M2 M2A

Report for RDFIA Homework

Section 2
Deep learning applications

Students :
Kai MA, Ruyi TANG, Jinyi WU

December 19, 2024

Contents

2-a: Transfer Learning

In this section, we will use the VGG16 Network pre-trained on the *ImageNet* dataset and attempt to transfer it to the *15 Scene* dataset for prediction. First, we extract features using the pre-trained model, and then using linear SVM on the extracted features to obtain classification results.

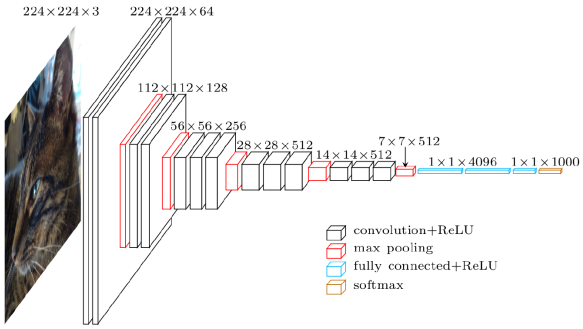


Figure 1: VGG16 Networks

1 VGG16 Architecture

As shown in Figure 1, VGG16, as the name suggests, consists of 16 weight layers, including 13 convolutional layers and 3 fully connected layers, excluding pooling and activation layers. For better understanding, the structure of VGG16 can be divided into 5 blocks, as clearly demonstrated in the following code:

```
class VGG16(nn.Module):
    def __init__(self, num_classes=1000):
        super(VGG16, self).__init__()
        self.features = nn.Sequential(
            # Block 1
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 2
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
```

```
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 3
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 4
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Block 5
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Q1* Knowing that the fully-connected layers account for the majority of the parameters in a model, give an estimate on the number of parameters of VGG16 (using the sizes given in Figure 1).

Parameter Calculation Formula:

$$\text{Conv Parameters} = (\text{Kernel size} \times \text{Input Channel} + 1) \times \text{Output Channel}$$
$$\text{FC Parameters} = (\text{Input size} + 1) \times \text{Output size}$$

The detailed calculation is as follows:

Layer Name	Input Channel	Kernel Size	Output Channel	Parameter Number
Conv1-1	3	3×3	64	1,792
Conv1-2	64	3×3	64	36,928
Conv2-1	64	3×3	128	73,856
Conv2-2	128	3×3	128	147,584
Conv3-1	128	3×3	256	295,168
Conv3-2	256	3×3	256	590,080
Conv3-3	256	3×3	256	590,080
Conv4-1	256	3×3	512	1,180,160
Conv4-2	512	3×3	512	2,359,808
Conv4-3	512	3×3	512	2,359,808
Conv5-1	512	3×3	512	2,359,808
Conv5-2	512	3×3	512	2,359,808
Conv5-3	512	3×3	512	2,359,808

Table 1: Number of parameters in convolutional layers of VGG16

Layer Name	Input Size	Output Size	Parameter Number
FC1	25,088	4,096	102,764,544
FC2	4,096	4,096	16,781,312
FC3	4,096	1,000	4,097,000

Table 2: Number of parameters in fully connected layers of VGG16

Total Number of Parameters:

$$14,714,688 + 119,572,864 = 138,357,544$$

In summary, VGG16 has approximately 138 million 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 1000, it represents to the number of class in the ImageNet Dataset.

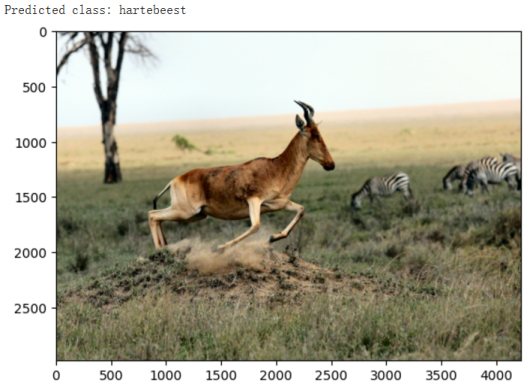
Q3* Apply the network on several images of your choice and comment on the results.

- **What is the role of the ImageNet normalization?**
- **Why setting the model to eval mode?**

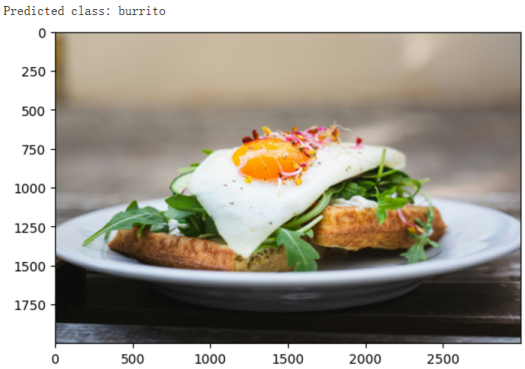
We applied the VGG16 network on several images, the results is as follows:

(a) Cat → Egyptian cat

(b) Dog → Wire-haired fox terrier



(c) Antelope → hartebeest



(d) Plate → burrito

Figure 2: Results on several images

VGG16 performs well in distinguishing general categories in images but may have some errors in finer details, such as differentiating between breeds of cats and dogs or identifying specific subcategories of objects. Therefore, further improvements are needed for VGG16 to handle classification tasks on external images more accurately.

ImageNet Normalization: Since the image sizes we selected vary, it is necessary to preprocess the images using ImageNet normalization. This ensures that the pixel values of the input images align with the distribution of the training data, which also helps improve the model's generalization ability and robustness. Specifically, the images are first resized to (3, 224, 224) and scaled to the range [0, 1], followed by normalization using the mean and standard deviation of ImageNet.

model.eval(): Setting the model to evaluation mode disables the dropout layers and ensures the Batch Normalization (BN) layers use global mean and variance, thereby guaranteeing stability and consistency during inference.

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

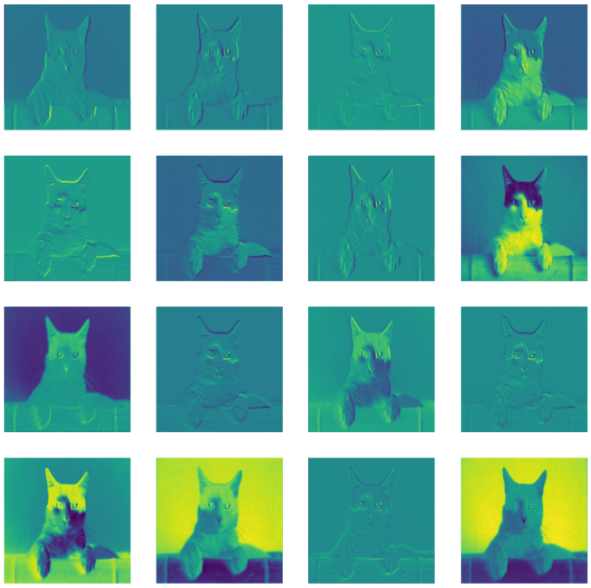


Figure 3: Activation Maps

We visualized the first 16 activation maps of the first convolutional layer. It can be observed that after one convolution operation, each kernel extracts different features from the input image and generates an activation map. Through these features, the network can understand the content of the input image, such as edges, shapes, and textures.

2 Transfer Learning wtih VGG16 on 15 Scene

In the previous section, we understood the structure and working principles of VGG16. Now, we aim to improve its classification performance on external images (15 Scene). To achieve this, we need to perform transfer learning.

2.1 Approach

Transfer learning consists of two steps: first, using a pretrained network for feature extraction, and then training a classification neural network with the extracted features. We will use the output of the relu7 layer in VGG16 as the result of feature extraction and employ an SVM classifier to complete the classification task for the 15 Scene dataset.

Q5* Why not directly train VGG16 on 15 Scene?

VGG16 has over 138 million parameters, while the 15 Scene dataset contains only 4,485 samples. The training set is too small, so directly training VGG16 on the 15 Scene dataset could lead to overfitting.

Q6* How can pre-training on ImageNet help classification for 15 Scene?

The early layers of convolutional neural networks typically learn general features such as edges, textures, and shapes. These features are universal across different images and can be directly applied to the 15 Scene dataset using the features learned

from ImageNet. This avoids redundant training, effectively prevents overfitting, reduces computational costs, and accelerates convergence.

Q7 What limits can you see with feature extraction?

Feature extraction can have several limits:

- Feature extraction heavily relies on the features learned from the pretraining dataset. If there is a significant difference between the target dataset and the pretraining dataset, the performance may be suboptimal. For example, ImageNet primarily consists of natural scenes, so if the target dataset involves satellite images or medical images, it may fail to extract effective features.
- Feature extraction with a pretrained model fixes the model parameters, making it impossible to optimize them based on the target dataset. This limitation can prevent the model from achieving optimal performance.

2.2 Feature Extration with VGG16

To implement feature extraction, we defined a new class VGG16relu7 to copy all layers of VGG16 up to relu7, effectively removing the last fully connected layer.

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

- Lower layers (closer to the input) extract more general features that are applicable to various types of images and have lower computational costs; however, their expressive power is limited, making them incapable of capturing higher-level information.
- Higher layers (closer to the output) extract richer semantic information and perform better on data that is highly similar to the training data; however, they have weaker adaptability and typically require higher computational costs.

Thus, in practical applications, the choice of feature extraction layers should consider the similarity between the target data and the pretrained data, as well as the available computational resources.

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

The images in the 15 Scene dataset do not meet the input requirements of VGG16, so preprocessing is necessary. To ensure the input images are in RGB format, we duplicate the single channel of grayscale images to expand them into a pseudo-RGB three-channel format, then convert them into Pillow image objects. Next, we normalize the pixel values of the images. Finally, we use transform.Compose to combine the preprocessing steps.

```
def duplicateChannel(img):
    img = img.convert('L')
    np_img = np.array(img, dtype=np.uint8)
    np_img = np.dstack([np_img, np_img, np_img])
```

```
        img = Image.fromarray(np_img, 'RGB')
        return img
    def resizeImage(img):
      return img.resize((224,224), Image.BILINEAR)

    # Add pre-processing
    train_dataset = datasets.ImageFolder(path+'/train',
        transform=transforms.Compose([
            transforms.Lambda(duplicateChannel),
            transforms.Lambda(resizeImage),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std
                =[0.229, 0.224, 0.225])
        ]))
    val_dataset = datasets.ImageFolder(path+'/test',
        transform=transforms.Compose([
            transforms.Lambda(duplicateChannel),
            transforms.Lambda(resizeImage),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std
                =[0.229, 0.224, 0.225])
        ]))
```

2.3 Training SVM classifiers

Next, we will use the extracted features to train a multi-class SVM classifier and evaluate its performance.

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

Theoretically, it is possible to use only a neural network to perform classification tasks by simply adding a fully connected layer after the frozen CNN. By adding new convolutional layers, the neural network can further optimize the extracted features and improve the model's accuracy. However, this approach increases computational costs and carries the risk of overfitting, especially with small-scale datasets. In contrast, using an SVM classifier can work efficiently and stably with fixed extracted features.

2.4 Going further

Q11 For every improvement that you test, explain your reasoning and comment on the obtained results.

There are several methods to improve the model, and we have tried some of them and created a bar chart to compare the accuracy of different models:

- **Change the layer at which the features are extracted.** We defined VGG16relu6, which uses the relu6 layer and all preceding layers of VGG16 for

feature extraction. The output shape remains as (batch size, 4096), but with one less fully connected layer compared to VGG16relu7. This allows the model to focus more on general features, enhancing its generalization ability on new data, which leads to better performance.

- **Try other available pre-trained networks.** We used ResNet as the pre-trained model and trained it on the 15Scene dataset, finding that it slightly outperformed VGG16.
- **Tune the parameter C to improve performance.** We adjusted the SVM parameter C to 10, but the difference compared to the initial value C=1 was minimal.

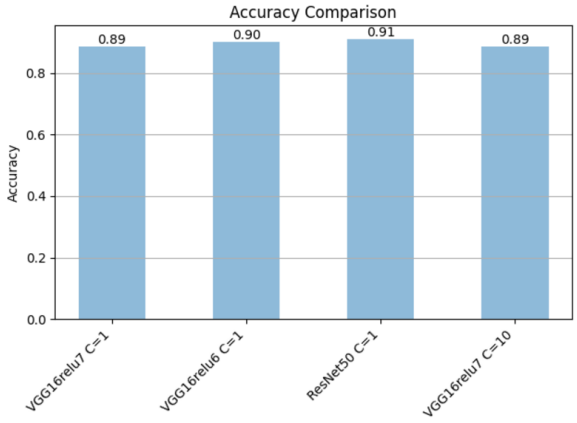


Figure 4: Comparison of accuracy

3 Conclusion

In this chapter, we studied the framework of VGG16 and applied the pre-trained VGG16 model on ImageNet to train on the 15Scene dataset through transfer learning. We also explored different optimization methods, such as adjusting the feature extraction layers, selecting different pre-trained models, and tuning the classifier parameters.

The key points of transfer learning are as follows:

1 Ensure that the format of the target data matches the format of the pre-trained data, making normalization an essential step.

2 During the feature extraction stage, it is important to choose the appropriate extraction layer to balance model performance and generalization ability. Additionally, ensure that the extracted feature dimensions are compatible with the classifier for training.

2-b: Visualizing Neural Networks

Neural network visualization involves techniques to analyze the internal mechanisms of neural networks (e.g., weights, activations, and gradients) to interpret their behavior.The objective is to : understand how the network processes input features, identify the regions the network focuses on during decision-making and assess the model's robustness and performance (e.g., adversarial examples).

This section aims to study techniques for analyzing the behavior of convolutional neural networks (CNNs) by computing the gradient of the input image with respect to the output of a specific class.

We use the **pre-trained SqueezeNet model** (trained on ImageNet), freeze its weights, and modify the input image to indirectly explore the network's behavior.Gradients and image generation techniques are used to observe how the model reacts to different features. And in this section we introduce : **Saliency Map**, **Adversarial Examples**, and **Visualization of Classes**.

1 Saliency Map

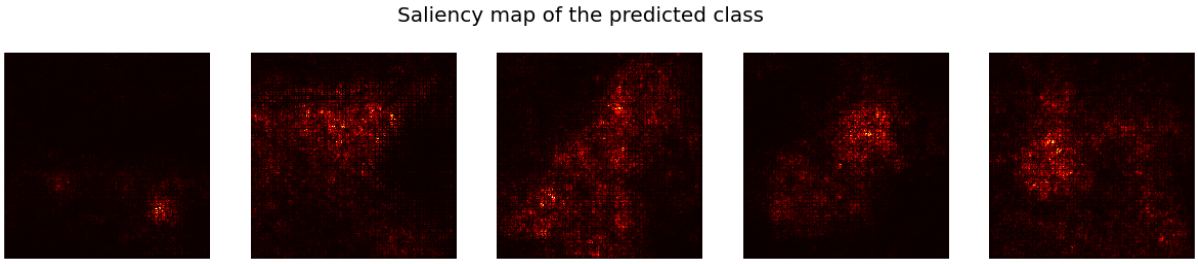
This method is proposed by Simonyan et al.(2014). It focuses on the visualization of the most impactful pixels for predicting the correct class. To produce the maps, it is proposed to approximate the neural network in the neighborhood of image.

Q1 Show and interpret the obtained results.

To better demonstrate and compare the performances of the model with different results of predictions, we show the following saliency maps with the true class and the predicted class.

(a) Original images with probabilities of true label and predicted class

(b) Saliency maps highlighting relevant regions of true class



(c) Saliency maps highlighting relevant regions of predicted class

Figure 5: Visualization of saliency maps using pre-trained SqueezeNet

Generally the saliency maps are consistent with the features shown in the original images. Except for the fourth image with lower accuracy of identifying the true label and making correct prediction, the rest images are shown with high accuracy of prediction, meaning that the model correctly highlights the important pixels/features in the saliency maps, making it easy to recognize the images.

And as for the exception of the fourth image, the model fails to predict "Border Terrier" correctly, but identify it as "Leonberg", which is also under the species of dogs. So in this case, the model fails to focus on the different facial features and outlines of the object with the highlighted area.

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

From the previous results, the saliency maps seem to work well as a high-lighter to visualize which part of the picture influences the model's decision. However we also observe that the model may be confused and unable to identify or distinguish, which leads to questions about the liability of saliency maps.

The limits of This Technique can be concluded as the following:

1. **Lack of Robustness:** The reliance on arbitrary reference points (e.g., black images in IG) can lead to non-invariant results, making the attributions sensitive to input shifts or noise.
2. **Susceptibility to Manipulation:** Saliency maps can be easily altered using imperceptible adversarial changes, reducing their trustworthiness as diagnostic tools.
3. **Incomplete Interpretations:** Saliency methods often fail to provide a holistic understanding of complex interactions among input features, limiting their utility for debugging or decision justification.

