DLCV-HW3

Carlos Marzal Romon, A08922106

Carlos Marzal

Contenido

DLCV-HW3

1. GAN

For this problem we created a GAN that generated fake 64x64x3 size images of faces.

1.1. Architecture and Implementation of GAN model

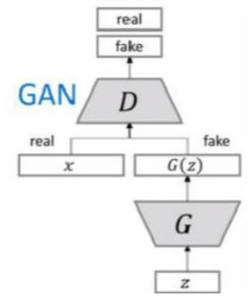

Figure 1 GAN Architecture

To do this problem I followed the architecture suggested in the slides, having a generator from a noise of dimension 100 that outputted images of size 3x64x64 and a discriminator that got fake and real images to train and output a value from 0 to 1, being 0 a fake image and 1 a real one (the discriminator would show a value in between these two numbers representing how likely it is for the image to be real/fake). The discriminator was trained knowing if the image was real of fake, but when training the generator said the discriminator that the images where real, this way we could try to minimize the generator error as if the images where real for the discriminator.

Now, commenting some individual architectures:

Generator:

For the generator I have used the model suggested by PyTorch for GANs[1] with some slight modification in the numeric values to fit our image sizes. The architecture in particular is the one used in DCGANs (as suggested in the slides) and it uses ConvTranspose2d, Batchnorm2d and ReLU in each of the steps, and a tanh at the end of the model. The input of de Generator is a [batch_size, 100] shape tensor (with random noises of size 100) and the output is a [batch_size, 3, 64, 64] shape tensor (containing the output images in 3 colors).

Discriminator:

For the discriminator we also used the model suggested in DCGANs, again modifying some parameters to work with our dimensions. This particular architecture of the model includes 5 sections, each containing a Conv2d, BatchNorm2d and LeakyReLU to resize the tensor, except for the last section which only has a Conv2d to a 1-dimension vector and a Sigmoid. The input to the discriminator is a [batch_size, 3, 64, 64] shape tensor (containing the input images in 3 colors) and the output is a [batch_size, 1] shape tensor which contains the value of how likely it has calculated the image to be real(1) or fake(0).

Overall architecture:

To try and maximize our results we followed most of the tips and tricks provided in GitHub by Ganhacks[2]. Some basic things in the architecture include Adam optimizers (since it has been shown they are the best for GANs generally[3]), random seeding, initial weights and using BCELoss as a criterion.

The reason why we use BCELoss as a criterion is because, as suggested by the slides of the homework, we should try to minimize the loss: $L_{GAN}(G,D) = E\left[\log\left(1 - D\big(G(x)\big)\right)\right] + E[logD(y)]$

BCE is perfect to obtain this formula because it's loss is defines as:

ℓ(x,y)=L={l1,…,lN}⊤,ln=−[yn⋅logxn+(1−yn)⋅log(1−xn)]

Which, when we pass a fake image (y=0) we obtain the output log(1−xn) and when we pass a real image(y=1) we have log(xn), therefore, if we add these two errors, we end with the desired GAN loss formula we want to minimize.

Also, in the code I'm showing 10 times per epoch the average errors from the discriminator and for the generator to know if the code is working correctly from the beginning and change parameters around if it's not. The general errors from the discriminator and generator are also stored in a parameter to then be showed once the training is over.

1.2. Images generated by GAN generator model

Once the model has been trained, we can apply to it 32 random 100 dimension noises to obtain 32 random faces that can be plot to observe the results. The file save_image_GAN.py in my GAN folder will do this for all the stored models and save the seed created and the images in a file to be checked and select the best output.

The chosen example that will be stored in the fig1_2.png file is the following, coming from the 44s epoch of the GAN training:

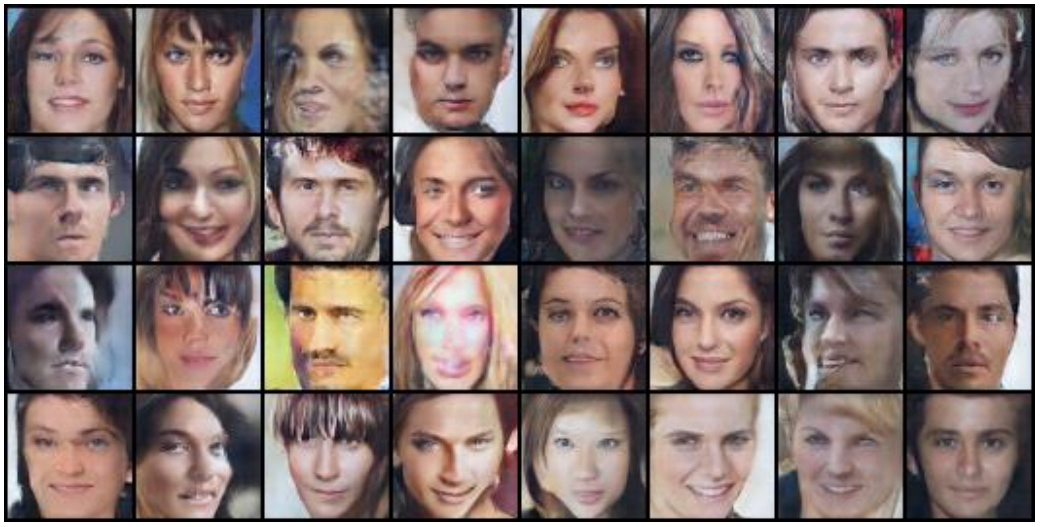

Figure 2 GAN generated images

As we can see, there are some faces that look very realistic and others that, due to the noise being not the best (maybe not as trained as other options) we have some faces that are clearly not real. Over all the results are more than decent since we have faces that could even make a human think it's a normal face (like the ones in the corners of the image).

