RDFIA : Section 2

Report 2 : Deep Learning Applications

Simay Celik, 28713301

Aylin Soykok, 28711545

Sorbonne University - Faculty of Science and Engineering

Master DAC

Academic year: 2024/2025

Contents

1 Introduction

In this report, we discuss different deep learning applications and experiment on their practical usage. We concentrate on four main categories :

- Transfer Learning
- Neural Network Visualization
- Domain Adaptation
- Generative Adversarial Networks

We explore these different architectures and methodologies both in theory and in practice with hands-on experiments, including recreating the model, training it, tuning the parameters and trying different methods to improve the performance. Each section concentrates on the application of a specific category from above.

2 2-a: Transfer Learning

2.1 Introduction

Transfer learning aims to transfer knowledge across different but related tasks. To achieve this efficiently, we reuse an already trained model (on a different task) to detect and extract features which are then used by another smaller model to classify the image[1].

In this practical work, we will use pre-trained VGG16[16] as the model to reuse followed by an SVM on 15 Scene Data which is different from the ImageNet dataset that was used to train VGG16.

Before going further, we will start by VGG16 alone and investigate how it works.

2.2 Section 1 - VGG16 Architecture

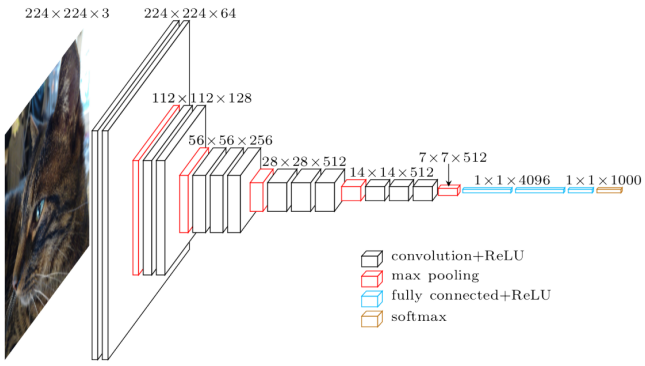


Figure 1: VGG16 Network

Figure 1: VGG Network Architecture

⋆ **Question 1.** In our previous lab work we calculated the number of weights per filter : Given our convolutional filter of size $k \times k \times z$ as the kernel needs to have the same depth as the input, we need

a weight for each element. Therefore :

$$nb_{\text{weight\_per\_filter}} = k \times k \times z$$

If we count the bias as a weight, we will have :

$$nb_{\text{weight\_per\_filter}} = (k \times k \times z) + 1$$

If we have c filters, the total number of weights to learn would be:

$$nb_{\text{weight}} = c \times (k \times k \times z) + 1)$$

We can use this equation to calculate the number of parameters in VGG16 given that VGG16 uses kernels of size 3×3.

Looking at Figure 1 given in the instructions, we can calculate the number of parameters for each layer and sum them to find the total number. Between each convolutional layer, we apply the equation above and for max pooling, we do not have any parameters.

- $224 \times 224 \times 3 \rightarrow 224 \times 224 \times 64$: The depth changes from 3 to 64 so $z = 3$ and for each output, we have a filter causing $c = 64$.

$$nb_{\text{weight\_1}} = 64 \times (3 \times 3 \times 3) + 1) = 1792$$

- $224 \times 224 \times 64 \rightarrow 224 \times 224 \times 64$: We repeat this layer.

$$nb_{\text{weight\_2}} = 64 \times (3 \times 3 \times 64) + 1) = 36928$$

- $224 \times 224 \times 64 \rightarrow 112 \times 112 \times 128$: The depth changes from 64 to 128 so $z = 64$ and for each output, we have a filter causing $c = 128$.

$$nb_{\text{weight\_2}} = 128 \times (3 \times 3 \times 64) + 1) = 73856$$

With this logic, we can calculate the value for each layer :

- $112 \times 112 \times 128 \rightarrow 56 \times 56 \times 258$:

$$nb_{\text{weight\_3}} = 256 \times (3 \times 3 \times 128) + 1) = 295168$$

- $56 \times 56 \times 256 \rightarrow 28 \times 28 \times 512$:

$$nb_{\text{weight\_3}} = 2512 \times (3 \times 3 \times 256) + 1) = 295168$$

$$\dots$$

Fully connected layers are richer in weights and in the first FC, we have an explosion of parameters passing from $(7 \times 7 \times 512)$ to $(1 \times 1 \times (7 \times 7 \times 512)) = (1 \times 1 \times 25088)$:

- $1 \times 1 \times 25088 \rightarrow 1 \times 1 \times 4096$:

$$nb_{\text{weight\_3}} \approx 100M$$

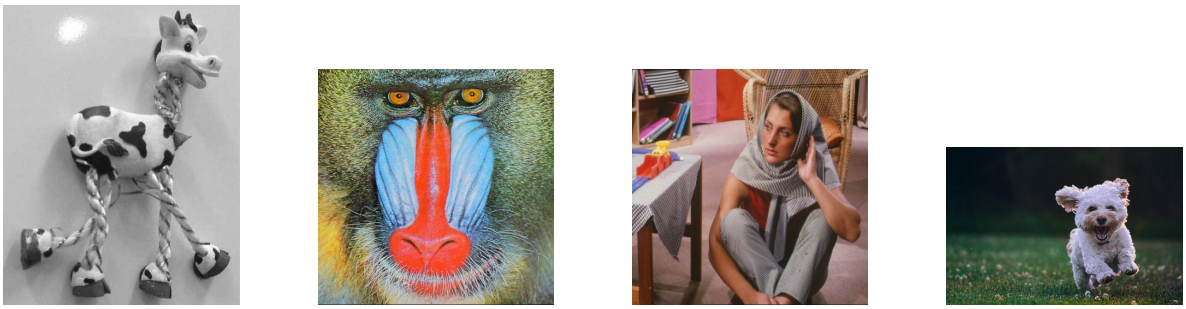
$$\dots$$

As the number of input and output increases, the number of parameters increases almost exponentially even when we repeat the layers, giving us more than 100 Million parameters. The majority of this number comes from the first fully connected layer.

⋆ **Question 2.** The output size of the last layer of VGG16 is 1000 corresponding to the 1000 classes of images in the dataset (ILSVRC-2012) which was used to train the model. It is important to note that because of the last few Fully Connected layers, VGG16 requires fixed-size inputs although the convolutional layers can adapt to different sizes. [16]

Image	**Prediction**
Cow toy	dalmatian, coach dog, carriage dog
Baboon	baboon
Barbara	rocking chair, rocker
Dog	wire-haired fox terrier

Table 1: Predictions for different images using VGG16.



(a) Cow Toy (b) Baboon (c) Barbara (d) Dog (Cockapoo)

Figure 2: Images tested on VGG16

⋆ **Question 3.** As seen in Table 1 above, VGG16 predicts classes based on what it learned from ImageNet images. Here, we fed different types of images (seen in Figure 2 through to see how it would classify known and unknown classes. Baboon and exists in ImageNet so the model could easily predict its class. For the dog image, it classified the dog breed by wire-haired fox terrier, however, it should be Cockapoo. So, even if the model classified it as a dog, as there is no Cockapoo class in ImageNet, it classified it as a close relative. For the famous image processing example Barbara, it could not detect the woman because there is no human or woman class in ImageNet so it detected the chair behind her instead. For the other famous toy image, it classified it as some breeds of dogs. This shows us how the training set of a pre-trained classifier can perform with examples outside of the dataset. The predictions make sense most of the time; however, if the domain or the task is unfamiliar, it does not perform well.

What is the role of ImageNet normalization?

ImageNet normalization standardizes the image pixels based on the data distribution that VGG16 was trained with. By doing so, our images become compatible with the model's expectations and we optimize gradient descent (time & performance) by making the input features contribute more proportionally to the gradients.

Why setting the model to eval mode?

We set the model to eval mode because VGG16 uses Dropout layers to prevent overfitting during the training period and in evaluation mode. These Dropout layers are deactivated because we want all neurons to contribute to the prediction. Without eval mode, we would deactivate neurons during prediction, causing unreliable results with inconsistencies.

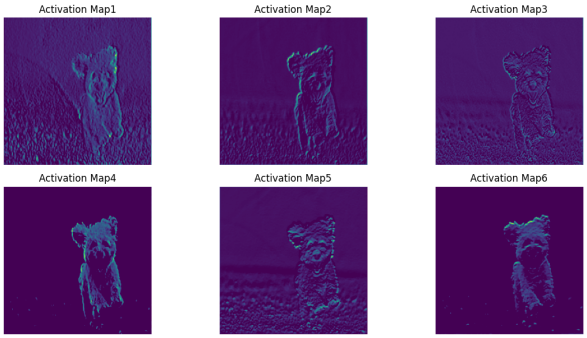


Figure 3: Activation maps after the first convolutional layer

Question 4. Bonus Activation maps highlight features that a convolutional layer detects. For the first layer, these are mostly edges, textures, and simple patterns in the input image. We can interpret them by looking at the bright areas where the filter detected a strong feature. In Figure 3, we see multiple maps, each map corresponds to a different filter.

We can speculate on what these filters are focused on based on these activation maps. For example, the first one looks noisy and highlights little details such as *texture*, but in the sixth one, we can easily distinguish between the background and the dog. In the 3rd and the 4th ones, we can see highlighted edges surrounding the dog.

To go a bit further, we decided to print activation maps from deeper layers.

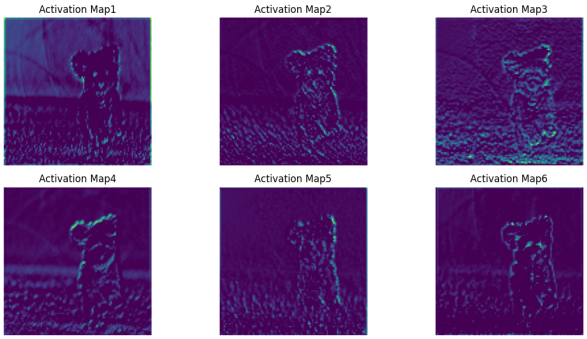


Figure 4: Activation maps after the second convolutional layer

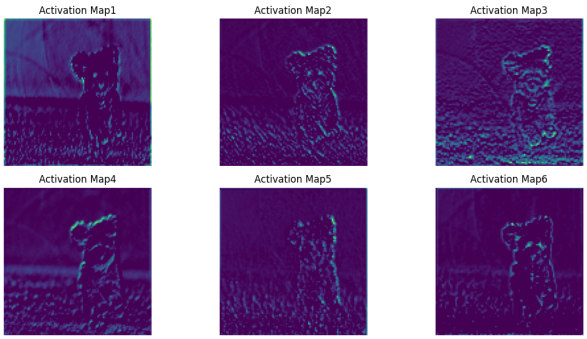


Figure 5: Activation maps after the last layer

As we go deeper, the maps become less detailed and more general, demonstrating how the convolutional layers in VGG down-sample and apply max pooling to focus increasingly on the general features of the image.

2.3 Section 2 - Transfer Learning with VGG16 on 15 Scene

2.3.1 2.1 Approach

⋆ **Question 5.** With transfer learning, we want to use a model that was trained on a task on another (related but different) task. To do this, we could directly train VGG16, however, it is a very deep network with millions of parameters, which was trained on a huge dataset (ImageNet) while 15 Scene is a small dataset. With limited data, direct training can lead to overfitting. It would memorize the training data instead of generalizing them.

So instead, we use this big pre-trained model with frozen features to extract features and feed them into a smaller model (SVM) which needs much less computational power.

⋆ **Question 6.** Pre-training on ImageNet enables VGG16 to capture and extract multiple features (such as edges, shapes, objects, etc.) from over 1 million images with 1000 classes leading to a rich knowledge. So, the model already knows how to detect important features, and we can transfer them in this case to another dataset (15 Scene).

Question 7. The general limit is the adaptability: the method works best when the tasks and the domains (data distributions) are similar.

If the features of the new dataset are significantly different from the transferred ones, the end result may be unreliable.

If the tasks are different, the important features to extract will be different for the two datasets and the model can struggle to find relevant features.

The pre-trained model should be effective and compatible with the new task at hand. If it is not well trained and the dataset lacks diversity, it might produce insufficient or biased results.

2.3.2 2.2 Feature Extraction with VGG16

Question 8. The layer influences the quality and specificity of the features.

In deep networks like VGG16, earlier layers extract shallow features (such as edges, textures, etc.) whilst as we travel further along the network, the features get more complex (patterns, objects, etc.). So, choosing a layer in the earlier stages will be useful for more general and non-specialized tasks. If the task in hand requires more detail, then we prefer intermediate or deeper levels with higher level features.

Question 9. We can duplicate the grayscale image across the color channels which would convert it to an RGB image or we can apply a colormap on it to maintain some information on the colors. This conversion will make them more compatible with VGG16 and its features.

2.3.3 2.3 Training SVM classifiers

Question 10. We can use the NN directly by either retraining it on the new data or fine tuning it. By retraining it, we can obtain better performance as it will be adapted to new data. However, these scenarios require creating the NN from scratch or updating/modifying every layer once again which can lead to overfitting if new dataset is small and it is computationally expensive and time consuming to train a large model like VGG. So Transfer Learning by using an independent classifier like SVM is preferred because we make use of pre-trained features so it is computationally cheaper. Furthermore, a linear classifier like SVM is easy to interpret compared to deep neural networks and they work faster and can still achieve satisfying performance.

Results of VGG16relu7 After using Transfer Learning stopping at relu7 in VGG16, we tested it on 15-SceneData which contains different types of scenarios in grayscale.

It took 37 seconds to run the entire training pipeline starting from initializing the VGG models to finishing training. After training, we used the model to predict an image in the train test and it took 0.2 seconds to correctly classify the scene (Figure 6a) and we also used the original pre-trained VGG16 model to predict the class of the same image without doing any modifications on it and it misclassified in 0.0003 (Figure 6b.

The accuracy of VGG16relu7+SVM reached 0.88 which is significantly high for the amount of training that was done.

(a) Classification by pre-trained VGG16

(b) Classification by VGG16relu7+SVM

Figure 6: Comparison of classifications by pre-trained VGG16 and VGG16relu7+SVM

2.3.4 2.4 Going further

Question 11.

