Faculty of Science and Engineering - Sorbonne Université

Master's in Computer Science - DAC / IMA

RDFIA - Reconnaissance Des Formes pour l'Analyse et l'Interprétation d'Images

Practical Works Report

Deep learning applications

Authored by:
Faten Racha SAID - M2 DAC
Mounir MAOUCHE - M2 IMA

Supervised by:
Nicolas Thome
Matthieu CORD

December 2025

Contents

List of Figures

Chapter 1

Transfer Learning

Transfer learning is a machine learning technique where a model developed for one task is reused for a related task. It is especially useful in deep learning, as it reduces the computational cost of training large networks from scratch by leveraging prior knowledge.

The main objective of this session is to familiarize with the well-known deep CNN network VGG16, introduced in 2015 by Simonyan and Zisserman. We will focus on understanding its architecture and functionality.

The session will also explore practical applications of transfer learning, particularly methods to extract key features from pre-trained networks. Finally, these features will be applied in a practical scenario using SVM classification, demonstrating how deep features from networks like VGG16 can be used effectively for classification tasks.

Q1. Knowing that the fully-connected layers account for the majority of the parameters in a model, give an estimation on the number of parameters of VGG16 (using the sizes given in Figure1)

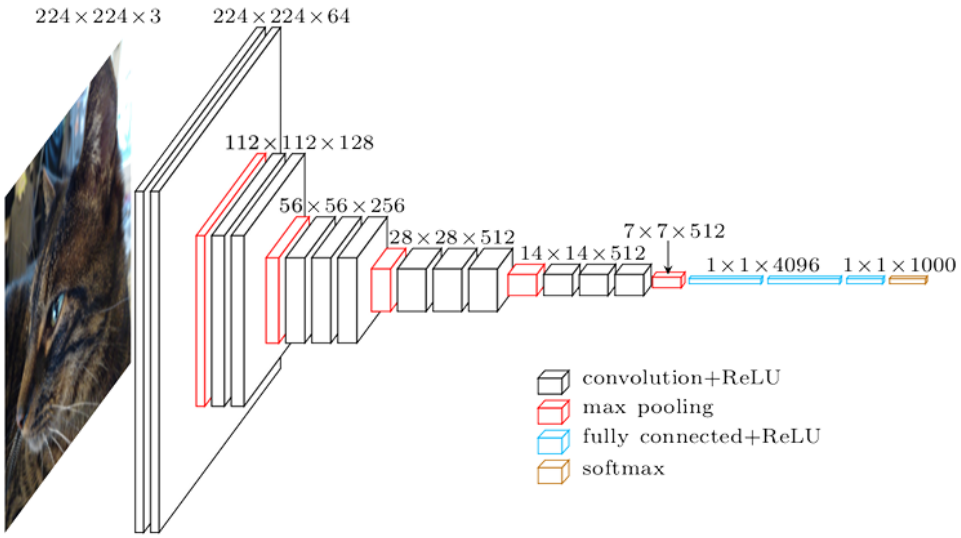

Figure 1.1: VGG16 architecture

Convolutions

The number of parameters of each convolutional leayer is given by:

Number of parameters = Number of kernels×(height of the kernel×weight of the kernel×number of chan

Where each kernel has a bias. Here's the calculation for each convolutional bloc:

- **Bloc Conv1** : 64 kernels, each of size 3×3 with 3 input channels:

$$64 \times (3 \times 3 \times 3 + 1) = 1\,792 \text{ parameters.}$$

- **Bloc Conv2** : 128 kernels, each of size 3×3 with 64 input channels :

$$128 \times (3 \times 3 \times 64 + 1) = 73\,856 \text{ parameters.}$$

- **Bloc Conv3** : 256 kernels, each of size 3×3 with 128 input channels:

$$256 \times (3 \times 3 \times 128 + 1) = 295\,168 \text{ parameters.}$$

- **Bloc Conv4** : 512 kernels, each of size 3×3 with 256 input channels :

$$512 \times (3 \times 3 \times 256 + 1) = 1\,180\,160 \text{ parameters.}$$

- **Bloc Conv5** : 512 kernels, each of size 3×3 with 512 input channels :

$$512 \times (3 \times 3 \times 512 + 1) = 2\,359\,808 \text{ parameters.}$$

Fully-connected layers (FC)

The number of parameters for the fully connected layers is given by :

$$\text{Number of parameters} = \text{Number of output neurones} \times (\text{Number of input neurones} + 1)$$

- **FC1** : 4096 ouput neurones, each connected to $7 \times 7 \times 512$ input neurones :

$$4096 \times (7 \times 7 \times 512 + 1) = 102\,764\,544 \text{ parameters.}$$

- **FC2** : 1000 ouput neurones, each connected to 4096 input neurones :

$$1000 \times (4096 + 1) = 4\,097\,000 \text{ parameters.}$$

Total

The total number of parameters is the sum of the parameters of the convolutions and FC layers:

$$\text{Total} = 1\,792 + 73\,856 + 295\,168 + 1\,180\,160 + 2\,359\,808 + 102\,764\,544 + 4\,097\,000 = 110\,765\,760$$

Thus, VGG16 contains approximatively **110,8 millions of parameters.**

Q2. What is the output size of the last layer of VGG16? What does it correspond to?

The output size of the last layer of VGG16 is $\mathbf{1 \times 1000}$.

The last layer of VGG16 is a fully connected layer with 1000 neurons. Each neuron corresponds to one of the 1000 classes in the ImageNet dataset.

Q3. Apply the network on several images of your choice and comment on the results.

- **What is the role of the ImageNet normalization?** The role of ImageNet normalization is to ensure consistency with the distribution of the training images used for VGG, maintaining compatibility and improving model performance.
- **Why setting the model to eval mode?** During inference we need to put eval mode because VGG has a dropout layer so otherwise we would lose the training.

  The model is set to eval mode during inference to disable dropout and freeze the model in its final training state.

Q4. Bonus : Visualize several activation maps obtained after the first convolutional layer. How can we interpret them?

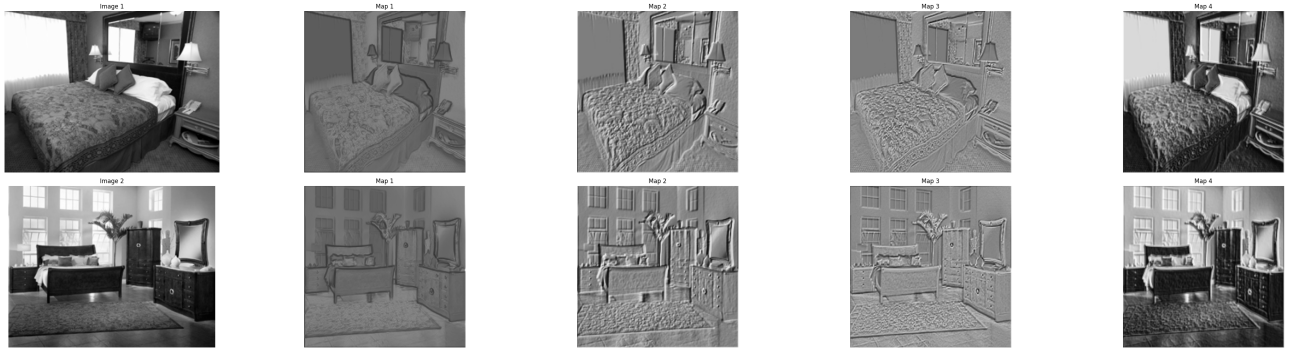

Figure 1.2: Visualisation of activation maps at the first layer

The activation maps obtained after the first convolutional layer can be interpreted as feature detectors that highlight different low-level image characteristics, such as edge detection, which is likely represented by the first feature map, texture Patterns, as other maps capture textures or repetitive patterns present in the image, such as the fabric of the bed or the structure of the furniture. Finally, the 4th one seems to simulate a contrast enhancement of the input image.

Q5. Why not directly train VGG16 on 15 Scene?

Directly training VGG16 on 15 Scene is not ideal due to the large number of parameters, leading to a high risk of overfitting with such a small dataset.

Q7. What limits can you see with feature extraction?

To be efficient, feature extraction needs to be performed using a model pre-trained on a sufficiently large dataset to capture relevant knowledge that can be transferred to a wide range of downstream tasks. Additionally, the domain of the target dataset should align with that of the source dataset to ensure generalization. This alignment can be challenging in cases such as medical or satellite imaging, where the image characteristics are highly domain-specific, and data is difficult to collect and label. Finally, feature extraction alone can be limiting if the model needs to adapt to the target dataset's features. In such cases, fine-tuning the model could be a viable solution.

Q8. What is the impact of the layer at which the features are extracted?

- **Early layers**: Early layers capture low-level features such as edges, textures, and simple shapes. These features are generic and highly transferable to any range of tasks and domains, which is especially useful if the downstream dataset differs from the training one. On the other hand, they can be more computationally expensive because of their high dimensionality (cf. VGG16 architecture.
- **Deeper Layers**: Capture high-level features that are highly specific to the original task. They are most useful for tasks closely related to the source dataset like other natural image classification tasks similare to ImageNet. They are less computationally expensive because of their smaller dimensionnality and require less additional work to train the new model since the features are "ready to use".

Q9. The images from 15 Scene are black and white, but VGG16 requires RGB images. How can we get around this problem ?

To address this issue, we can think of:

- Replicating the grayscale image across all three channels to create an RGB-like input with identical values in each channel.
- Using a transformation or model to estimate realistic RGB values from the grayscale image, leveraging shape and intensity information.

Q10. Rather than training an independent classifier, is it possible to just use the neural network? Explain.

It would be difficult to use the neural network as-is because the classifier in VGG is designed for 1000 ImageNet classes, not the 15 15Scene classes, making the outputs incompatible. Additionally, while the latent representations before the classifier are designed to be linearly separable, they are not tailored to the specific classes of 15Scene. Zero-shot learning could be an alternative, but it would require creating a meaningful latent representation for the classes inside the model, which would required modifying it. Moreover, the large number of parameters in VGG poses a significant risk of overfitting on the small 15Scene dataset.

Going further

Tune the parameter C to improve performance

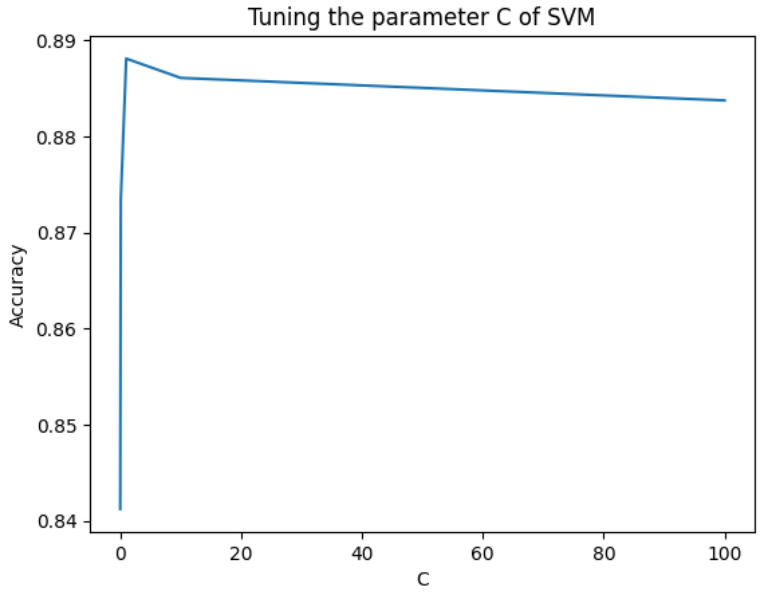

Figure 1.3: Tuning the parameter C of the SVM

According to our experimentations, the optimal value for the regularization constant of the SVM is C=2.

Instead of training an SVM, replace the last layer of VGG16 with a new fully-connected layer and continue to train the network on 15 Scene (with or without propogating the gradients to the rest of the network)

In this experiment, we created a classifier made of a fully connected layer (4096, 1000), a ReLU activation unit, and a final FC layer (1000, 15).