1.3. Observations from GAN

Naturally our GAN didn't work correctly the first time we run it and a lot of modifications had to be done over time to end with a decent result. These where some of the observations from the GAN after all these changes and understanding the system:

- It's really complicated to know when the GAN is training the generator correctly since you don't really have an efficiency value. This is because, even if the generator error went up, it could be due to de discriminator becoming better and therefore the generator had a tougher time trying to trick it, even if it was making better faces. This can be seen in the following graphs obtained after the GAN training, where we can see how the error of the Generator and Discriminator changed over time:

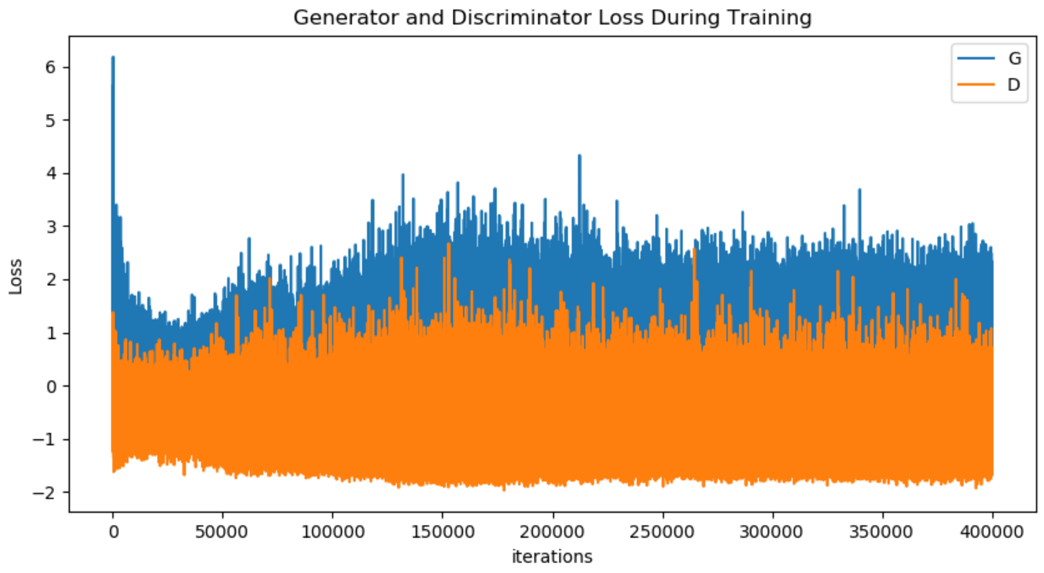

Figure 3 GAN generator and discriminator losses

As we can see, the smaller generator losses where after around 20k iterations (second epoch) but the images created at the last epochs where better than the on created after 2, seen here:

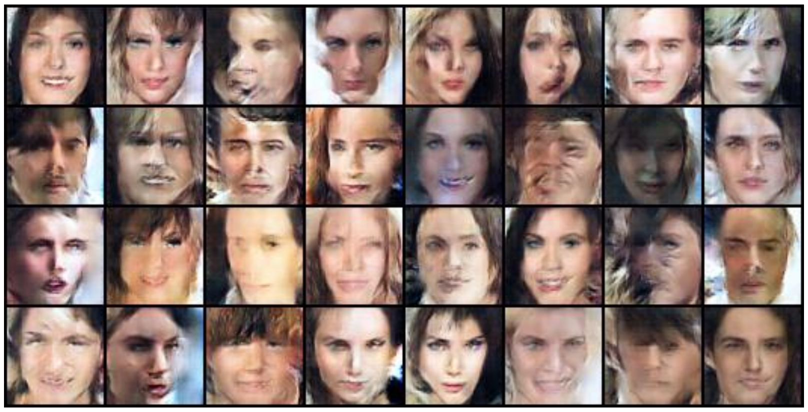

Figure 4 GAN discriminator images after 2 epochs

This is because the discriminator was not as correctly trained as at the end so the generator was tricking it with much worse images than the ones it needed after 30/40 epochs. This made knowing the "best generator model" impossible based on only numeric values, so that's why we needed to find images for all models and find the best one

- Overall, the GAN seemed to be trained better with smaller batch sizes because it would obtain more backward information per image. If the batch size was too big (>100) the images from the generator would come out like in small patches (worst resolution). The optimal batch size for my GPU was between 8 and 12, depending on the learning rate used.

- Images started to seem very similar after around epoch 35, from where only some particular faces changed a lot, while other just shifted a bit. This shows that the GAN converged at around 35 epochs (30k iterations from the graph of Figure 3).

- System worked better with a small weight decay in the Adam optimizer for the discriminator (around 0.005) because it would let the generator “catch up” with it a bit more and generate better images slowly, which gives better results.

- Training with the same code but different seed and order of images can change completely the way a model is trained and give you completely different results. For example, the following image shows the faces created by the same noise as the ones from Figure 2 and also the model from epoch 44 from a different training session with no changes to the code:

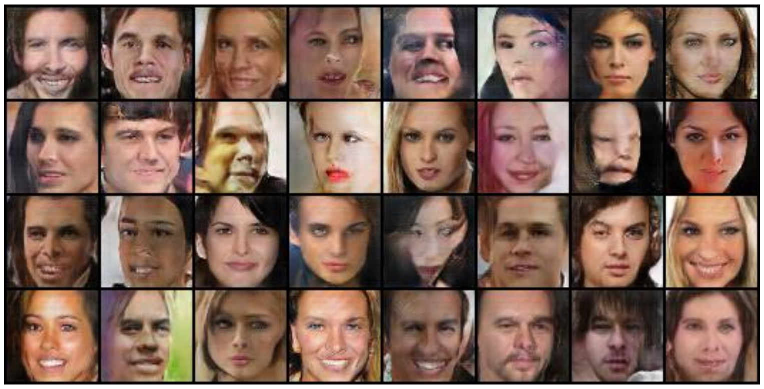

Figure 5 GAN epoch 44 other training

As we can see, the images don’t look at all like the ones previously shown, which comes to show how unstable and unpredictable the training of the GAN can be.

2. ACGAN

This problem is similar to the previous one, but instead of only training the discriminator to know if faces are real or fake, now you also have to train it to know if the faces are smiling or not, and the discriminator should receive an parameter if the face is smiling or not and generate the appropriate face.