Changing the layer at which the features are extracted *What is the representation size and what does this change?* Changing where we extract the features can provide us with insight into how the feature extraction and the new classifier work. VGG16 is a CNN. As mentioned above, the architecture of CNNs were designed to detect increasingly global patterns in an image, and the last Fully Connected layers aim to capture the relationships such as semantic representation. So, where we stop and extract features changes the type of features we obtain to use in classification. Therefore, shallow layers capture low-level features and retain spatial information while they lack semantic information and deeper features keep semantic information but they lose spatial details after pooling and flattening.

What is the representation size and what does this change? Representation size is the number

of features in the output of a layer. It depends on the depth of the layer (as we go deeper, spatial dimension changes) and the number of filters : $RepresentationSize = H \times W \times C$. Hence, depending on the layer we extract features, representation size will change.

To see if relu7 makes any difference, we tried relu6 which is in FC layers and some layers in the convolutional part.

Model	Train Accuracy	Test Accuracy	Training Runtime (s)	Single Test Runtime (s)
relu7	1.000000	0.885762	40.4069	0.2689
relu6	1.000000	0.869682	49.4845	0.2291
relu4	1.000000	0.850921	41.9435	0.2294
relu1	1.000000	0.901508	37.0251	0.3870

Table 2: Training and testing results for different VGG16 layers.

In Table 2, we see different results obtained by extracting features from different layers. We see that all of them achieved perfect training accuracy which can indicate overfitting but test accuracies are also quite high but the gap is large so the generalization is not perfect. But, all depths gave similar results so VGG16 might be providing excessively complex features for SVM causing it to overfit.

For the test accuracy, VGG16relu1 performed the best and VGG16relu7 followed behind closely while relu6 and relu4 performed worse. This could be because relu1, being a shallower layer, extracts simpler features. These features might be more suitable for the SVM, as it performs better on less complex data. In contrast, at relu7, although the features are more detailed and semantically rich, they encapsulate the entirety of the feature space, providing sufficient information for the SVM to classify effectively. The features extracted at intermediate layers may not have been as suitable or comprehensive for the SVM, potentially leading to lower performance.

In terms of runtime, all models performed similarly overall. While relu1 was faster during training, it was over 10 seconds slower than the others when testing a single image. Considering both performance and speed, relu7 appears to be the most suitable option.

Trying other pre-trained networks We tried AlexNet and ResNet with the same methodology.

AlexNet [10] being one of the first CNN to perform well based on ImageNet images and a shallower version of VGG16, it can provide us insights on how the depth of VGG16 influenced the results.

ResNet50 [7] is much deeper than VGG16 and solves the vanishing gradient problem and has much less parameters compared to VGG16.

Model	Train Accuracy	Test Accuracy	Training Duration (s)	Single Test Duration (s)
relu7	1.000000	0.885762	40.4069	0.2689
AlexNet	1.000000	0.862647	23.9372	0.2382
ResNet50	1.000000	0.908208	25.9784	0.2343

Table 3: Performance and runtime comparison of different models.

In the table above Table3, we see how these three models performed with Transfer Learning. To ensure consistency across the experiments, we utilized the final FC for all models.

Both models, AlexNet and ResNet50, outperformed VGG in terms of runtime. This can be attributed to AlexNet being a shallower model and ResNet50 having fewer fully connected layers and, consequently, fewer parameters.

In terms of performance, AlexNet was the weakest, VGG performed moderately, and ResNet50

achieved the best results. This indicates that the greater depth of VGG compared to AlexNet improved the performance. Similarly, the even greater depth of ResNet50 appears to improve its performance further. These results demonstrated the benefit of increased model depth when designed effectively and the adaptability of Transfer Learning across different models.

Tuning the parameter C to improve performance C is a regularization parameter which controls the trade-off between maximizing margin and minimizing misclassification. Larger C prioritizes minimizing the margin and is more strict with misclassification but is more prone to overfitting whilst smaller C relaxes the constraint and allows a wider margin, is more tolerant to misclassification but is prone to underfitting. By tuning this parameter, we can adapt the model to have better performance based on the data so it is an important step. To tune this parameter, we used cross validation grid search on VGG16relu7 on 15 Scene data.
In Figure 7 below, we see how C affected the performance of our model. The best accuracy (of 0.89) was achieved with a smaller $C = 0.001$ indicating that reducing C helped to relax the margin and prevent overfitting, thereby improving generalization.

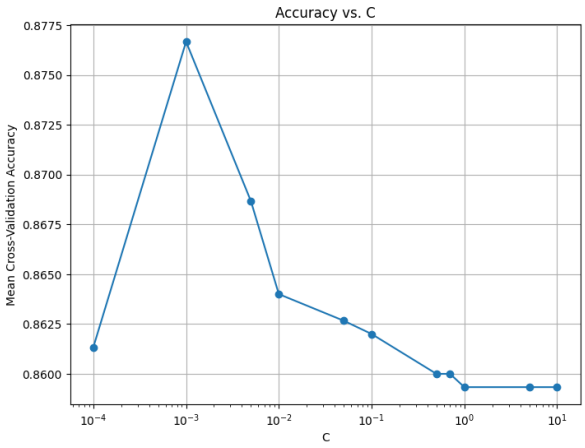


Figure 7: Effect of Regularization Parameter C on SVM Accuracy

2.4 Conclusion

In this work, we experimented with transfer learning using VGG16 and other models. Our experiments highlighted how significantly the pretrained model, feature extraction layer and SVM parameters impact the performance. This showed us the importance of tuning and choosing the correct model.

Additionally, we demonstrated the methodology behind transfer learning and how it is applied in real data with similar yet different tasks.

3 2-b: Visualizing Neural Networks

3.1 Introduction

In this section, we will explore techniques to try to understand the behavior of convolutional neural networks. Before, we looked at calculating the gradient of the loss with respect to the parameters, now we will calculate the gradient with respect to the pixels of the image.

We look at 3 techniques:
Saliency Map: Highlights the important parts of the image for the classification decision.

Fooling Samples: By adding small modifications to an image that aren't noticeable to the human eye, classifying it as a different class.
Class visualization: Visualizing what patterns are important for the network for classification of certain classes by generating images.

3.2 Section 1 - Saliency Map

⋆ **Question 1.** The saliency map highlights the most impactful pixels for predicting the correct class. In figure 8, the most influential pixels correspond well to the pixels that we expect to be important for the correct class. For example, in the third image, the highlighted pixels clearly outline the dog.

In figure 9, for the second and third images, we see examples of saliency maps that aren't very informative, the highlighted pixels are scattered around the images without a coherent form. However, most of the visualizations show well-defined saliency maps that provide insights.

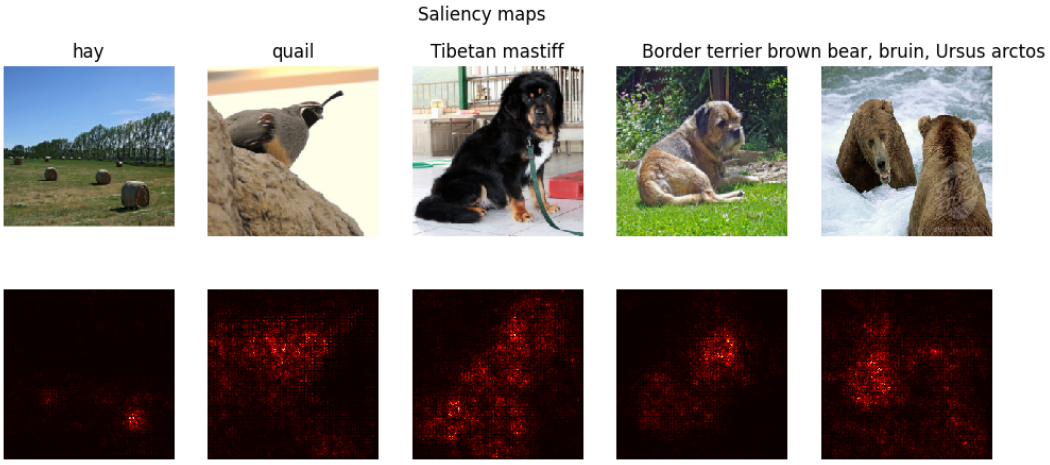


Figure 8: Example saliency maps

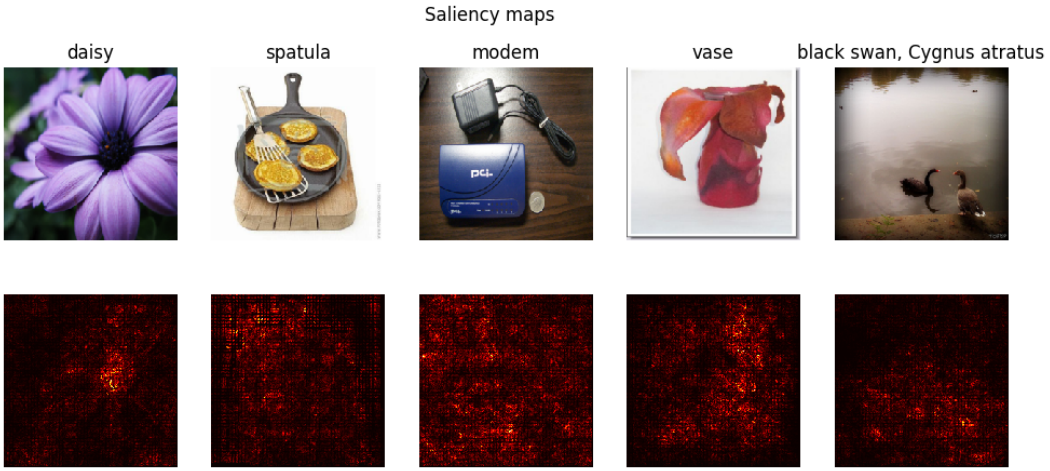


Figure 9: Example saliency maps

Question 2. Saliency maps have many limitations.
They are sensitive to the hyperparameters and can result in inconsistent results when the hyperparam-

eters are changed. They are also dependent on the model, different models can give varying saliency maps for the same input. [5]

They can be noisy. There are many studies about this issue. For example, [9] proposed a new hypothesis to address this problem, claiming that saliency maps are noisy because deep neural networks don't filter out irrelevant features during forward propagation.

Their interpretations aren't clear. For instance in medical imaging, the saliency maps may highlight some regions but we're not sure if those are the areas that we should be looking at for diagnosis. [18]

Question 3. Saliency maps have applications in object detection, content-aware image resizing, and picture collage. [8] They can also be used in dataset bias assesment. [12]

Question 4. We used VGG16 to obtain saliency maps. In figure 10 we compare the saliency maps of SqueezeNet and VGG16. When we look at the 'hay' pictures' saiency maps, we can see that SqueezeNet put most of the focus on the far right hay, whereas VGG16 highlighted the other hays present in the image as well. For the 'Tibetan mastiff', the saliency map of Squeeze Net is more interpretable, as it highlights the entire dog. For 'brown bear', SqueezeNet identifies the bear on the left as the most impactful, while VGG16 highlights both bears. So there are variations in the saliency maps generated by the two models. This shows the importance of the model.

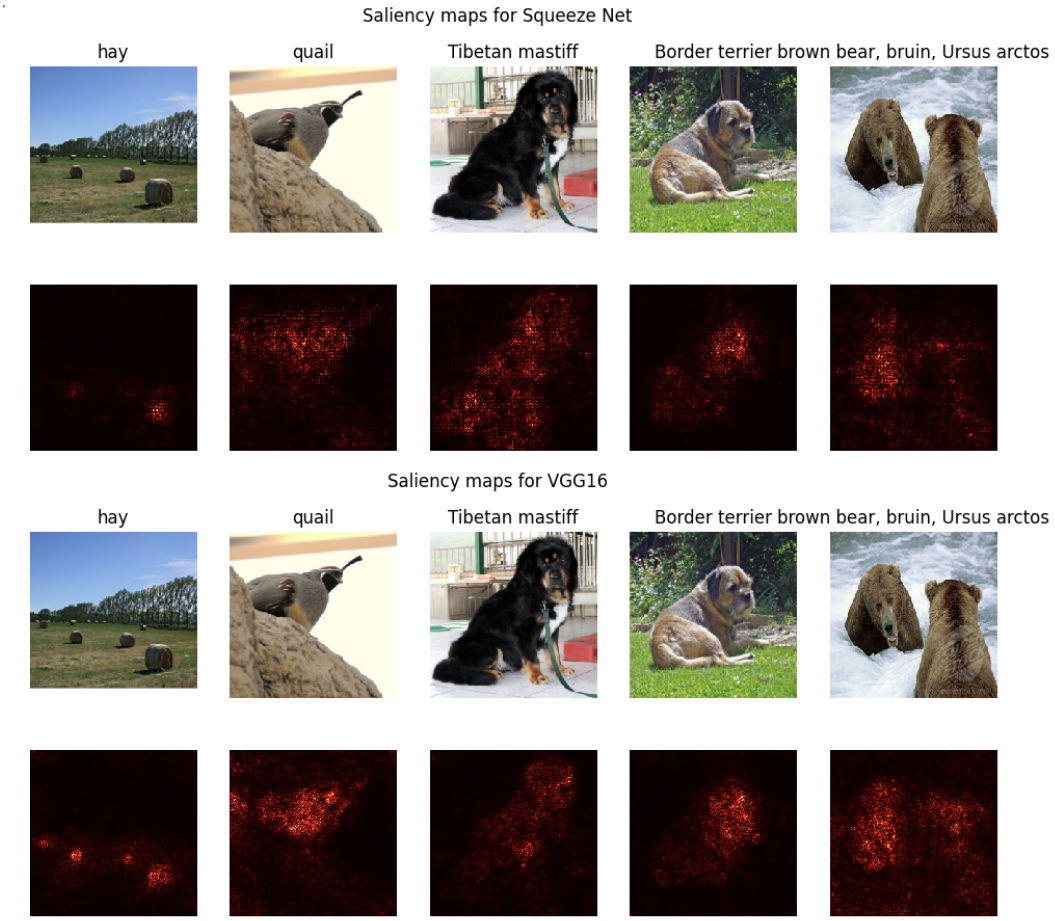


Figure 10: Saliency maps produced by SqueezeNet and VGG16

In figure 11, for the first image, the saliency map produced by VGG16 is more intuitive than

Squeeze Net's. The saliency map of Squeeze Net highlights many pixels scattered around the image, not really telling us something, while the map of VGG shows the structure of the scarf and highlights the neck region.