We decided to freeze the feature extractor as we consider the images in 15scene to be close enough to the ImageNet training dataset. Results of the training are as follows:

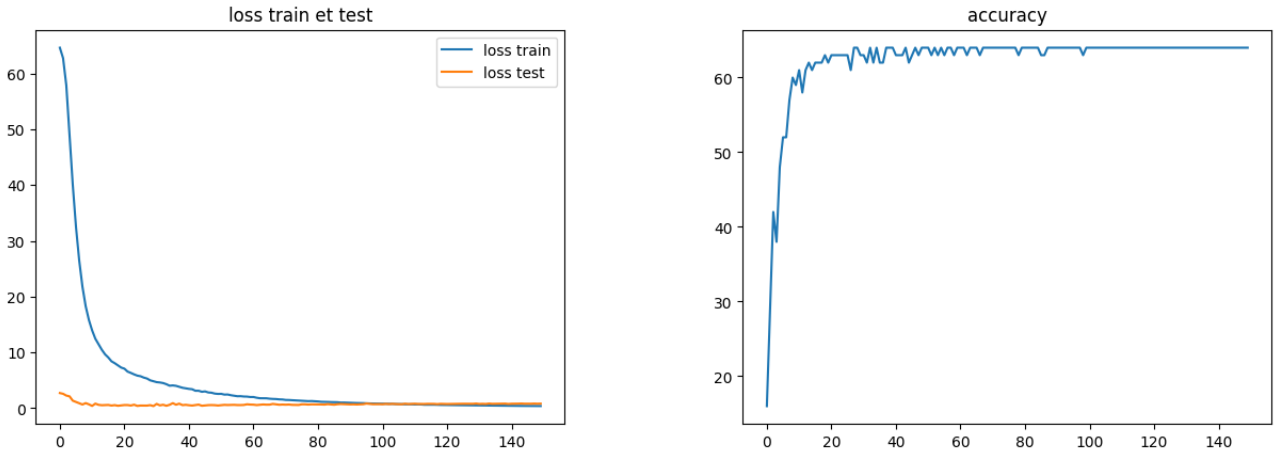

Figure 1.4: Train and test loss (left) and test accuracy (right) using a fully connected classifier instead of an SVM

We can see a convergence after approximately 100 epochs in training, while the loss in the test case converged faster (10 epochs) with a lower starting value. This behavior could be due to the already efficient features used from the feature extractor, while the pleateauing in train and test loss could be explained by the relatively small number of parameters of the fully connected layers, and the one in the accuracy by the non update of the parameters of the feature extractor, as the performance is now limited by the training capacities of the new classifier.

Conclusion

This chapter introduced the concept of transfer learning, emphasizing its ability to leverage pre-trained models like VGG16 to solve new tasks with limited data. By exploring techniques such as feature extraction and fine-tuning, we demonstrated the importance of aligning source and target datasets for effective knowledge transfer. Additionally, we discussed the challenges and solutions for adapting deep learning models to domain-specific problems. The chapter laid the foundation for using transfer learning as a powerful tool in deep learning workflows.

Chapter 2

Visualizing Neural Networks

This practical work deals with visualizing neural networks and the aspects of the images that affect their internal behavior through gradient-based methods. We will explore 3 techniques: saliency maps, adversarial examples, and class visualization.

2.1 Saliency maps

A saliency map identifies the important regions of an image that contribute most significantly to the prediction of the correct class in a convolutional neural network.

To produce the heatmap, we will compute the absolute value of the derivative of the class score $\tilde{y}_i$ with respect to the input image I. For each pixel, the heatmap is then generated by taking the maximum value across the three RGB channels, resulting in a 2D matrix. Mathematically, this can be expressed as:

$$\text{Heatmap}(x, y) = \max_{c \in \{R,G,B\}} \left| \frac{\partial \tilde{y}_i}{\partial I(x, y, c)} \right|$$

Here, (x, y) represents the pixel coordinates, and c denotes the RGB channels.

Q1. Show and interpret the obtained results.

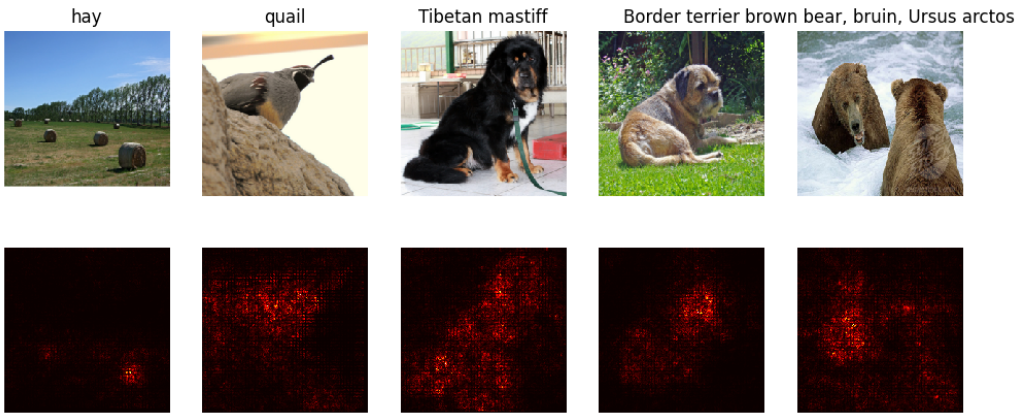

Figure 2.1: Saliency map

The saliency maps highlight the most critical pixels that influence the model's prediction for the correct class. The red points represent areas with the highest gradients, which indicate regions of the image that the model focuses on to make its decision.

These points outline the shapes or regions corresponding to the main objects in the image. For example: In the dog images, the points trace the silhouettes of the dogs. In the "hay" image, the points are concentrated on the hay bales' locations.

These highlighted areas suggest which parts of the image are most informative for the model to correctly classify the input, which are coherent as to what human logic would expect.

Q2. Discuss the limits of this technique of visualizing the impact of different pixels.

The limits of this technique lie in the face that it considers pixels as individual elements independently from the others, whereas we have seen earlier that convolutional networks use higher level features captured through the local context of convolutions in order to classify an image. This shows that this method is a good first step in understanding the behavior of the model, but is insufficient to completely modelize and predict its decision.

Q3. Can this technique be used for a different purpose than interpreting the network?

Saliency maps could be utilized as silver standard labels for tasks like segmentation or object detection, or as additional features for a new model designed specifically for these purposes. They could also be incorporated into the loss function of the new model to produce smoother error values, leveraging the continuity of gradient values compared to binary masks.

Finally, as we will see next, this information could be leveraged to generate adversarial data aimed at deceiving the model's classification. For instance, a second model could be trained on (image, modified saliency map that misleads the first model) pairs to achieve this.

Q4. Test with a different network, for example VGG16, and comment

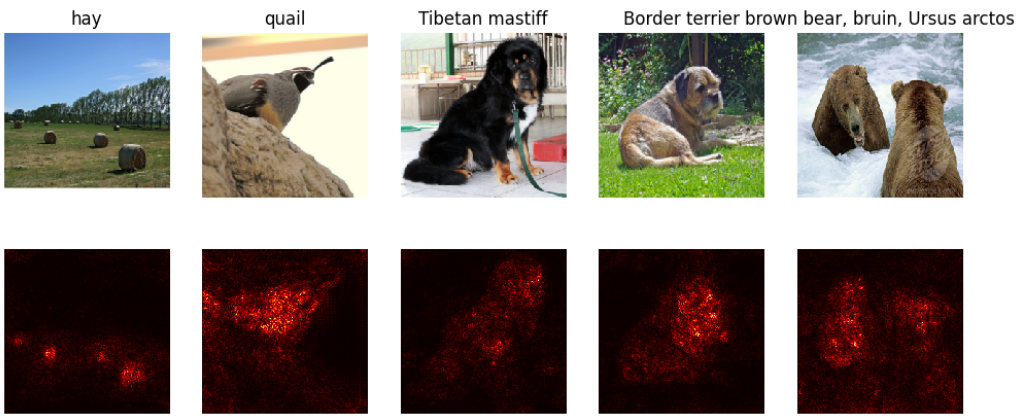

Figure 2.2: Saliency maps VGG16

Comparison between SqueezeNet and VGG16 saliency maps

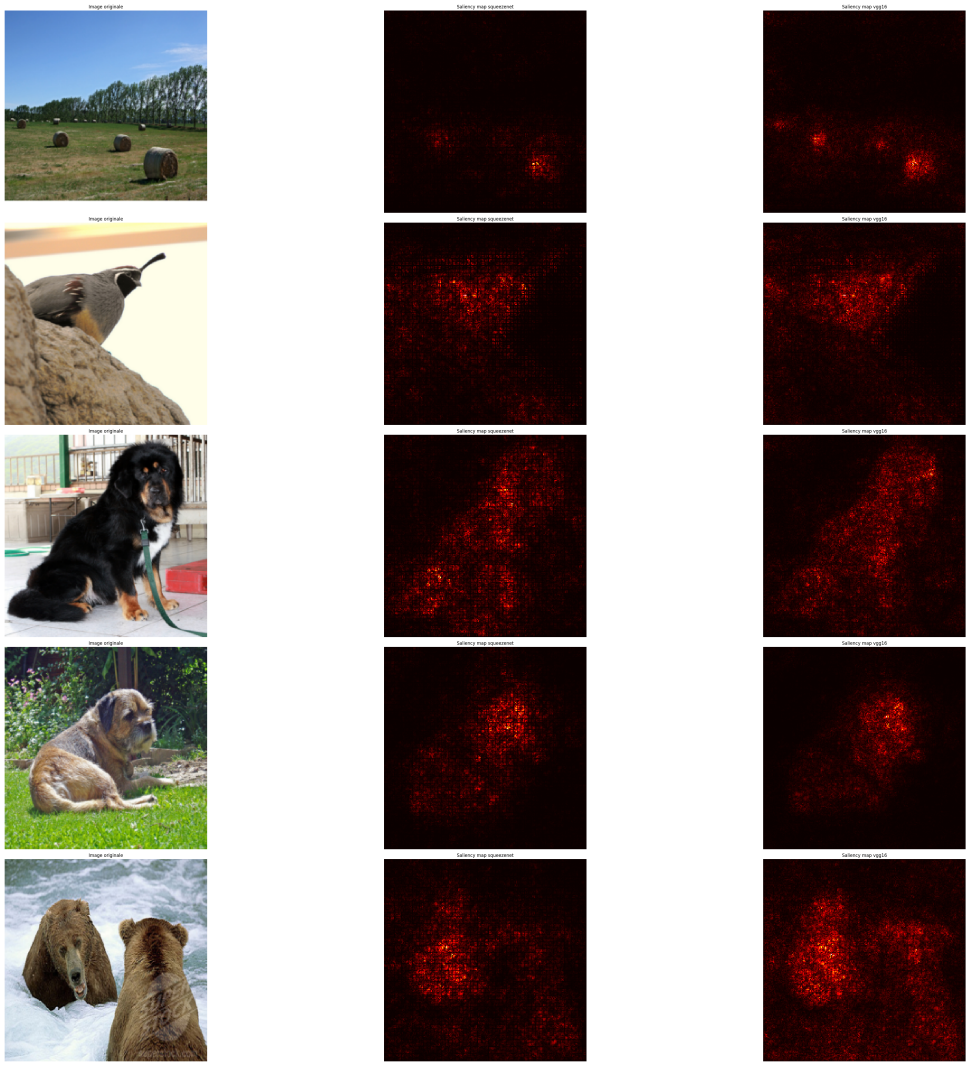

Figure 2.3: Saliency maps squeezenet (middle) vs VGG16 (right)

With VGG16, we notice that the saliency maps are much more vivid and bright, plus the shapes defined by them seem sharper and faithful to the object to be detected. Specifically, we can notice that in the bear example, it highlights not only the front facing bear, but also the rear facing one, as opposed to SqueezeNet that focuses on the first one. This is probably due to the higher number of parameters in the model compared to squeezenet, which is designed to be compact, which allows it to capture more precisely the details in the images.

2.2 Adversarial examples

Adversarial examples are images belonging to classes recognized by the model, but that are intentionally altered to mislead the model into incorrectly classifying them.

In order to achieve that, we will use a gradient-based image modification scheme. By modifying the input image x, rather than the model, we aim to change its classification from the correct class i to a target class j. This is achieved through gradient backpropagation:

$$g = \frac{\partial \tilde{y}_j}{\partial x},$$

which computes the gradient of the score $\tilde{y}_j$ for class j with respect to the input x. The image is progressively updated according to the rule:

$$x \leftarrow x + \eta \frac{g}{\|g\|_2},$$

where η is the learning rate. This process is iteratively applied until the model predicts the target class j.