2.1. Architecture and Implementation of ACGAN model

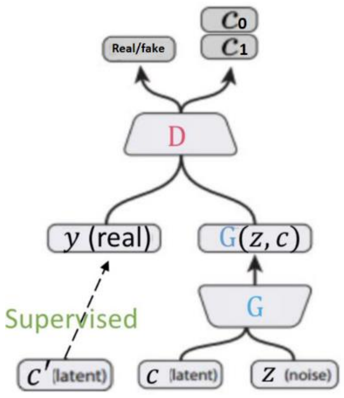

Figure 6 ACGAN model

There were different options to implement this model from resources online, but for the models of the discriminator and generator I ended up creating one by myself based on the one from the GAN, but changing things based on other observed models[4]. Overall, the model is the same as the one suggested in the slides, with a C latent space (smiling digit from the smiling column in the corresponding csv file in this case) for the real images and a random C' latent space for the fake images. Another big change here is also that the discriminator has two outputs, therefore it will also contain two errors, which will be calculated differently.

Some more specific explanations:

Generator:

The ACGAN generator basically is the same as the one from the GAN, but has two small differences in the input of itself. The input now also contains two new columns (the first two are the class columns, being column 0 = 1 if the face is not supposed to be smiling and column 1 = 1 if it is smiling, if one of the columns is 1 the other will be 0) and the rest of the columns are the noise like before.

The other slight change to the generator is that the size of the input noise + class is now of dimension 10 instead of 100. The explanation to this will be given in section 2.3.

Discriminator:

Similar to the generator, the discriminator follows the same basic architecture than the generator with some small changes to fit the extra class classification from ACGAN. The normal real/fake output is the same as before, but now we have a new output corresponding to the classes. This output come from removing the last layer of the normal output and changing it to a Conv2D to dimension 2 and a SoftMax of this output. This new output is two dimensional; the first column is how likely it is for the face to not be smiling and the second one is the probability of the face being smiling (due to SoftMax they add up to 1).

Overall Architecture:

We have two different ways of calculating the two errors, the first one, which is if the image is fake or real is the same as in the GAN (using BCE Loss), but for the loss of the smiling parameter, we will be using Cross Entropy Loss because it's a multiclass array that we want to analyze, so we cant use BCE.

The total error from which we do the backward function to train the discriminator is the sum of the losses from the real and fake images and smiling or not (sum of 4 losses). As for the generation, the backward function will be done over the losses of the two classes for the fake images.

2.2. Images generated by ACGAN generator model

For this section we generated 10 random images with a 10-dimensional noise and the smiling parameter set to 0 (not smiling) and then, using the same noise generate another 10 images but this time smiling, to see how the ACGAN distinguished between smiling and non-smiling faces:

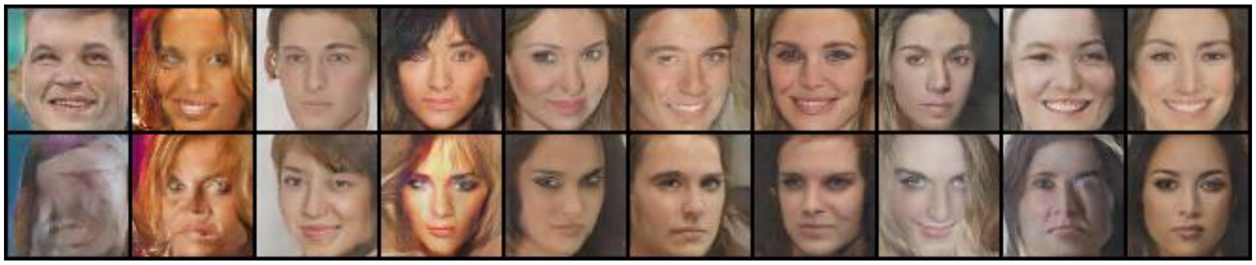

Figure 7 ACGAN images

It can be observed how, generally speaking, the images on the top are smiling and the ones on the bottom are not. There are two main exceptions in this case in images 3 and 8, where the bottom one seams to be the one smiling, and there will obviously be some small errors in this, but overall the results are more than pleasing, specially for the smiling faces, which look very realistic compared to the non-smiling ones.

Also, even though we are using the same noise for both images, sometimes the top and bottom faces are not exactly the same because of the influence of the smiling columns (for example in face 4, the faces are really similar but one of the girls has brown hair while the other is blonde); but generally speaking, the faces from the top and bottom rows do seem to be the same person.

2.3. Observations from ACGAN

When doing the ACGAN we faced some of the problems and observations mentioned in section 1.3. But now we'll focus on the ones specific to the ACGAN. Some general observations from the ACGAN from what we've learned after having to change values around to get pleasant results:

- Overall the biggest decision making that had to be done when implementing the ACGAN using my model was the dimension of the noise that went into the generation to create faces. When the dimension was set to 100, like in the previous exercise, the two extra columns on the input of the generator didn't have enough impact on the generated faces for it to recognize that column was the smiling one. So, when creating the images, it would generate two similar ones but it wouldn't really distinguish when it was smiling or not.

On the other hand, if you made the dimension of the noise too small (<10), the smiling columns would have such a big influence that the two faces wouldn't look similar at all (like the example in the previous section where a girl was blonde and the other had brown hair).

- Another change that had to be done in this case compared to the one of the GAN was to train the generator more than the discriminator, because it was falling back even after applying dropout to the discriminator. The point that gave me better results where with a batch size of 20 and a training multiplier for the generator of 1.5 (it would train with 30 images instead of the 20 from the discriminator).

3. DANN

3.1. Lower Bound

The lower bound of this example is obtained by training the model using only the source train data and find the maximum accuracy we get on the target test data. For this you don't need the second part of the model, since we only need the value of the classes, not the domain the numbers come from, so it's essentially just doing a normal train of the model on recognizing different numbers.

Obtained Lower Bounds:

(1) MNIST-M -> SVHN: 0.3835 (38.35%)
(2) SVHN -> MNIST-M: 0.4171 (41.71%)

These numbers represent the minimum values we should obtain in the DANN training when using the whole architecture. They where obtained after 3 and 5 epochs respectively, with a batch size of 100. After getting these values, the accuracy started to decline slowly, going as low as a 20%, probably because of how specifically it was training the images from the source.