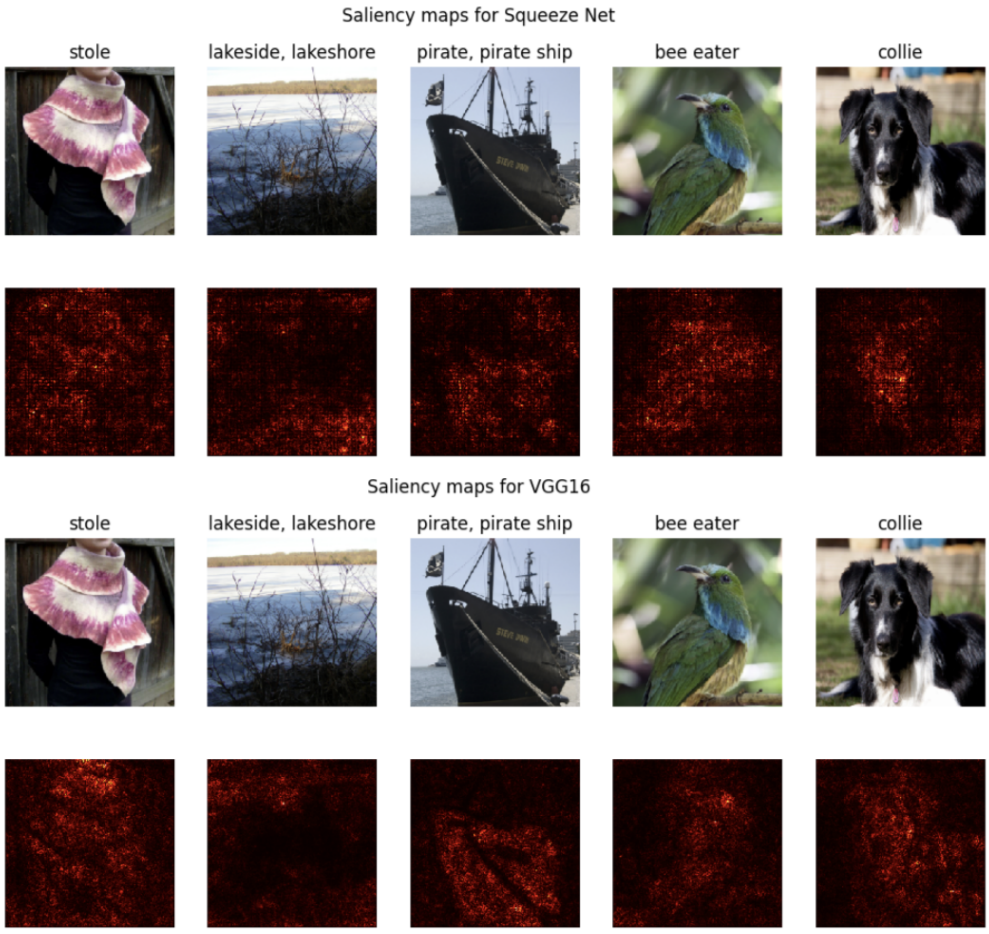


Figure 11: Saliency maps produced by SqueezeNet and VGG16

3.3 Section 2 - Adversarial Examples

⋆ **Question 5.** We tested different input images and target classes.
In figure 12, when trying to misclassify daisy as brown bear and as pirate ship, we see that the pixels that were changed are mostly in the center part of the flower (disc floret). This makes sense since this part is important to identify daisies, and in the saliency maps section we also saw that this part of the daisy was highlighted by the saliency maps, so they're important for the classification of the image as 'daisy'.

Modifying the classification of the 'Tibetan mastiff' dog in figure 13, in the first row we're classifying it as a different breed of dog (corgi), and we see that the changed pixels outlined the dog's silhouette. On the second row, we classify the dog as 'toilet paper', and in this case not only the pixels around the dogs silhouette are changed, but also the pixels in the background. This suggests that for classification as toilet paper, the expected background might be a bathroom setting.

Figure 12: Fooling images on 'daisy'

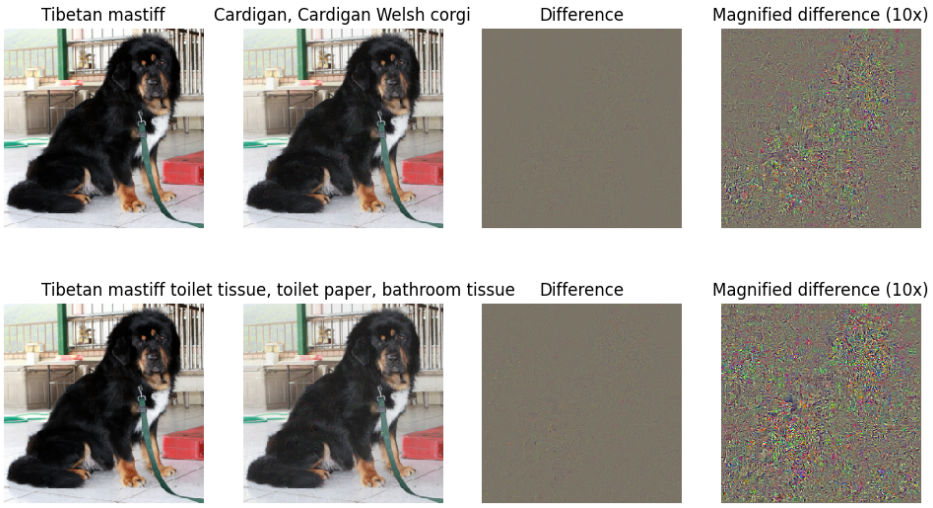


Figure 13: Fooling images on 'Tibetan mastiff'

In figure 14, classifying 'soap dispenser' as 'bee eater' (a bird) and 'corgi' didn't show an intuitive difference. There are some pixels that were changed scattered around the difference image, but they aren't very interpretable. This also reminds us of the saliency map of the soap dispenser shown in figure 15, which wasn't very interpretable either. This may suggest that the soap dispenser lacks characteristic features for its classification.

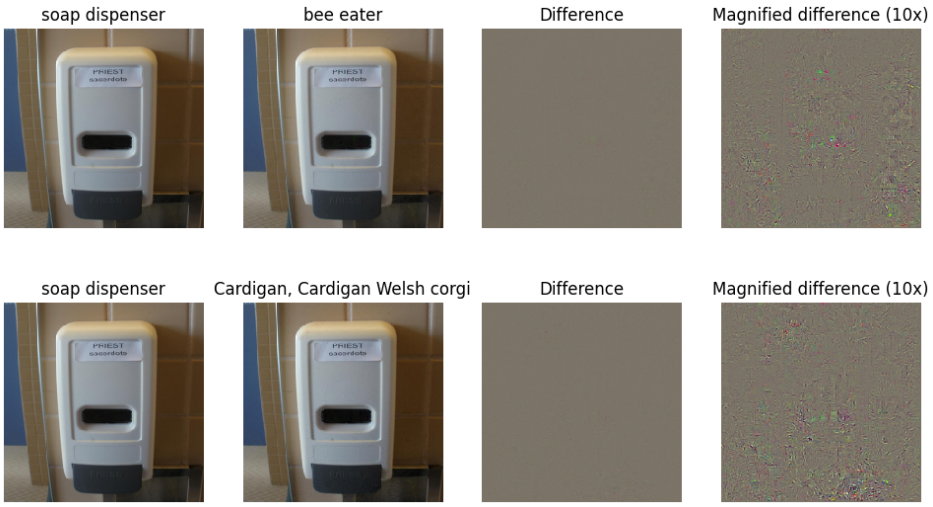


Figure 14: Fooling images on 'soap dispenser'

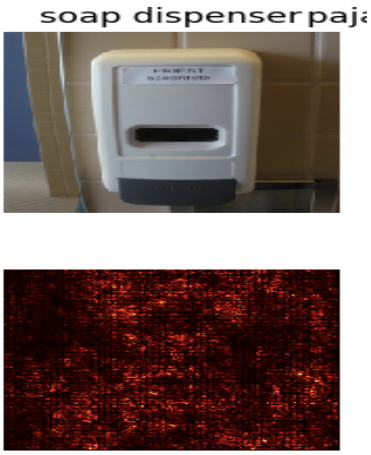


Figure 15: Saliency map of SqueezeNet for soap dispenser

Overall, in all tests, when looking only at the input image and the resulting image, we don't see a difference to the naked eye, but the images are classified differently. The 10x magnified difference provides better insight into the changed classification.

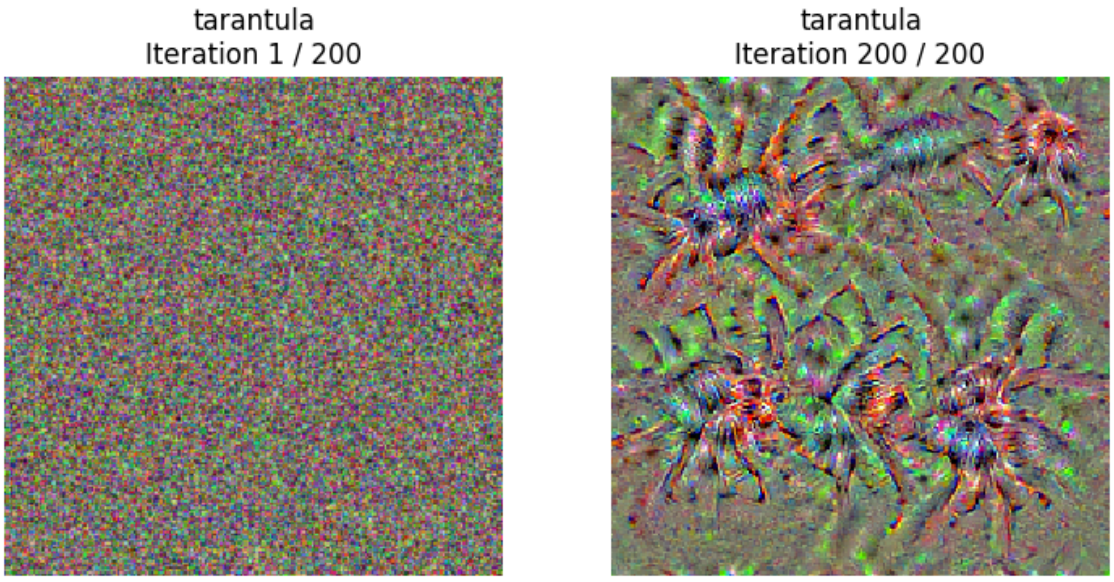
Question 6. Adversarial examples that fool deep networks with slight differences can introduce risky consequences. They expose vulnerabilities of neural networks and can be used to disguise malicious content such as malware. [17]
A work [15] showed that facial recognition systems can be fooled with adversarial attacks by generating glasses and the attacker can impersonate someone else or evade recognition. In autonomous driving, adversarial attacks can mislead object detection models to misclassify objects like traffic signs.[4]
For medical imaging, adversarial attacks can lead to incorrect diagnoses, putting the patient's well-being at risk. [19]

Question 7. The naive approach of progressively applying gradient ascent can have limitations such as it's computational efficiency since it's an iterative method. A method proposed in [13] where

generative models are used to create adversarial examples is shown to be faster than iterative methods. The naive approach is using just the target class score, a more sophisticated approach can result in minimally perceptible perturbations in the input image. Work in [2] uses a cost function that includes regularization terms to generate adversarial examples that are very similar to the input images.

3.4 Section 3 - Class Visualization

⋆ **Question 8.** Using the default settings given in the notebook, we started from random noise image and modified it so that it's classified as a tarantula. The image was progressively adjusted. In figure 16 (b) there are a few tarantula patterns in the image, although with random colors and not real life-like, they show us the patterns the network associates with tarantulas. This method provides insight into the learned features and allows us to better understand the network.



(a) Random noise

(b) Resulting image for 'Tarantula' class

Figure 16: Generation with default settings (l2_reg=1e-3, learning_rate=5)

In figure 17, we can see patterns associated to gorillas like their eyes, furry texture, and wrinkly faces.

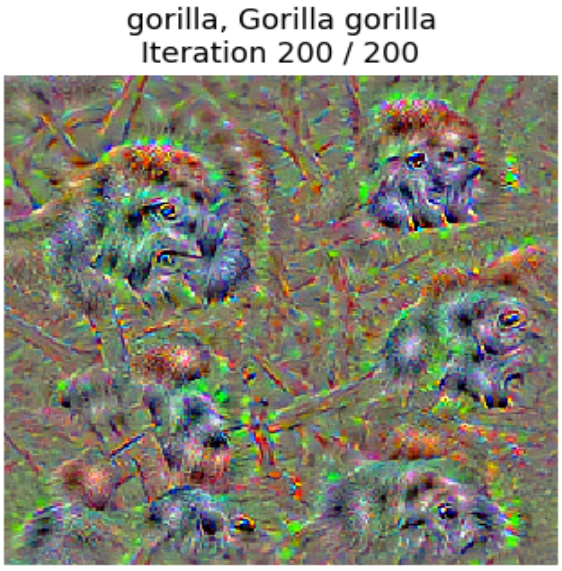
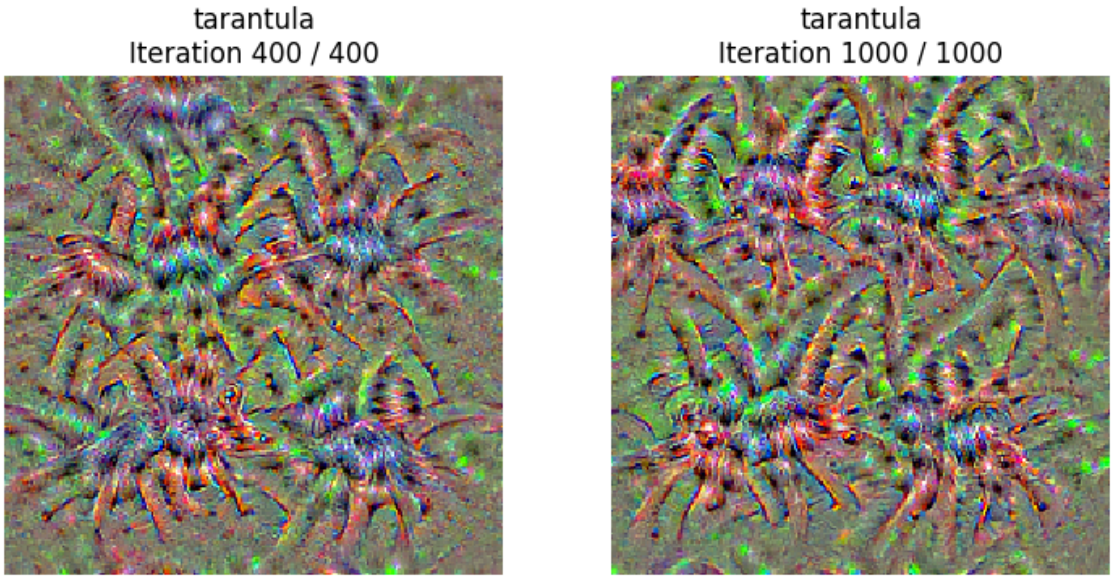


Figure 17: Resulting image for 'gorilla' class with default settings (l2_reg=1e-3, learning_rate=5)

Question 9. We looked at the impact of different numbers of iterations.
Training for 1000 iterations resulted in brighter images and more distinct tarantula-like shapes compared to the images in figure 16 generated after 200 iterations. With a longer training period, the network learned more refined features.