Q5. Show and interpret the obtained results

Figure 2.4: Saliency maps VGG16

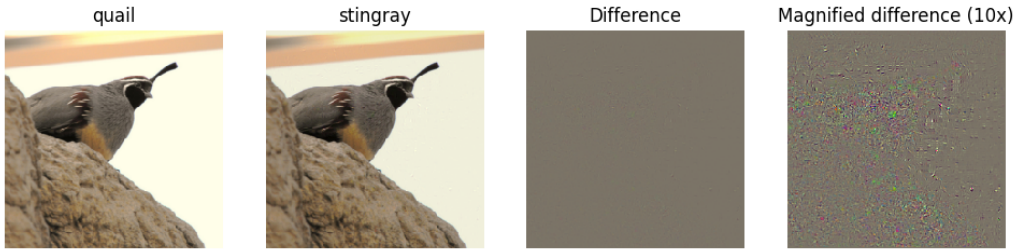

Figure 2.5: Saliency maps VGG16

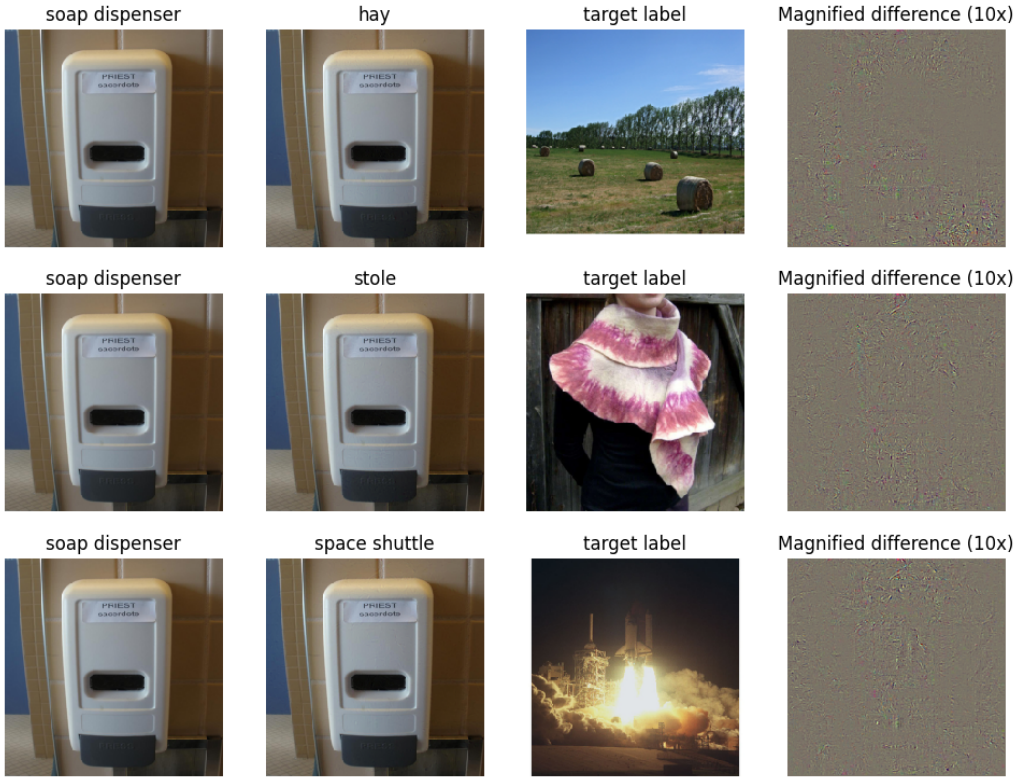

Figure 2.6: Influence of the target class on the adversarial disturbance.

The results demonstrate that adversarial examples require only minimal, almost imperceptible, modifications to the original image to mislead the model into predicting an incorrect class. These subtle changes are typically undetectable to the human eye but are sufficient to alter the model's decision into a significantly different class.

Moreover, the modified pixels align closely with the most critical areas identified in the **saliency maps**, which makes sense as these regions contribute most to the classification process. This highlights the model's vulnerability to targeted manipulation in its most sensitive areas.

Lastly, the modifications seem to be influenced not only by the defining features of the source class but also by the characteristics of the target class (straight lines in the space shuttle, rough silhouette of a person in the stole example, pixels in the bottom of the image in the hay example). The difference maps often show patterns or shapes that resemble features of the target label, illustrating how the desired class influences the distribution of the perturbations in order to shift the model's decision toward it.

Q6. In practice, what consequences can this method have when using convolutional neural networks?

This method highlights the vulnerability of convolutional neural networks to noise, as even small, targeted perturbations can significantly alter their predictions. This lack of robustness raises concerns about the reliability of such models, particularly in critical domains like medical imaging or autonomous driving, where incorrect decisions can have serious consequences. Furthermore, adversarial attacks can be exploited by malicious persons to deceive models, leading to potential security and safety risks in real-world applications.

Q7. Discuss the limits of this naive way to construct adversarial images. Can you propose some alternative or modified ways? (You can base these on recent research)

Limits

First, we can imagine that, with different types of images that we used here, the adversarial noise can be less subtle and more easily perceptible by detection algorithms, for example in highly contrasted images. Another problem is that the image obtained through this type of methods may not be transferable to other models, as they may rely on different pixels or areas to make their predictions. Finally, the fact that the model is sensitive to noise with regards to the original images suggests that it can also be sensitive to perturbations in these adversarial images, which could lead to adversarial predictions different than the ones that were intented.

Alternative ways

- **Fast-Sign gradient method**: in this method proposed by [Goodfellow, 2014], the original image is transformed by adding or subtracting a small error to each pixel, depending on whether the sign of the gradient for the pixel is positive or negative.

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

  where $\nabla_x J$ is the gradient of the model's loss function with respect to the original input pixel vector x, y is the true label vector for x, θ is the model parameter vector, and $\epsilon istheaddederror.$

- **Expectation Over Transformation (EOT) algorithm [Athalye et al., (2017)]**:

  The main idea behind EOT is to optimize adversarial examples across many possible transformations. Instead of minimizing the distance between the adversarial example and the original image, EOT keeps the expected distance between the two below a certain threshold, given a

selected distribution of possible transformations. This EOT method modelizes the search for adversarial examples as an optimization problem where we try to find an adversarial example that maximizes the probability for the selected class, similarly to the studied method in this practical.

2.3 Class visualization

Class visualization is a technique used to understand the patterns and features that a neural network associates with a particular class. By modifying an input image to maximize the score of a specific class, we can gain insights into what the model detects and considers important for that class.

Principle

starting from an initial image x (e.g., random noise), we iteratively modify the input to maximize the class score for the desired class i. To improve the quality of the resulting visualizations, we incorporate regularization techniques to produce cleaner and more interpretable images.

- Add an L_2 norm of the image as a regularization loss to prevent extreme pixel values.
- Use implicit regularization techniques, such as translations and blurs, to further smooth the image.

Formulation:

For an image x that we want to modify to maximize the score for class i, the loss function is defined as:

$$\mathcal{L} = \tilde{y}_i - \lambda \|x\|_2,$$

where $\tilde{y}_i$ is the score for class i, and $\lambda\|x\|_2$ is the regularization term controlled by the hyperparameter λ.

Update Rule:

The image is iteratively updated using the gradient of the loss $\mathcal{L}$ with respect to x, following the rule:

$$x \leftarrow x + \eta \nabla_x \mathcal{L},$$

where η is the learning rate. This process is repeated until the image prominently displays patterns associated with the desired class.

Q8. Show and interpret the obtained results

Figure 2.7: Real turnstile image

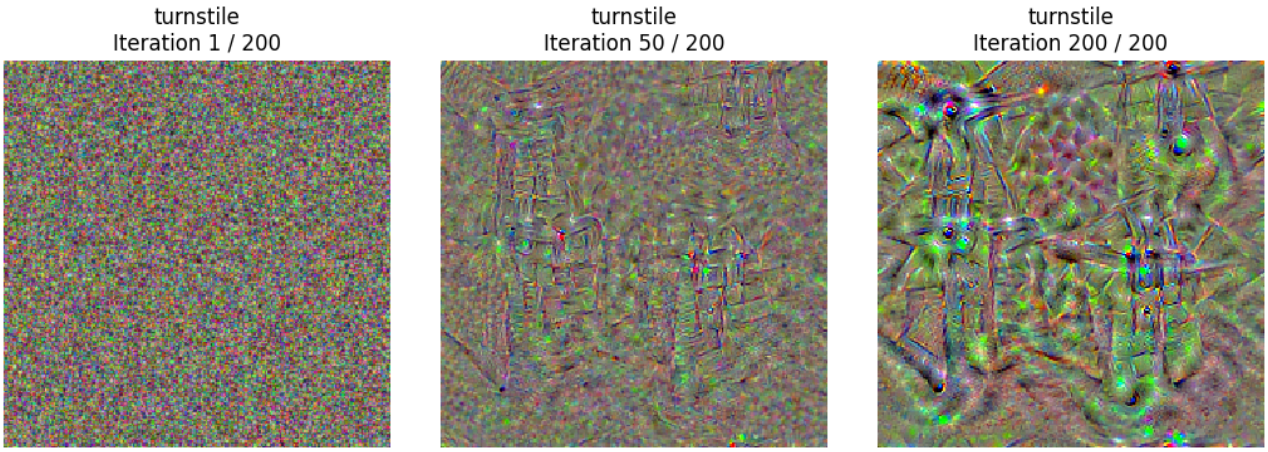

Figure 2.8: Iterative modification of a noise image into a turnstile-predicted one by SqueezeNet

This experiment shows that it doesn't take for the image to be realistic in order to be classified in a certain class, as the object of interest is unrecognizable and the image is still mostly noise. However, we can still recognize the general shape of the turnstiles. This showcases how the model bases its predictions on abstract features related to the patterns and geometry of the image in addition to local intensity variations, while it seems that the colors themselves play a secondary role.

Q9. Try to vary the number of iterations and the learning rate as well as the regularization weight

Varying the regularization weight

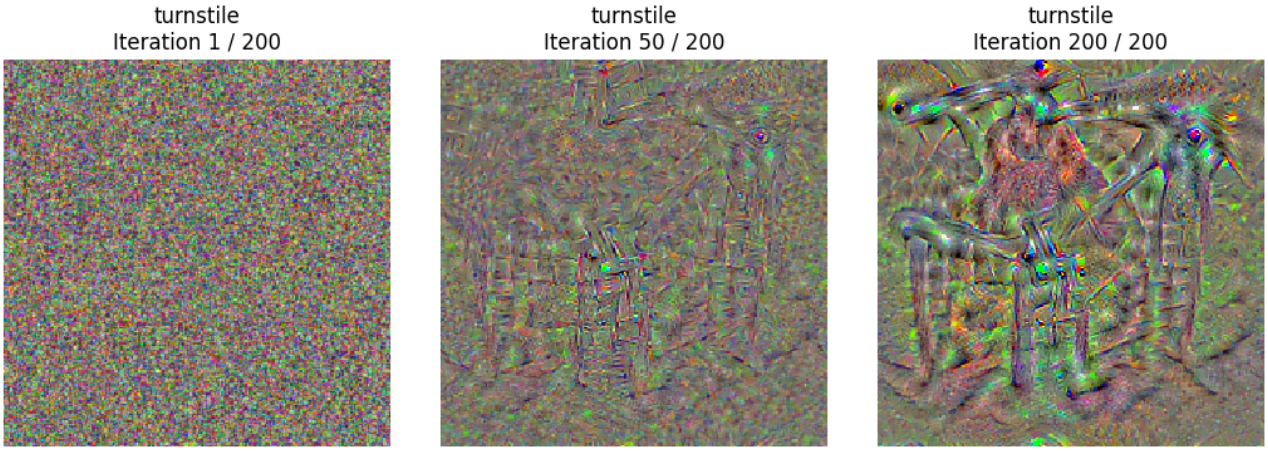

Figure 2.9: Increase of the L2 regularization constant (1e-1)

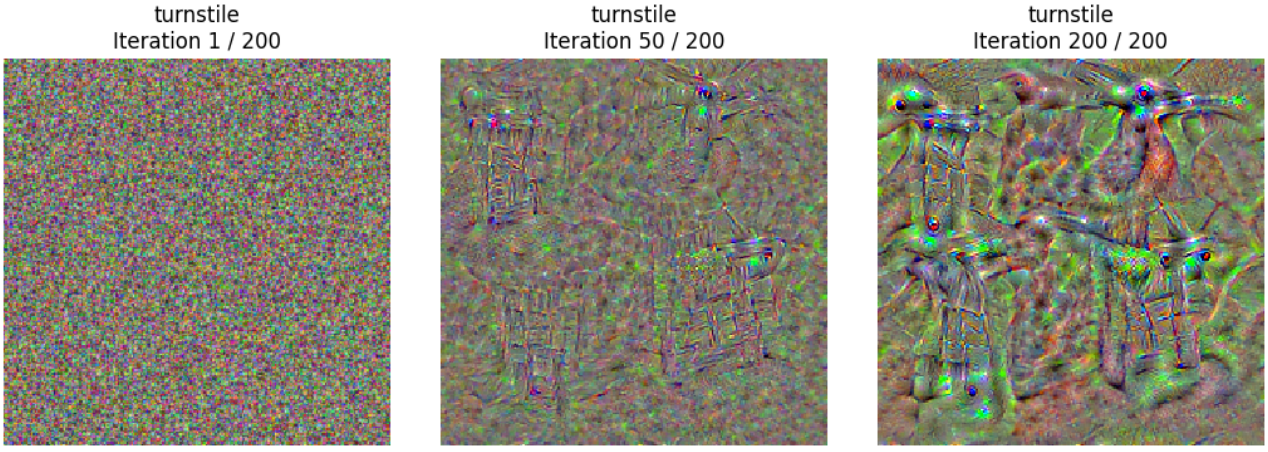

Figure 2.10: Decrease of the L2 regularization constant (1e-7)

Figure 2.11: Comparison between 1e-1 (left) and 1e-7 (right) constant regulariztions at the 50th iteration

The results of the experiments do not allow us to make conclusions on the effect of the regularization constant on ther esulting image, as they seem very close between a small and large value. Although, we would expect that a strong regularization constant would restrain the model from transforming too much the original image, and adding just enough modifications in order to achieve the targeted classification.