3.2. Domain Adaptation

The domain adaptation will be obtained by training the source domain images on both the class and domain and train the target training domain on only the domain (because it's supposed that we don't have access to the classes of the target numbers). A better explanation of the methodology can be seen in section 3.5.

Obtained Domain Adaptation accuracy:

(1) MNIST-M -> SVHN: <u>0.5013 (50.13%)</u>
(2) SVHN -> MNIST-M: <u>0.4695 (46.95%)</u>

Both these numbers clearly surpass the baseline scores for the DANN that where set in the homework slides (40%), and they are both better than the lower bound. This shows that the DANN is working correctly and obtaining desired values. To obtain these I needed 3 and 7 epochs respectively, with a batch size of 10 (because it gave better results than with 100). After these epochs the value fluctuated close to the maximum but never improved it (maybe after more epochs it would surpass it, but there is no need to improve it).

3.3. Upper Bound

The upper bound is obtained by training the model only on the classes of the target and ignoring the source. This way we know what the best possible percentage is that we can get on the target; although obviously it's impossible for the DANN to reach these percentages since it doesn't have the information on the labels.

Obtained upper bound:

(1) MNIST-M -> SVHN (only train and get percentage on SVHN): 0.9121 (91.21%)
(2) SVHN -> MNIST-M (only train and get percentage on MNIST-M): 0.9712 (97.12%)

As we can see these upper bound values are a very big accuracy which is naturally impossible to reach without the appropriate labels.

3.4. Latent Space Visualization

We can visualize the latent space created by the features of the models. To do this we changed the model so it returned the features as well as the other two outputs and then use these features from the best trained model to generate these graphs. To reduce the dimension of the features from 800 to 2 we are using t-SNE which is a function that reduces space based on the distribution and probability of the given dataset. We are getting the graph and showing firstly the different numbers as different colors and then the different domains as different colors.

The overall expected result is that the latent space is domain invariant (there shouldn't be much difference in the domain location) but the different numbers should be divided correctly intro sections. The obtained results are the following:

(1) MNIST-M -> SVHN:

a) Different digit classes 0-9:

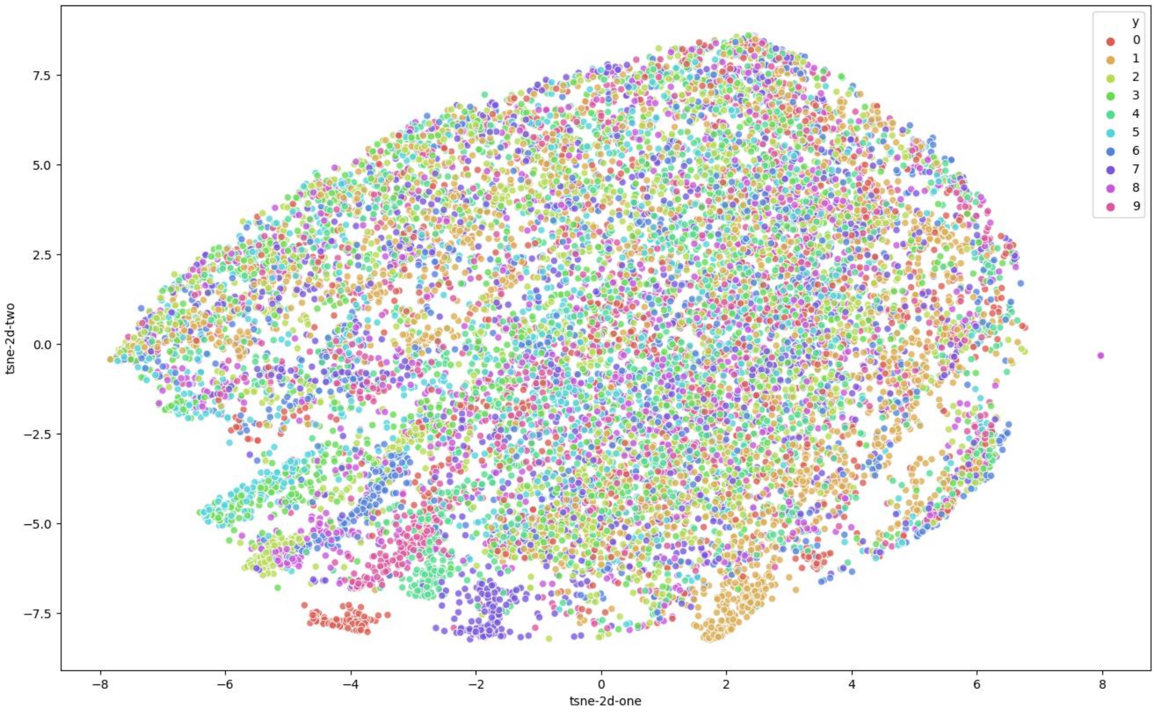

Figure 8 DANN MNISTM-SVHN digit latent space

b) Different domains (0=MNIST-M, 1=SVHN):

