Springer, Cham, 2016.

[15] Cordts, Marius, et al. "The cityscapes dataset for semantic urban scene understanding." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

Appendix

CycleGAN Script:

```
## libararies
from os import listdir
from numpy import asarray
from numpy import vstack
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from numpy import savez_compressed
from matplotlib import pyplot
import matplotlib.pyplot as pyplot
from random import random
from numpy import load
from numpy import zeros
from numpy import ones
from numpy import asarray
from numpy.random import randint
from keras.optimizers import Adam
from keras.initializers import RandomNormal
from keras.models import Model
#from keras.models import Input
from tensorflow.keras.layers import Input
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from keras.layers import Activation
from keras.layers import Concatenate
from matplotlib import pyplot
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, LeakyReLU, Flatten, Dense
from tensorflow.keras.optimizers import Adam
```

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import img_to_array, load_img
from tensorflow.keras.utils import image_dataset_from_directory
import os
import numpy as np
path ='/content/drive/MyDrive/ComputerVisionAssignment4/GTA/Output_Dir/'

def load_images(path, size=(256, 256), limit=3000):
    dataset = image_dataset_from_directory(
        directory=path,
        label_mode=None,    # No labels as we're only loading images
        image_size=size,
        batch_size=None,    # Load images one by one
        shuffle=False    # Maintain the order
    )

    data_list = []
    count = 0
    for img_tensor in dataset:
        if count >= limit:   # Stop after loading 1000 images
            break
        img_array = img_tensor.numpy()    # Convert tensor to numpy array
        data_list.append(img_array)
        count += 1

    return np.asarray(data_list)

# Example usage:
# images = load_images('/path/to/dataset')
```

```
import numpy as np

# Dataset A
dataTrainA1 = load_images(path + 'trainA/')
dataTestAB = load_images(path + 'testA/')
dataA = np.concatenate((dataTrainA1, dataTestAB), axis=0)
print('Loaded dataA: ', dataA.shape)

# Dataset B
dataTrainB1 = load_images(path + 'trainB/')
dataTestB2 = load_images(path + 'testB/')
dataB = np.concatenate((dataTrainB1, dataTestB2), axis=0)
print('Loaded dataB: ', dataB.shape)

filename = 'horse2zebra_256.npz'## Save outside the \drive
np.savez_compressed(filename, dataA, dataB)
print('Saved dataset: ', filename)
```

```
# Plot the image with random sampling
#Set the number of samples
n_samples = 4

# Set the overall figure size
pyplot.figure(figsize=(12, 6))  # Width, Height in inches

# Plot images from Dataset A (Source Images)
for i in range(n_samples):
    pyplot.subplot(2, n_samples, 1 + i)
    pyplot.axis('off')
    pyplot.title(f'Data A - Image {i+1}')
    pyplot.imshow(dataA[i].astype('uint8'))

# Plot images from Dataset B (Target Images)
for i in range(n_samples):
    pyplot.subplot(2, n_samples, 1 + n_samples + i)
    pyplot.axis('off')
    pyplot.title(f'Data B - Image {i+1}')
    pyplot.imshow(dataB[i].astype('uint8'))

# Show the plot
pyplot.show()
```

Cycle GAN Implementation

```
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
```

a. Discriminator

```
[10] #  Discriminator model
     def define_discriminator(image_shape):
             # weight initialization
         init = RandomNormal(stddev=0.02)
         # source image input
         in_image = Input(shape=image_shape)
         d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(in_image)
         d = LeakyReLU(alpha=0.2)(d)

         d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
         d = InstanceNormalization(axis=-1)(d)
         d = LeakyReLU(alpha=0.2)(d)

         d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
         d = InstanceNormalization(axis=-1)(d)
         d = LeakyReLU(alpha=0.2)(d)

         d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
         d = InstanceNormalization(axis=-1)(d)
         d = LeakyReLU(alpha=0.2)(d)