Varying of the number of iterations

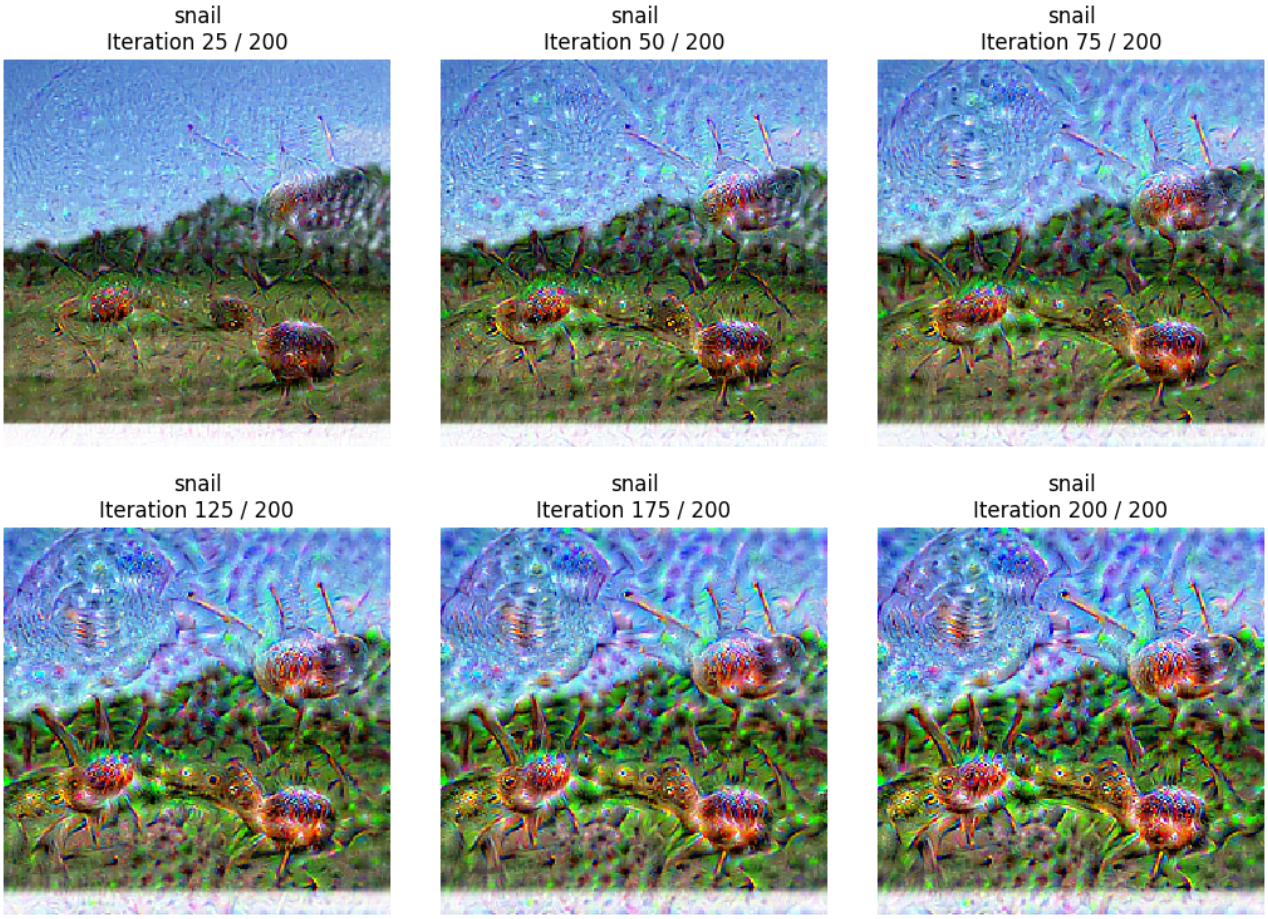

Figure 2.12: Resulting image across different number of iterations

As the number of iterations increases, the modifications toward the target class become progressively more distinct and evident.

Varying the learning rate

Small learning rate

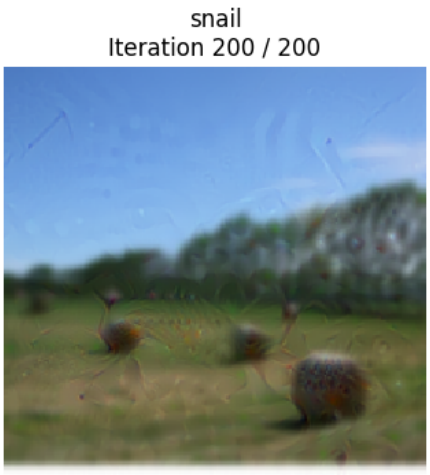

Figure 2.13: Result after 200 iterations with learning rate = 1

We notice that with a small value of the learning rate the modifications are very slight from iteration to iteration, making for a slow convergence that requires a high number of repetitions in order to converge.

Large learning rate

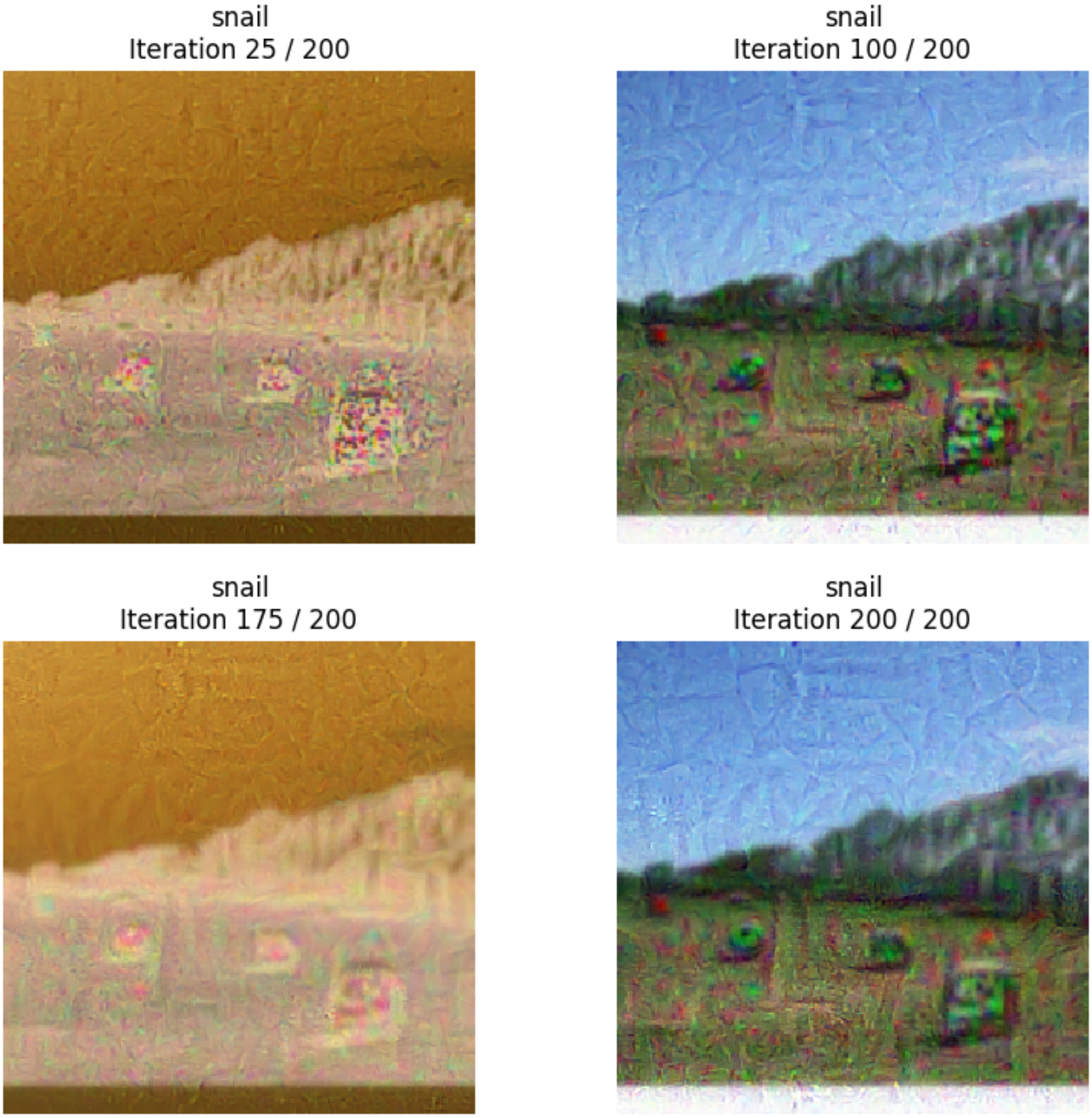

Figure 2.14: Images résultante à différentes itérations avec un learning rate=100

A learning rate that is too large can cause instability during the optimization, leading to oscillations in the solution space and preventing convergence to the optimal point, a similar behavior occurs during the training of neural networks.

Q10. Try to use an image from ImageNet as the source image instead of a random image (parameter init_img). You can use the real class as the target class. Comment on the interest of doing this.

We can see that this methods works similarly with actual images as with random noise, modifying local intensity values variations in order to create patterns related to the targeted class (for example, the horns of the snail and its shell). Interestingly, we can notice how the algorithm took advantage of the hay bales to create the snales shells, as they have a similar shape.