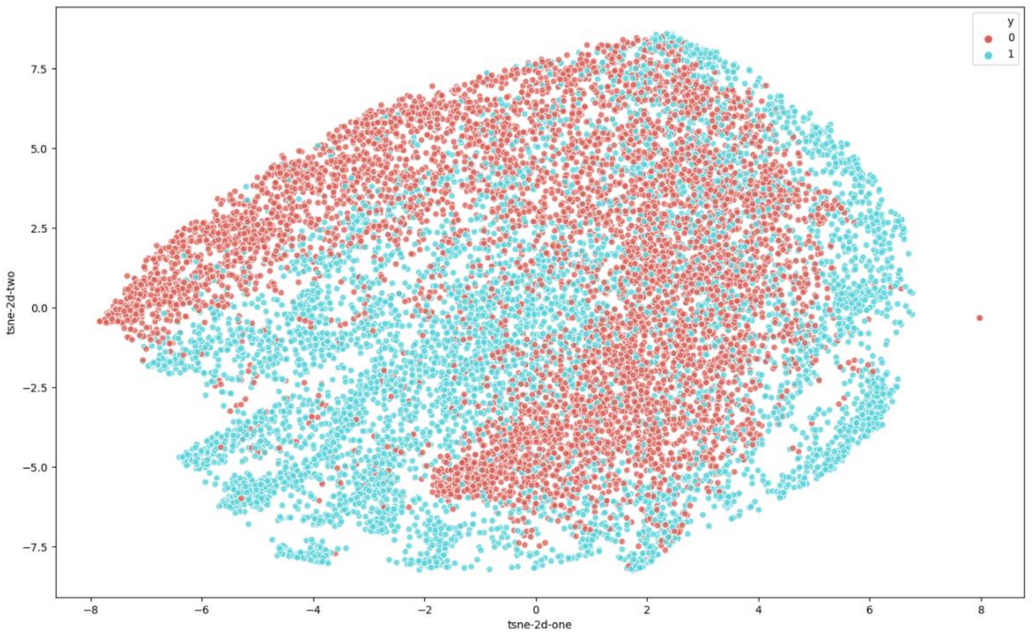

Figure 9 DANN MNISTM-SVHN domain latent space

(2) SVHN -> MNIST-M

a) Different digit classes 0-9:

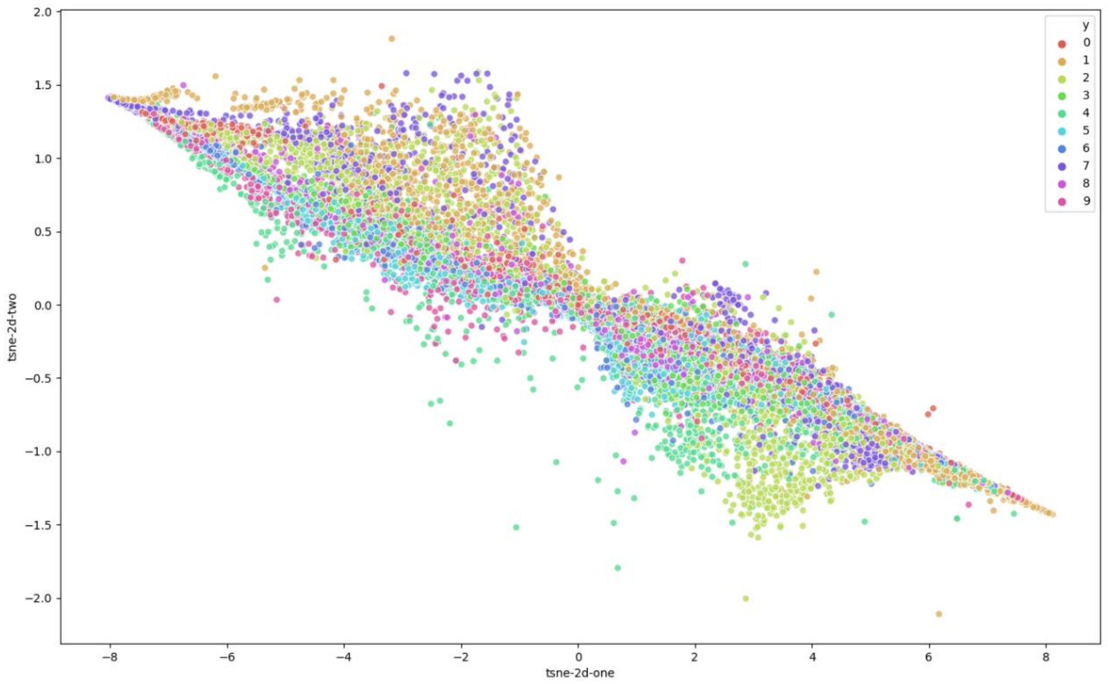

Figure 10 DANN SVHN-MNISTM digit latent space

b) Different domains (0= SVHN , 1=MNIST-M):