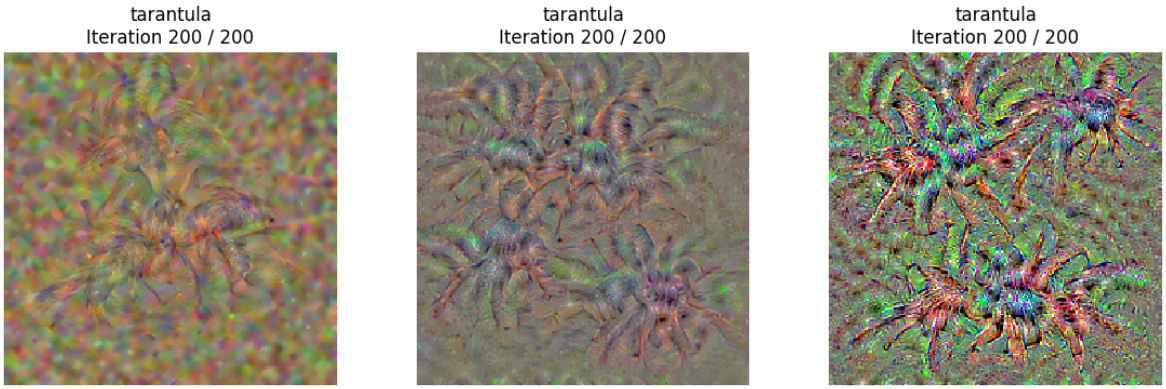
(a) 400 iterations

(b) 1000 iterations

Figure 18: Class visualization with different iterations and default settings (l2_reg=1e-3, learning_rate=5)

Experimenting with the learning rate, we changed the default learning rate of 5. In figure 19 using a very small learning rate (0.1) resulted in poor-quality, uninterpretable images (0.1), this learning rate was too low. The modifications were too minimal to get meaningful features. With a learning rate of 1, the modifications were still not good enough. Increased learning rate of 10 led to a bright

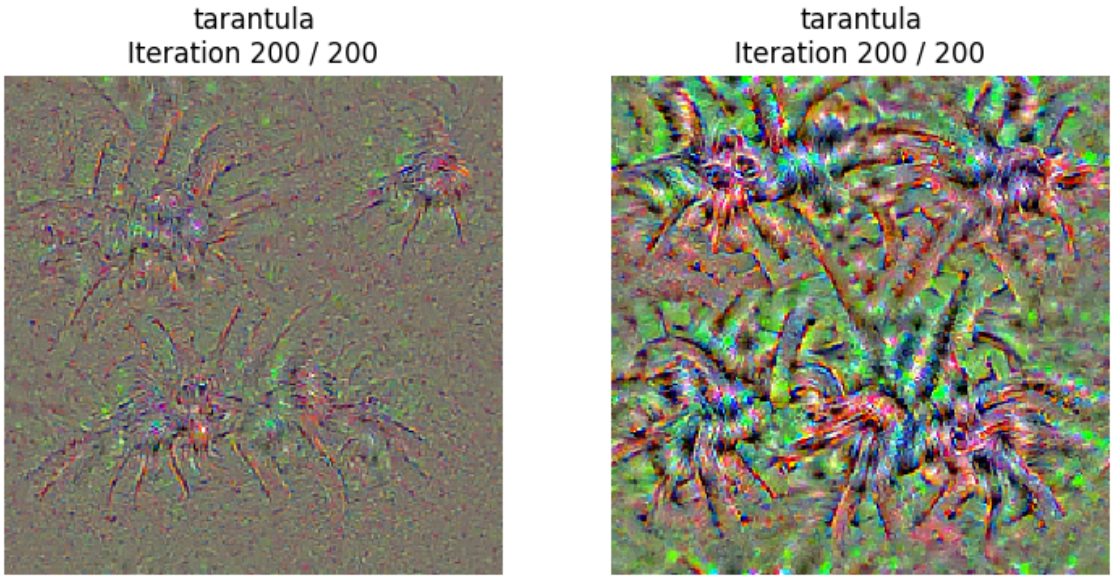
colored image but the features with learning rate 5 were more detailed and easier to interpret.



(a) learning_rate=0.1 (b) learning_rate=1 (c) learning_rate=10

Figure 19: Class visualization with different learning rates and default settings

Adjusting the regularization weight from the default 1e-3, we observed some differences (figure 20. A higher value (1e-2) resulted in images with less distinctive features. A lower regularization (1e-5) led to noisier images that were hard to interpret compared to those with 1e-3. The default 1e-3 was a better choice in this setting given the other hyperparameter values (lr=5, nb_iterations=200 etc.)

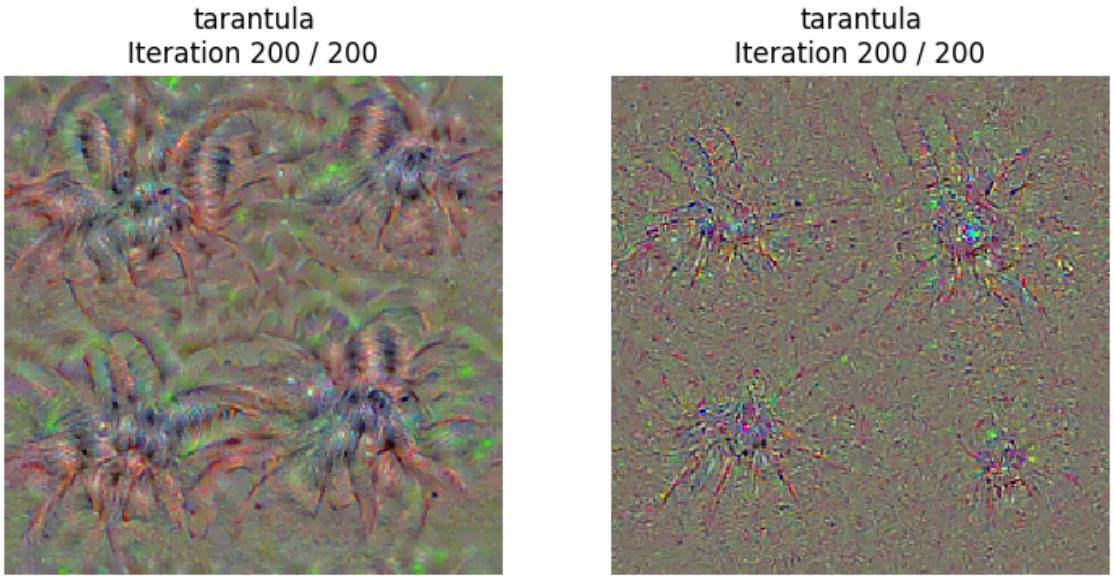


(a) l2_reg=1e-2 (b) l2_reg=1e-4

Figure 20: Class visualization with different regularization strengths and default setting (learning_rate=5)

Given our results, we also tried combining some values for both the learning rate and regularization weight. In figure 21 on the left we used a low learning rate and a low regularization weight. The resulting image wasn't as bright as when we used the same regularization weight but higher learning rate (5). With this combination the spider shapes look more distinct compared to using a learning rate of 1 with higher regularization.
On the right, combining a high learning rate with high regularization gave a noisy and poor quality image. While using a high regularization weight with a lower learning rate earlier produced a slightly

more interpretable image, it was still not satisfying. In figure 19 when we had used a high learning rate with the default regularization (1e-3), the image was more colorful and brighter.

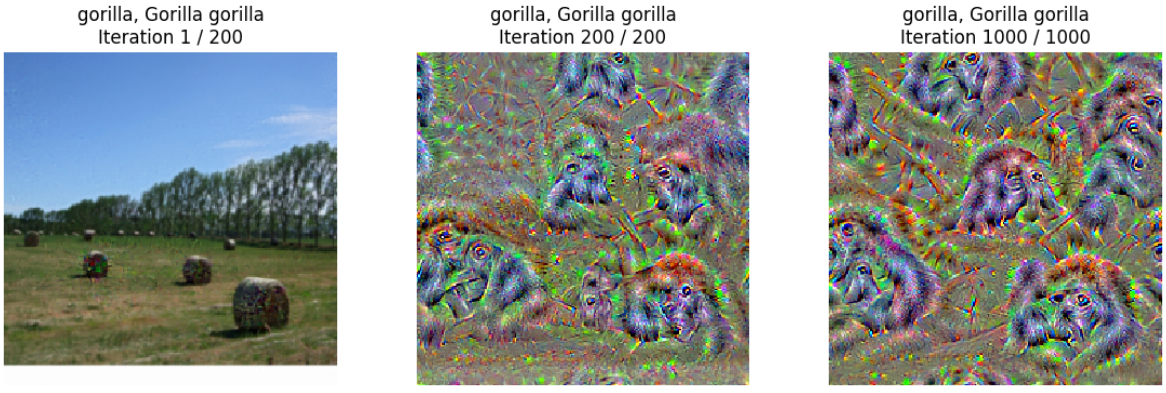


(a) learning_rate=1 and l2_reg=1e-4 (b) learning_rate=10 and l2_reg=1e-2

Figure 21: Class visualization with different regularization strengths and learning rates

Question 10. Instead of a random noise image, we used an image from the ImageNet dataset as source image.

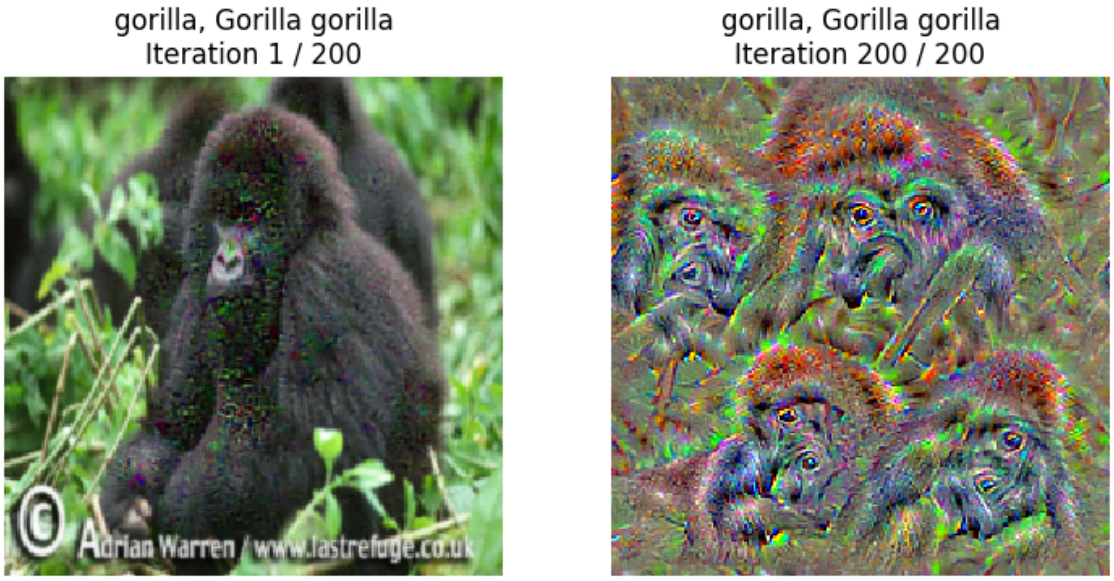
In figure 22, we started from the hay image to classify the image as gorilla. In early iterations, we can observe how the network begins generating features, starting by covering the hay with patterns resembling gorilla features. Using an image from ImageNet instead of random noise can help us see how the existing features of the image are changed to incorporate the features of the target class, allowing us to have an insight into how the network adapts the features to align with the target class.



(a) hay image as input (b) with 200 iterations (c) with 1000 iterations

Figure 22: Class visualization with image from ImageNet as source image

In figure 23, we give the gorilla image as input and we observe that the network generates a gorilla face pattern around the input gorilla's face, and also generates additional gorilla patterns throughout the image.

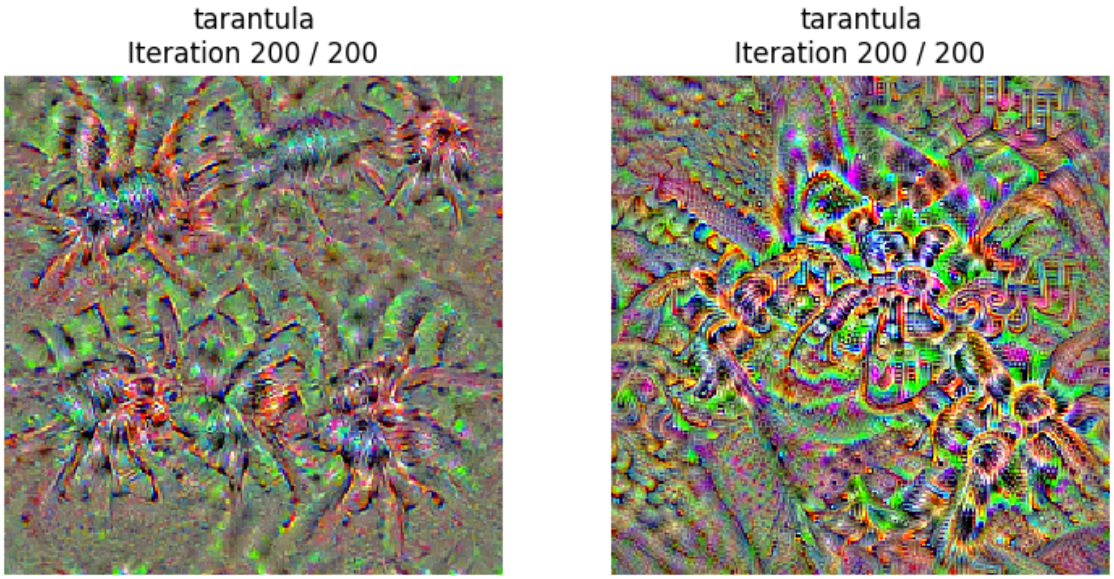


(a) Input gorilla image

(b) Resulting image

Figure 23: Class visualization with the same image as the target class for source image

Question 11. We tried VGG16 instead of SqueezeNet. In figure 24, starting from random noise, the generated tarantula patterns using VGG16 on the right are different to those of Squeeze Net. VGG's image is brighter and has more prominent outlines for the tarantulas. SqueezeNet's image has simpler details, with less pronounced tarantula outlines. Despite these differences, both networks produce patterns that resemble tarantulas.