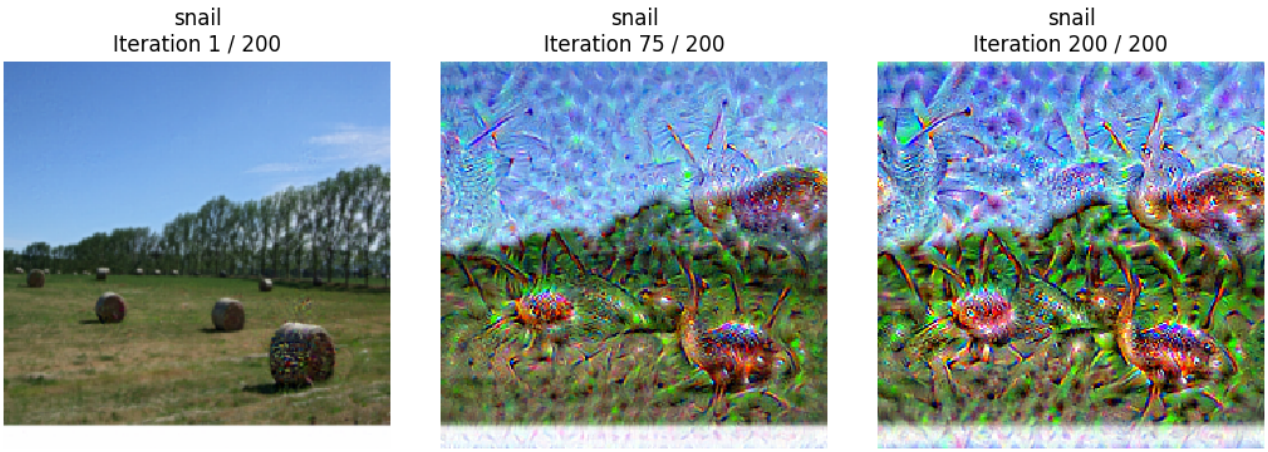

Figure 2.15: Turning a hay image into a snail-predicted one by SqueezeNet

11. Test with another network, VGG16, for example, and comment on the results.

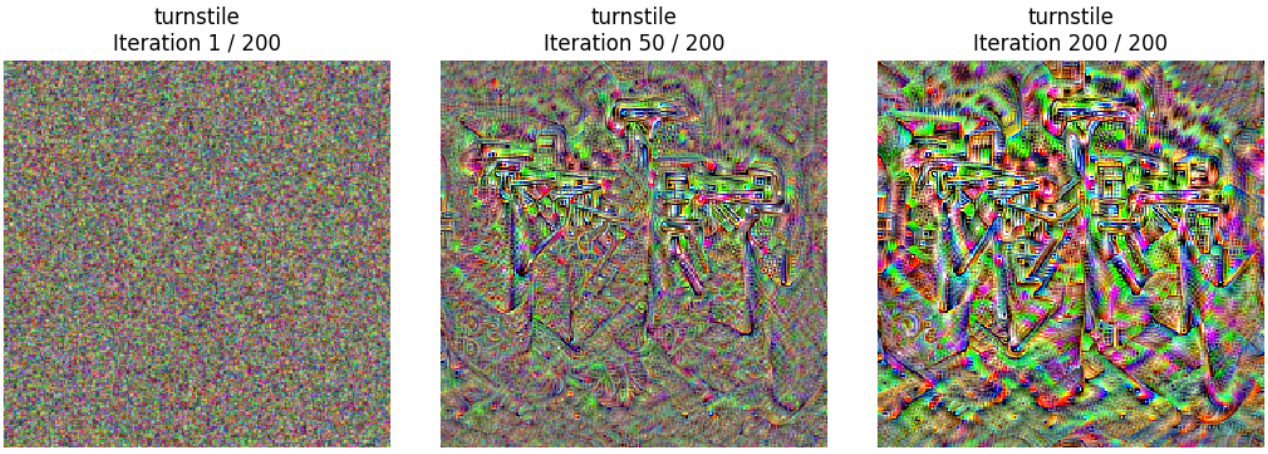

Figure 2.16: Turning noise into a turnstile-predicted image with VGG16

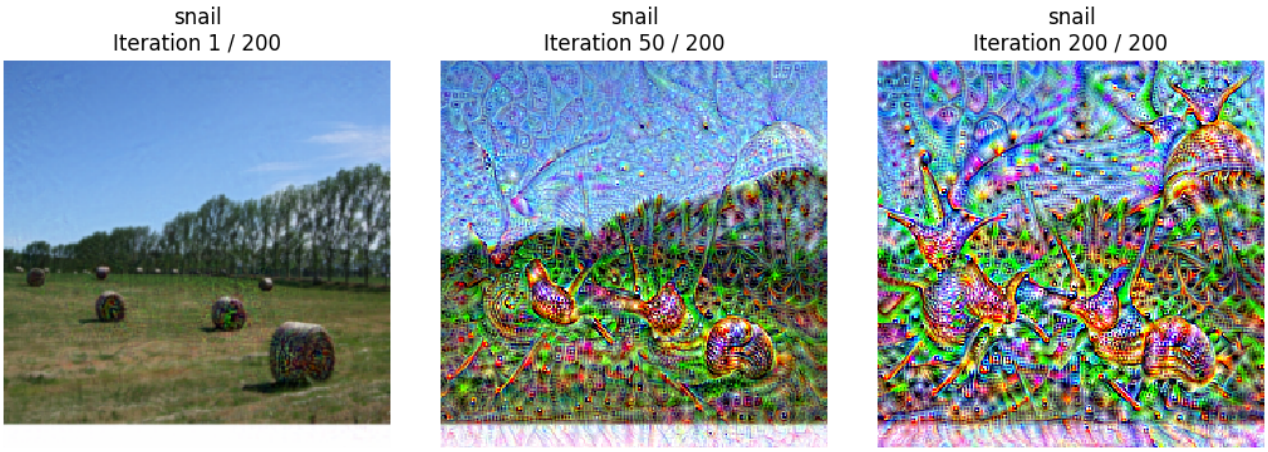

Figure 2.17: Turning a hay image into a snail-predicted image by VGG16

From the experiments with VGG16, we notice that the resulting patterns are much more pronounced and visible, especially in the hay image where it has become barely recognizable and we can recognize more clearly the shapes of the snails. Similarly to the activation maps experiments, this can be explained by the over-parametrization of VGG16 compared to SqueezeNet.

Conclusion

This chapter focused on interpreting neural networks through visualization techniques such as saliency maps, adversarial examples, and class visualization. We analyzed how saliency maps reveal the most influential regions of an image, while adversarial examples highlighted the vulnerability of models to subtle perturbations. Through class visualization, we explored how networks associate patterns and features with specific classes. These methods provided valuable insights into the inner workings of neural networks, fostering a deeper understanding of their decision-making processes.

Domain Adaptation

Domain Adaptation with the DANN Model and Gradient Reversal Layer (GRL)

Domain adaptation addresses the challenge of training a model on a labeled source dataset while achieving good performance on an unlabeled target dataset. This is particularly relevant when the source and target domains share the same classes but differ in pixel distribution. For instance, images from the MNIST and Color-MNIST datasets, which we will use in our exprimentations, represent similar classes (numbers) but exhibit significant differences in visual characteristics.

Figure 3.1: MNIST and Color-MNIST datasets

The DANN Model and its Components

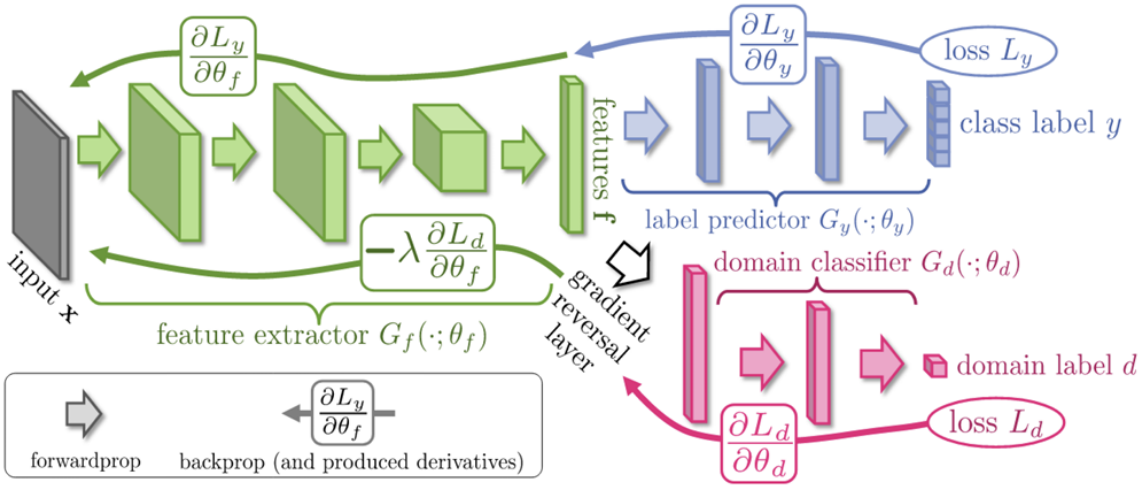

Figure 3.2: The DANN architecture

The Domain-Adversarial Neural Network (DANN) model consists of three main components:

- **Feature Extractor (ConvNet)**: The convolutional neural network (green layers) learns a feature representation of the input data that is useful for classification.

- **Task-specific classifier** The blue classifier uses the extracted features to predict the class of the input image.
- **Domain Classifier**: The pink classifier predicts whether the input data comes from the source or target domain.
- **Gradient Reversal Layer**: To achieve domain adaptation, the model incorporates a Gradient Reversal Layer (GRL) between the feature extractor and the domain classifier. The GRL reverses the gradient during backpropagation (by multiplying it by a negative number). enabling domain-agnostic learning of features by the ConvNet.

Formally, the GRL modifies the gradients during backpropagation as follows:

$$\text{GRL Gradient: } \frac{\partial L_{\text{domain}}}{\partial \text{features}} \rightarrow -\lambda \frac{\partial L_{\text{domain}}}{\partial \text{features}},$$

where L_{domain} is the loss from the domain classifier, and λ controls the strength of the gradient reversal.

Questions for Discussion

1. **Without the GRL**: If the GRL is not used, the feature extractor may retain domain-specific information, allowing the domain classifier to easily distinguish between source and target domains. This would prevent the feature extractor from learning domain-invariant representations.

2. **Degradation of Source Performance**: Adapting to the target domain can cause slight degradation in performance on the source domain because the features are optimized for domain invariance rather than source specificity.

3. **Impact of λ**: The value of λ in the GRL determines the extent of gradient reversal. A small value may result in weak domain adaptation, while a large value could excessively disrupt the features, degrading performance on both domains.

4. **Pseudo-Labeling**: Pseudo-labeling involves assigning labels to unlabeled target data using the model's predictions and incorporating them into training. This technique can iteratively improve performance on the target domain by refining the model's understanding of target data.

Q1. If you keep the network with the three parts (green, blue, pink) but didnt use the GRL, what would happen?

Without the GRL, the network would learn domain-specific features in order for the domain classifier to easily discriminate between the source (MNIST) and target (MNIST-M) domains. As a result, the feature extractor would focus on separating the two domains instead of finding

domain-invariant representations. This would lead to poor generalization on the target dataset (MNIST-M), as the learned features would not align across domains. In contrast, using the GRL encourages the feature extractor to produce domain-agnostic representations by making it harder for the domain classifier to distinguish between domains, thereby improving generalization. This phenomenon can be seen in the T-SNE visualization of the two datasets put together below:

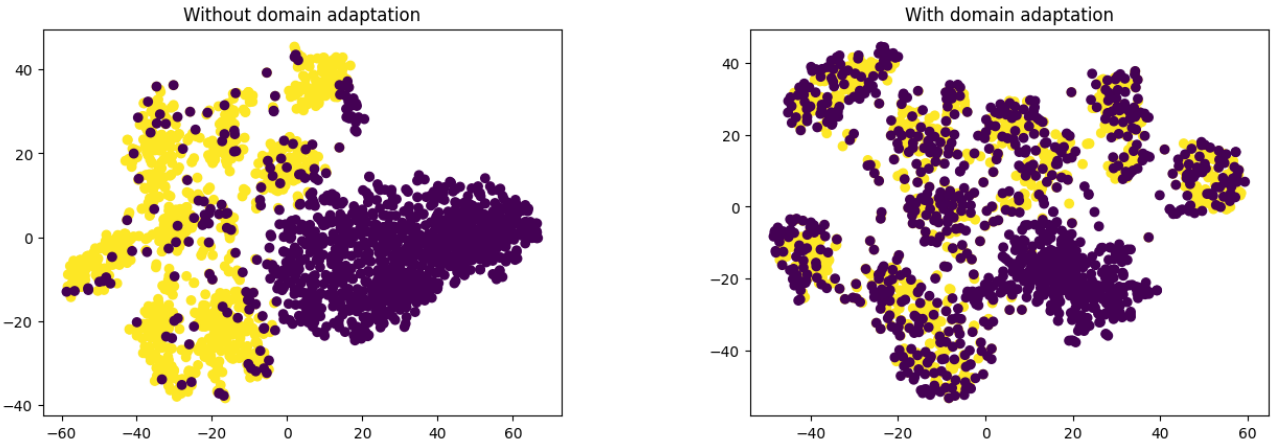

Figure 3.3: T-SNE visualization of source and target datasets without (left) and with (right) domain adaptation

Q2. Why may the performance on the source dataset degrade a bit

The performance on the source dataset may degrade slightly because the model is constrained to learn features that generalize across both the source and target domains, instead of being fully optimized for the source dataset alone. This forces the model to adapt to features common to both domains, which slightly reduces its specificity for the source one.

Training results

Epochs	Dataset	Class Loss / Acc	Domain Loss / Acc
20	SOURCE	0.03722 / 98.66%	0.46794 / 88.88%
20	**TARGET**	1.23199 / **73.03**%	0.63732 / 53.7%
100	SOURCE	0.06678 / 98.22%	0.5191 / 78.81%
100	**TARGET**	1.07924 / **78.08**%	0.62658 / 57.71%

Table 3.1: Résultats des performances sur les ensembles de données SOURCE et TARGET après 20 et 100 epochs.

As shown in the table, the model keeps a significant performance on the source dataset (here we notice a drop in the accuracy from 98.6% to 98.22%, but it is not significant enought to talk about a drop) while improving on the target dataset across the epochs (73% at 20 epochs, 78% at 100). In parallel, we can see a constant and significant drop in the domain classification

performance, going from 88.88% and 53.7% to 78.88% and 57.71% respectively in the source and target datasets.

Q3. Discuss the influence of the value of the negative number used to reverse the gradient in the GRL

We gradually increase the weight of the GRL as described in the graph below:

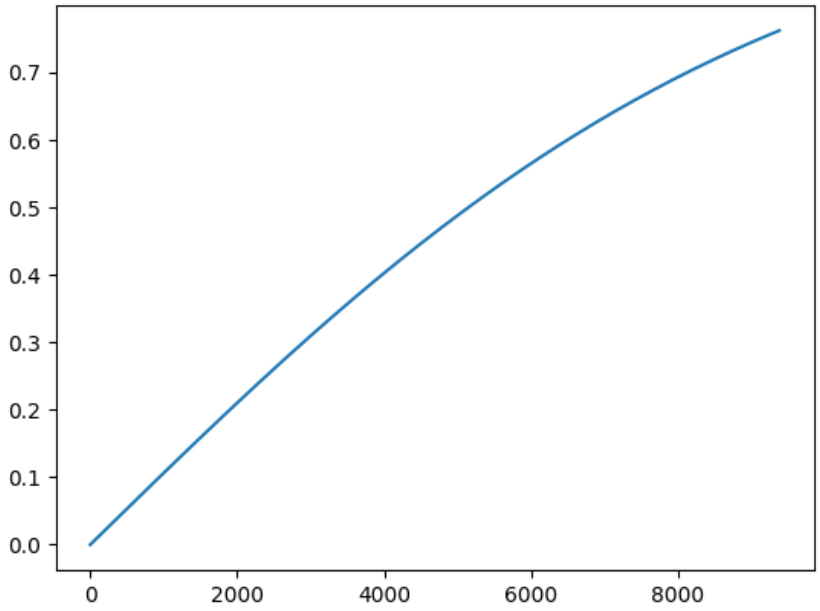

Figure 3.4: Evolution of the factor across the training

At first, the factor is in the lower values in order to stabilize the training and focus on learning meaningful representations for both domains. As the factor increases, domain alignment becomes more prominent, enabling the feature extractor to shift its features into more domain-invariant representations without prematurely compromising task performance.