And in the paper by Kindermans et al.(2017), the researchers evaluate the robustness and interpretability of saliency maps, highlighting two main findings:

- **Input Invariance**: Many saliency methods, like Integrated Gradients (IG) and Deep Taylor Decomposition (DTD), fail to maintain consistency under input transformations like constant shifts. For instance, the attributions depend on the choice of reference points, which are arbitrary and can lead to inconsistent results.
- **Adversarial Sensitivity**: Saliency maps are vulnerable to adversarial perturbations, which can drastically alter visual explanations without changing model predictions. This undermines their reliability for understanding model decisions.

These findings emphasize the need for developing more robust and interpretable saliency methods that are invariant to input shifts and resistant to adversarial attacks.

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

Saliency maps can also be used for other purposes beyond network interpretation.

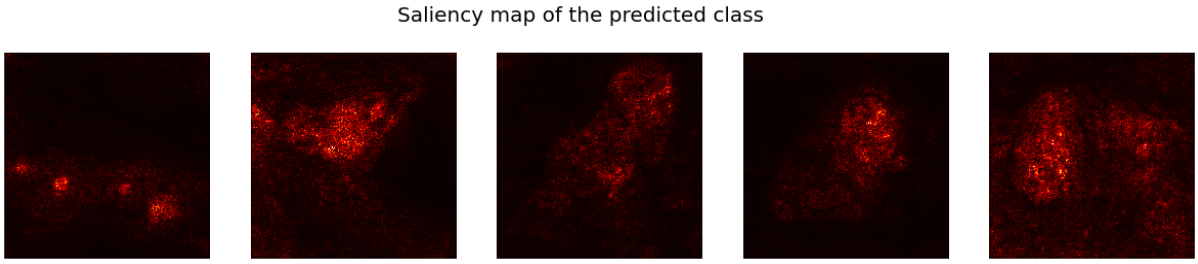
1. **Data Augmentation:** From Simonyan et al.(2014), by identifying the most critical regions for a task, saliency maps guide targeted data augmentation. For instance, focusing transformations like cropping or rotation on these regions preserves relevant features, enhancing model training efficiency.
2. **Style Transfer and Image Generation:** From Liu et al.(2019) Saliency maps assist in style transfer tasks by ensuring the content of significant regions is preserved while applying the style of another image. This improves the quality and meaningfulness of generated images.
3. **Feature Selection:** From Shrikumar et al., 2017 (DeepLIFT)., in tasks involving tabular or structured data, saliency maps can help select the most informative features, reducing input dimensionality and making models faster and more interpretable.

These applications highlight the versatility of saliency maps, extending their utility into areas like generative models, and feature optimization.

Q4 Test with a different network, for example VGG16, and comment. To make comparison, we generate the saliency maps using pre-trained VGG16 on the same sample images.

(a) Original images with probabilities of true label and predicted class on VGG16

(b) Saliency maps highlighting relevant regions of true class on VGG16



(c) Saliency maps highlighting relevant regions of predicted class on VGG16

Figure 6: Visualization of saliency maps using pre-trained VGG16

From the results above, we can observe that the VGG16 makes no missclassification with high accuracy. Therefore the saliency maps generated are demonstrated with less noise and clearer indicator to the importance pixels. For example, we can see that the first saliency maps for the "hay" are clearly highlighting the four hays of the corresponding image instead of just one in the precious saliency maps of SqueezeNet. We can also see that the corresponding maps for "quail" and "Tibetan mastiff" are also shown with less noise in the background and clearer outline of the object.

As for the previously missidentified "border terrier", the saliency maps of VGG16 shows more concentration on the facial features with lighter pixels, which contributes to the correct classification this time,

This result demonstrates the robust performance of VGG16 in image classfication with more reliable and consistent saliency maps. VGG16's ability to generate better saliency maps underscores the trade-off between model efficiency (as in SqueezeNet) and feature richness, suggesting that deeper, more expressive models are often better

suited for interpretability tasks, while SqueezeNet is more suited for real-time, low-latency applications where efficiency is critical.

2 Adversarial Examples

In this section we analyze neural networks by generating adversarial examples (or fooling samples), which is to confuse the model with slightly modified images (imperceptible for humans) that lead to missclassfication. The idea is to deceive the model therefore to learn about the limitations and unusual behaviors of the neural network.

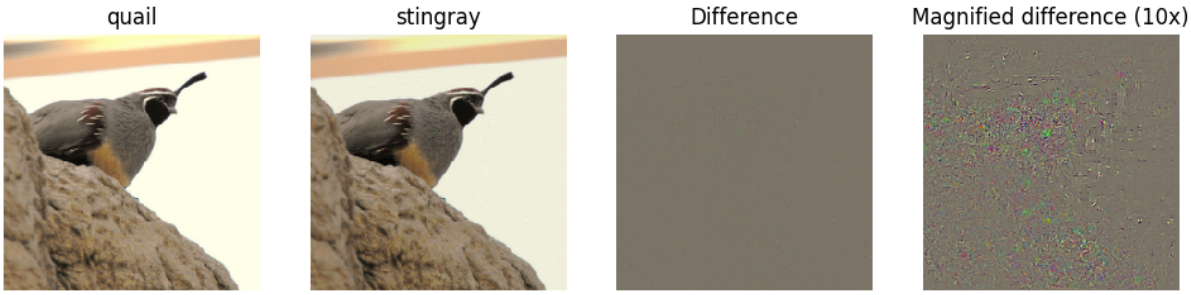
Q5 Show and interpret the obtained results.

To generate a fooling image that the model will classify as the target class we perform gradient back propagation on the score of the target class, stopping until the model is fooled.

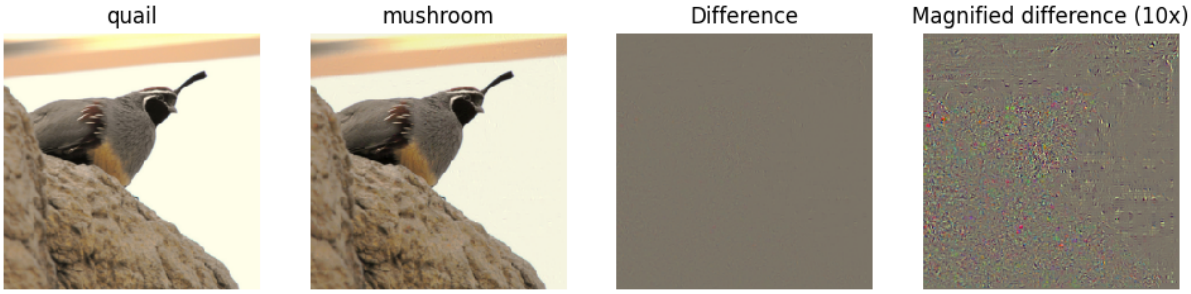
In the following Figure 7, we show the results with the left column presenting the original image of "quail", the second column showing the modified image missclassified as the target class, and the rest columns showing the differences and magnified differences between the original images and the modified images.

Even though it's hard to just visually distinguish the modified images from the original one, the neural network identifies them to be in an different class. And with magnification of the differences, the ones with "stingray" and "mushroom" are presented with more noises compared to the "bee eater".

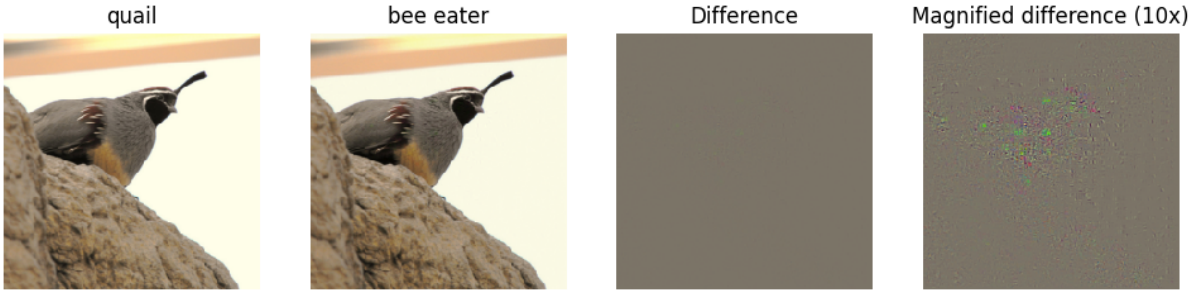
And we also observe from the results of iteration and scores, for the "stingray" and "mushroom", the model is fooled after 14 and 16 iterations, respectively achieving the max score of 59.641 and 52.659, compared with iteration of 6 and max score of 46.023 of the target class "bee eater". The higher scores and number of iterations means it is harder to fool the model with the more distinct targets. Compared with "bee eater" which is also a kind of bird as "quail", "stingray" and "mushroom" are of two entirely different species that results in more obvious differences.



(a) "quail" V.S. "stingray"



(b) "quail" V.S. "mushroom"



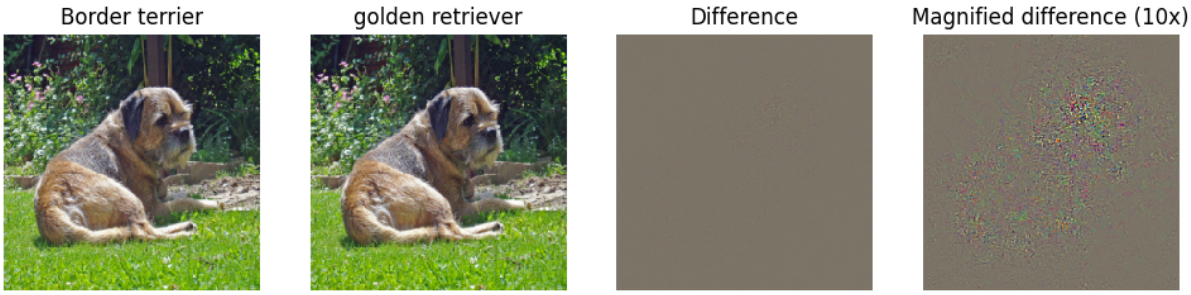
(c) "quail" V.S. "bee eater"

Figure 7: Adversarial examples of "quail" with different target classes

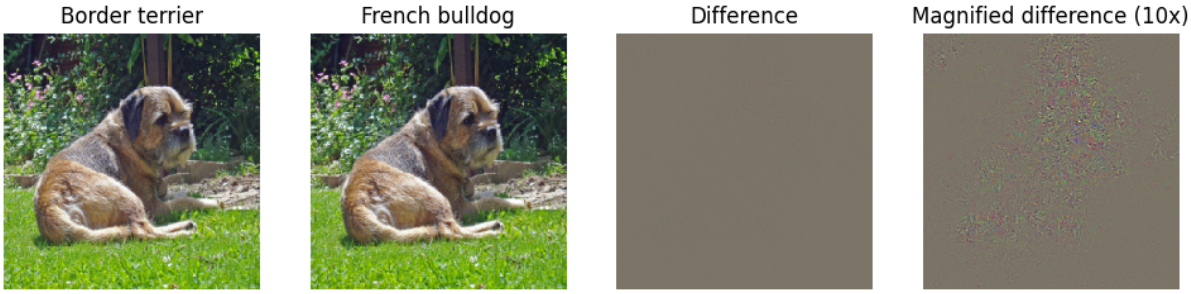
We also test with another input image of "border terrier".

In the following Figure 8, we can see that the Figure 8c has the maginified difference with most noises and the model needs 87 iterations reaches the max score of 74.994 to be fooled. It means that the image of "desk" is evidently distinct from the original "border terrier".

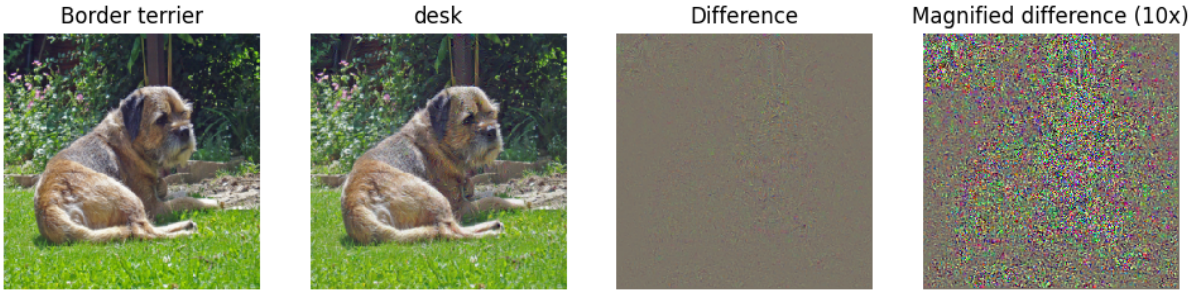
The results of Figure 8a and Figure 8b show that even the target classes "golden retriever" and "French bulldog" are under the same species as the original "border terrier" with smaller differences compared with the last one, the "French bulldog" share more similarities with the original image with less noise in the background of magnified difference, and needs less iteration with smaller max score to fool the model.



(a) "border terrier" V.S. "golden retriever"



(b) "border terrier" V.S. "french bulldog"



(c) "border terrier" V.S. "desk"

Figure 8: Adversarial examples of "border terrier" with different target classes

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

Adversarial examples expose critical vulnerabilities in CNNs, leading to significant practical consequences across various domains.

1. **Security risks**
   (a) **Model Vulnerability**: CNNs are highly susceptible to adversarial attacks, where imperceptible perturbations can mislead models to produce incorrect predictions. In the paper of Szegedy et al.(2014) ,this poses risks in high-stakes applications such as autonomous vehicles, facial recognition, and healthcare diagnostics. For example, Slight modifications to a stop sign image can cause a self-driving car to misclassify it as a speed limit sign, leading to dangerous outcomes.
   (b) **Malicious Exploitation**: According to Goodfellow et al.(2015) attackers can craft adversarial examples to bypass security systems, compromising authentication (e.g., facial ID) or financial fraud detection systems.

2. **Challenges in Interpretability**
   (a) **Unreliable Explanations**: In the paper of Ilyas et al.(2019), adversarial examples exploit features imperceptible to humans but influential to CNNs, complicating efforts to interpret predictions.
   (b) **Debugging Difficulty**: In the studies of Athalye et al.(2018), the presence of adversarial examples makes understanding and diagnosing CNN errors more challenging.
3. **Fairness Implications** Adversarial examples can disproportionately affect certain groups, exacerbating biases in sensitive applications like hiring or healthcare. Obermeyer et al.(2019) discusses how adversarial susceptibility can exacerbate biases in AI systems, especially in sensitive domains like healthcare. A healthcare algorithm could be manipulated that results in over-priority or neglect issues.

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).

Adversarial images constructed naively often lack robustness in real-world settings, as environmental variations (e.g., lighting, viewing angles, and noise) disrupt adversarial perturbations. To address the problem,Athalye et al.(2018) proposes the Expectation Over Transformation (EOT) method that explicitly models these variations by averaging over multiple transformations of the input during adversarial image construction. This ensures the adversarial examples remain effective under physical-world conditions.

To also solve the problem of poor robustness,in the research of Goodfellow et al., 2015. it uses adaptive adversarial training to dynamically generate adversarial examples during model training, ensuring the model learns to counter diverse and evolving perturbations. This reduces vulnerability to both known and unknown attacks.

Adversarial examples created naively are also often overfitted to a specific target model and fail to transfer across different architectures or parameters. To this end, Liu et al.(2017) proposes ensemble-based methods, which generate adversarial examples using multiple models simultaneously. By optimizing perturbations to work across various models, the method improves the transferability of attacks to black-box or unknown models.

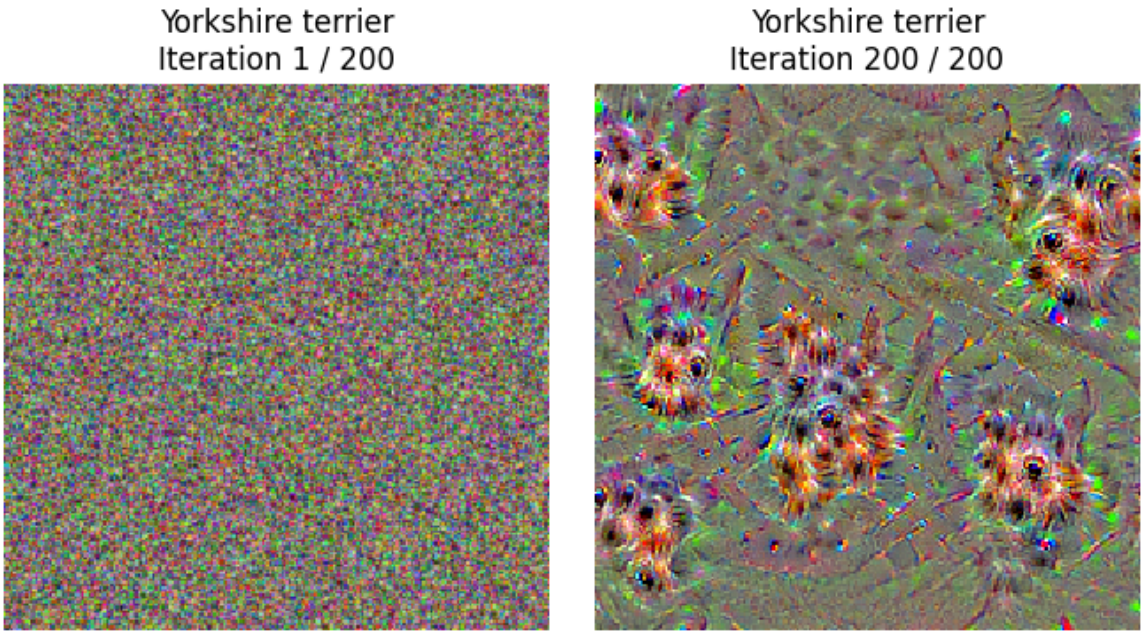
3 Class Visualization

In this section we utilize the method proposed by Simonyan et al. (2014) and developed by Yosinski et al. (2015) to generate images that highlight the type of patters likely to produce a particular class prediction, and so indirectly analyzing what a network prioritizes for classfication prediction.