         # 2nd last layer (Output)
         d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
         d = InstanceNormalization(axis=-1)(d)
         d= LeakyReLU(alpha=0.2)(d)
         # Patch Output
         patch_out = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
         # define model
         model = Model(in_image, patch_out)
         # compilation of model
         model.compile(loss='mse', optimizer=Adam(lr=0.0002, beta_1=0.5), loss_weights=[0.5])
         return model
```

b.Generator

```
[11] # Resnet block
     def resnet_block(n_filters, input_layer):
         # weight initialization
         init = RandomNormal(stddev=0.02)
         # first layer convolutional layer
         g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(input_layer)
         g = InstanceNormalization(axis=-1)(g)
         g = Activation('relu')(g)
         # second convolutional layer
         g = Conv2D(n_filters, (3,3), padding='same', kernel_initializer=init)(g)
         g = InstanceNormalization(axis=-1)(g)
         # concatenate merge channel-wise with input layer
         g = Concatenate()([g, input_layer])
         return g
```

```
#  Generator model
def define_generator(image_shape, n_resnet=9):
  init = RandomNormal(stddev=0.02)
  in_image = Input(shape=image_shape)

  g = Conv2D(64, (7,7), padding='same', kernel_initializer=init)(in_image)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)

  g = Conv2D(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)

  g = Conv2D(256, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)

  for _ in range(n_resnet):
    g = resnet_block(256, g)

  g = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)

  g = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)
  g = Activation('relu')(g)

  g = Conv2D(3, (7,7), padding='same', kernel_initializer=init)(g)
  g = InstanceNormalization(axis=-1)(g)

  out_image = Activation('sigmoid')(g)
  # define model
  model = Model(in_image, out_image)
  return model
```

```
[13] # Composite model fo updating generators : adversarial and cycle loss
     def define_composite_model(g_model_1, d_model, g_model_2, image_shape):
         # ensure the model we're updating is trainable
         g_model_1.trainable = True
         # mark discriminator as not trainable
         d_model.trainable = False
         # mark other generator model as not trainable
         g_model_2.trainable = False
         # discriminator element
         input_gen = Input(shape=image_shape)
         gen1_out = g_model_1(input_gen)
         output_d = d_model(gen1_out)
         # identity element
         input_id = Input(shape=image_shape)
         output_id = g_model_1(input_id)
         # forward cycle
         output_f = g_model_2(gen1_out)
         # backward cycle
         gen2_out = g_model_2(input_id)
         output_b = g_model_1(gen2_out)
         # define model graph
         model = Model([input_gen, input_id], [output_d, output_id, output_f, output_b])
         # define optimization algorithm configuration
         opt = Adam(lr=0.0002, beta_1=0.5)
         # compile model with weighting of least squares loss and L1 loss
         model.compile(loss=['mse', 'mae', 'mae', 'mae'], loss_weights=[1, 5, 10, 10], optimizer=opt)
         return model
```

```
[15] # load and prepare training images
     def load_real_samples(filename):
         # load the dataset
         data = load(filename)
         # unpack arrays
         X1, X2 = data['arr_0'], data['arr_1']
         # scale from [0,255] to [-1,1]
         X1 = (X1 - 127.5) / 127.5
         X2 = (X2 - 127.5) / 127.5
         return [X1, X2]

     # select a batch of random samples, returns images and target
     def generate_real_samples(dataset, n_samples, patch_shape):
         # choose random instances
         ix = randint(0, dataset.shape[0], n_samples)
         # retrieve selected images
         X = dataset[ix]
         # generate 'real' class labels (1)
         y = ones((n_samples, patch_shape, patch_shape, 1))
         return X, y

     # generate a batch of images, returns images and targets
     def generate_fake_samples(g_model, dataset, patch_shape):
         # generate fake instance
         X = g_model.predict(dataset)
         # create 'fake' class labels (0)
         y = zeros((len(X), patch_shape, patch_shape, 1))
         return X, y
```