Q4. Another common method in domain adaptation is pseudo-labeling. Investigate what it is and describe it in your own words.

Pseudo-labeling is the process of generating labels for a target dataset using the model predictions on its data. This is for example a method used in the advanced training phases of the Segment Anything Model [Kirillov, 2023], by training it on new images labeled by its own predictions, allowing for the use of an extensive amount of images.

In a paper published by [Chhabra, 2024], it is described that traditional non-supervised approaches for data alignment across different modalities have the inconvenient of considering the data distribution as a whole and not taking into account the classes themselves, which does not guarantee that examples related to the same class but in two different modalities will be correctly aligned in the feature space.

The paper proposes class-prediction on the target dataset using a pretrained model, then refining that prediction using multiples filter, which will then be used to better align the class distributions of the two datasets allowing for more efficient generalization.

Conclusion

This chapter addressed domain adaptation using the Domain-Adversarial Neural Network (DANN) model, demonstrating its ability to generalize across distinct data distributions. By employing techniques like the Gradient Reversal Layer, we achieved domain-invariant feature representations, improving target domain performance while preserving source domain accuracy. We discussed the trade-offs and challenges of domain adaptation, including pseudo-labeling and its impact on aligning data distributions. This chapter highlighted the importance of adapting models to new environments for real-world applications.

Chapter 4

Generative Adversarial Networks

In recent years, the field of computer vision has shifted from traditional discriminative approaches, which focus on modeling the conditional probability $P(Y|X)$, to generative models that capture the joint distribution $P(X, Y)$. While discriminative models are primarily used for tasks like classification, generative models enable a broader range of applications, including data generation and completion.

One of the most impactful generative models is "Generative Adversarial Networks (GANs)", introduced in 2014. GANs are designed to generate new data (such as images) from random noise. Unlike traditional generative models, which explicitly model the joint distribution, GANs use two networks: a generator that creates synthetic data and a discriminator that attempts to distinguish real from fake data. Through this adversarial process, GANs have proven highly effective in generating realistic images, often requiring little to no labeled data.

In addition to standard GANs, "Conditional GANs (cGANs)" allow for more control over the generated output by incorporating additional information (like class labels, images, or text) into the generation process. This makes cGANs particularly useful for tasks where specific, context-dependent outputs are needed, such as generating images of particular categories or editing images based on given conditions.

In this report, we will explore the fundamentals of GANs and cGANs, their differences, and their impact on the field of computer vision, particularly in areas like image generation.

4.1 Generative Adversarial Networks (GANs)

A "Generative Adversarial Network (GAN)" consists of two main components: the **generator** G and the **discriminator** D.

The **generator** G transforms a random noise vector z (sampled from a fixed distribution $P(z)$, such as $U[-1, 1]$ or $N(0, I)$) into a synthetic image $x = G(z)$, aiming to generate realistic data that mimics the unknown distribution of real data $P(X)$.

The **discriminator** D receives an image as input and predicts whether it is a real image x from the training set or a fake image generated by G. The goal of D is to correctly classify real images as $D(x) = 1$ and fake images as $D(x) = 0$.

Training a GAN is a game-theoretic process where the generator attempts to fool the discriminator by creating increasingly realistic images, while the discriminator tries to improve its

ability to distinguish real from fake. The training objective is a minimax problem:

$$\min_G \max_D \mathbb{E}_{x\sim P(X)}[\log D(x)] + \mathbb{E}_{z\sim P(z)}[\log(1 - D(G(z)))]$$

This adversarial setup alternates between updating the generator to maximize $D(G(z))$ and updating the discriminator to correctly classify real and generated data.

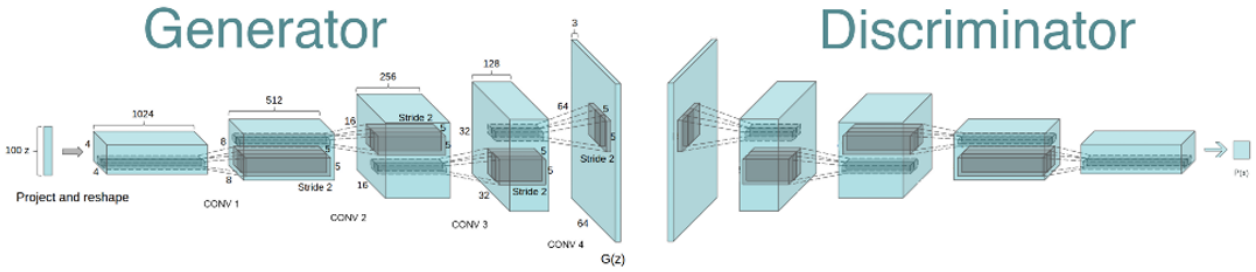

Figure 4.1: Architecture of the classic DCGAN (Radford et al., 2016).

Q1. Interpret the equations. What would happen if we only used one of the two ?

The equations represent the objective functions of a Generative Adversarial Network (GAN) :

- Equation for the Generator (G):

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

  The generator G takes as input a random noise vector z, sampled from a prior distribution $P(z)$ (such as a Gaussian or uniform distribution), and maps it to a synthetic image $\tilde{x} = G(z)$. The noise vector z introduces randomness into the generator, enabling it to explore the latent space and produce diverse outputs.

  The goal of the generator is to produce synthetic images $\tilde{x}$ that are indistinguishable from real images by the discriminator D. The generator seeks to maximize $\log D(G(z))$, which corresponds to the probability that the discriminator incorrectly classifies the generated image $G(z)$ as real.

- Equation for the Discriminator (D):

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

  The discriminator D is responsible for distinguishing between real images x^* from the dataset D_{data} and fake images $\tilde{x} = G(z)$ generated by the generator. The discriminator outputs a probability $D(x)$, where $D(x^*)$ represents the probability that a real image

is classified as real, and $D(G(z))$ represents the probability that a generated image is classified as real. The discriminator maximizes two terms: $\mathbb{E}_{x^* \sim D_{data}}[\log D(x^*)]$, which encourages the discriminator to correctly classify real images as real, and $\mathbb{E}_{z \sim P(z)}[\log(1 - D(G(z)))]$, which encourages the discriminator to classify generated images as fake.

The discriminator acts as the adversary to the generator, ensuring that it not only correctly identifies real images but also accurately detects fake images, providing feedback that forces the generator to improve over time.

If only the generator's objective was used: The generator would produce synthetic images $\tilde{x} = G(z)$ without meaningful feedback from the discriminator, leading to stagnant performance and low-quality or random images.

If only the discriminator's objective was used: The discriminator would become highly proficient at distinguishing real from fake images, but the generator would fail to evolve, consistently producing unrealistic images.

Important note :

The original **minimax objective** seeks to find a balance where the generator minimizes $\log(1 - D(G(z)))$, while the discriminator maximizes $\log D(x) + \log(1 - D(G(z)))$. This formulation, however, suffers from a key limitation: when $D(G(z))$ is small (as it often is early in training), the generator receives very small gradients due to the saturation of $\log(1 - D(G(z)))$, making it harder to improve the generated images and slowing down convergence.

To address this issue, the **non-saturating objective**, described in the equations above, modifies the generator's loss to maximize $\log D(G(z))$ instead. This change significantly improves training stability by ensuring that the gradient does not vanish, even when the discriminator is poorly trained. With this formulation, the generator receives stronger feedback, allowing for more effective and stable updates, especially in the early stages of training when the generator's images are still far from realistic.

The gradient of the sigmoid function in this case is:

$$\frac{d\sigma(a)}{da} = \sigma(a) \cdot (1 - \sigma(a)) .$$

With:

$$D(x) = \sigma(a) = \frac{1}{1 + e^{-a}}.$$

Q2. Ideally, what should the generator G transform the distribution P(z) to?

In GANs, the generator G aims to transform the input noise distribution $P(z)$ into a distribution that closely resembles the real data distribution $P(X)$. The goal is for the generator to produce

synthetic data that is indistinguishable from real data. Ideally, this results in an equilibrium where the discriminator performs no better than random guessing, i.e., $\forall x, D(x) = \frac{1}{2}$, indicating that the generator has successfully learned to replicate the real data distribution.

Q3. Remark that the equation (6) is not directly derived from the equation (5). This is justified by the authors to obtain more stable training and avoid the saturation of gradients. What should the "true" equation be here?

In the original minmax GAN formula (5), the generator's objective is expressed as minimizing the likelihood that the discriminator successfully classifies its generated outputs as fake. This (6) can be represented by the following "true" equation:

$$\min_G \mathbb{E}_{z \sim P(z)}[\log(1 - D(G(z)))]$$

4.1.1 Architecture of the networks

Most GAN models for image generation are based on the Deep Convolutional GAN (DCGAN) architecture. This architecture uses convolutional layers, batch normalization, ReLU activations, etc.

For our purposes, we will implement a variant of this architecture to generate 32x32 pixel images from the MNIST dataset.

Q4. Comment on the training of of the GAN with the default settings (progress of the generations, the loss, stability, image diversity, etc.

The training of the GAN begins by generating images from random noise, gradually improving over time. Initially, the generated images are blurred, with vague figures. As training progresses through several hundred iterations, the background darkens, and the digit outlines become clearer, i.e, the generator produces more realistic images indicated by a decreasing loss. By the 600th iteration, the digits' lines become significantly sharper, with the results becoming acceptable only around the 1000th iteration.

However, the stability of the training remains a challenge, with frequent spikes in the loss curves, indicating fluctuating performance.

The g_loss shows higher variability, which is typical in GAN training as the generator continuously adapts to improve sample quality. In contrast, the d_loss remains relatively stable and low, indicating that the discriminator is effective in identifying real versus fake samples. The average scores at the bottom confirm this balance: the high average real score and low average fake score highlight the discriminator's robustness.

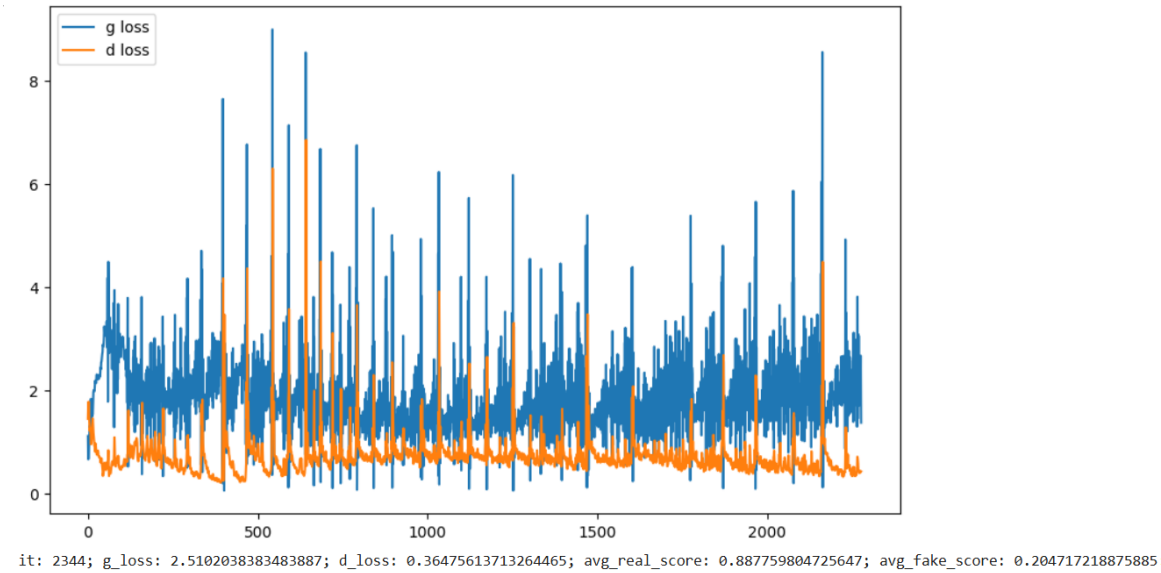

Figure 4.2: GAN Training

A notable issue is the lack of diversity in the generated images. Analysis of 64 generated images reveals an uneven distribution, with certain digits like '0', '3', '6', '7', '8', and '9' appearing more often, while others like '2' and '5' are almost absent.