Q8 Show and interpret the obtained results.

Class visualization is a technique used to understand what a neural network has learned about specific classes by generating synthetic images that maximize the activation of a given class. This approach reveals the patterns or features the network associates with that class, offering insights into how the network processes and recognizes different categories.

We start the experiment by visualizing Yorkshire Terrier from random noise. In the following Figure 9, we show a class visualization for a Yorkshire Terrier after 200 epochs and using default parameters with L2-regularization factor of 10^{-3}, learning rate at 5, bluring every 10 epochs.



(a) initial state (b) last iteration

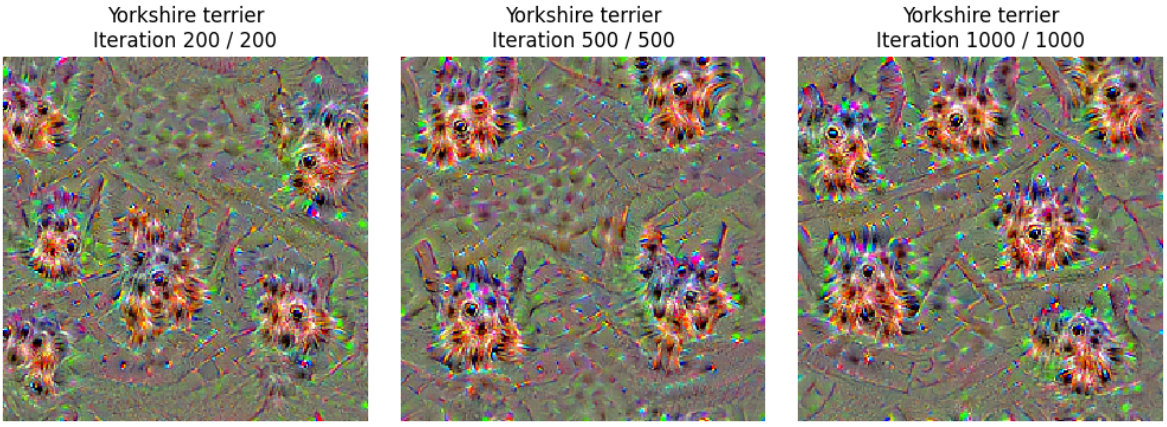
Figure 9: Class visualization: started from random noise, maximizing the score from the Yorkshire Terrier class with default parameters.

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

- **number of iteration:**

  In the following experiments, we are showing results from last iteration with number of iteration at 200,500 and 1000.

  Since the results from Figure 10 are generated from different random noises, so it ends up with different number and rotations of the object. But we can still observe from the results that with the increase of the number of iterations, the surrounding green points that highlight the outline of the multiple objects shown on the figure are brighter. We can also see that the features (the facial features and the fur texture etc.) are more amplified for the network to activate the correct classification neurons in the network.



(a) iteration=200 (b) iteration=500 (c) iteration=1000

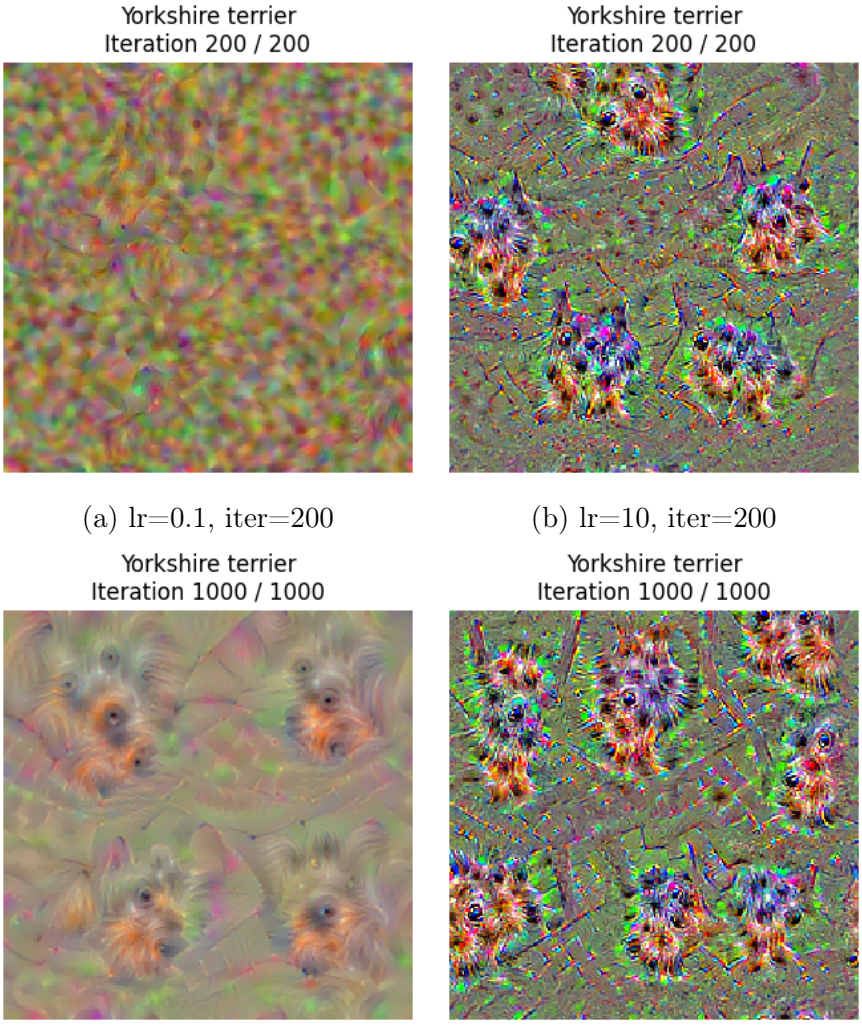
Figure 10: Class visualization of Yorkshire terrier with different number of iterations

- **Learning rate**

  Learning rate controls the step size during the optimization process, where the input image is iteratively updated to maximize the activation of a specific class in the neural network. It has significant impact on the quality and interpretability of the generated visualizations.

  In this part of experiments, we are testing with cases with learning rate as 0.1 and 10.

  From Figure 11, we can see that using the higher learning rate of 10, the generated figures are noisier, and combined with higher iterations the figure becomes harder to identify. But with lower learning rate of 0.1, the generated figure is relatively more smooth. But with smaller iterations, the features of the object are not produced. After compensating with higher iterations at 1000, we can see that with lower learning rate, the objects are shown with more perceptible features, meaning that the lower learning rate allows for more effective convergence and that it should keep the balance between the moderate learning rate and the iterations.



(a) lr=0.1, iter=200 (b) lr=10, iter=200

(c) lr=0.1, iter=1000 (d) lr=10, iter=1000

Figure 11: Comparison of Yorkshire terrier class visualization with different learning rates and with (a) and (b) at 200 iterations, (c) and (d) at 1000 iterations

- **L2-regularization parameter**

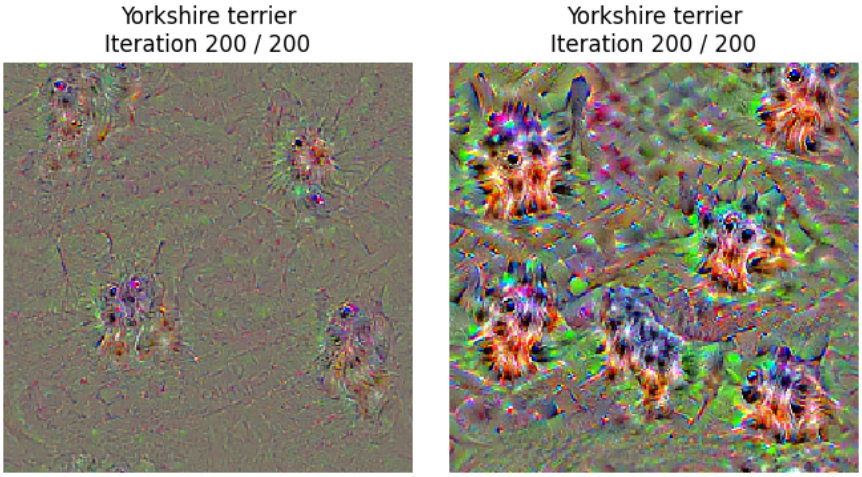
  L2-regularization factor represents the strength of the penalty term added to the optimization objective to limit large pixel changes in the generated image.

  We now experiment with different L2-regularization parameters. And in the following figures in Figure 12 we are presenting results using $1e^{-2}$ and $1e^{-5}$ regularization factor for 200 and 1000 iterations.

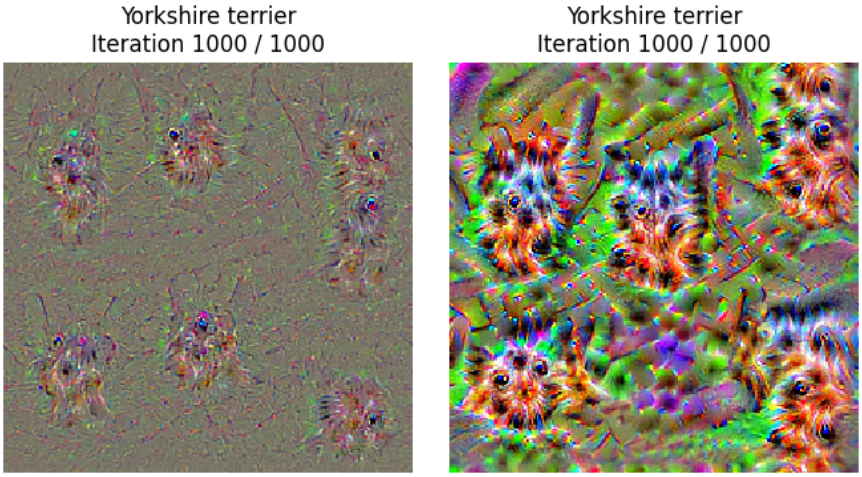
  When the regularization factor is at $1e^{-5}$, the generated images are noisier and contain high-frequency patterns that maximize the class score. We can see that when the iteration hits 1000, the image is demonstrated with more artifacts and irrelevant details. This means that with weak penalty on the pixel changes, the optimization prioritizes maximizing the class score without much regard for the interpretability.

  But when the regularization factor is larger at $1e^{-2}$, the generated visualizations though are less noisy and contain smoother patterns, fail to capture finer

details on the object and the situation is not improved with the increased iterations.



(a) l2-reg=$1e^{-2}$, iter=200 (b) l2-reg=$1e^{-5}$, iter=200



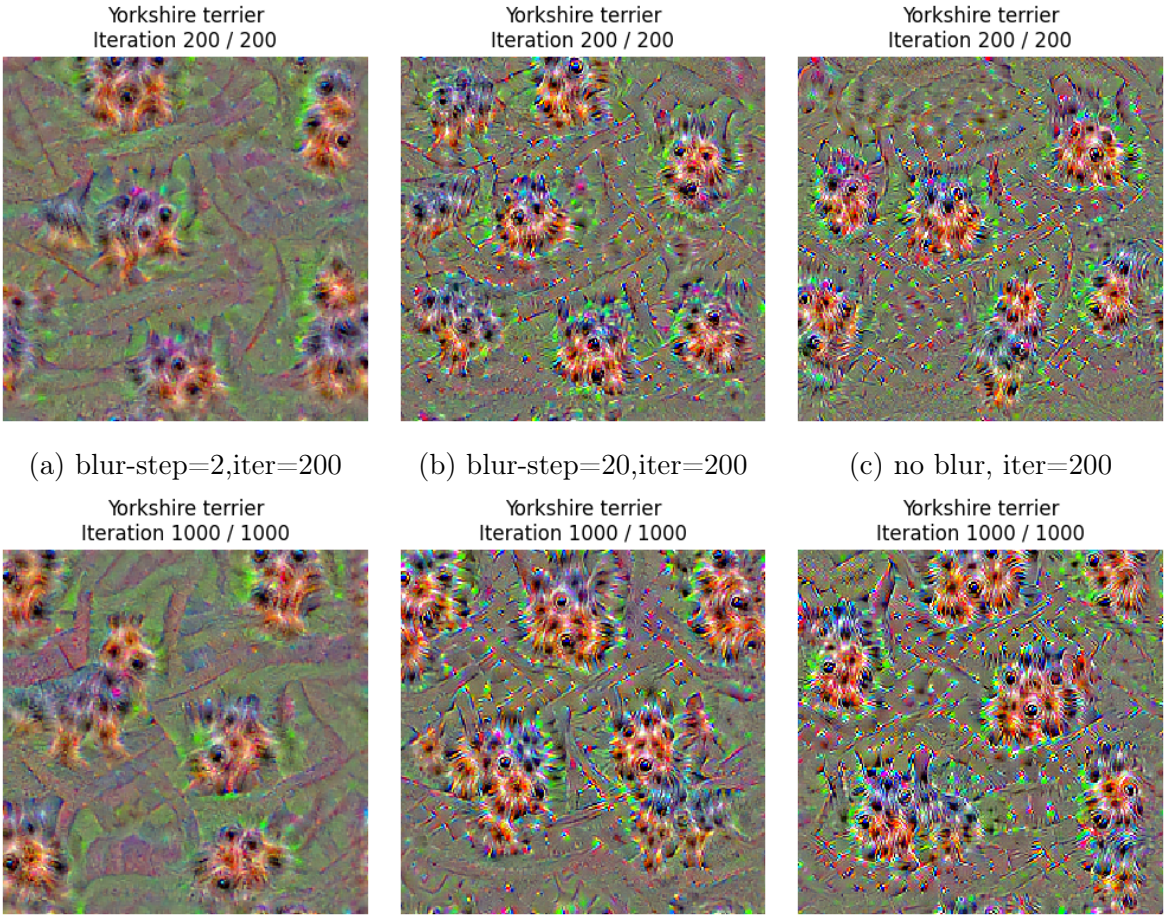
(c) l2-reg=$1e^{-2}$, iter=1000 (d) l2-reg=$1e^{-5}$, iter=1000

Figure 12: Comparison of Yorkshire terrier class visualization with L2-regularization factors and with (a) and (b) at 200 iterations, (c) and (d) at 1000 iterations.

- **blur frequency**

  We also experiment on various blur frequencies. The blur frequency is the process of applying a low-pass filter or smoothing operation to the input image during optimization. It can suppress high-frequency artifacts and focus on broader and more interpretable features of the target class.

  In the following experiments, we test on blurring every two steps blurring every 20 steps, and no blur at all.



(a) blur-step=2,iter=200 (b) blur-step=20,iter=200 (c) no blur, iter=200

(d) blur-step=2,iter=1000 (e) blur-step=20,iter=1000 (f) no blur, iter=1000

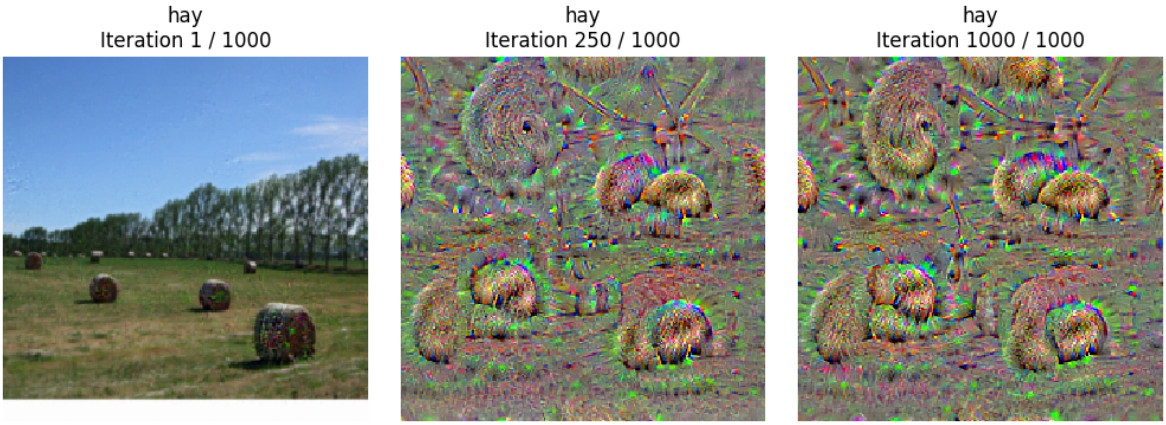
Figure 13: Class visualization of Yorkshire terrier with different blurring frequency

From Figure 13 we can see that with smaller blurring step at 2 which equals to higher blurring frequency results in smoother generated visualizations, the performance is slightly more explicit with higher iterations. And with larger blurring step at 20, the visualization balances between the fine details and the large-scale patterns with high-frequency patterns. However the noise still partially affect the visualization. And without blur, the images are highly saturated and appears relatively visually chaotic, they contain noisy and fragmented patterns which are harder to interpret.