```
[16] # save the generator models to file
     def save_models(step, g_model_AtoB, g_model_BtoA):
         # save the first generator model
         filename1 = 'g_model_AtoB_%06d.h5' % (step+1)
         g_model_AtoB.save(filename1)
         # save the second generator model
         filename2 = 'g_model_BtoA_%06d.h5' % (step+1)
         g_model_BtoA.save(filename2)
         print('>Saved: %s and %s' % (filename1, filename2))
```

```
# generate samples and save as a plot and save the model
def summarize_performance(step, g_model, trainX, name, n_samples=5):
    # select a sample of input images
    X_in, _ = generate_real_samples(trainX, n_samples, 0)
    # generate translated images
    X_out, _ = generate_fake_samples(g_model, X_in, 0)
    # scale all pixels from [-1,1] to [0,1]
    X_in = (X_in + 1) / 2.0
    X_out = (X_out + 1) / 2.0
    # plot real images
    for i in range(n_samples):
        pyplot.subplot(2, n_samples, 1 + i)
        pyplot.axis('off')
        pyplot.imshow(X_in[i])
    # plot translated image
    for i in range(n_samples):
        pyplot.subplot(2, n_samples, 1 + n_samples + i)
        pyplot.axis('off')
        pyplot.imshow(X_out[i])
    # save plot to file
    filename1 = '%s_generated_plot_%06d.png' % (name, (step+1))
    pyplot.savefig(filename1)
    pyplot.close()
```

```
# update image pool for fake images
def update_image_pool(pool, images, max_size=50):
    selected = list()
    for image in images:
        if len(pool) < max_size:
            # stock the pool
            pool.append(image)
            selected.append(image)
        elif random() < 0.5:
            # use image, but don't add it to the pool
            selected.append(image)
        else:
            # replace an existing image and use replaced image
            ix = randint(0, len(pool))
            selected.append(pool[ix])
            pool[ix] = image
    return asarray(selected)
```

4 . Train

```
[19] # train cyclegan models
     def train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset):
         # Properties of the training

         n_epochs, n_batch, = 70, 1
         # determine the output square shape of the discriminator
         n_patch = d_model_A.output_shape[1]
         # unpack dataset
         trainA, trainB = dataset
         # prepare image pool for fakes
         poolA, poolB = list(), list()
         # calculate the number of batches per training epoch
         bat_per_epo = int(len(trainA) / n_batch)
         # calculate the number of training iterations
         n_steps = bat_per_epo * n_epochs
         # manually enumerate epochs
         for i in range(n_steps):
             # select a batch of real samples
             X_realA, y_realA = generate_real_samples(trainA, n_batch, n_patch)
             X_realB, y_realB = generate_real_samples(trainB, n_batch, n_patch)
             # generate a batch of fake samples
             X_fakeA, y_fakeA = generate_fake_samples(g_model_BtoA, X_realB, n_patch)
             X_fakeB, y_fakeB = generate_fake_samples(g_model_AtoB, X_realA, n_patch)
             # update fakes from pool
             X_fakeA = update_image_pool(poolA, X_fakeA)
             X_fakeB = update_image_pool(poolB, X_fakeB)
             # update generator B->A via adversarial and cycle loss
             g_loss2, _, _, _, _  = c_model_BtoA.train_on_batch([X_realB, X_realA], [y_realA, X_realA, X_realB, X_realA])
             # update discriminator for A -> [real/fake]
             dA_loss1 = d_model_A.train_on_batch(X_realA, y_realA)
             dA_loss2 = d_model_A.train_on_batch(X_fakeA, y_fakeA)
             # update generator A->B via adversarial and cycle loss
             g_loss1, _, _, _, _ = c_model_AtoB.train_on_batch([X_realA, X_realB], [y_realB, X_realB, X_realA, X_realB])
             # update discriminator for B -> [real/fake]
             dB_loss1 = d_model_B.train_on_batch(X_realB, y_realB)
             dB_loss2 = d_model_B.train_on_batch(X_fakeB, y_fakeB)
             # summarize performance
             print('>%d, dA[%.3f,%.3f] dB[%.3f,%.3f] g[%.3f,%.3f]' % (i+1, dA_loss1,dA_loss2, dB_loss1,dB_loss2, g_loss1,g_loss2))
             # evaluate the model performance every so often
             if (i+1) % (bat_per_epo * 4 )== 0:
             # plot B->A translation
               summarize_performance(i, g_model_AtoB, trainA, 'AtoB')
             # plot B->A translation
               summarize_performance(i, g_model_BtoA, trainB, 'BtoA')
             if (i+1) % (bat_per_epo * 4) == 0:
             # save the models
               save_models(i, g_model_AtoB, g_model_BtoA)
```