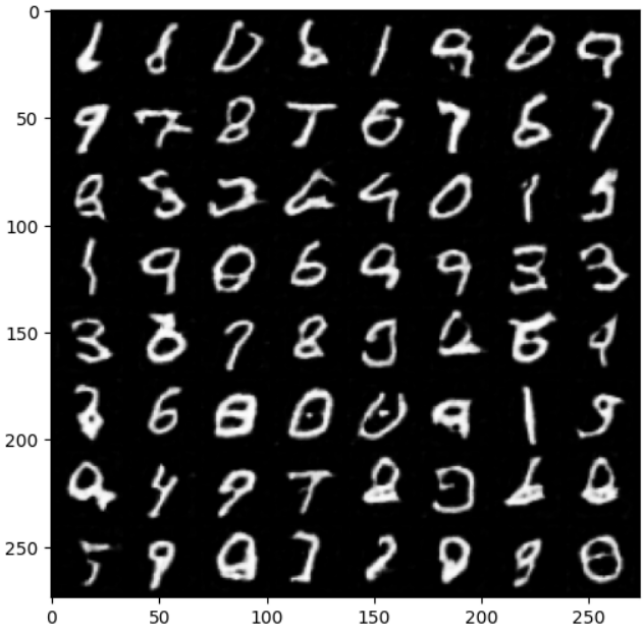

Figure 4.3: Digits results - GAN

This pattern raises concerns about a potential mode collapse issue, where the generator learns to produce only a subset of outputs sufficient to deceive the discriminator but fails to generate a wide range of diverse and accurate images.

Some images also contain unrecognizable digits, highlighting problems with accuracy. Despite improvements in realism, the generated digits remain synthetic and lack diversity.

Q5. Comment on the diverse experiences that you have performed with the suggestions above. In particular, comment on the stability on training, the losses, the diversity of generated images, etc.

This section presents experiments with the DCGAN architecture. The goal is to better understand its functionality, strengths, and weaknesses, and to identify areas for potential improvement.

Increase *ndf*

Here, we want to increase the value of *ndf* to strengthen the discriminator. By doing so, we enhance its ability to distinguish between real and generated images, making it harder for the generator to fool the discriminator.

Note : *ndf* refers to the Number of Discriminator Features. Specifically, it indicates the number of feature maps (or channels) in the first layer of the discriminator network. More generally, *ndf* is used to describe the depth or capacity of the convolutional layers in the discriminator.

The trade-off is that the generator should improve the quality of its images in response to the stronger discriminator, without falling into mode collapse, where it produces limited and repetitive outputs.

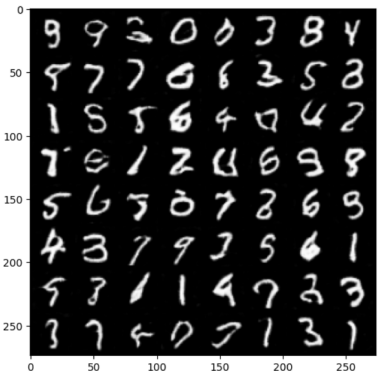

Figure 4.4: Digits results - GAN - ndf increased

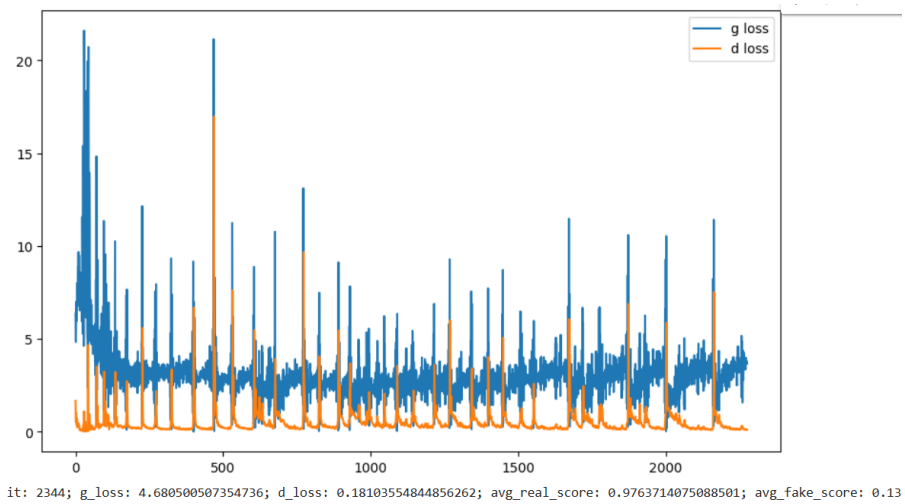

Figure 4.5: GAN training - ndf increased

We can observe that the results are generally more realistic compared to the first execution (see figure 4.2). Indeed, the generated images are both clearer and more diverse, with digits like '2', '4' and '5' appearing. This aligns with our goal of achieving a balanced distribution of digits, while maximizing their realism, although some examples remain difficult to identify.

Regarding the loss function, the discriminator shows a lower loss compared to figure 4.2, which is consistent with the increase in the *ndf* parameter. However, the generator's loss increases, suggesting an imbalance between the strengths of the two modules. This indicates

a problem, as the goal is to maintain a balance between them, allowing for a dynamic and beneficial interaction during training.

Decrease *ndf*

Here, we aim to reduce ndf to weaken the discriminator, making it easier for the generator to deceive it.

Weakening the discriminator may compromise the core principle of GANs, which relies on the adversarial relationship between the generator and discriminator.

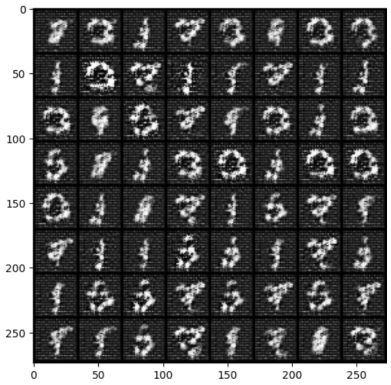

Figure 4.6: Digits results - GAN - ndf decreased

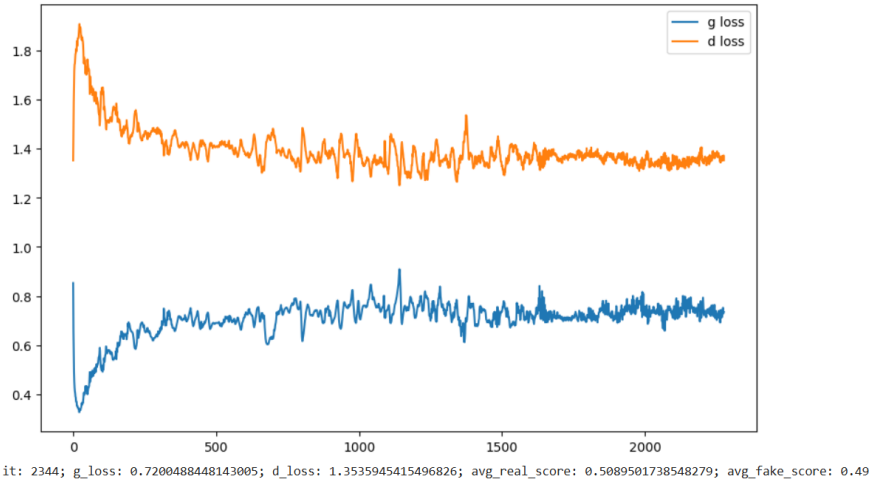

Figure 4.7: GAN training - ndf decreased

We can observe 4.6 that reducing the ndf (number of filters in the discriminator explained above) in a GAN leads to lower-quality image generation by the generator. In fact, while this might temporarily simplify the generator's training, it results in poorer overall performance, as the generator fails to improve adequately without effective adversarial feedback from the discriminator.

Regarding the loss curves 4.7, it can be observed that both g_loss and d_loss are very high and far apart, indicating a model that is not robust and performs poorly. These curves show the imbalance between the two modules, which often leads to less realistic and diverse outputs, consistent with the generated digits. This highlights the importance of maintaining a balanced adversarial dynamic between the generator and the discriminator.

Replace the custom weight initialization with Pytorch's default initialization

Based on the results presented in Figure 4.2 and in comparison to Figure 4.9, it can be concluded that custom weight initialization plays a critical role in stabilizing GAN training, leading to a smoother discriminator learning curve and preventing imbalances between the generator and discriminator. Indeed, this conclusion is supported by Figure 4.9, which illustrates that the default PyTorch initialization results in instability, characterized by fluctuating training dynamics, suboptimal generator performance, and a noticeable decline in the quality of the

generated images. This underscores the importance of tailored initialization in enhancing both training stability and the fidelity of the generated outputs.

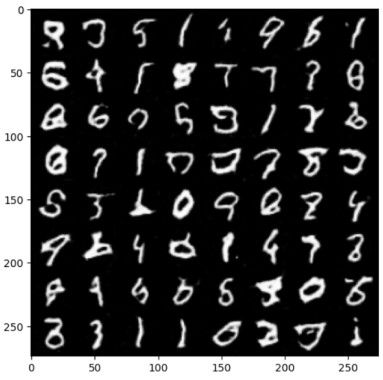

Figure 4.8: Digits results - GAN - Python's weight init

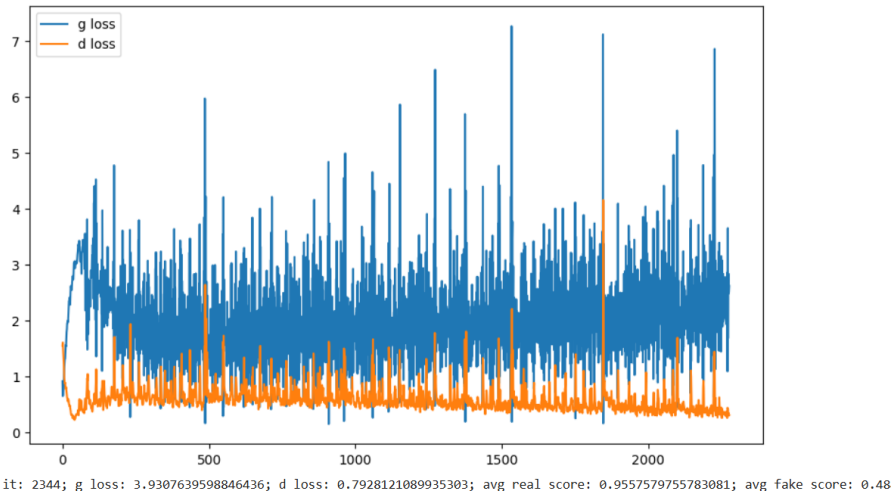

Figure 4.9: GAN training - Python's weight init

Change the learning rate

- **Increase *lr_d*** : With an increased learning rate for the discriminator ($lr_d = 0.02$), it becomes more efficient at classifying generated images as fake, which is reflected in the low loss value. However, this can make it too dominant, leading to difficulties for the generator in progressing, although the curves remain generally stable. This reflects a slight imbalance in favor of the discriminator.

- **Increase *lr_g*** : When the learning rate of the generator is increased ($lr_g = 0.05$), the curves become very unstable, particularly at the beginning of the training. This phenomenon can be explained by an overly aggressive exploration of the parameter space, preventing the generator from converging properly. The discriminator loses efficiency, reflecting a marked imbalance where the generator struggles to produce high-quality results.

- **Increase both *lr_d* and *lr_g*** : With high learning rates for both networks ($lr_d = 0.02$ and $lr_g = 0.05$), training remains unstable, but the balance between G and D is maintained. This can be explained by synchronization in their respective progressions, with each network adapting quickly to the other's changes. However, this instability limits the quality of the final generations and may reflect excessive competition between the two networks.

Increase the number of epochs

When we extend the training of a DCGAN excessively, we observe a phenomenon of progressive divergence between the losses of the generator (loss_g) and the discriminator (loss_d), indicating an increasing imbalance between the two networks.