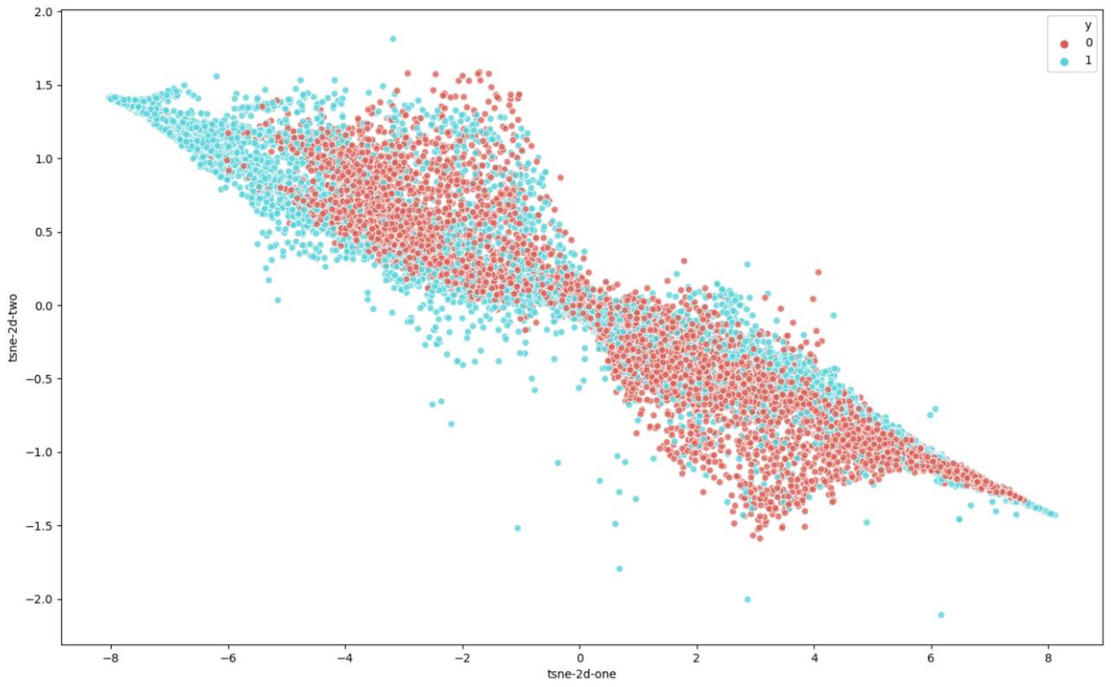

Figure 11 DANN SVHN-MNISTM domain latent space

3.5. Architecture and Implementation of model

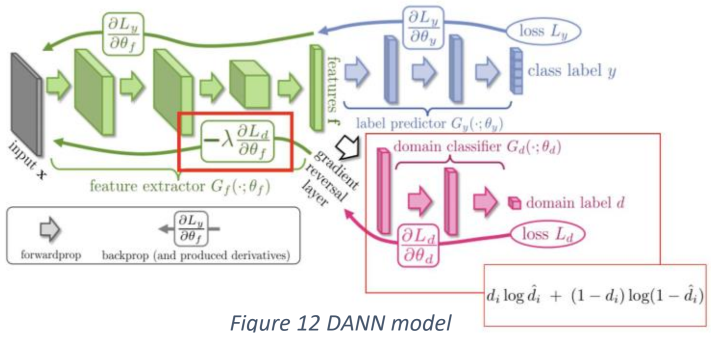
$\frac{\partial L_y}{\partial \theta_f}$
$\frac{\partial L_y}{\partial \theta_y}$
loss L_y
class label y
features $\mathbf{f}$
label predictor $G_y(\cdot;\theta_y)$
$-\lambda \frac{\partial L_d}{\partial \theta_f}$
input $\mathbf{x}$
feature extractor $G_f(\cdot;\theta_f)$
gradient reversal layer
domain classifier $G_d(\cdot;\theta_d)$
domain label d
$\frac{\partial L_d}{\partial \theta_d}$
loss L_d
forwardprop
backprop (and produced derivatives)
$d_i \log \hat{d}_i + (1-d_i)\log(1-\hat{d}_i)$

Figure 12 DANN model

For the architecture of the DANN we have followed the suggested model in the slides of the homework. For the convolutions and normalizations of the model itself we have used one suggested online by Fungtion[5], which also has the alpha and lambda mentioned in the slides that calculate the gradient reversal layer.

Model:

The model seen in the image above is the one implemented. It contains three main layers divided into smaller layers: The features, the label predictor and the domain classifier.

The features are calculated by a series of Conv2d, BatchNorm2d, MaxPool2d and RELUs. These features are used to create the graphs from the previous section and the features themselves are the input of the other layers of the model. The features array is then squished into a 1-dimensional array to be used more conveniently.

The Label predictor uses the features to calculate the prediction of the class label of the images. This is done with some linear, BatchNorm1d and ReLU, as well as some dropouts to add some noise to the results. It ends with a LogSoftmax to a 10-dimensional output (one per number that it tries to recognize).

The final layer is the one on the bottom right of the image, which is the domain classifier. This layer basically returns a 2-dimensional output with the probabilities of the image being from one domain or the other.

Other implementations:

Other important implementations in our system include: Adam optimizer, NLLLoss (if you do LogSoftmax in the model it is recommended to use this loss, while if you only used Softmax you would have to use CrossEntropyLoss[6]), require gradient in the model parameters…

For the training section we are adding the three losses: domain from source, digits from source and domain from target; and doing a backward on the total error to then do the step of the optimizer.

The code is set to show the average error 10 times per epoch to know how the code is running, and I have set 6 different options to run de code (we have lower and upper bound and DANN for both options of target and domain). Also, there are two types of training, one more simple one for the lower and upper bounds, which simply consists on the training of one of the sources and the other which is the DANN training mentioned above.

As for the creation of the graphs from previous section, we have used the seaborn function which greatly facilitated the whole process[7].

All the code can be seen in the GitHub in train_DANN.py in the DANN folder.

3.6. Observations of DANN

Throughout the development of the DANN I have found problems which we had to solve and learn through them. Some observations from the development of the DANN are:

- With high learning rate the values of the accuracy went down really fast after a couple of epochs.
- The training sometimes starts with high accuracy as soon as the first epoch and then decreases slowly, which could be because of how small and simple the images are, there might be oversampling really soon. This happens specially when training the MNIST-M data.
- When creating the graphs, you can see how the one from training in the MNIST-M has worse divisions of numbers than the other one. This is due to the first one being done after only 3 epochs, where the model wasn't as trained in general even though it gave the best values for the SVHN. Also, the division created in the first one is rounder than the second one, probably due to the complexity of the images in the MNIST-M data compared to the simpler SVHN ones, which are divided in a less complex way.
- In this case I found very important to correctly show all the average errors on screen a couple of times per epoch, since it really showed when the code would end up running correctly or not (in this case this is more important than in the ACGAN or GAN) and it was useful when knowing what parameters to change for next time.

4. Improved UDA model

For the improved UDA model, I found the simplest and most accurate solution was to use Associative Domain Adaptation models. I had to do some previous research to find this and found that, according to one of the most important Domain Adaptation libraries for python, Salad[8], that there are two methods to get better results than DANN for this kind of problem both for MNIST-SVHN and SVHN-MNIST: Associative Solver and VADA solver[9], and since the associative gave us more immediate results, we went for this option.

4.1. Domain Adaptation

Using a similar method than in the DANN, we train the model for around 50 epochs and obtain the bet accuracy possible on the target domain.

Obtained Domain Adaptation accuracy:

(1) MNIST-M -> SVHN: **<u>0.58424 (58.424%)</u>**
(2) SVHN -> MNIST-M: **<u>0.7611 (76.11%)</u>**

As we can see the results in this model is much better than the one observed in the regular DANN, especially when the SVHN data was the source (which fits with the table provided by Salad in [9], that shows a small increase in the (1) case and a big one in the (2).