```
# load image data
dataset = load_real_samples('horse2zebra_256.npz')
print('Loaded', dataset[0].shape, dataset[1].shape)
# define input shape based on the loaded dataset
image_shape = dataset[0].shape[1:]
# generator: A -> B
g_model_AtoB = define_generator(image_shape)
# generator: B -> A
g_model_BtoA = define_generator(image_shape)
# discriminator: A -> [real/fake]
d_model_A = define_discriminator(image_shape)
# discriminator: B -> [real/fake]
d_model_B = define_discriminator(image_shape)
# composite: A -> B -> [real/fake, A]
c_model_AtoB = define_composite_model(g_model_AtoB, d_model_B, g_model_BtoA, image_shape)
# composite: B -> A -> [real/fake, B]
c_model_BtoA = define_composite_model(g_model_BtoA, d_model_A, g_model_AtoB, image_shape)
# train models
train(d_model_A, d_model_B, g_model_AtoB, g_model_BtoA, c_model_AtoB, c_model_BtoA, dataset)
```

```
## Addditional Lib
# example of using saved cyclegan models for image translation
from numpy import load
from numpy import expand_dims
from keras.models import load_model
from keras_contrib.layers.normalization.instancenormalization import InstanceNormalization
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from matplotlib import pyplot
```

```
# load and prepare training images
def load_real_samples(filename):
  # load the dataset
  data = load(filename)
  # unpack arrays
  X1, X2 = data['arr_0'], data['arr_1']
  # scale from [0,255] to [-1,1]
  X1 = (X1 - 127.5) / 127.5
  X2 = (X2 - 127.5) / 127.5
  return [X1, X2]

# select a random sample of images from the dataset
def select_sample(dataset, n_samples):
  # choose random instances
  ix = randint(0, dataset.shape[0], n_samples)
  # retrieve selected images
  X = dataset[ix]
  return X

# plot the image, the translation, and the reconstruction
def show_plot(imagesX, imagesY1, imagesY2):
  images = vstack((imagesX, imagesY1, imagesY2))
  titles = ['Real', 'Generated', 'Reconstructed']
  # scale from [-1,1] to [0,1]
  images = (images + 1) / 2.0
  # plot images row by row
  for i in range(len(images)):
    # define subplot
    pyplot.subplot(1, len(images), 1 + i)
    # turn off axis
    pyplot.axis('off')
    # plot raw pixel data
    pyplot.imshow(images[i])
    # title
    pyplot.title(titles[i])
  pyplot.show()
```

UDA Script:

train:

```
from math import ceil

import torch
from torch import Tensor
from torch.autograd import Variable

from cycle_gan import CycleGAN
from data_loader import DataLoader
from logger import logger
from utils import ensure_dir, get_opts

project_root = "./"
data_root = "./gta/images/"
models_prefix = project_root + "saved_models/"
images_prefix = project_root + "saved_images/"

device = 'cuda:0' if torch.cuda.is_available() else 'cpu'


def train_cycle_gan(data_root, semi_supervised=False):
    opt = get_opts()

    ensure_dir(models_prefix)
    ensure_dir(images_prefix)

    cycle_gan = CycleGAN(device, models_prefix, opt["lr"], opt["b1"],
                         train=True, semi_supervised=semi_supervised)
    data = DataLoader(data_root=data_root,
                      image_size=(opt['img_height'], opt['img_width']),
                      batch_size=opt['batch_size'])

    total_images = len(data.names)
    print("Total Training Images", total_images)

    total_batches = int(ceil(total_images / opt['batch_size']))

    for epoch in range(cycle_gan.epoch_tracker.epoch, opt['n_epochs']):
        for iteration in range(total_batches):

            if (epoch == cycle_gan.epoch_tracker.epoch and
                        iteration < cycle_gan.epoch_tracker.iter):
                continue

            y, x = next(data.data_generator(iteration))

            real_A = Variable(x.type(Tensor))
            real_B = Variable(y.type(Tensor))

            cycle_gan.set_input(real_A, real_B)
            cycle_gan.train()

            message = (
                "\r[Epoch {}/{}] [Batch {}/{}] [DA:{}, DB:{}] [GA:{}, GB:{}, cycleA:{}, cycleB:{}, G:{}]"
                    .format(epoch, opt["n_epochs"], iteration, total_batches,
                            cycle_gan.loss_disA.item(),
                            cycle_gan.loss_disB.item(),
                            cycle_gan.loss_genA.item(),
                            cycle_gan.loss_genB.item(),
                            cycle_gan.loss_cycle_A.item(),
                            cycle_gan.loss_cycle_B.item(),
                            cycle_gan.loss_G))
            print(message)
            logger.info(message)

            if iteration % opt['sample_interval'] == 0:
                cycle_gan.save_progress(images_prefix, epoch, iteration)
        cycle_gan.save_progress(images_prefix, epoch, total_batches, save_epoch=True)


if __name__ == "__main__":
    train_cycle_gan(data_root)
```

test:

```
import torch
from torch import Tensor
from torch.autograd import Variable

from cycle_gan import CycleGAN
from data_loader import DataLoader
from logger import logger
from utils import ensure_dir, get_opts

project_root = "./"
data_root = "./gta/images/"
models_prefix = project_root + "saved_models/test_"
images_prefix = project_root + "saved_images/"

device = 'cuda:0' if torch.cuda.is_available() else 'cpu'


def test_cycle_gan(semi_supervised=True):
    opt = get_opts()

    ensure_dir(models_prefix)
    ensure_dir(images_prefix)

    cycle_gan = CycleGAN(device, models_prefix, opt["lr"], opt["b1"], train=False,
                         semi_supervised=semi_supervised)
    data = DataLoader(data_root=data_root,
                      image_size=(opt['img_height'], opt['img_width']),
                      batch_size=1, train=False)

    total_images = len(data.names)
    print("Total Testing Images", total_images)

    loss_A = 0.0
    loss_B = 0.0
    name_loss_A = []
    name_loss_B = []


          for i in range(total_images):
              print(i, "/", total_images)
              x, y = next(data.data_generator(i))
              name = data.names[i]

              real_A = Variable(x.type(Tensor))
              real_B = Variable(y.type(Tensor))

              cycle_gan.set_input(real_A, real_B)
              cycle_gan.test()
              cycle_gan.save_image(images_prefix, name)
              loss_A += cycle_gan.test_A
              loss_B += cycle_gan.test_B
              name_loss_A.append((cycle_gan.test_A, name))
              name_loss_B.append((cycle_gan.test_B, name))

          info = "Average Loss A:{} B :{}".format(loss_A/(1.0 * total_images), loss_B/(1.0 * total_images))
          print(info)
          logger.info(info)
          name_loss_A = sorted(name_loss_A)
          name_loss_B = sorted(name_loss_B)
          print("top 10 images")
          print(name_loss_A[:10])
          print(name_loss_B[:10])


      if __name__ == "__main__":
          test_cycle_gan()
```