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.

In class visualization, the choice of the initial image affects the optimization process and the quality of the generated visualization. Using an image from ImageNet as source image instead of random noise combines the network's learned features with meaningful patterns already present in the source image, resulting in more interpretable visualizations.When the source image already belongs to the target class (e.g. Figure 14), the optimization process starts closer to a solution that maximizes the target class score.Starting from an ImageNet image helps maintain

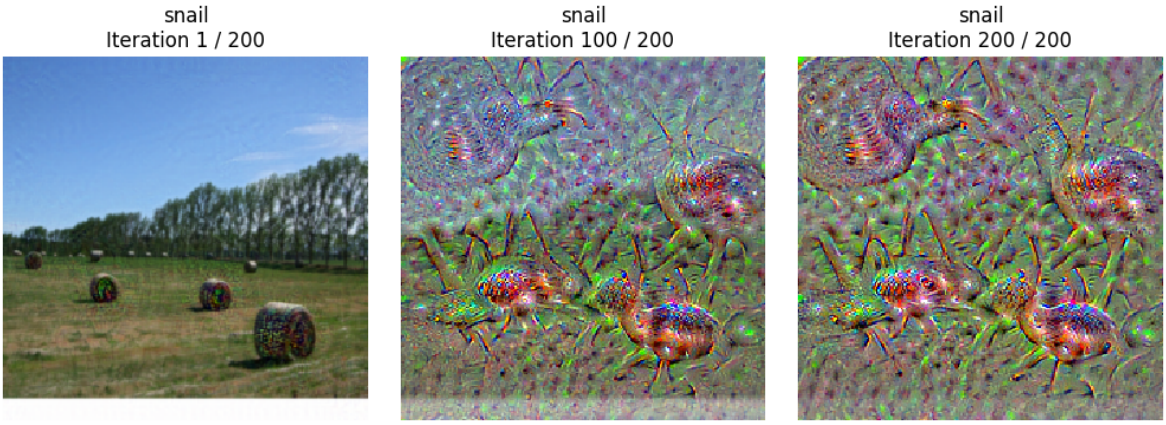
the semantic structure of the input and increases the likelihood of converging to a global or near-global minimum that better represents the target class.



(a) initial state (b) iteration=250 (c) last iteration

Figure 14: Class Visualization using SqueezeNet: starting from an image of the same class as the target class

We also experiment on initiating the image from ImageNet with different target class. In the following images Figure 15 we create a snail class visualization starting from the image of hays.



(a) initial state (b) iteration=100 (c) last iteration

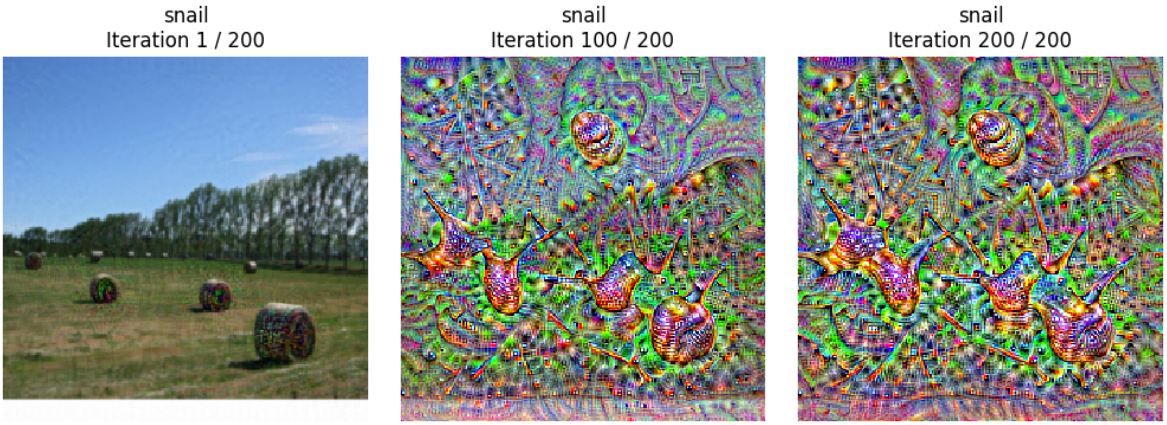
Figure 15: Class Visualization using SqueezeNet: starting from an image of hays from ImageNet with target class as snail

Q11 Test with another network, VGG16, for example, and comment on the results.

To compare the performances of different networks, we do some identical experiments for the network of VGG16.

To compare with Figure 15, we do the same experiment on VGG16 network. VGG16 network has more layers compared to SqueezeNet, making it significantly deeper to

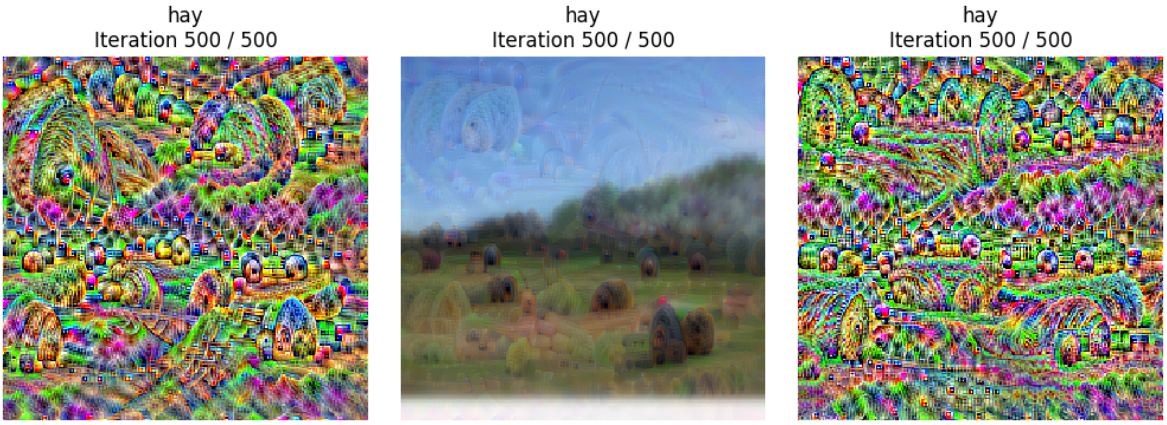
capture more complex and hierarchical features which we can see from the patterns in Figure 16.



(a) initial state (b) iteration=100 (c) last iteration

Figure 16: Class Visualization using VGG16: starting from an image of hays from ImageNet with target class as snail

In the following experiments, we repeat our test on different learning rates, regularization factors and blur frequencies. And we test on initiating the image of same class as the target class from ImageNet,



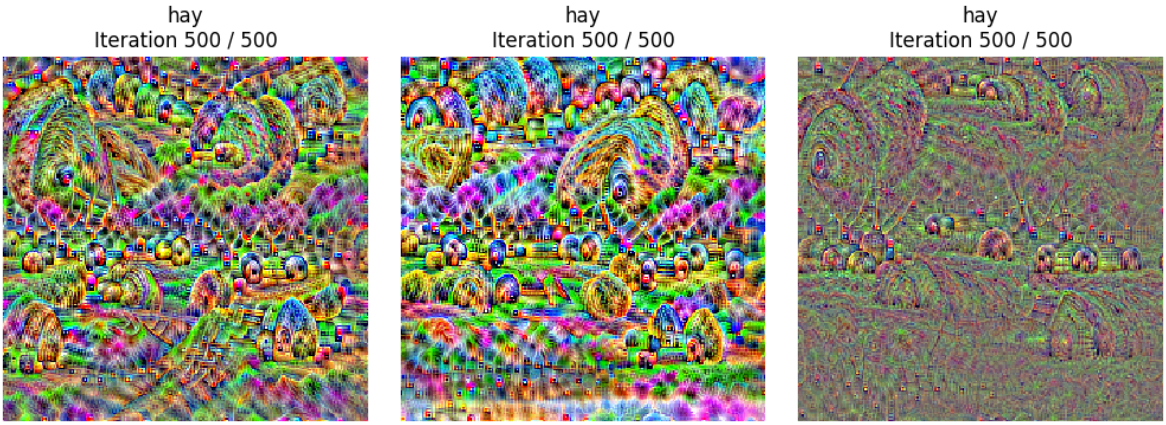
(a) default:lr=5 (b) lr=0.1 (c) lr=10

Figure 17: Class Visualization using VGG16: starting from an image of the same class as the target class with different learning rates.

As can be seen in Figure 17, the differences between figures using various learning rates become more apparent with VGG16. And the fig. 17a is the one using VGG16 with default parameters for comparison. But compared with the result of Figure 11, the patterns shown on fig. 17b is too smooth to observe the features and the pixels are much more saturated with the higher learning rate.

And as for the case with regularization factors, from the following figures we can

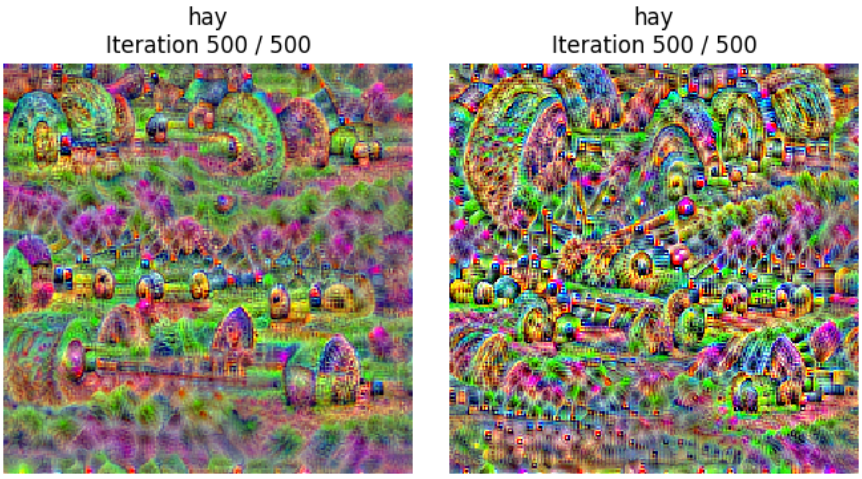
see that with lower regularization factor, it becomes more challenging to obtain the features of the pattern.



(a) default:l2-reg=$1e^{-3}$ (b) l2-reg=$1e^{-5}$ (c) l2-reg=$1e^{-2}$

Figure 18: Class Visualization using VGG16: starting from an image of the same class as the target class with different regularization factors.

And finally for the blurring frequencies, we here compare the default blurring frequency (when blurring step=10) with higher blurring frequency (when blurring step=2):



(a) blur step=2 (b) default: blur step=10

Figure 19: Class Visualization using VGG16: starting from an image of the same class as the target class with different blurring steps.

With higher blurring steps we can get a smoother pattern of the objects, which is the same conclusion as before with SqueezeNet.

But generally we take much longer time to receive results from VGG16 since it is more computationally expensive than smaller models due to its depth and larger number of parameters.

2-c: Domain Adaptation

Domain adaptation tackles the problem of training a model on one dataset (source domain) and ensuring it performs well on a different dataset (target domain) with distinct distributions. In this practical, we will implement **Domain-Adversarial Neural Networks (DANN)**, a method that uses adversarial training to help the model learn domain-invariant feature representations, and analyse how it works in domain adaptation.

1 DANN and the GRL layer

Domain-Adversarial Neural Networks (DANN) are designed to tackle the challenge of domain adaptation, where the goal is to train a model on a labeled source domain while ensuring it performs well on an unlabeled target domain with distinct feature distributions.DANN introduces a novel **Gradient Reversal Layer (GRL)** to achieve domain invariance, enabling the model to bridge the gap between different domains effectively.

1.1 Architecture of the DANN model

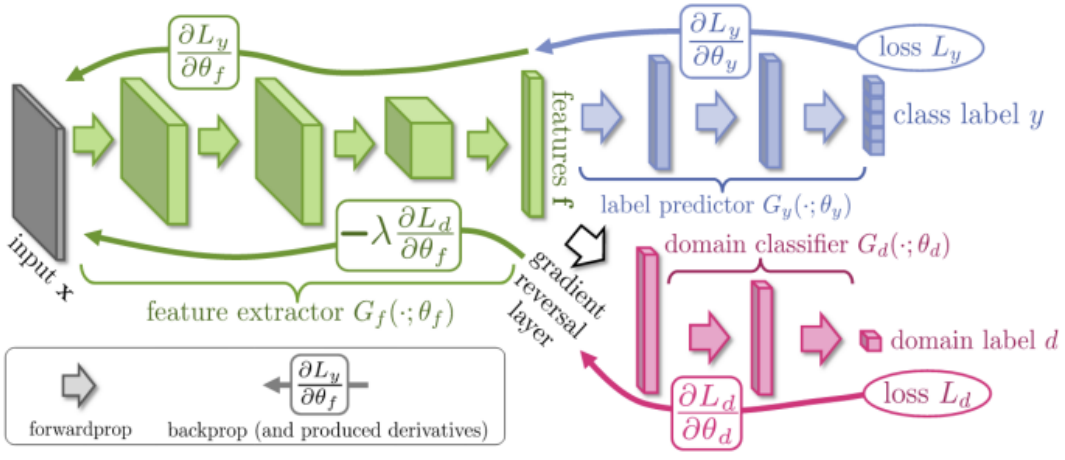


Figure 20: DANN model with its GRL layer

As it is shown in the **Figure 20**, the DANN model consists of three main components:

[HTML]D6F5D6[HTML]FFFFFF **Component**	**Description**
[HTML]D6F5D6[HTML]D6F5D6 Feature Extractor (G_f)	A neural network that learns representati
[HTML]D6F5D6[HTML]D6E6F5 Label Predictor (G_y)	A classifier trained on labeled source doma
[HTML]D6F5D6[HTML]F5D6E6 Domain Classifier (G_d)	A binary classifier that predicts whether source or target domain.

Table 3: Components of the DANN Model

Notably, a **Gradient Reversal Layer (GRL)** is inserted between the feature extractor and the domain classifier.

1.2 General Principle

The training process of DANN simultaneously optimize the loss of Label Predictor (L_y) and the loss of Domain Classifier (L_d).

$$L_y(\theta_f, \theta_y) = \frac{1}{N} \sum_{i=1}^{N} \text{CrossEntropy}(G_y(G_f(x_i; \theta_f), \theta_y), y_i)$$

$$L_d(\theta_f, \theta_d) = \frac{1}{N} \sum_{i=1}^{N} \text{BinaryCrossEntropy}(G_d(G_f(x_i; \theta_f), \theta_d), d_i)$$

where θ_f, θ_y and θ_d are the corresponding parameters for Feature Extractor, Label Predictor and Domain Classifier.

In adversarial training, the feature extractor aims to extract more generalized features that can "confuse" the domain classifier. To achieve this, we minimize the label prediction loss (L_y) to ensure good classification performance while maximizing the domain classification loss (L_d) to reduce the domain classifier's accuracy. This adversarial process helps the feature extractor learn domain-invariant representations.

Then the overall loss function for feature extractor is:

$$E(\theta_f, \theta_y, \theta_d) = L_y(\theta_f, \theta_y) - \lambda L_d(\theta_f, \theta_d)$$

where λ is a trade-off parameter that balances the label prediction and domain alignment tasks (which can be dynamic during the training).

For two classifiers, we aim to enhance their ability to correctly classify the data. In that case, we need to minimize the two corresponding losses.

By backpropagation, each parameter is updated by the following rules:

$$\theta_f \leftarrow \theta_f - \mu \left(\frac{\partial L_y}{\partial \theta_f} - \lambda \frac{\partial L_d}{\partial \theta_f} \right)$$

$$\theta_y \leftarrow \theta_y - \mu \frac{\partial L_y}{\partial \theta_y}$$

$$\theta_d \leftarrow \theta_d - \mu \frac{\partial L_d}{\partial \theta_d}$$

where μ is the learning rate.

We can see the gradient of Domain Classification Loss L_D is only reversed when we update the parameter of Feature Extractor θ_f and that's where we introduce GRL layer.