4.2. Latent Space Visualization

Using the same method as in section 3.4:

(1) MNIST-M -> SVHN:

a) Different digit classes 0-9:

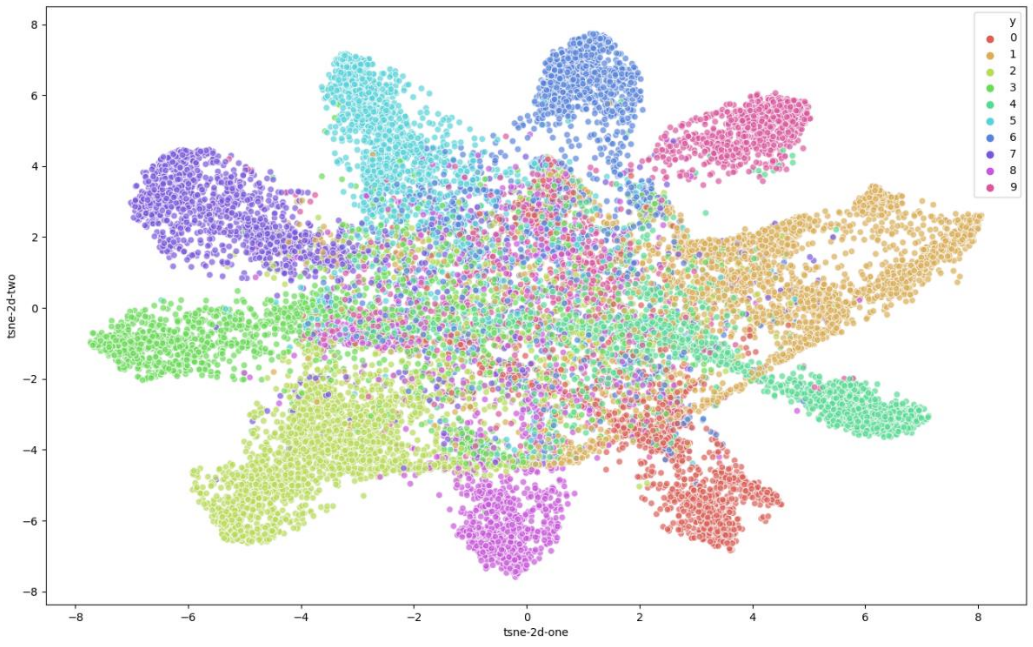

Figure 13 Improved UDA MNISTM-SVHN digit latent space

b) Different domains (0=MNIST-M, 1=SVHN):

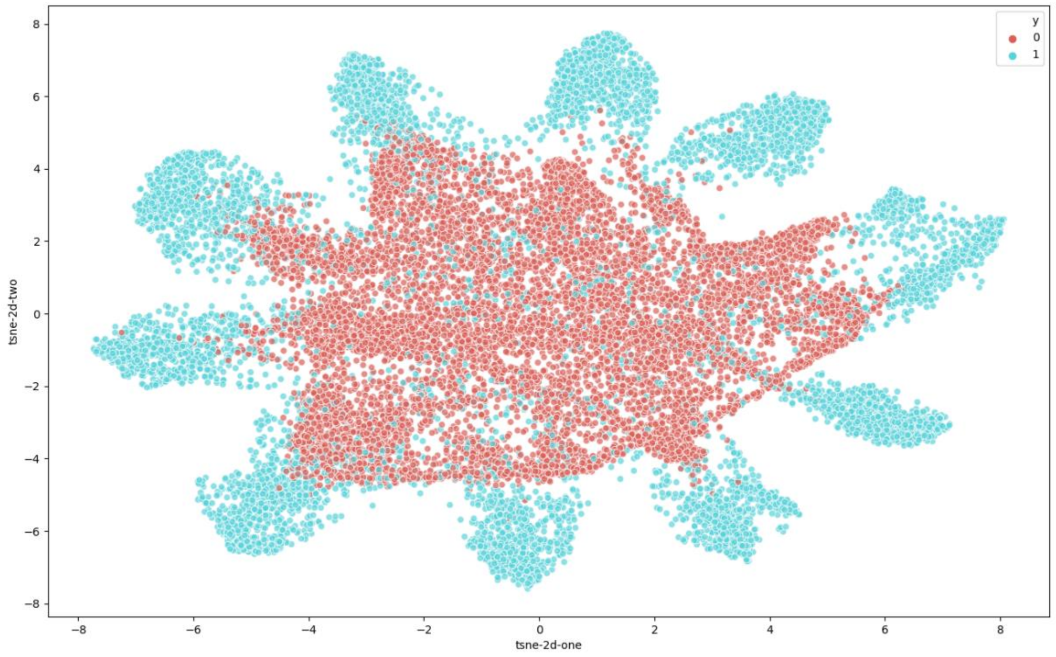

Figure 14 Improved UDA MNISTM-SVHN domain latent space

(2) SVHN -> MNIST-M

a) Different digit classes 0-9:

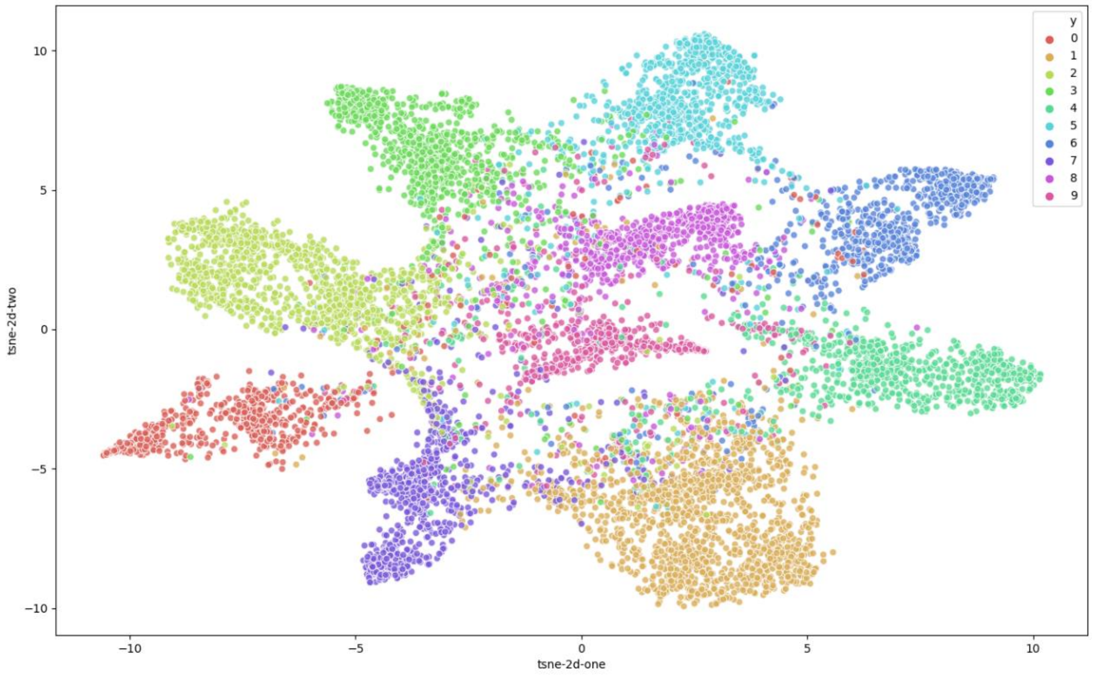

Figure 15 Improved UDA SVHN -MNISTM digit latent space

b) Different domains (0= SVHN, 1= MNIST-M):