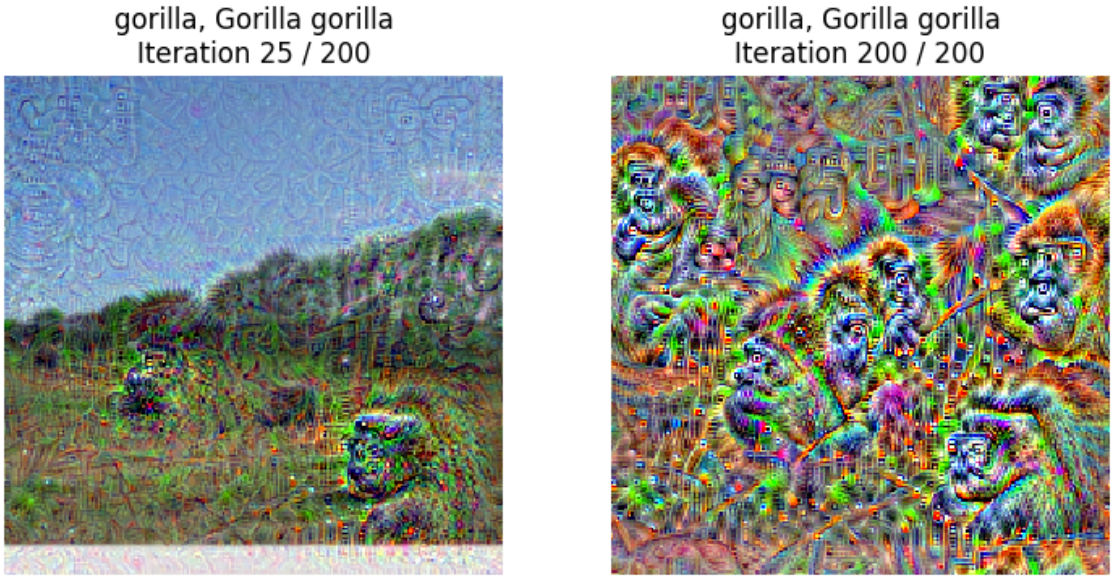
(a) Squeeze Net

(b) VGG16

Figure 24: Class visualization with default setting for Squeeze Net and VGG

In figure 25, we observe how VGG16 constructs gorilla patterns starting from the 'hay' image from ImageNet as the source. Compared to the corresponding image generated by SqueezeNet in figure 22

(b), VGG16's visualization is brighter and has more defined patterns. This difference can be attributed to the architectural differences between the models, SqueezeNet is designed to be efficient and compact whereas VGG16 is deeper.



(a) hay as source image

(b) VGG16

Figure 25: Class visualization using VGG16 with image from ImageNet as source image

3.5 Conclusion

In this lab, we gained a deeper understanding of how convolutional neural networks function and make decisions. We discovered different methods to improve the explainability and interpretability of these networks.

4 2-c: Domain Adaptation

4.1 Introduction

In domain adaptation, the aim is to adapt the models to suit both train and test domains by aligning feature distributions between the source data and the target data. To achieve this, we use an adversarial approach where we want the discriminator to be unable to discriminate between these two domains.[6]

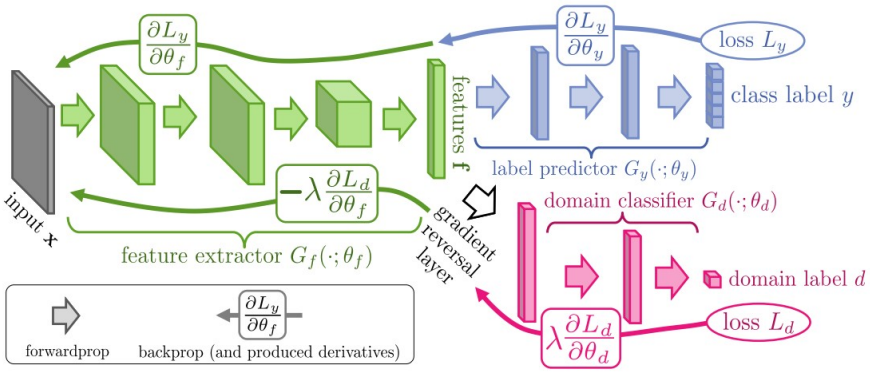


Figure 26: DANN Model Architecture. Source: [6]

In Figure 26 extracted from DANN's paper, we see the architecture of DANN. Green part is the feature extractor, blue part is the label predictor and pink part is the domain classifier. These two classifiers are adversaries, G_y back-propagates its gradient towards G_f with the intention of minimizing the loss between the predicted label and the real label. G_d on the other hand back-propagates the domain classification loss gradient, however, with GRL (Gradient Reversal Layer) between G_f and G_d, the gradient is reversed, causing G_f to concentrate on domain invariant features instead.[6]

So, the main goal is to minimize the performance gap between the source domain and the target domain.

In this section, we will investigate this domain adaptation method on MNIST data and MNIST-M data. We decided to respond to the questions at relevant times so they are scattered throughout the section.

4.2 Section 2 - Practice

4.2.1 Experiments and Results

To test this method, we need two datasets with different domains and a task to apply on both. For this, we used MNIST dataset and MNIST-M where the only difference between the two is that MNIST-M has colors and textures, making it RGB whilst MNIST is grayscale. We can see their differences in figures: Figure27 and Figure28.

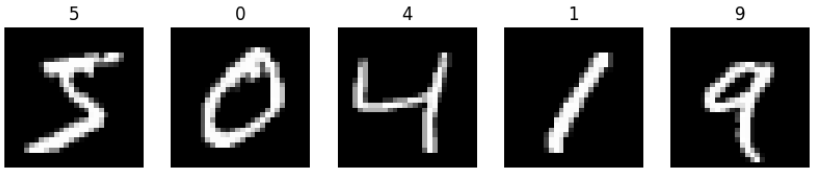


Figure 27: MNIST Examples

Figure 28: MNIST-M Examples

Implementing Naive Version To have a baseline result, we implemented a naive net which does not contain the adversarial architecture with domain classifier to see if a model trained only on the source data (MNIST) can perform well on target data (MNIST-M).

Dataset	Test Loss	Test Accuracy (%)
Source MNIST	0.02608	99.12
Source MNIST-M	1.27998	57.82

Table 4: Test results for MNIST and MNIST-M datasets.

In Table 4, we can see a massive drop between the accuracy on MNIST (source) and the accuracy on MNIST-M (target) without domain adaptation despite the similarity between two tasks and datasets.

Implementing DANN With a supplementary domain classifier and GRL, we can make the features as agnostic as possible from the domain. Both datasets contain digits with the only difference being their domain, hence, by ignoring domain specific features, we only keep the digit information.

It is important for GRL to apply a classic forward but multiplying the gradient with the negative number. For the model, we add the domain classifier and both during forward and backward, we calculate the result using GRL in between.

Question 1 : If you keep the network with the three parts (green, blue, pink) but didn't use the GRL, what would happen ? Let's go back to the DANN architecture in Figure 26. If we keep the network with the three parts but did not use GRL, the adaptation would not work. GRL is the Gradient Reversal Layer and is an essential part of the DANN architecture. GRL reverses the gradient calculated from the back-propagation in domain classifier G_d by a given factor. This causes feature extractor G_f to learn features that are domain invariant instead of the classic scenario where it would learn features indicating a domain.

Therefore, without GRL, G_f does not learn domain invariant features and the system has no adversarial effect. G_d is no longer penalized for not making the correct predictions because G_f learns in favor of both classifiers. As there is no domain confusion, G_d will probably not fail in the end and G_f will keep information on domain which will defeat the purpose of the architecture.

Directly reversing the gradient with a factor of -1 might cause unimpressive results because it does not apply any scaling and has no control over the magnitude. It is not our intention to completely blindside the domain classifier; rather, we aim to assimilate the target data toward the source data so we will start with a small factor and gradually increase it (shown in Figure 29) as instructed in the Notebook.

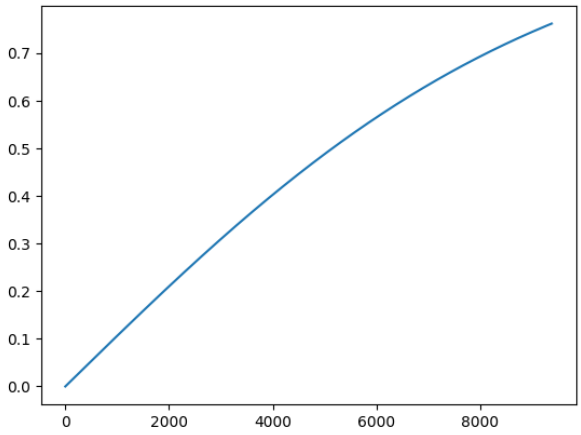


Figure 29: Gradual Factor Increase

Let's train our DANN using both out datasets. We will use Cross Entropy Loss for label classifier and Binary Cross Entropy Loss for domain classifier. We do not have access to the labels of target data MNIST-M during training, so, they do not contribute to G_y's loss but G_y is still applied to them. For G_d, both datasets contribute to the alignment of domain distributions.

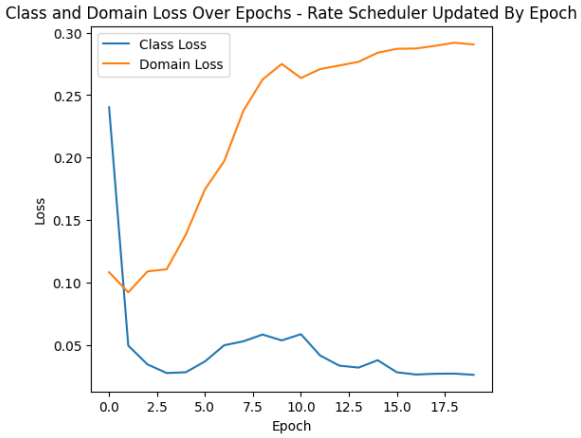


Figure 30: Loss results of our DANN model (20 epochs)

Domain	**Class Loss**	**Class Accuracy (%)**	**Domain Loss**	**Domain Accuracy (%)**
Source MNIST	0.03975	98.66	0.48861	82.47
Target MNIST-M	1.07824	74.81	0.65926	55.77

Table 5: Results of the DANN experiment for source and target domains.

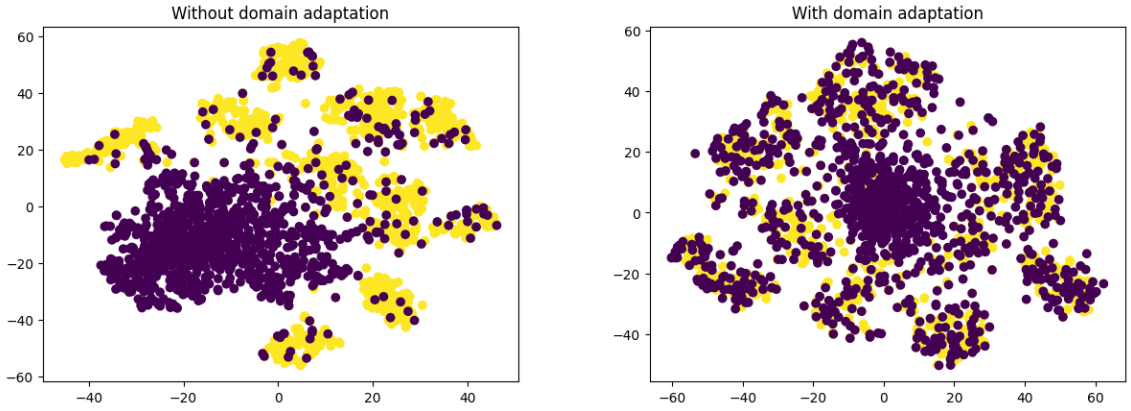
The results of the experiment are presented in Figure 30 and Table 5 above.

As seen on the table, the target set accuracy increased to 74% and the source classification performance dropped slightly to 98.66% compared to the results we had without domain adaptation in Table 4, 99% and 57% respectively. Domain classification on the other hand performs better with the source data (MNIST) than with the target data (MNIST-M). This shows us that the domains are yet to be ambiguous enough to totally confuse G_d but it still improved the performance on the target dataset by 20%.

Question 2 We observe a slight drop in classification task on the source data and we can see it clearly in Figure 30 where while the class loss decreases gradually, the domain loss increases.

Performance on the source data may degrade a bit because we are un-adapting the features extracted from the source data. As the feature extractor creates domain invariant features, we lose information on the characteristics of the source data. This loss leads to a small performance drop and is a part of the trade-off mechanism to improve generalization.

Embedding Visualization Let's visualize how the domains became ambiguous by extracting embeddings despite the drawbacks of TSNE to have a concrete idea of what happens with domain adaptation.



(a) Embedding Visualization Without Domain Adaptation (20 epochs)

(b) Embedding Visualization With Domain Adaptation

Figure 31: Embedding Visualization Comparison Based On Domain Adaptation (20 epochs)

In Figure 31, we can see how the embedding space changed after domain adaptation. Yellow dots are the source (MNIST) dataset and the purple dots are the target dataset (MNIST-M). In Figure 31a, the scattered points are not significantly distanced because both datasets contain digits. However, because their domains are different, their embeddings are different to it is hard to generalize them without domain adaptation. The next Figure 31b demonstrates how two domains became ambiguous after 20 epochs of training with DANN. They are still not perfectly domain-agnostic, as our results also revealed but we can see the improvement, the purple and yellow dots overlap more.