1.3 GRL Layer

As the key component of DANN is the **Gradient Reversal Layer (GRL)**, it reverses the gradient flowing from the domain classifier to the feature extractor during training as mentioned above, in order to encourage the feature extractor to learn features that make it difficult for the domain classifier to distinguish between the source and target domains.

In general:

- **Forwardly**, it acts as an identity function, passing data from the feature extractor to the domain classifier without modification.
- **Backwardly**, it multiplies the gradient from the domain classifier by a negative scalar, effectively reversing the gradient's direction

As a result, the model learns domain-invariant features, aligning the feature distributions across domains as it learns to "confuse" the domain classifier.

2 Practice

In this section, we firstly built a naive model trained on labeled MNIST (source domain) and evaluated it on unlabeled MNIST-M (target domain) without applying any domain adaptation techniques. Next, we reproduced the DANN model described by Ganin and Lempitsky in their paper "Unsupervised Domain Adaptation by Backpropagation" to assess its effectiveness in addressing domain shifts between MNIST and MNIST-M.

The DANN model extends the naive model by adding a new branch consisting of a GRL and a domain classifier. Here's the architecture of the DANN model we used:

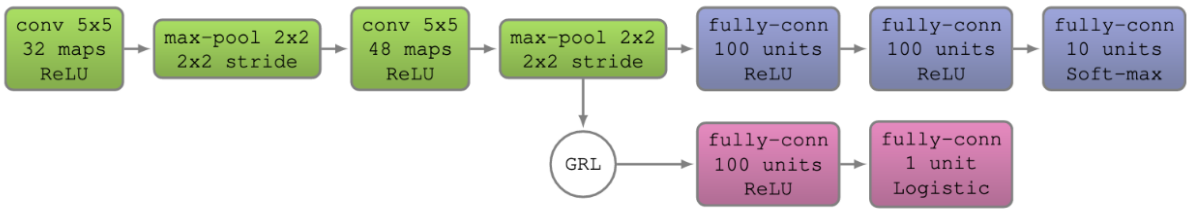


Figure 21: DANN model in paper

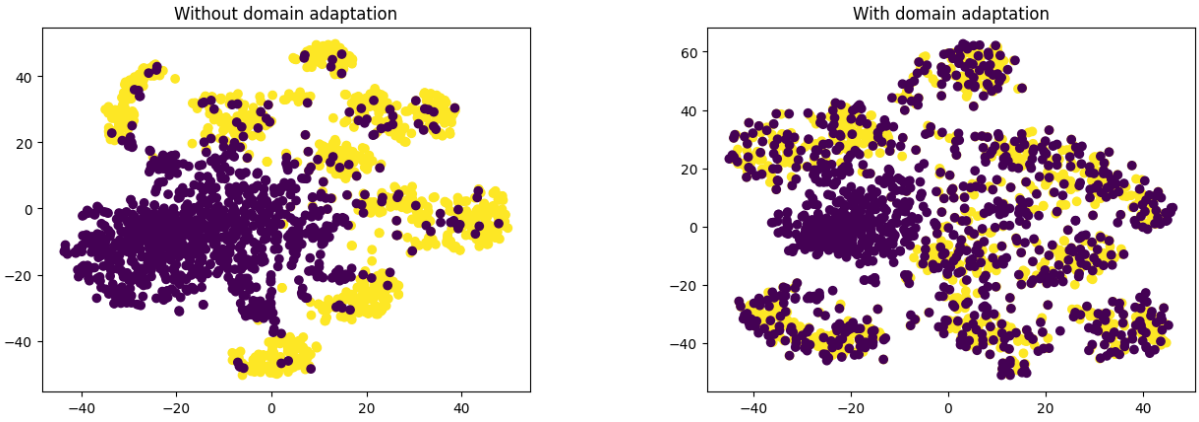
Notably, We used a learning rate scheduler and also dynamically adjusted the GRL factor during the training(we will discuss the reasons in Question 3). The GRL factor is upadted at each training step i by:

$$\lambda(i) = -1 + \frac{2}{1 + \exp\left(\frac{-2 \cdot i}{\text{total training steps}}\right)}$$

2.1 Experiments and results

Based on our experiments, the naive model achieved an impressive accuracy of 98.94% on the source dataset but only 58.9% on the target dataset, highlighting its inability to generalize across domains. In contrast, after 20 epochs of training, our DANN model demonstrated significant improvements in domain adaptation. On the source dataset, it achieved a class accuracy of 98.62%. More importantly, on the target dataset, the DANN model achieved a class accuracy of 76.33%. These results highlight the effectiveness of DANN in learning domain-agnostic features and improving performance on the target domain.

To gain a better understanding of how well the feature extractor aligns the source and target domains, we utilized t-SNE, a dimensionality reduction technique, to observe patterns and relationships in the feature space more intuitively in a 2D plan.



(a) Naive Model (b) DANN Model

Figure 22: Visualization of the embeddings learned by the networks

The t-SNE visualizations highlight the difference in domain alignment between the Naive Model and the DANN Model.

In the Naive Model (a), the source (purple) and target (yellow) features are clearly separated, indicating that the model learned domain-specific features and failed to generalize to the target domain, as reflected by the low target accuracy of 58.9%.

In contrast, the DANN Model (b) shows significant overlap between the source and target feature distributions, demonstrating successful domain alignment through adversarial training with the Gradient Reversal Layer. This alignment led to a substantial improvement in target domain accuracy (76.33%), though with a slight trade-off in source domain performance. These results confirm the effectiveness of DANN in learning domain-agnostic features for domain adaptation.

Further Study

However, our model still performs below the reported performance in the original paper (81.49%). To improve results, we introduced a step decay learning rate scheduler and extended the number of epochs to 100.

Additionally, the GRL factor is updated at each training step i using the formula:

$$\lambda(i) = -1 + \frac{2}{1 + \exp\left(\frac{-10 \cdot \mathrm{i}}{\text{total training steps}}\right)}$$

Since the total training steps have increased fivefold (from 20 epochs to 100 epochs), we scaled the rate of λ adjustment to maintain the same growth rate as before.

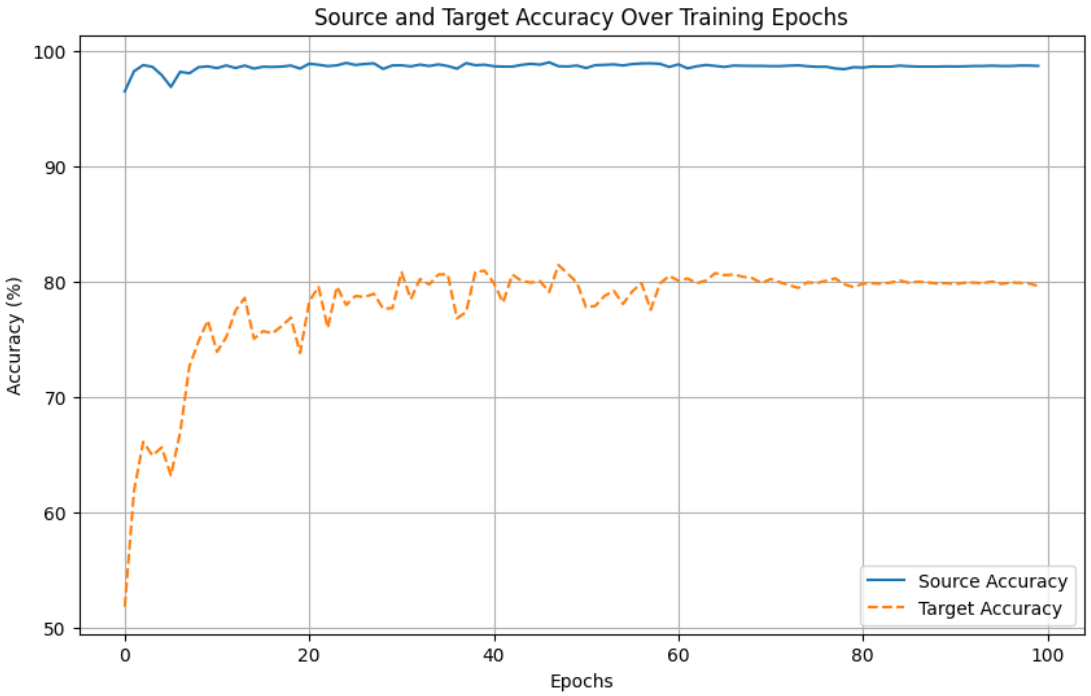


Figure 23: Source and Target Accuracy Over 100 Training Epochs

After a fortunate attempt, our target accuracy stabilized at around 80%, reaching a peak close to the reported performance in the original paper.

2.2 Questions & Answers

Q1 If you keep the network with the three parts (green, blue, pink) but didn't use the GRL, what would happen ?

If the Gradient Reversal Layer (GRL) is removed, the feature extractor would no longer receive adversarial gradients from the domain classifier. This means that during the generator's update step, it would aim to minimize the domain classification loss rather than maximize it. As a result, the feature extractor would produce features that are easier for the domain classifier to distinguish between source and target domains, reinforcing the domain-specific nature of the features. This undermines the goal of learning domain-invariant features, leading to poor alignment between source and target domains.

Consequently, the target domain performance would degrade significantly, as the label classifier would fail to generalize. Meanwhile, the source domain performance

might improve slightly since the feature extractor no longer faces the competing objective of confusing the domain classifier. Overall, the removal of GRL fundamentally breaks the adversarial dynamic that drives domain adaptation.

Q2 Why does the performance on the source dataset may degrade a bit ?

In our experiment, the source dataset accuracy started at 98.94% and dropped slightly to 98.62% after 10 epochs. This slight decrease can be explained by the fact that the feature extractor is not solely focused on optimizing for the source domain classification task. To achieve a more generalized and domain-invariant representation, it must align the source and target domains, which inevitably sacrifices a small degree of local specificity in the features that are particularly beneficial for the source domain. This trade-off, aimed at improving the model's ability to adapt to the target domain, results in a slight reduction in the source domain's performance.

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

The value of the negative number used in the Gradient Reversal Layer (GRL) controls the strength of the adversarial signal from the domain classifier to the feature extractor, and it significantly affects the training and model performance.

This trade-off parameter that balances the label prediction and domain alignment tasks. The larger the absolute value of this parameter, the more it pushes the feature extractor to confuse the domain classifier and focus on creating features that work well for both domains.

However, if the value is too large, the feature extractor might focus too much on confusing the domain classifier and lose important details needed for accurate predictions, which could hurt performance on both the source and target datasets.

If it's too small, the feature extractor won't get enough pressure to confuse the domain classifier. This means it will mostly focus on the source domain and won't learn features that generalize well to the target domain. As a result, the domain classifier will easily distinguish between the two domains, and the model's performance on the target dataset will likely remain poor.

If the domain classifier is too weak, trying to confuse it becomes meaningless because the feature extractor won't need to work hard to fool it. This is why we need to dynamically adjust the parameter to find the right balance, ensuring that the training progresses smoothly and steadily. By gradually increasing the adversarial strength, the feature extractor can effectively adapt while avoiding instability or premature convergence.

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

The concept of pseudo-labeling originates from semi-supervised learning and typically involves three steps:

1. Train the model using labeled source domain data.
2. Use the trained model to predict labels for the unlabeled target domain data, thereby creating pseudo-labels.

3. Retrain the model using both the labeled source domain data and the pseudo-labeled target domain data together.

In practical, in step 3 we often select high-confidence pseudo-labeled target domain data based on the predicted probability.

To conclude, this method heavily relies on the predictions of the initial model. If the initial model's predictions are accurate, the pseudo-labels will be reliable and help improve the model's performance. However, if the initial predictions are poor, the pseudo-labels may contain errors, which can propagate during training and negatively impact the final model.

3 Conclusion

In this practical, we implemented and analyzed the Domain-Adversarial Neural Network (DANN) to tackle the challenge of domain adaptation. By introducing the Gradient Reversal Layer (GRL), we enabled the model to learn domain-invariant features through adversarial training, bridging the gap between source and target domains.

Our experiments demonstrated that the naive model, while achieving high accuracy on the source domain, struggled to generalize to the target domain due to domain shifts. In contrast, the DANN model significantly improved target domain performance, with a target accuracy of 76.33% after 20 epochs and further stabilized at around 80% after extending training to 100 epochs. These improvements were validated through t-SNE visualizations, which highlighted the effective alignment of source and target feature distributions.

In conclusion, the DANN model provides a powerful framework for domain adaptation, enabling models to generalize across domains with distinct distributions.

2-de: Generative Adversarial Networks

1 Introduction

In the previous sections, we implemented some classification tasks using transfer learning and domain adaptation. In this section, we will utilize generative models GAN and cGAN. In **GAN**, we generate images $\tilde{x}$ using random noise z:

$$\tilde{x} = G(z),\ z \sim P(z)$$

In **cGAN**, we introduce an additional input y (which can be a class label, image, sentence, etc.):

$$\tilde{x} = G(z, y),\ z \sim P(z)$$

2 Generative Adversarial Networks

2.1 General Principle

In this subsection, we will use a Deep Convolutional Generative Adversarial Network (DCGAN) to generate realistic data. The DCGAN consists of two components: a generator and a discriminator.

Generator: We sample an input z from a fixed distribution (generally $U_{[-1,1]}$ or $\mathcal{N}(0, I)$) to generate realistic data $\tilde{x}$.

$$\tilde{x} = G(z),\ z \sim P(z)$$

Discriminator: To evaluate the images generated by the generator, we need a criterion. However, since the distribution $P(X)$ of real data x^* is unknown, we cannot explicitly define a loss function. Thus, we introduce a second neural network D to distinguish between real image x^* and fake image $\tilde{x}$.

$$D(x) \in [0, 1],\ \text{ideally},\ \begin{cases} D(\tilde{x}) = 0, & \tilde{x} = G(z) \\ D(x^*) = 1, & x^* \in \mathcal{D}_{\text{data}} \end{cases}$$

Adversarial training: We aim for the generator G to learn how to produce outputs that are classified as 1 by the discriminator D, while simultaneously training the discriminator

D to classify real images as 1 and fake images as 0. This creates an adversarial relationship between the two networks.

We choose binary cross-entropy (BCE) as the loss function for the discriminator, which is defined as:

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i)\right]$$

We separately calculate the binary cross-entropy for real images and fake images:

- For real images, the class label is 1, i.e., $y_i = 1$:

$$\begin{aligned}\text{BCE}_{x^*} &= -\frac{1}{N_1}\sum_{i=1}^{N_1}\left[1\cdot\log(p_i) + 0\cdot\log(1-p_i)\right], \quad \text{where } p_i = D(x_i^*)\\ &= -\mathbb{E}_{x^*\in\mathcal{D}}\left[\log(D(x^*))\right]\end{aligned}$$

- For fake images, the class label is 0, i.e., $y_i = 0$:

$$\begin{aligned}\text{BCE}_{\tilde{x}} &= -\frac{1}{N_2}\sum_{i=1}^{N_2}\left[0\cdot\log(p_i) + 1\cdot\log(1-p_i)\right], \quad \text{where } p_i = D(G(\tilde{x}_i))\\ &= -\mathbb{E}_{z\in P(z)}\left[log(1-D(G(Z)))\right]\end{aligned}$$

By summing up both two equations, the problem we aimed to optimize becomes:

$$\min_G \max_D \mathbb{E}_{x^*\sim\mathcal{D}}\left[\log D(x^*)\right] + \mathbb{E}_{z\sim P(z)}\left[\log\left(1-D(G(z))\right)\right] \tag{5}$$

We optimize the generator and discriminator alternately (similar to the coordinate ascent method):

- fixed D, optimize G

$$\begin{aligned}&\min_G \mathbb{E}_{x^*\sim\mathcal{D}}\left[\log D(x^*)\right] + \mathbb{E}_{z\sim P(z)}\left[\log\left(1-D(G(z))\right)\right]\\ \Leftrightarrow &\min_G \mathbb{E}_{z\sim P(z)}\left[\log\left(1-D(G(z))\right)\right]\\ \Leftrightarrow &\max_G \mathbb{E}_{z\sim P(z)}\left[\log\left(D(G(z))\right)\right]\end{aligned} \tag{6}$$

- fixed G, optimize D

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

Q1 Interpret the equations (6) and (7). What would happen if we only used one of the two ?

In Equation (6), for a fixed discriminator D, we aim for the generator G to produce images that are classified as real (i.e., classified as 1 by D), making the generated

images as close as possible to real images. In Equation (7), for a fixed generator G, we aim for the discriminator D to classify real images as 1 and the generated images as 0, meaning the discriminator should effectively distinguish between real and fake images.

If we only use Equation (6), the loss will converge quickly, but because the discriminator's ability to distinguish is weak, the generator will not receive meaningful feedback during training. As a result, the generated images will significantly differ from real images. On the other hand, if we only use Equation (7), the discriminator will easily classify real and fake images, while the generator will remain unchanged, failing to produce a well-trained generator.

Therefore, we need to alternate between the two optimization steps to allow the generator and discriminator to compete with each other effectively.