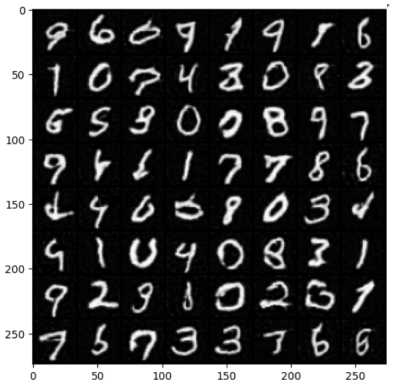

Figure 4.10: Digits results - GAN - Increase the number of epochs

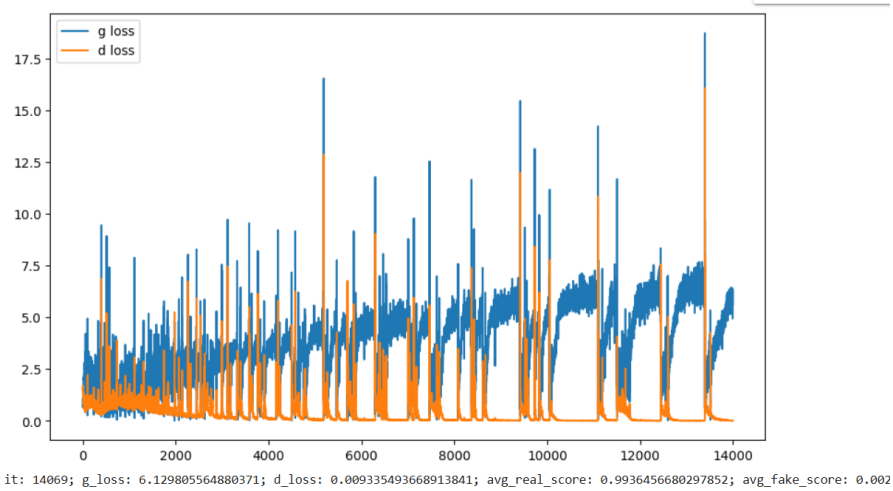

Figure 4.11: GAN training - Increase the number of epochs

Here (see Figure 4.11), after approximately 1500 iterations, the generator over-learns to deceive the discriminator, resulting in a very low loss for D and a high loss for G. Although the generated digits remain discernible, some digits (such as 0 and 4) become underrepresented, indicating that the generator is exploring the latent space less effectively. This imbalance is likely due to the saturation of the discriminator, which consistently classifies generated images as fake without actually improving its ability to generalize (mode collapse).

Increase *nz*

Increasing the noise vector size (n_z) from 100 to 1000 dimensions did not improve the quality of the generated images, suggesting that simply expanding the input noise space does not inherently lead to better GAN outputs.

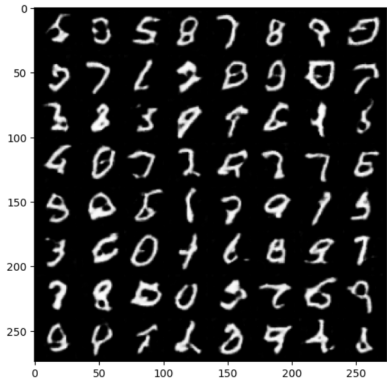

Figure 4.12: Digits results - GAN - Increase nz

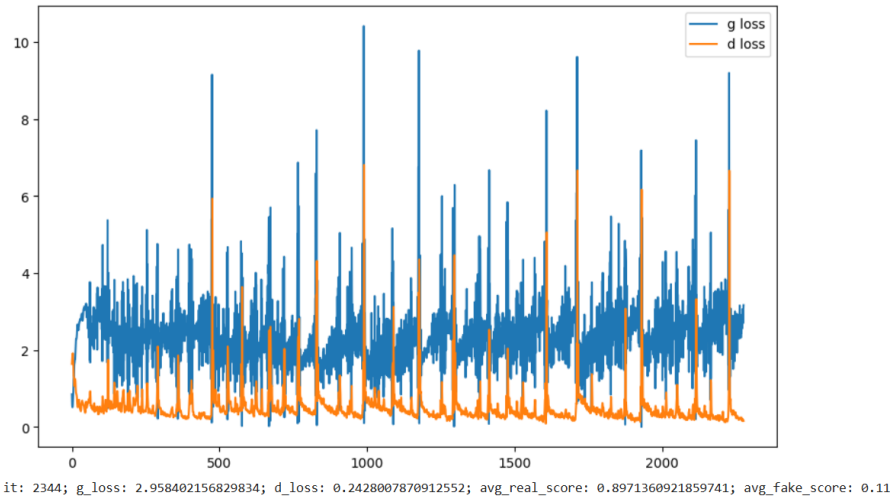

Figure 4.13: GAN training - Increase the nz

One possible explanation is that a higher-dimensional noise space increases the complexity of the mapping task for the generator, which may struggle to effectively leverage this added dimensionality, particularly with a relatively simple architecture like ours. This emphasizes the

importance of designing generator architectures and training procedures that can utilize larger input spaces effectively, typically through empirical experimentation; such as grid search).

Applying Linear Interpolations

The linear interpolation experiment between two latent vectors on the MNIST dataset reveals insights into the structure of the learned latent space.

Figure 4.14: Digits results - α=0

Figure 4.15: Digits results - α=0.5

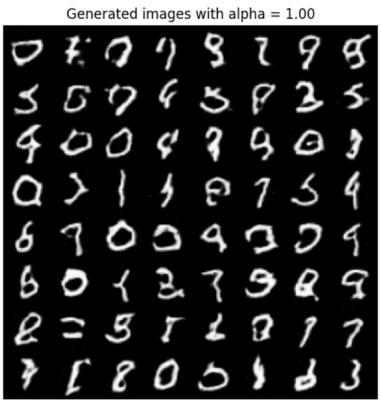

Figure 4.16: Digits results - α=1

As α varies from 0 to 1 (see more examples in the notebook), the generated images are expected to transition smoothly between the two latent representations, reflecting how the model encodes and generalizes features across the space. For instance, transitions between digits (e.g., from a "8" to an "9") or variations in style (such as thickness or orientation) highlight the generator's ability to capture meaningful relationships within the data.

Note that if the interpolations appear coherent and realistic, this suggests that the latent space is well-structured and continuous. Conversely, irregular transitions or unrealistic outputs could indicate gaps in the model's learning or limitations in its ability to generalize. This experiment provides a valuable qualitative assessment of the generator's capacity to produce diverse and plausible outputs while navigating the latent space.

4.2 Conditional Generative Adversarial Networks (cGANs)

Generative Adversarial Networks (GANs) can be extended to conditional models by conditioning both the generator and discriminator on an additional factor, y. For example, in MNIST, this could be the class label, while in facial image generation, attributes like glasses or hair color can be used. Conditional Generative Adversarial Networks (cGANs) allow for more controlled generation, useful in applications like image completion and style transfer.

In this section, we focus on the cDCGAN variant, where the generator $cG(z, y)$ generates images based on a noise vector z and attribute vector y. The discriminator $cD(x, y)$ distin-

guishes between real and generated images, using branching and concatenation techniques to incorporate label information effectively into the generation and classification process.

Q6. Comment on your experiences with the conditional DCGAN.

The digits generated by cGANs exhibit significantly better quality compared to those produced by standard GANs as we can see in Figure 4.17. This improvement can be attributed to the additional conditioning information provided to the cGAN, which helps the generator learn the unique characteristics of the training data, leading to more realistic outputs. The core hypothesis is that conditioning on the class label allows the cGAN to generate more authentic representations of individual digits, as it gains a clearer understanding of each digit's features.

Moreover, the diversity in the generated digits is more pronounced, likely because the cGAN has access to richer information (the data distribution, further supported by the labels), resulting in a broader and more varied set of outputs.

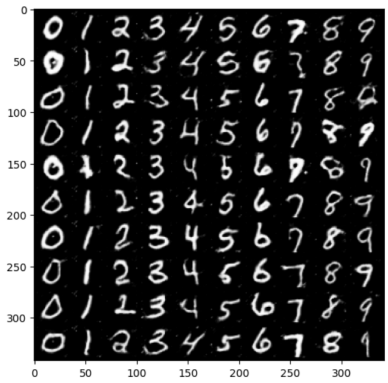

Figure 4.17: Digits results - cGAN

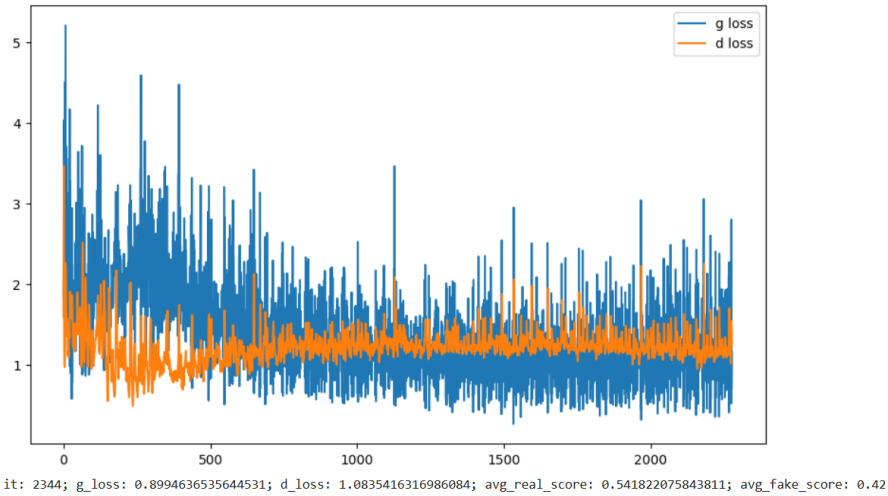

Figure 4.18: cGAN training

As shown in Figure 4.18, the cGAN's loss curve is notably more stable, with fewer significant spikes and a steadier convergence, indicating a more consistent and effective learning process.

Q7. Could we remove the vector y from the input of the discriminator (so having cD(x) instead of cD(x,y))?

Removing y from the discriminator essentially transforms the cGAN into a classic GAN, while keeping an unnecessary redundancy in the generator's input. Such a modification could be explored in exploratory contexts, but it goes against the main objectives of a cGAN.

The key role of y is to ensure that the discriminator does not simply evaluate the realism of the samples, but also verifies their adherence to the specified condition. Without y, the discriminator becomes blind to the contextual information, breaking the conditional learning process. The generator, even though it still receives y as input, would lose critical supervision,

as it is no longer constrained to produce samples that are consistent with y. This could lead to two phenomena: on the one hand, a degradation of the quality of the conditioned samples, and on the other, a potential bias in the generator towards generic outputs that are realistic but disconnected from the conditions.

Q8. Was your training more or less successful than the unconditional case? Why?

To evaluate whether the training was more or less successful than the unconditional case, we observe (see Figure 4.18) that the cGAN exhibits more stable and structured training dynamics, primarily due to the guidance provided by the conditioning vector y.

While the unconditional GAN must learn to model the entire data distribution without context—leading to instability and slower convergence—the cGAN leverages y to narrow the generative task to specific sub-distributions. This simplifies optimization, resulting in steadier loss evolution, better generator-discriminator alignment, and superior generation of targeted outputs.

However, the improvement depends on the quality of y and the model's robustness. Poorly chosen / biaised conditions can undermine these benefits, highlighting the importance of careful design. Overall, cGANs outperform unconditional GANs for targeted generation tasks.

Q9. Test the code at the end. Each column corresponds to a unique noise vector z. What could z be interpreted as here?

The experiment illustrates how variations in the noise vector z influence the diversity of data generated by the Conditional GAN.

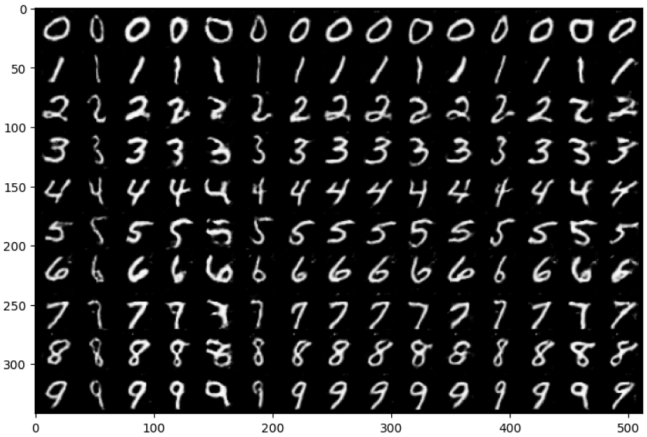

Figure 4.19: cGAN Testing

Each column in the resulting grid corresponds to a unique noise vector, with all digits in the same column sharing the same z but having different class labels. This setup demonstrates

the generator's ability to produce distinct instances of the same class based on variations in z, showcasing its capacity to sample from the latent space.

The noise vector z can be interpreted as the underlying latent variable that governs the appearance of the generated digits, enabling subtle variations within the same class. By altering z, the generator navigates through different representations of the same number, illustrating its potential to create a wide range of plausible and diverse outcomes from the same conditioning input.

This highlights the critical role of z in controlling the diversity of generated data and emphasizes the generator's ability to produce varied instances of each digit.

Conclusion

This chapter explored Generative Adversarial Networks (GANs), detailing their architecture, training dynamics, and applications in data generation. By examining the interplay between the generator and discriminator, we demonstrated how adversarial learning fosters the creation of realistic synthetic data. Additionally, we introduced Conditional GANs (cGANs), which enhance control over generated outputs by incorporating additional input conditions. Through experiments with both GANs and cGANs, the chapter illustrated their potential for creative tasks and highlighted the challenges of stability and diversity in generative models.