Improvement 1 : Increasing the number of epochs To improve the performance of our model, we can try to increase the number of epochs to see if training more helps with generalization. In Table 6 below, we see performance comparison between different number of epochs. We do not observe a big change in performance, we can see that the performance on source domain classification deteriorated significantly but it also deteriorated for target domain and now performs like random classification. However, this did not necessarily improve the performance of G_y for target label classification, 50 epochs of training showed best results while with 100 epochs, it dropped slightly. So, the number of epochs might improve performance but we can not be sure of its stability. Let's look at how the losses varied through epochs and what is the final embedding visualization looks like to understand what happened better.

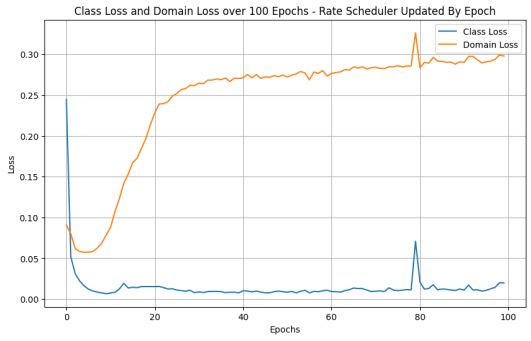
Epochs	Src G_y Loss/Acc	Src G_d Loss/Acc	Tgt G_y Loss/Acc	Tgt G_d Loss/Acc
20	0.03975 / 98.66%	0.48861 / 82.47%	1.07824 / 74.81%	0.65926 / 55.77%
50	0.05759 / 98.5%	0.5093 / 80.25%	1.21943 / 78.17%	0.68328 / 53.02%
100	0.077 / 98.44%	0.5795 / 73.85%	1.5318 / 77.79%	0.68239 / 51.65%

Table 6: Loss and accuracy results for source and target domains across epochs.

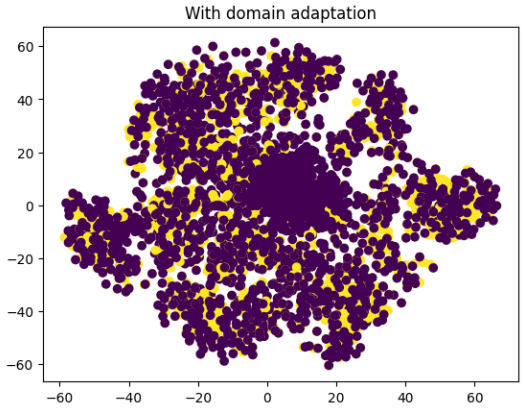
In Figure 32a, we see how the loss ranged in 100 epochs; at some point, after approximately 30 epochs, it became steady (except the sudden peak in both losses towards 80 epochs). This shows us how increasing the number of epochs does not necessarily improve the performance beyond a certain point and the adversary system reaches a stable state with slight changes.

Also, in Figure 32, we see that the embedding visualization became much more ambiguous and it is harder to discriminate with naked eye between two classes -we still have to consider the drawbacks of this embedding method-.

From these observations, we can speculate that another parameter or hyperparameter has more impact on the performance rather than the number of epochs.



(a) Loss results of our DANN model (100 epochs)



(b) Embedding Visualization With Domain Adaptation (100 epochs)

Figure 32: Embedding Visualization Comparison Based On Domain Adaptation (100 epochs)

Improvement 2 : Modifying the learning rate scheduler to be updated per step To investigate the cause of this performance stability, we can try to update learning rate scheduler per step instead of per epoch. This would mean that the scheduler gets updated much frequently and can lead to faster convergence. The results in Table 7 and in Figure 33 show exhibit a slight improvement in target class (MNIST-M) accuracy from 74.81% to 77.32% and a slight improvement in domain alignment (rows of Source Domain and Target Domain). The change is not drastic, this might be due to other parameters like α and β or the learning rate scheduler itself. It is a meticulous job to find the optimal choice of parameters.

Table 7: Comparison of 20 Epoch Results (Update Per Epoch vs Update Per Step)

Metric	Update Per Epoch	Update Per Step
Source Class Loss/Accuracy	0.03975/ 98.66%	0.03993/98.66%
Source Domain Loss/Accuracy	0.48861/82.47%	0.60412/73.67%
Target Class Loss/Accuracy	1.07824/74.81%	0.85065/77.32%
Target Domain Loss/Accuracy	0.65926/55.77%	0.62128/58.84%

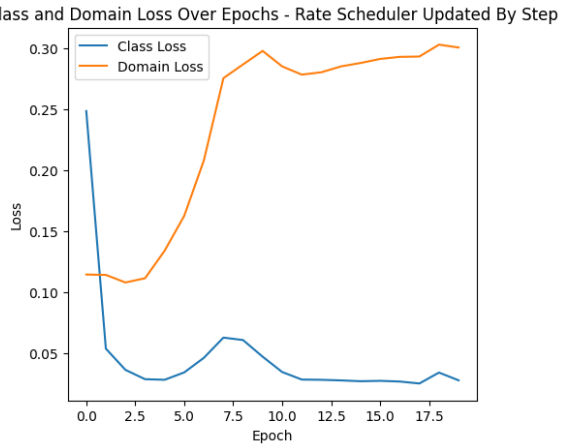


Figure 33: Loss results of our DANN model (20 epochs + Update Per Step)

Improvement 3 : Testing Different Standardizations Previously, we standardized the datasets based on the mean and std values calculated from the MNIST dataset. This was logical because it helped reducing differences between the source and target distributions. Normalization is an important step in NNs because they prevent large gradient updates and depending on two different gradient updates, it is important for DANN.

We will try a few different standardizations (independent from both data distributions) to see how different normalizations effect the performance.

MNIST-based Normalization transforms both datasets based on Normalization with the mean and std values of MNIST dataset, so it depends on the source data.

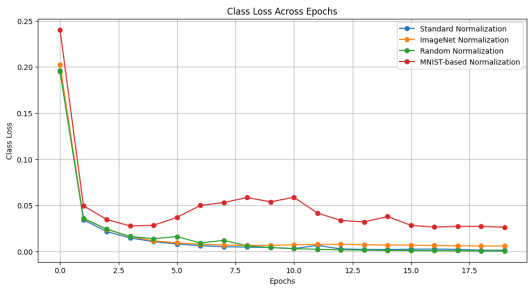
Standard Normalization performs simple $(0.5, 0.5)$ normalization.

ImageNet Normalization is similar to MNIST-based Normalization but here, we extract the mean and std values from the ImageNet dataset.

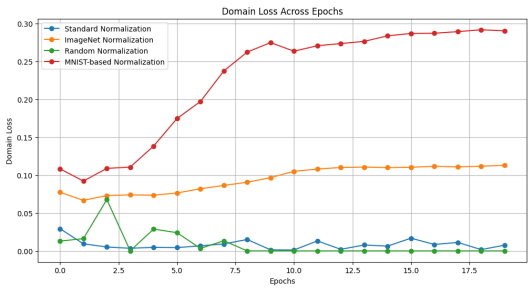
Random Normalization has random mean and std values and is not meant to make sense.

In the two plots in Figure 34, we see how different standardizations impacted training performance over 20 epochs and in Table 8, we can see the final results. Looking at loss graphs, MNIST normalization seems to perform poorly next to others, however we see in Figure 34b that it was more efficient when aligning the domains. In the table where we have test results, we see that it performed the best target label classification thanks to its ability to adapt domains and make their features more ambiguous.

An interesting observation that we can make is how ImageNet Normalization increased the target domain accuracy significantly, passing the source domain accuracy. It reversed the domain adaptation system towards the target domain and failed to give domain-agnostic results. We can assume that this happened because ImageNet dataset contains colorful images, which is more similar to MNIST-M dataset than grayscale MNIST dataset. This shows us how the dataset we use to standardize impacts the domain alignment process.



(a) Class Loss over Epochs for Different Standardizations



(b) Domain Loss over Epochs for Different Standardizations

Figure 34: Embedding Visualization Comparison Based On Domain Adaptation (100 epochs)

Table 8: Performance Metrics for Different Normalization Methods

Normalizaion	**Dataset**	**Class Loss**	**Class Accuracy**	**Domain Loss**	**Domain Accuracy**
Standard	Source	0.04091	98.92%	0.02559	99.16%
	Target	1.30212	61.70%	0.02068	99.37%
ImageNet	Source	0.04978	98.48%	0.71588	59.08%
	Target	1.52530	67.51%	0.34054	84.10%
Random	Source	0.03514	99.12%	1e-05	100.00%
	Target	5.62319	26.72%	0.0	100.00%
MNIST-based	Source	0.03975	98.66%	0.48861	82.47%
	Target	1.07824	**74.81**%	0.65926	55.77%

Improvement 4 : Different GRL Factor Scheduling Previously, we used exponential factor scheduling. This time we used linear and cosine versions. 1. Exponential :

$$\lambda : -1 + \frac{2}{1 + \exp\left(\frac{-2e}{\text{nb\_iters}}\right)}$$

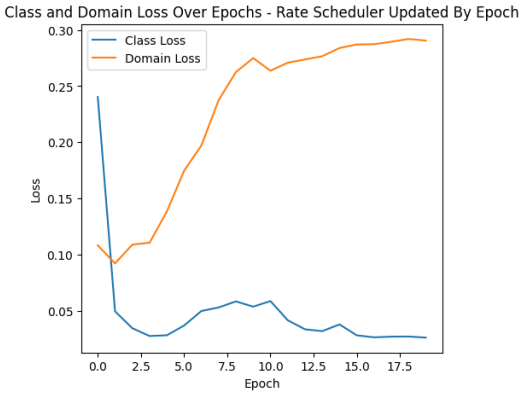
2. Linear :

$$\lambda : 1.0 - (1.0 - 0.0) \cdot \frac{e}{\text{epochs}}$$

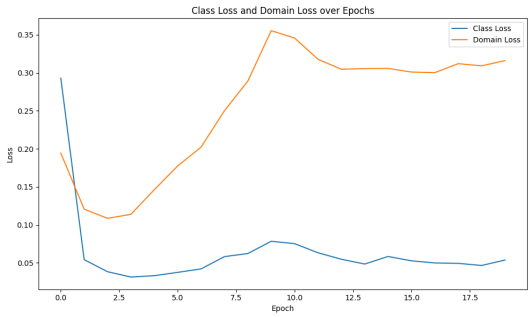
3. Cosine :

$$\lambda : 0.0 + (1.0 - 0.0) \cdot \frac{1 + \cos\left(\frac{\pi e}{\text{epochs}}\right)}{2}$$

Firstly, linear decay failed after a few epochs, there are multiple possible causes for this instability, it increases at a constant rate without smooth increase and it might cause aggressive gradient reversal. So we will not be using it in our comparison but it shows us the importance of factor scheduling.



(a) Loss results of our DANN model with Exponential Factor Scheduling (previous result)



(b) Loss results of our DANN model with Cosine Factor Scheduling

Figure 35: Loss Comparison Between Two Factor Schedulers

We do not observe ant significant difference between the behaviors of the losses in Figures 35a and 35b. They both seem to work well with the DANN model.

Decay Type	**Src G_y Loss/Acc**	**Src G_d Loss/Acc**	**Tgt G_y Loss/Acc**	**Tgt G_d Loss/Acc**
Exponential	0.03975 / 98.66%	0.48861 / 82.47%	1.07824 / **74.81%**	0.65926 / 55.77%
Cosine	0.05711 / 98.21%	0.72944 / 45.73%	1.33143 / 68.36%	0.54946 / 78.18%

Table 9: Comparison of Class and Domain Losses with Accuracy for Exponential and Cosine Decay

In Table 9, we can see that our initial exponential factor scheduler performed better than the cosine scheduler for target label classification task with 74.81% over 68.36%. We can see that cosine decay increased the target domain classification significantly, once again, reversing the domain adaptation and keeping features that the discriminator G_d can recognize.

Question 3 The value of the negative number (λ) influences the degree of gradient reversal, instead of features that help the domain classifier separate two domains, it learns those who are similar in both domains. Larger λ will have a stronger effect, increasing the domain confusion. However, if it is too strong, it can only concentrate on domain invariant features and can lose information on class-specific features, decreasing the classification performance. And just like the learning rate, extreme values can cause unstable gradients, leading to divergence or insufficient convergence. This is why, during this lab, we gradually grew λ.

To see how this negative factor impacts the performance, we multiplied the gradually increasing factor by 2,3,4 and 5. These will result in harsher jumps between the factors.

For 4 and 5, the jumps were too unstable and the model stopped working after a few epochs. For 1 (that does not modify the original gradual increase), 2 and 3, we can see the results in Figure 36 and in Table 10. In the table, we can see that Multiplier=1 gave the best results for target classification and as Multiplier increased, the performance worsened both for label classification and domain alignment as the domain accuracy for the target domain increased and the domain accuracy for the source domain decreased. When the factor is too high, instead of generalizing and making the domain gap reduce, we increase it by favoring the target class. These results show us the importance of λ and its impact on the performance.

In the figures showing us how the loss evolved during training in Figure 36, multipliers 2 and 3 seem to perform better than 1 but in concrete tests after training, we can assume their performance were based on overfitting and instability.

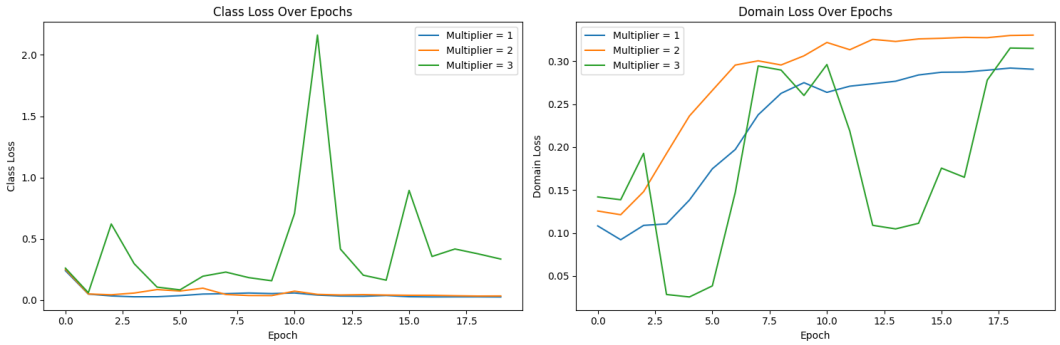


Figure 36: Loss Over Epochs for Different λ Multipliers

Multiplier	**Src G_y Loss/Acc**	**Tgt G_y Loss/Acc**	**Src G_d Loss/Acc**	**Tgt G_d Loss/Acc**
1	0.03975 / 98.66%	1.07824 / **74.81**%	0.48861 / 82.47%	0.65926 / 55.77%
2	0.05595 / 98.18%	1.47031 / 70.81%	0.67737 / 66.48%	0.66093 / 56.91%
3	0.34231 / 90.77%	2.93005 / 49.28%	0.59986 / 63.25%	0.65647 / 71.81%

Table 10: Class and Domain Losses/Accuracy for Source and Target Domains by Multiplier

Question 4 Another common method in domain adaptation is pseudo-labeling [11].