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

The distribution of z is typically initialized as a Gaussian distribution $\mathcal{N}(0, I)$. However, as GAN training progresses, the distribution of generated images $P(G(z))$ should become as close as possible to the distribution of real images $P(x)$.

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 ?

From the derivation above, we can see that the "true" equation should be:

$$\min_G \mathbb{E}_{z\sim P(z)} \left[\log\left(1 - D(G(z))\right)\right] \tag{6'}$$

However, in practice, if the discriminator D can easily distinguish generated samples from real samples, $D(G(z))$ will be very close to 0 in the early stages of training. In this case, $1 - D(G(z)) \approx 1$, and the generator's loss approaches 0, providing almost no significant gradient feedback to the generator, leading to the vanishing gradient problem.

In contrast, in equation (6), even if the discriminator is very strong, $\log(D(G(z)))$ approaches $-\infty$ when $D(G(z))$ is close to 0. The loss function can still provide sufficiently large gradients to ensure the generator's updates. Therefore, this modification helps prevent the vanishing gradient problem and improves training stability.

2.2 Architecture of the networks

We will use a deep convolutional GAN (DCGAN) architecture to generate MNIST images. This architecture consists of a generator and a discriminator mainly based on convolutional layers.

Define the batch size as 128, the latent dimension size as 1000, the base number of filters in the generator(ngf) as 32, and the base number of filters in the discriminator(ndf) as 32.

Batch size $= 128$, $n_z = 1000$, $ngf = 32$, $ndf = 32$.

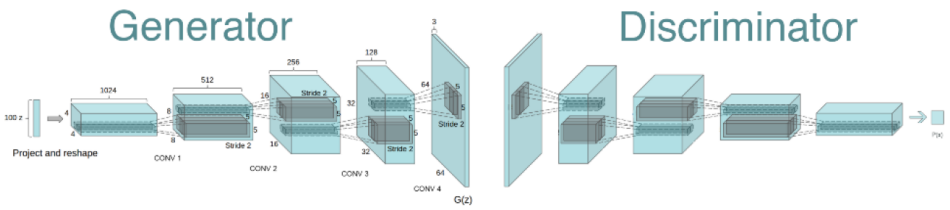


Figure 24: Architecture of the classic DCGAN

- **Generator**
  We sample a tensor z of size (128, 1000) from a Gaussian distribution and reshape it to (128, 1000, 1, 1). Then, we apply a sequence of convolutional transpose operations:

  - Convolution Transpose (128 filters, kernel 4, stride 1, padding 0, no bias) + Batch Norm + ReLU $outputsize : (128, 128, 4, 4)$

  - Convolution Transpose (64 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU $outputsize : (128, 64, 8, 8)$

  - Convolution Transpose (32 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + ReLU $outputsize : (128, 32, 16, 16)$

  - Convolution Transpose (3 filters, kernel 4, stride 2, padding 1, no bias) + Tanh activation $outputsize : (128, 3, 32, 32)$

- **Discriminator**
  We pass real and fake images into the classifier. The input size is (128, 3, 32, 32). The discriminator consists of the following convolutional operations:

  - Convolution (32 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$) $outputsize : (128, 32, 16, 16)$

  - Convolution (64 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$) $outputsize : (128, 64, 8, 8)$

  - Convolution (128 filters, kernel 4, stride 2, padding 1, no bias) + Batch Norm + LeakyReLU ($\alpha = 0.2$) $outputsize : (128, 128, 4, 4)$

  - Convolution (1 filter, kernel 4, stride 1, padding 0, no bias) + Sigmoid activation $outputsize : (128, 1, 1, 1)$

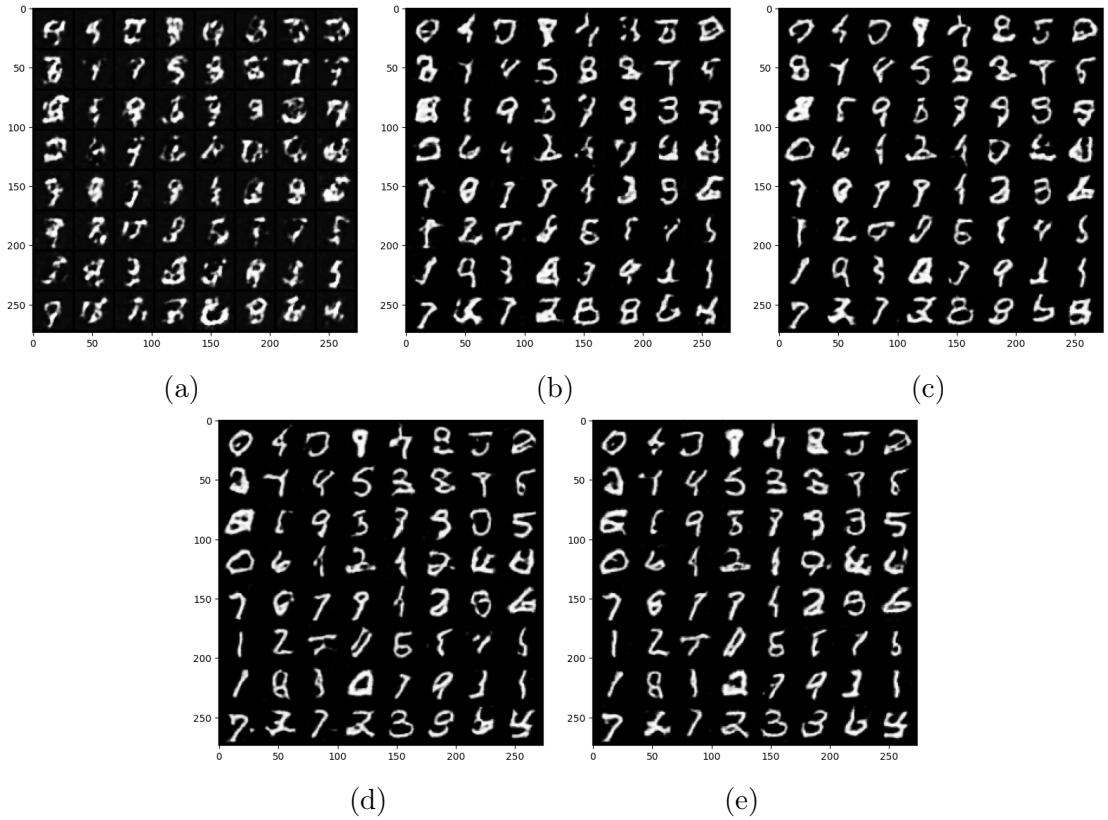
Q4 Comment on the training of of the GAN with the default settings (progress of the generations, the loss, stability, image diversity, etc.

From the visualization of the generated images, with the progress of the generations, the digits are shown clearer and more identifiable, which we can see from the Figure 25 below.

But from Figure 26 the loss curves are showing strong instabilities, oscillating across iterations, which is typical in GANs due to the adversarial objective, where the generator and discriminator are continuously trying to outperform each other. The discriminator loss is generally lower and appears more stable compared to the generator loss. Small spikes in the discriminator loss occur, likely when the generator

successfully produces images that are more challenging to classify. The lower overall discriminator loss suggests that it is consistently performing better than the generator in distinguishing real and fake samples, and in turn encourages the Generator to produce better images.

From Figure 25 below, though the generated figures improves over epochs, the instability in generator loss that we observe from Figure 26 could suggest the potential for mode collapse, where the generator produces limited variations of images to exploit weaknesses in the discriminator.



(a) (b) (c)

(d) (e)

Figure 25: Generated images from GAN during training with default settings after (a) 1 epoch, (b) 2 epochs, (c) 3 epochs, (d) 4 epochs, (e) 5 epochs

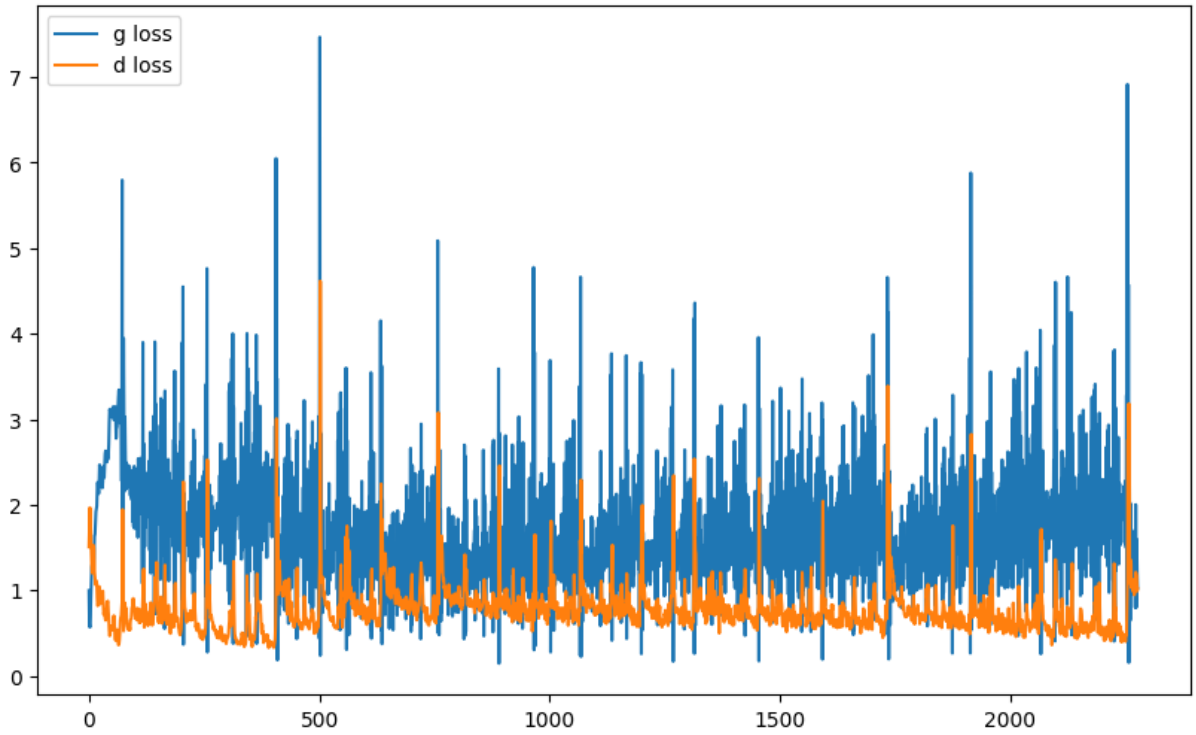
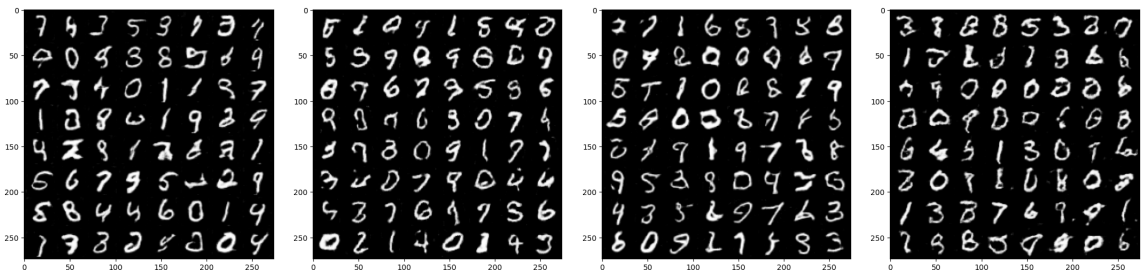


Figure 26: Generator loss and Discriminator loss for the GAN with default settings

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.

In this part we experiment on various tests under different circumstances which we will demonstrate with generated images, and training curves of losses and scores.

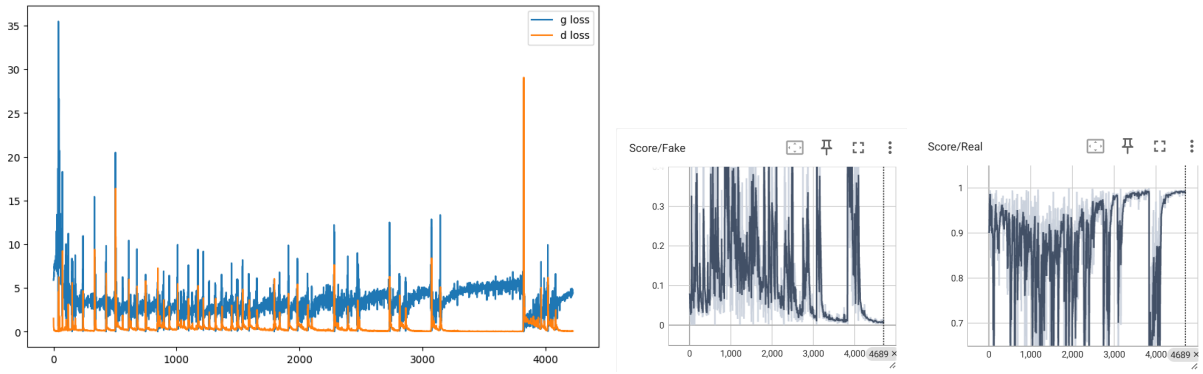
Modify ngf or ndf In the experiment, ngf and ndf respectively represent the feature map size of Generator and Discriminator, which directly influence the capacity and the feature extraction ability of the model. Even though training with different parameters of ngf and ndf, visually speaking, the generated images are not presenting much identifiable differences to human eyes as shown in Figure 27. But we are going to elaborate on the differences of each case later



(a) (b) (c) (d)

Figure 27: Generated images after 10 epochs with (a) $ndf = 128$, (b) $ndf = 8$, (c) $ngf = 128$, (d) $ndf = 8$.

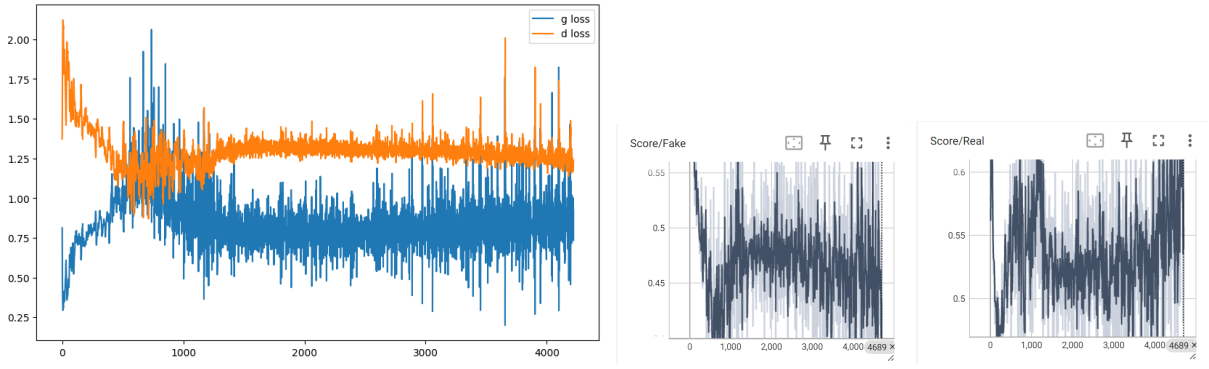
$ndf = 128$**:** We start with increasing ndf to 128 while keeping ngf at 32. From Figure 28 we can see that with larger ndf, the discriminator consistently distinguish the images, making it more challenging for the generator to learn.



(a) (b) (c)

Figure 28: Training Curves for $ndf = 128$: (a) loss(D) and loss(G), (b) $D(G(z))$, (c) $D(x)$

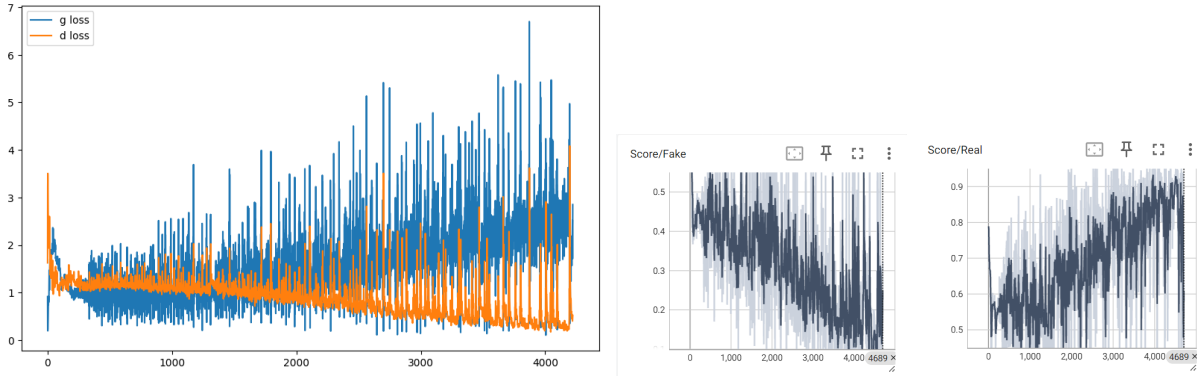
$ndf = 8$**:** However, when we deduce the ndf to 8 while keeping the ngf unchanged at 32, the results from Figure 29 show that the score of $D(G(z))$ and $D(x)$ still remain close to the equilibrium at 0.5.