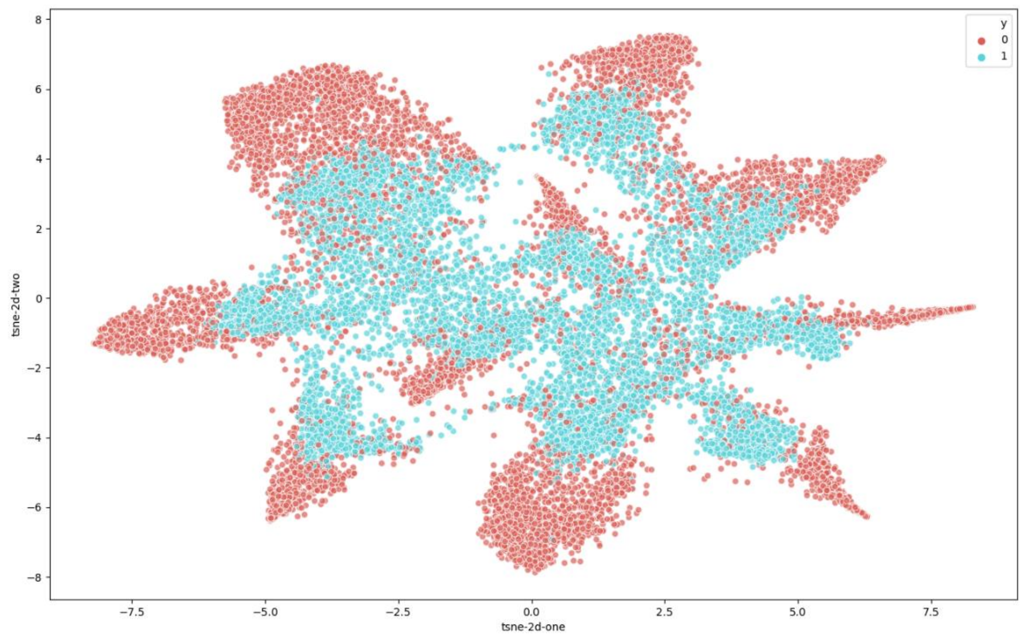

Figure 16 Improved UDA SVHN -MNISTM domain latent space

4.3. Architecture and Implementation

The architecture of the improved UDA model is very similar to the one of the DANN in terms of the training code, but have two mayor changes: The model and the loss calculation.

Model:

The model in this example is actually a lot simpler than the one in the DANN. It's based on an InstanceNorm2d and a series of conv2d and MaxPool2d to end up with a [128,9] dimensional tensor from which we find the mean for the second column. This returns to us the "features" (although they are not really "features" as in the last example since it is the actual output of the domain classification.

For the class classification we are doing a linear of the features into a 10-dimensional output, similar to the one obtained in the DANN.

Error calculation:

This is the main difference between this architecture compared to the one from the DANN.

For the loss of the class from the source we simply use CrossEntropyLoss, similar to before.

The difference comes with the error of the domain. Here we use a created error function which uses the domain output of the model from both the source and the target and the class label from the source. From these three values, the loss function does the following:

- Create Association matrix between the domain predictions of source and target.

- Implement the walker loss[10] between the matrix and the labels of the source.
- Implement the visit loss[10] on the second output of the matrix.
- Return the visit weight times the visit value plus the walker weight times the walker loss value.

This gives us a new loss that we add to the classes loss to obtain the total error from which we do the backward and step the optimizer.

The rest of the steps of the training are very straight forward and similar to the one explained in section 3.5.

4.4. Observations of Improved UDA

These are some of the observations I've learned after implementing the improved UDA model based on Associative Domain Adaptation models.

- The simplicity of the model makes the whole thing really fast to run, being about twice as fast as the DANN and getting much better results.
- The key to the whole thing working, as mentioned in the article[10] about Associative Domain Adaptation is the error calculation based on the returned domain of the source and target. This error tricks the system into not knowing if the image and label is coming from the source or the target domain.
- As seen from the graphs of the latent space, we get a much better division of the digits than what we did in the DANN model. This is also shown in the accuracy obtained on the target. As we can see, the model which gave us the much better accuracy on the target (SVHN->MNIST-M) had a better division on the numbers than the other model (although it was also pretty good).
- The training for this model started with much lower accuracy levels than the DANN, but didn't stop increasing until around the 20th epoch, contrary to the DANN that stopped really early on.
- The T-SNE space for this section took a bit more time than before to calculate, maybe because of the complexity of the latent space. To avoid having to wait for too long for it to work correctly I only got 14k testing images instead of the +30k we could get between the two test sources, but these 14k should be a good enough representation of the graphs.

Bibliography

[1] Pytorch ACGAN tutorial, link.
[2] Tips and tricks to make GANs work, link.
[3] Optimizers in GANs, link.
[4] ACGAN example, link.
[5] DANN model example, link.
[6] Using NLLLoss or CrossEntropyLoss, link.
[7] Using T-SNE for graphs of high dimensional data, link.
[8] Salad python library, link.
[9] Table showing best UDA model solutions, link.
[10]Walker loss explanation article, link.
[11]T-SNE in python not showing all images, link.
[12]Google
[13]Relevant StackOverflow error solving

Students I collaborated with:
Ricardo Manzanedo-R08942139
Javier Sanguino-T08901105
Celine Nauer-A08922116
Julia Maricalva-A08922107