Investigate what it is and describe it in your own words.

During domain adaptation, we do not have the labels of target data and try to align domains to mitigate differences between two data. Pseudo-labeling aims to use the fact that even if the target labels are unknown, because the tasks are the same, they have to belong to the same label space as the source. In order to initialize these pseudo labels, we start training the model using source labels and then, we introduce pseudo-label generation. To generate them, we use the trained classifier to predict labels for the target data as if we are testing unknown data. Then, we mix these pseudo-labels with already existing source labels and retrain the model with this augmented dataset. These steps tend to improve generalization because the classifier model is now working with more diverse data. However, it can also boost the bias of the model or overfit with inaccurate pseudo labels. To overcome these problems, we have to be selective with the pseudo label selection by only choosing high confidence predictions and retraining the model multiple times to improve pseudo label accuracy.

4.3 Conclusion

In this lab, we worked on domain adaptation networks where we use an adversarial methodology to adapt the domains of two different datasets where we have a similar task but different domains in order to classify the target dataset without needing target labels.

We experimented on different parameters and methodologies to improve the model's performance.

5 2-de: Generative Adversarial Networks

5.1 Introduction

So far, we have explored discriminative approaches. In this lab, we discover generative models, that given training data, generate new samples from the same distribution. We look at two types of GANs:

the "classic" unconditional GAN and the conditional cGAN which generates images conditioned on the input you give to it.

5.2 Section 1 - Generative Adversarial Networks

$$\max_G \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})} [\log D(G(\mathbf{z}))] \tag{6}$$

$$\max_D \mathbb{E}_{\mathbf{x}^*\in \mathcal{D}\text{ata}} [\log D(\mathbf{x}^*)] + \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))] \tag{7}$$

Figure 37: Equations of GAN

Question 1. In equation (6), the generator wants to fool the discriminator into thinking that the generated images G(z) are real, by trying to make D(G(z)) close to 1.

In equation (7), the discriminator wants to maximize objective such that D(x) is close to 1, so the discriminator correctly identifies the real images, and D(G(z)) is close to 0, so the discriminator is able to distinguish fake images.

If we only used equation (6), the generator wouldn't receive feedback from the discriminator to improve its generation of real-looking images, so it wouldn't generate images that are indistinguishable from real images.

If we only used equation (7), the discriminator would learn to perfectly identify real and fake images but the generator wouldn't be learning to generate real-looking images.

Question 2. The generator should transform the random noise distribution P(z) into P(Data), the distribution of the real images. So, the generator learns to generate images that are indistinguishable from the real images.

Question 3. The true equation deriving from equation (5) is: $\min_G \mathbb{E}_{\mathbf{z}\sim P(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$

The generator wants to minimize objective such that D(G($\mathbf{z}$)) is close to 1.

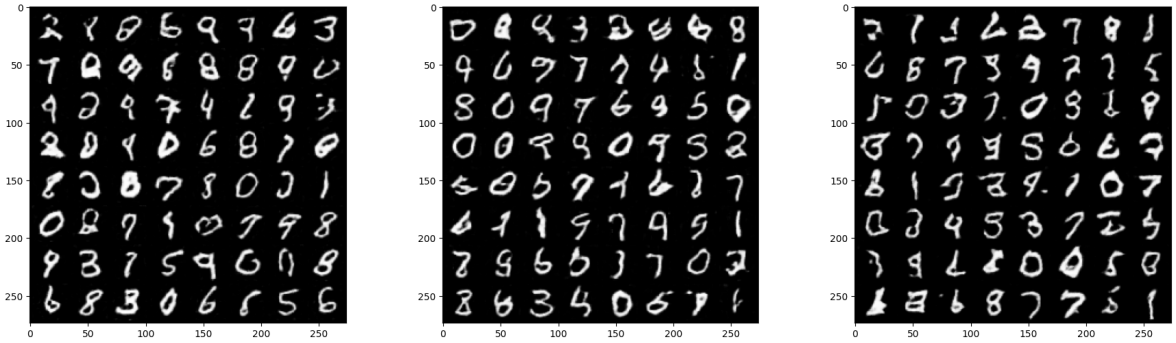
Question 4. With the default settings, the generated images start noisy. In the earlier iterations, the generator's loss is higher than the discriminator and both losses are noisy and fluctuate. As we progress in iterations, the losses become lower but still unstable. Towards the later iterations, the images look more realistic but not very diverse, most of the digits are 9, 0 or 3, which suggests possible mode collapse, and some digits still look 'fake-ish' indicating that the generator has not yet learned to generate good quality images across all classes.

Question 5. We tried different experiments:

Modify ngf: We increased ngf from 32 to 64. ngf corresponds to the size of feature maps in the generator.
We observed a larger gap between the discriminator (d) and generator (g) losses during the early iterations compared to the default setting. This suggests that the generator initially struggled more to compete with the discriminator due to the increased complexity of its architecture. However, as training progressed, the gap between the losses became smaller than in the default setting. The generated digits were not significantly different from the default setting. Despite the increased size

of the feature maps in the generator, the outputs did not show notable improvements in diversity or realism.

Modify nz: We decreased nz from 100 to 10, then increased to 1000. nz is the size of latent vector z. When we set nz to 10, the generator loss (g_loss) became noisier, likely due to the generator struggling with the small latent space. The discriminator (d) and generator (g) losses converged closer to each other earlier in training compared to the default setting. The generated images lacked diversity. Increasing nz resulted in more diverse images. But it also prolonged the runtime. The losses became noisier and the gap between the generator and discriminator losses increased. This suggests that the optimization process became more challenging.

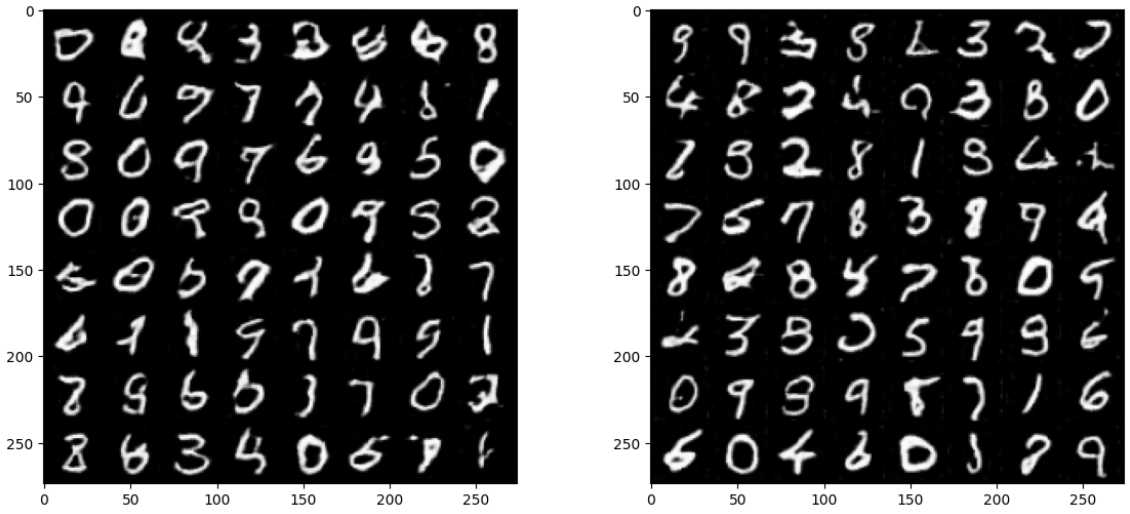


(a) nz=10 (b) nz=100 (c) nz=1000

Figure 38: Generated images with nz values

Learn for longer: We increased the number of epochs from 5 to 30. In figure 39, for the images generated in 30 epochs, we can observe gray dots appearing around the corners of some of the images when zoomed in. This may suggest the presence of noise. With many epochs, the model might be overfitting to noise in the data.

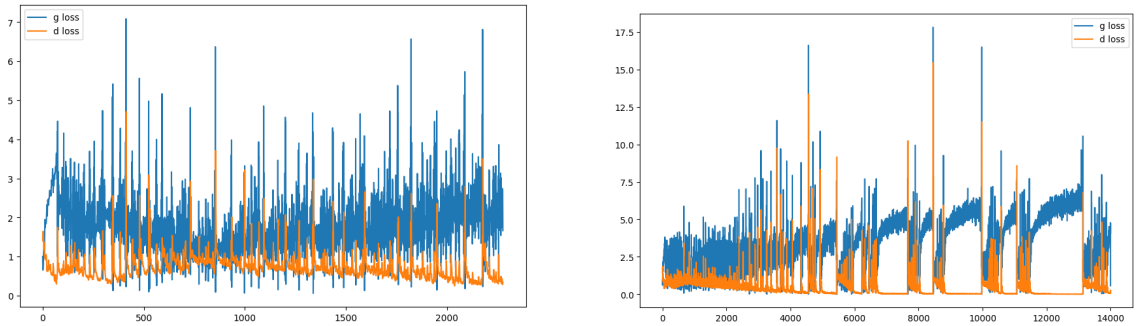
When we look at the loss functions in figure 40, for 5 epochs we observe high variability for the generator loss and see spikes for both losses. Discriminator's loss is generally lower than the generator's. Over time, both losses seem to stabilize.
For 30 epochs, the discriminator loss shows a pattern of high spikes followed by stable periods, while the generator loss shows upward patterns. The losses aren't stable.



(a) 5 epochs

(b) 30 epochs

Figure 39: Generated images with different epochs



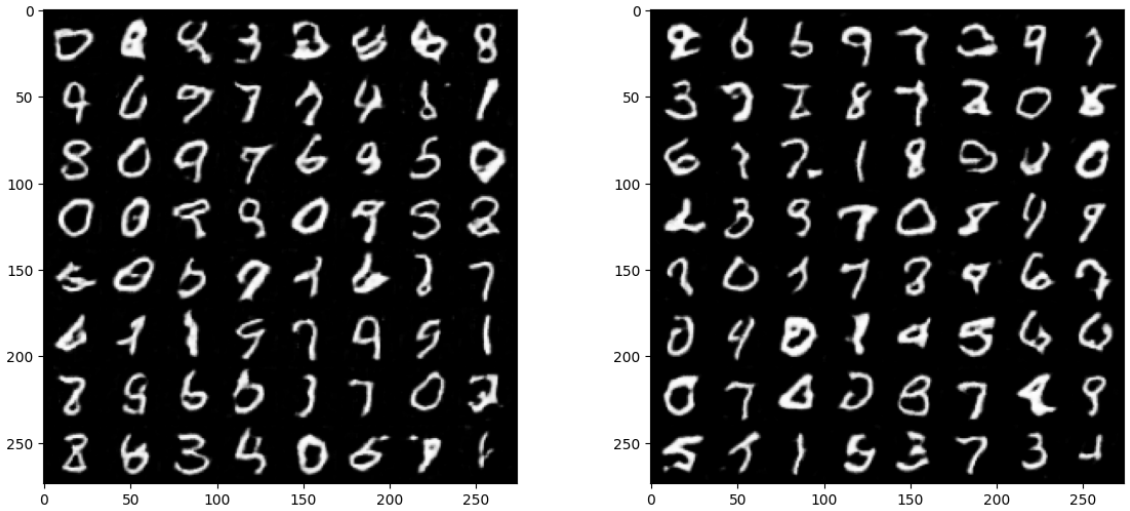
(a) Losses for 5 epochs

(b) Losses 30 epochs

Figure 40: Loss curves with different epochs

Replace the custom weight initialization with pytorch's default initialization:

When we used the default weight initialization instead of the custom function, during training in general the discriminator's loss was lower than in the custom weight setting. In figure 41, the digits generated in the custom weight setting seem more real.

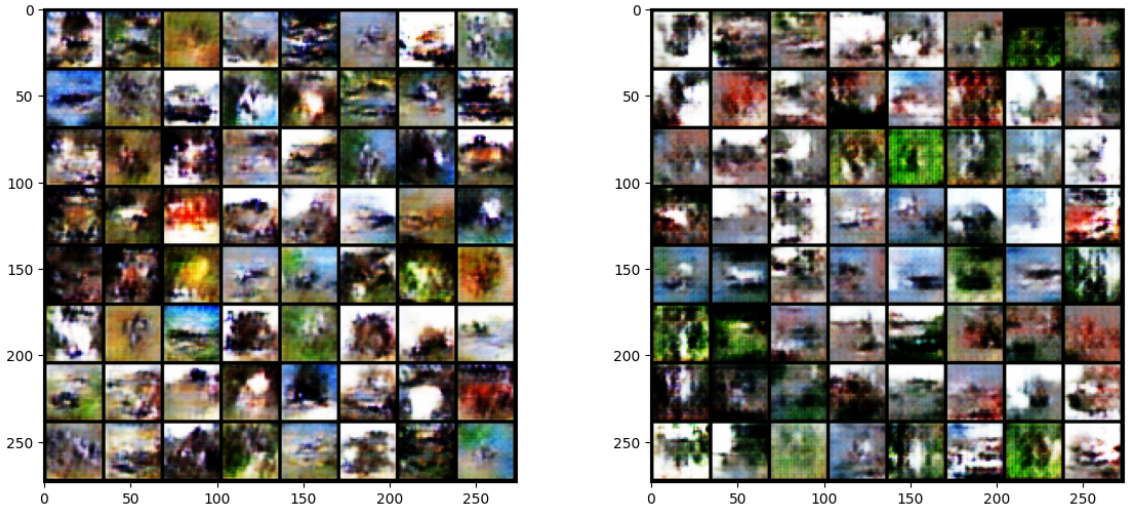


(a) Custom weight initialization

(b) Default weight initialization

Figure 41: Generated images with and without the custom weight initialization

Try on CIFAR-10 dataset: We tried our GAN on the CIFAR-10 dataset. The losses were higher compared to the ones for MNIST digits. In figure 42, on the left we see the images generated in 5 epochs. The generated images are blurry and difficult to interpret, but they look like images that might come from natural scenes, like the ones of cifar-10. 5 epochs were probably not enough to learn. After 10 epochs, the generated images show slight improvement in quality, though they still do not appear realistic. 5 or 10 epochs are probably insufficient for the GAN to learn meaningful features from such a complex dataset.