(a) (b) (c)

Figure 29: Training Curves for $ndf = 8$: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

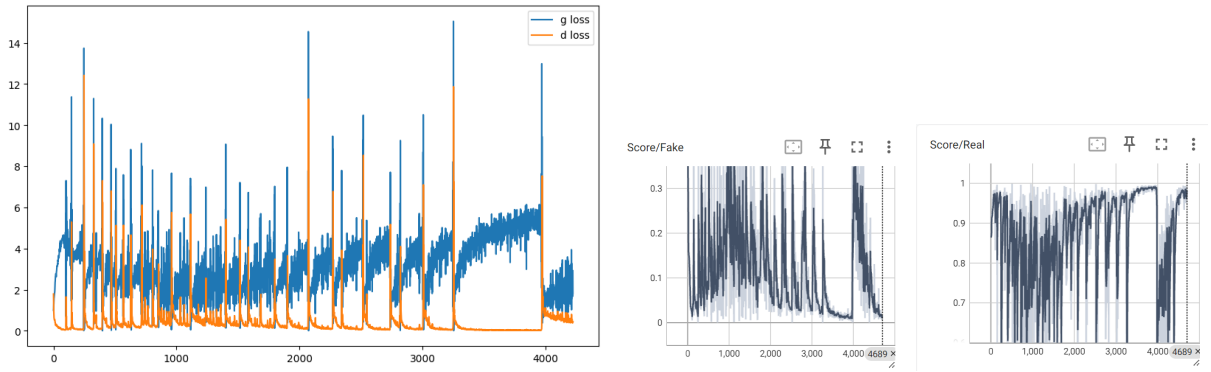
$ngf = 128$**:** As for ngf, we start with increasing it to 128 while keeping ndf at 32. Even though the increase of ngf enhances the performance of generator but the results in Figure 30a show that the generator still struggles to produce samples that the discriminator perceives as realistic, meanwhile the discriminator outperforms with confidence in both effectively identifying the generator's output as fake and also in classifying real images as real. So we can conclude that even with higher ngf, it's still hard for the generator to fool the discriminator.



(a) (b) (c)

Figure 30: Training Curves for $ngf = 128$: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

$ngf = 8$**:** As presented in Figure 31, with lower ngf, the discriminator consistently outperforms with significantly lower loss compared with generator, and higher confidence in more identifying more effectively the generator's output as fake and classifying correctly the real figures as real.

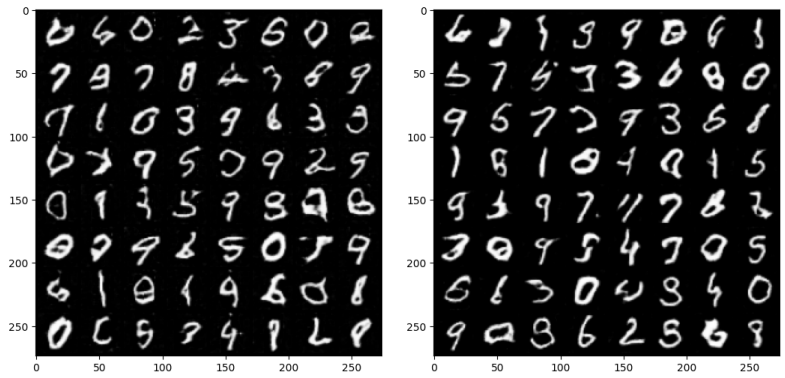


(a) (b) (c)

Figure 31: Training Curves for $ngf = 8$: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

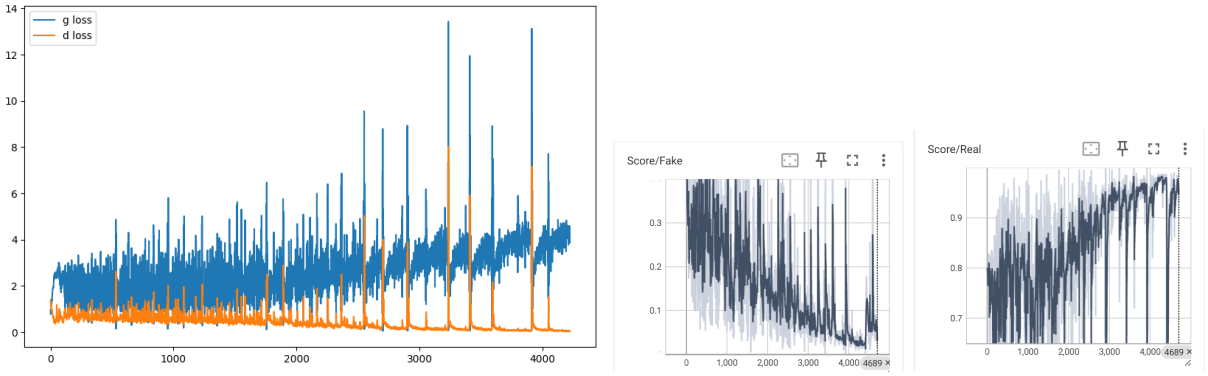
Replace the custom weight initialization with pytorch's default initialization. We now show the generated images with customized weight initialization for `'CONV'` and `'BatchNorm'` layers and with default initializations. From the following Figure 32, we observe that with default initializations though performs well on certain digits, but there are some digits on the generated image that are still less convincing than the one with customized initialization.

This is this result of what we observe in Figure 33, where we can see that with default initialization the discriminator manages to distinguish between real and fake images in early stages, making it difficult for the generator to learn and generate convincing images.



(a) (b)

Figure 32: Generated images after 10 epochs with (a) customized weight initialization, (b) PyTorch's default weight initialization



(a) (b) (c)

Figure 33: Training Curves for default initialization: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

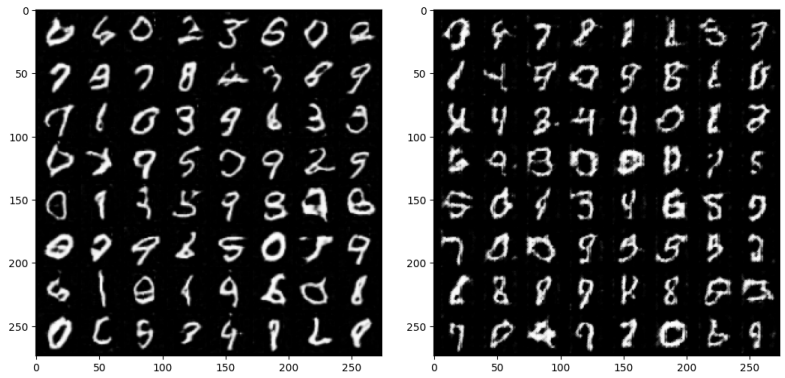
Replace the training loss of the generation with the "true" loss derived from the original equation In the model, we use BCE loss for the generator, which is

$$\text{BCE Loss (Generator):} \quad g\_loss = -\mathbb{E}[\log(D(G(z)))]$$

and in this experiment, we are replacing with its true loss, which is

$$\text{True Loss (Generator):} \quad g\_loss = \mathbb{E}[\log(1 - D(G(z)))]$$

Let's start with the visualization of the generated images. From Figure 34, the Figure 34b with true loss are shown with less identifiable digits compared with Figure 34a.

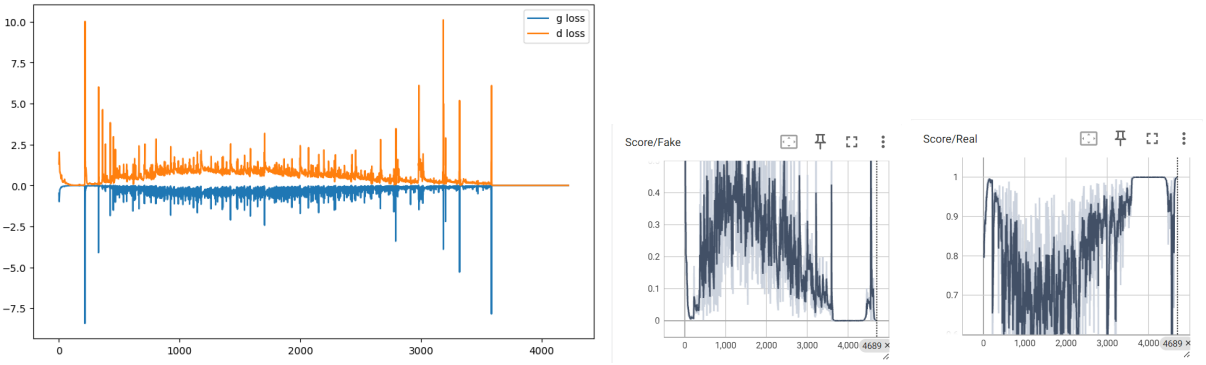


(a) (b)

Figure 34: Generated images after 10 epochs for generator with (a) BCE loss, (b) true loss.

And from Figure 35a, we can see that with the replacement of "true" loss, the generator loss is then calculated as negative values, which indicates the generator's difficulty in improving sample quality due to the gradient vanishing problem inherent in true Loss. Meanwhile, the discriminator remains dominant throughout training, making it difficult for the generator to match its performance.

In Figure 35b, persistent low fake sample scores suggest that the generator's progress is limited, and its outputs fail to convincingly fool the discriminator while from Figure 35c, the dominance of high real sample scores creates an imbalance, making it even harder for the generator to catch up.



(a) (b) (c)

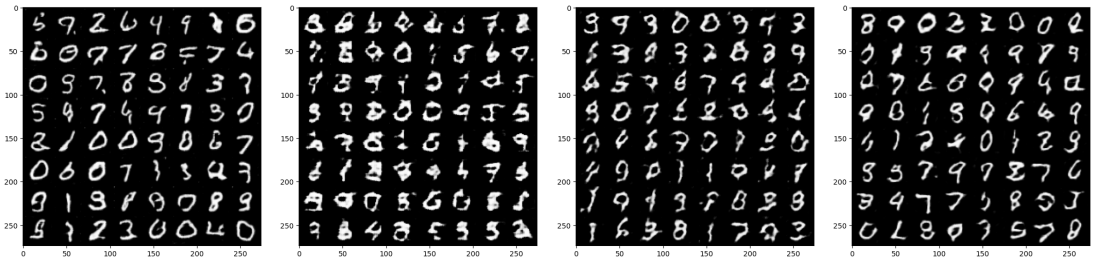
Figure 35: Training Curves for "true" loss of generator: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

And from the performances of the "true" loss above, we can conclude that it is better to keep using BCE loss to train the generator for stable gradients and improved generator performance.

Change the learning rate of on or both models. In this part we experimented on different combinations of learning rates and β_1 for the Adam optimizer of both

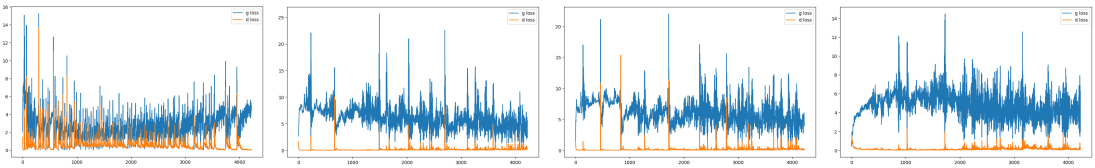
the discriminator and generator: higher learning rates for only discriminator (0.001); higher learning rates for discriminator (0.001) and higher β_1 (0.9); higher learning rates for both discriminator and generator (0.001) and higher β_1 (0.9); only higher β_1 (0.9).

Our results from Figure 36, Figure 37 , Figure 38 and Figure 39 correspond with the conclusion from Radford et al. (2016) that the higher learning rate at 0.001 and higher β_1 at 0.9 makes the model learn too quickly and results in training oscillation and instability. And we also observe that with same β_1 at 0.9, increasing only the discriminator's learning rate has much worse result than increasing both learning rates.



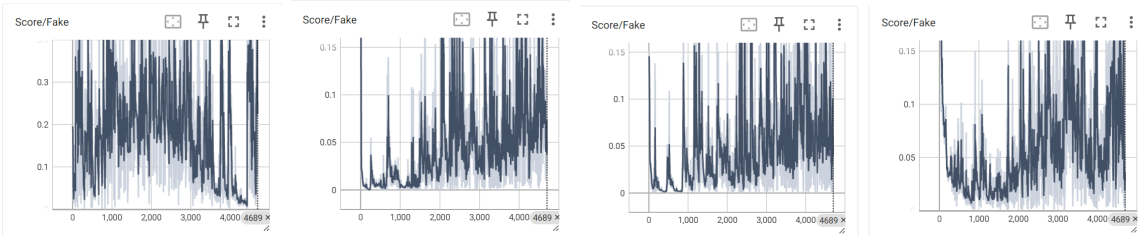
(a) (b) (c) (d)

Figure 36: Generated images after 10 epochs of different combinations of learning rates and β_1 with (a) $lr_d = 0.001$, (b) $lr_d = 0.001, \beta_1 = 0.9$, (c) $lr_d = lr_g = 0.001, \beta_1 = 0.9$, (d) $\beta_1 = 0.9$



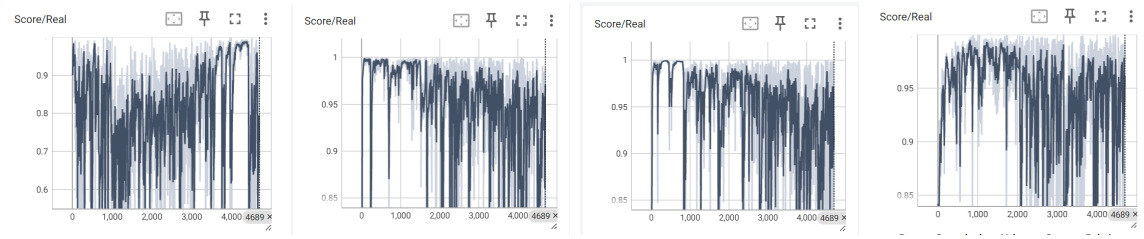
(a) (b) (c) (d)

Figure 37: Training curves of loss(D) and loss(G) of different combinations of learning rates and β_1 with (a) $lr_d = 0.001$, (b) $lr_d = 0.001, \beta_1 = 0.9$, (c) $lr_d = lr_g = 0.001, \beta_1 = 0.9$, (d) $\beta_1 = 0.9$



(a) (b) (c) (d)

Figure 38: Training curves of $D(G(z))$ of different combinations of learning rates and β_1 with (a) $lr_d = 0.001$, (b) $lr_d = 0.001, \beta_1 = 0.9$, (c) $lr_d = lr_g = 0.001, \beta_1 = 0.9$, (d) $\beta_1 = 0.9$

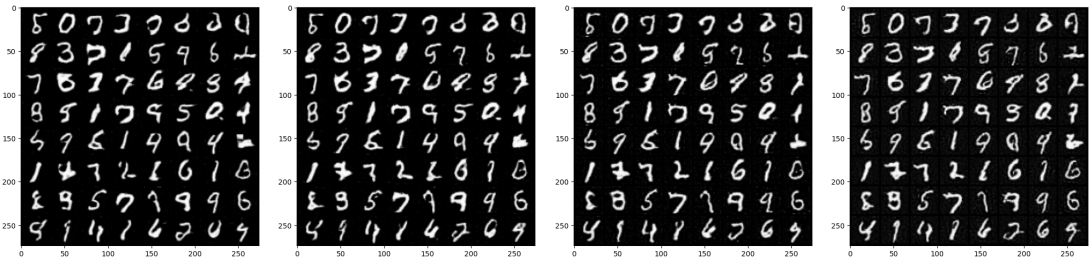


(a) (b) (c) (d)

Figure 39: Training curves of $D(x)$ of different combinations of learning rates and β_1 with (a) $lr_d = 0.001$, (b) $lr_d = 0.001, \beta_1 = 0.9$, (c) $lr_d = lr_g = 0.001, \beta_1 = 0.9$, (d) $\beta_1 = 0.9$

Learn for longer epochs In this experiment, we extend the training to 30 epochs as suggested.