(a) 5 epochs

(b) 10 epochs

Figure 42: Generated CIFAR-10 images

5.3 Section 2 - Conditional Generative Adversarial Networks

Question 6. The digits generated in figure 43 by the cGAN with the default settings and 5 epochs are more diverse and of better quality than the images generated by the classic GAN.

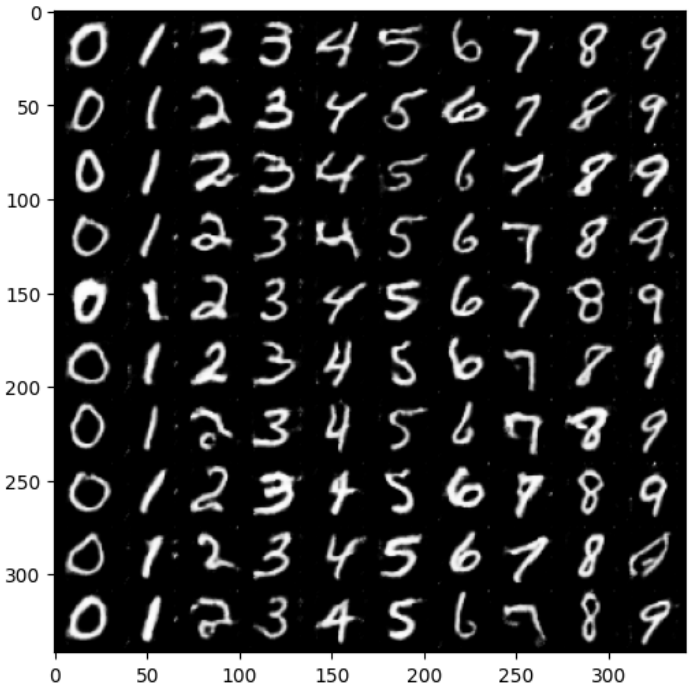
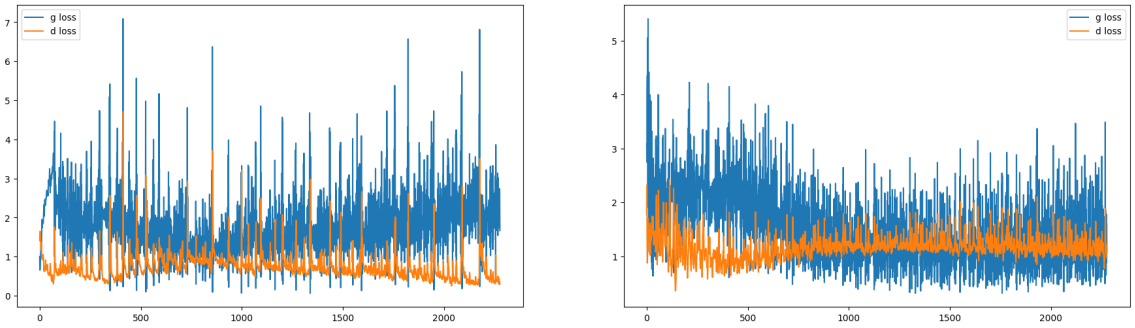


Figure 43: Generated images by cGAN

When using conditional GAN, in figure 44 we observe that the generator loss generally lower than the discriminator loss. In both GANs the losses oscillate. There seems to be more stability in the loss curves in the case of conditional GAN.

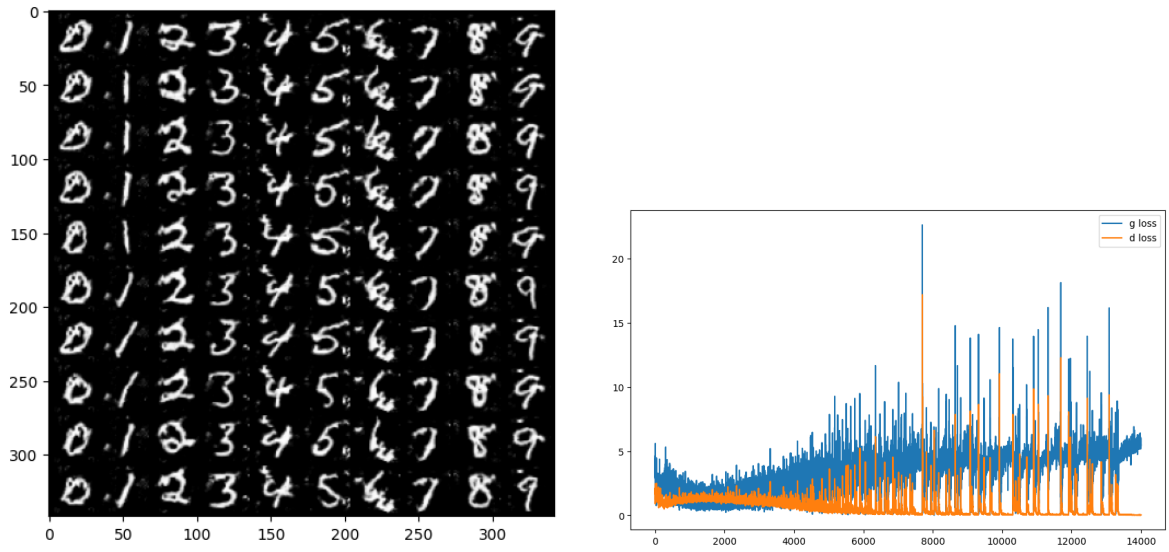


(a) Losses for classic GAN

(b) Losses for conditional GAN

Figure 44: Loss curves

We also tried training the conditional GAN for 30 epochs, like we did for classic GAN before. Training the conditional GAN for 30 epochs took a lot longer than training the classic GAN for 30 epochs. In figure 45 image (a), the generated digits are very noisy, their quality is significantly lower than the digits generated after 5 epochs. This indicates that the extended training doesn't necessarily lead to better images.
Looking at the loss curves in figure 45 (b), we notice that the gap between the generator and discriminator losses increases over time. Additionally, the losses are more unstable compared to the curves obtained when trained for 5 epochs.



(a) Generated images (b) Losses

Figure 45: Training cGAN for 30 epochs

Question 7. If we were to remove the input **y** from the discriminator, the generator could ignore the label and wouldn't learn to generate real-looking images of that class. With the presence of the label, the discriminator learns what a certain class of image should look like, and forces the generator to generate according to the label. [3] Without giving y to the discriminator, we would defy the difference in the purpose of the cGAN from the classic GAN.

In figure 46, we observe the lack of diversity and the mode collapse in the generated digits as a result of removing the input y.

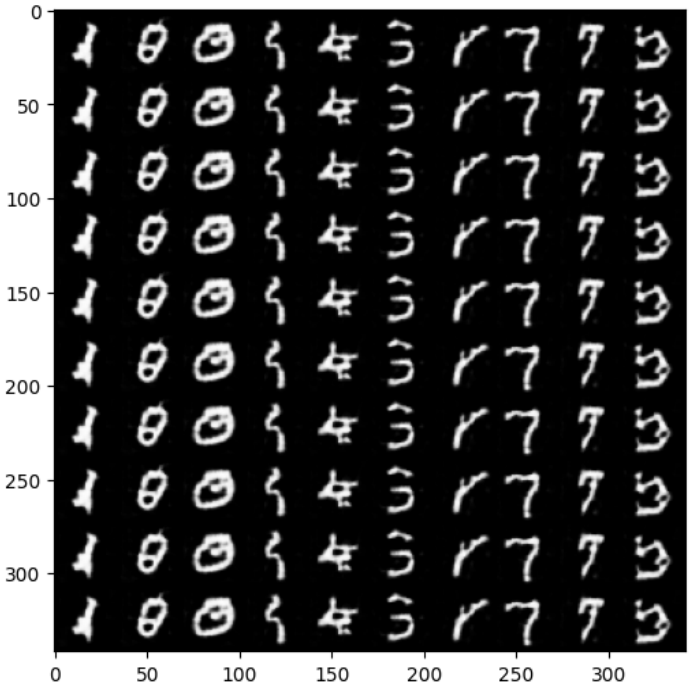


Figure 46: Generated images by cGAN without giving y input to the discriminator

Question 8. The training of the cGAN was more successful compared to the unconditional GAN. By leveraging class labels as additional input, the cGAN was able to guide the generation process, resulting in higher-quality images.

Question 9. In conditional GAN, the generator learns a deterministic mapping from input to output distributions (G : X×Z → Y). [20] Because of this deterministic property of the generator, in figure 47 the images in the same column (that have the same noise vector z) have similar properties. Images from different vectors z have slight differences. So with small differences in the noise vector, we can get different samples. [14]

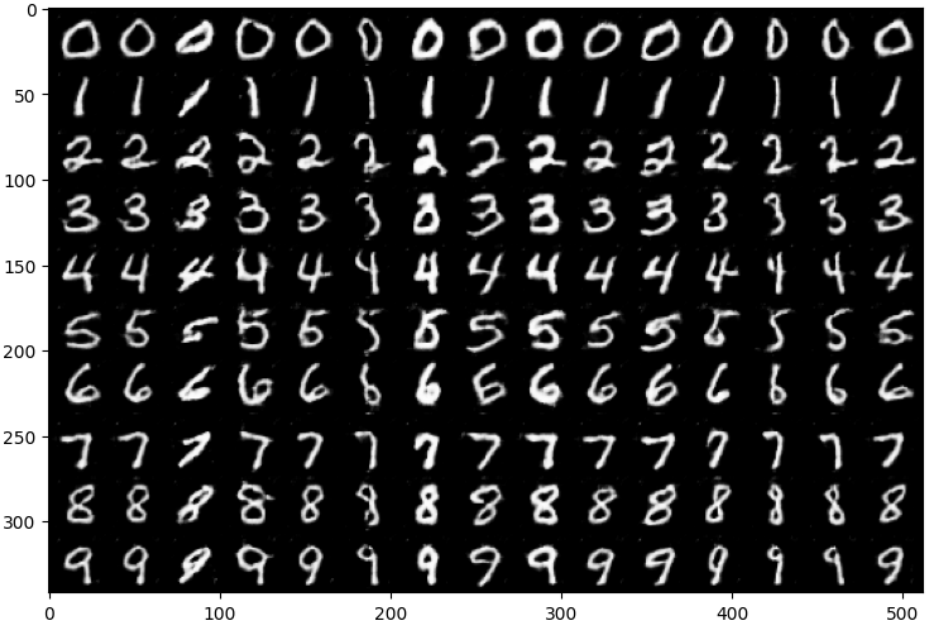


Figure 47: Impact of z on generation. All digits in the same column have the same noise vector z.

5.4 Conclusion

We explored the implementation of GANs and conditional GANs on MNIST. In addition, we investigated the effects of various hyperparameters on training.

6 Conclusion

In this homework, we experimented on different deep learning applications. With *Transfer Learning*, we leveraged pre-trained models to improve performance on similar datasets, we explored the interpretability of NNs with *Visualization*, applied the robust generalization strategy DANN for *Domain Adaptation* and applied data creation method *Generative Adversarial Networks*. We studied their theory, recreated their architectures, trained them and applied them on real data.

These lab works revealed the importance of theoretical knowledge, the model architecture, the depth, the parameters and the training to achieve ideal performances.

References

[1] What is transfer learning? | IBM.

[2] Adil Kaan Akan, Mehmet Ali Genc, and Fatos T. Yarman Vural. Just noticeable difference for machines to generate adversarial images, 2020.

[3] Alberto. In the conditional gan (cgan) architecture, why does the discriminator need conditional variable? https://ai.stackexchange.com/questions/42758/in-the-conditional-gan-cgan-architecture-why-does-the-discriminator-need-cond, 2024. Accessed: 2024-12-19.

[4] Amirhosein Chahe, Chenan Wang, Abhishek Jeyapratap, Kaidi Xu, and Lifeng Zhou. Dynamic adversarial attacks on autonomous driving systems, 2024.

[5] Massed Compute. What are the limitations of using saliency maps for model interpretability? https://massedcompute.com/faq-answers/?question=What+are+the+limitations+of+using+saliency+maps+for+model+interpretability%3F, 2024. Accessed: 2024-12-20.

[6] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition.

[8] Huaizu Jiang, Zejian Yuan, Ming-Ming Cheng, Yihong Gong, Nanning Zheng, and Jingdong Wang. Salient object detection: A discriminative regional feature integration approach. *International Journal of Computer Vision*, 123:251 – 268, 2013.

[9] Beomsu Kim, Junghoon Seo, SeungHyun Jeon, Jamyoung Koo, Jeongyeol Choe, and Taegyun Jeon. Why are saliency maps noisy? cause of and solution to noisy saliency maps, 2019.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

[11] Dong-Hyun Lee. Pseudo-label : The simple and efficient semi-supervised learning method for deep neural networks.

[12] Jacob Pfau, Albert T. Young, Maria L. Wei, and Michael J. Keiser. Global saliency: Aggregating saliency maps to assess dataset artefact bias, 2019.

[13] Omid Poursaeed, Isay Katsman, Bicheng Gao, and Serge J. Belongie. Generative adversarial perturbations. *CoRR*, abs/1712.02328, 2017.

[14] sfotiadis. Why different noise in gan generate different images? https://ai.stackexchange.com/questions/45098/why-different-noise-in-gan-generate-different-images, 2024. Accessed: 2024-12-19.

[15] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1528–1540, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.

[17] Mark Stone. Why adversarial examples are such a dangerous threat to deep learning. https://securityintelligence.com/articles/why-adversarial-examples-are-such-a-dangerous-threat-to-deep-learning/, 2024. Accessed: 2024-12-21.

[18] Anna Stubbin, Thompson Chyrikov, Jim Zhao, and Christina Chajo. The limits of perception: Analyzing inconsistencies in saliency maps in xai, 2024.

[19] Min-Jen Tsai, Ping-Yi Lin, and Ming-En Lee. Adversarial attacks on medical image classification. *Cancers*, 15:4228, 08 2023.

[20] Dingdong Yang, Seunghoon Hong, Y. Jang, Tianchen Zhao, and Honglak Lee. Diversity-sensitive conditional generative adversarial networks. *ArXiv*, abs/1901.09024, 2019.