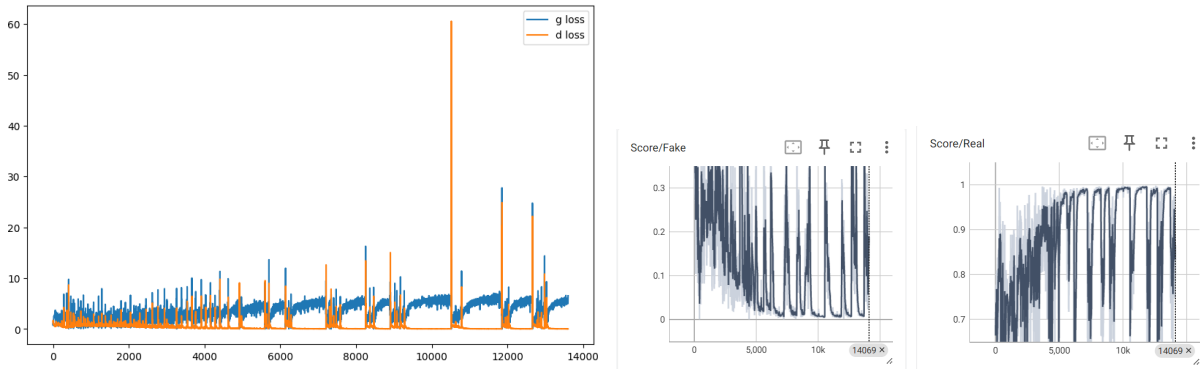
In the following evolution of the generated images at different epochs in Figure 40, we can observe that some digits are becoming more identifiable while some are turning into something different like from the digit resembling to 4 can turn into 9. The process shows the risk of increasing the number of epochs, since the discriminator becomes more proficient at identifying the real and fake images, forcing the generator to generate noise and make more effort to deceive the discriminator.



(a) (b) (c) (d)

Figure 40: Generated images after (a) 15 epochs, (b) 20 epochs (c) 25 epochs, (d) 30 epochs

As the training proceeds, we can see from Figure 41 that the spikes in discriminator loss indicate that the generator occasionally achieves substantial progress. though the discriminator's output approaches 1 and the discriminator's output for generated images approaches 0, meaning that the discriminator are distinguishing perfectly the real images from fake images, the output scores oscillates, indicating that the discriminator starts giving more random feedback and the generator might therefore struggles to maintain improvements over time. So the convergence of GANs is rather temporary than stable and permanent.



(a) (b) (c)

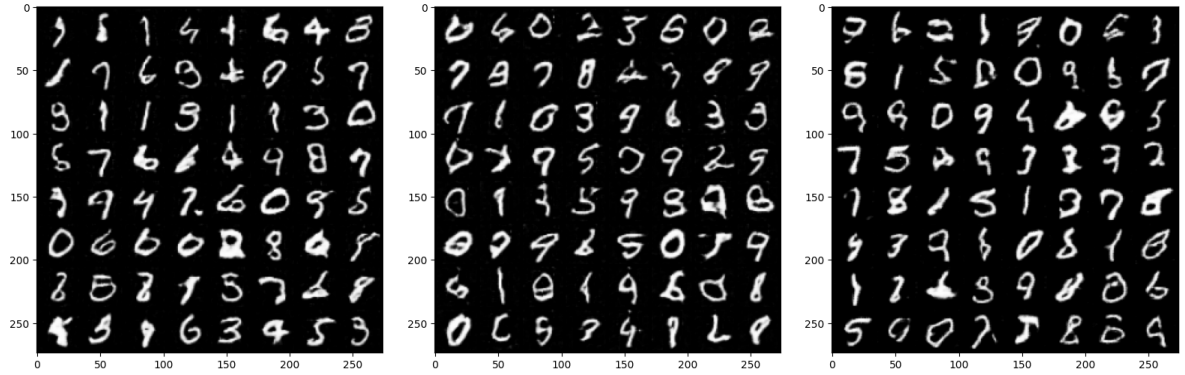
Figure 41: Training Curves for epoch=30 : (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

Influence of n_z n_z represents the size of the noise vector that directly affects the quality, diversity, and structure of the generated images. As suggested we compare the results of n_z at 10, 100(original) and 1000.

From the generated images in Figure 42, when $n_z = 10$, we see that many digits appear repetitive or overly simplified (e.g., similar "1"s and "7"s). and with the increase of n_z, the generated images are exhibiting more diversities. But with higher n_z at 1000, it risks overfitting or unnecessary complexity, which can negatively impact sample quality.

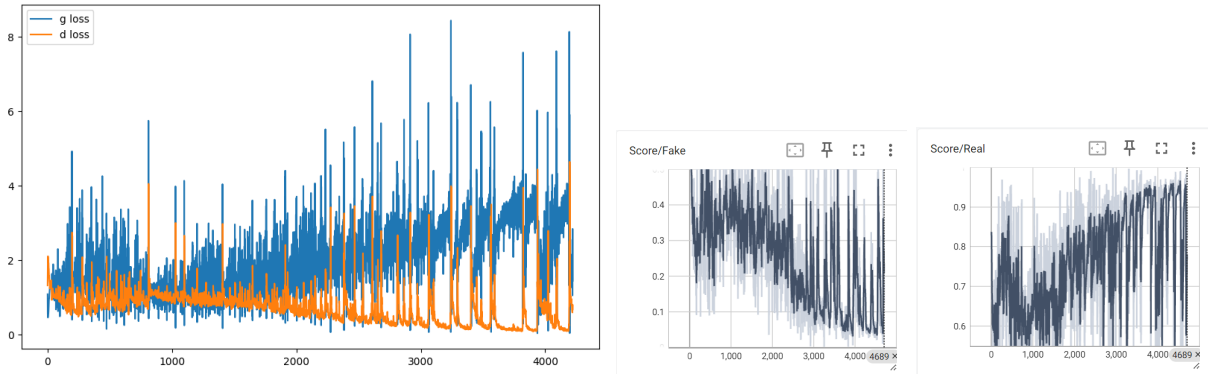
And with the comparison from Figure 43 and Figure 44, we see that when $n_z = 10$, the generator struggles to improve, likely due to insufficient latent space capacity and this limited latent space restricts the generator's ability to improve. And when $n_z - 1000$, the larger latent space increases the complexity of training, leading to occasional spikes and slower convergence, also making consistent improvements harder for the generator that occasionally produces better samples. The larger fluctuation in Figure 44c also reflects the instability introduced by the larger latent space.

So the takeaway is that we should keep using $n_z = 100$ as the default for most datasets, as it is moderate and most likely to provide a good balance of quality and diversity.



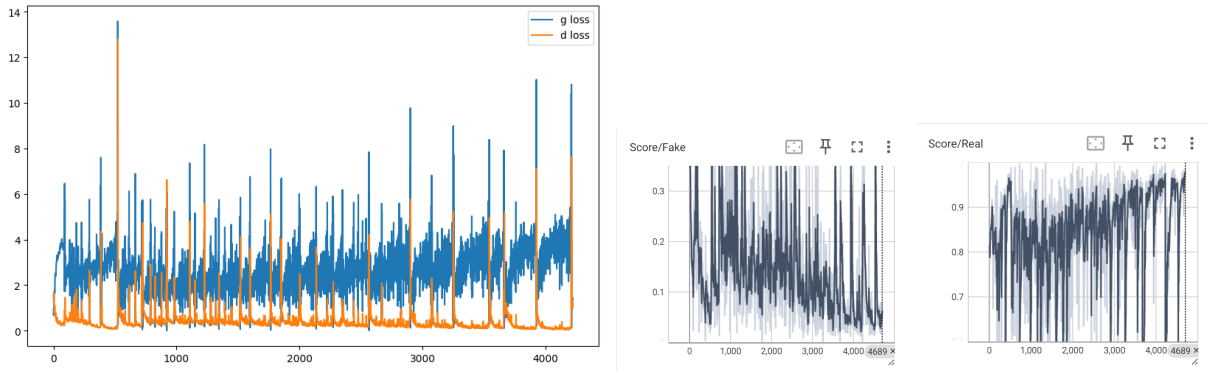
(a) (b) (c)

Figure 42: Generated images with (a) $n_z = 10$, (b) $n_z = 100$ (c) $n_z = 1000$



(a) (b) (c)

Figure 43: Training Curves for $nz = 10$: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$



(a) (b) (c)

Figure 44: Training Curves for $nz = 1000$: (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

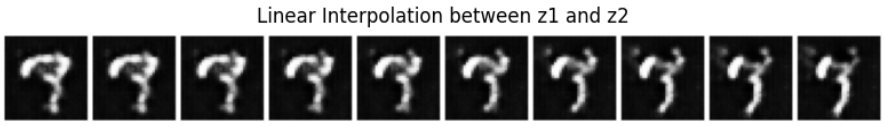
Using a learned GAN, take 2 noise vectors z_1 and z_2 and generate the images corresponding to several linear interpolations $\alpha z_1 + (1-\alpha)z_2, \alpha \in [0, 1]$.

To compare the result of interpolation in different training stages, we demonstrate the generated interpolation after 1, 6, 10 epochs in Figure 45.

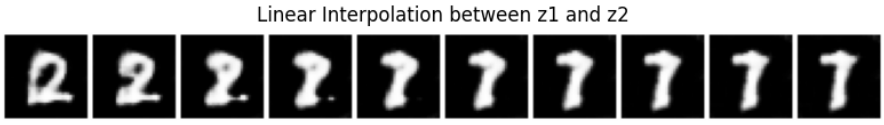
We can see from Figure 45a that only after one epoch, the adversarial training dynamic has not stabilized, and the generator has not captured sufficient information about the real data distribution and therefore the interpolations between z_1 and z_2 fail to result in meaningful transitions.

But as the training proceeds, in Figure 45b we see that The interpolated images start showing clearer shapes and resemble digits (from "2" to "7"). However, the diversity and smoothness of transitions are limited. Most images look quite similar, with only slight variations. So we could conclude at this stage, while the generator has improved, it struggles to produce diverse samples or meaningful interpolations.

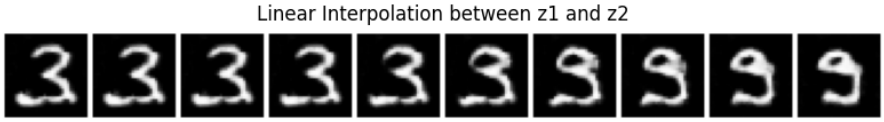
And eventually in Figure 45c, with the progress of training, the generator captures the data distribution effectively, resulting in high-quality interpolations. It learns a smooth mapping from the latent space to the data space, enabling realistic transitions (from "3" to "9").



(a)



(b)

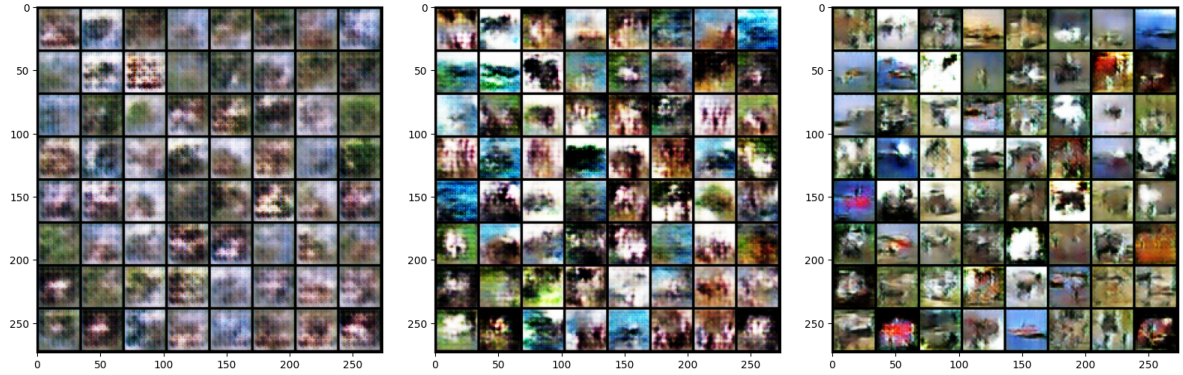


(c)

Figure 45: Linear interpolation generated after (a) 1 epoch, (b) 6 epochs, (c) 10 epochs

Try another dataset: CIFAR-10 To further explore the model, we try on the dataset of CIFAR-10 by training a GAN over 10 epochs on 32×32 generated images.

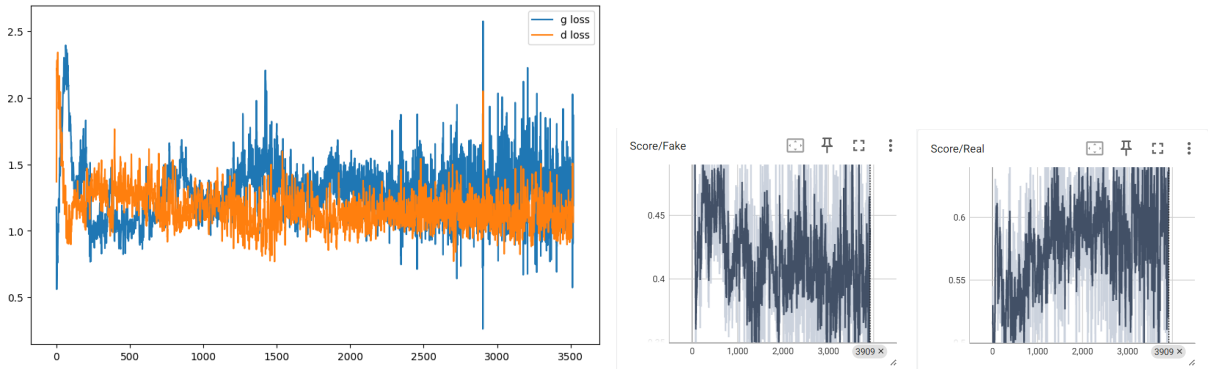
At first glance, the generated images in Figure 46 appear to be very visually distinct figures, but we can still manage to find some sharing features between the figures of different training stages even though under the limits of the resolution. This indicates that the generator has learned from the distribution of images.



(a) (b) (c)

Figure 46: Generated images of CIFAR-10 after (a) 3 epochs, (b) 5 epochs (c) 10 epochs.

And from Figure 47, we can see that the discriminator achieves relatively higher losses compared with all the previous experiments with MNIST before, and the scores of $D(G(z))$ and $D(x)$ that fluctuate around 0.43 and 0.57 respectively show that the discriminator finds it difficult to distinguish the generated images from the real ones.

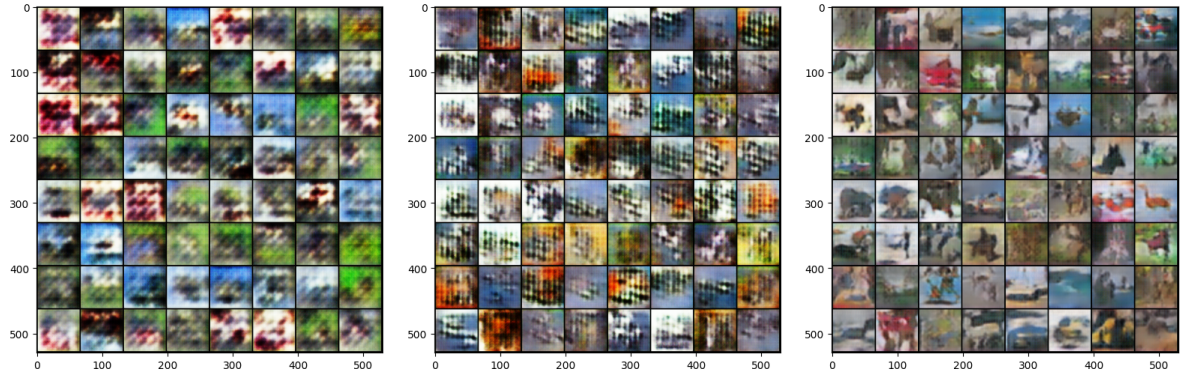


(a) (b) (c)

Figure 47: Training Curves for CIFAR-10 : (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

CIFAR-10 with higher resolution This time we try with higher resolution by generating 64×64 images of CIFAR-10.

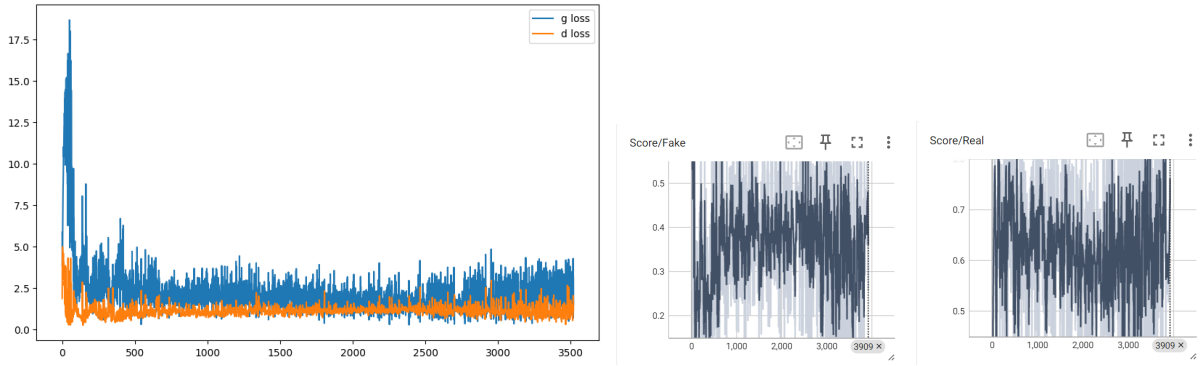
With higher resolutions the generated images are more identifiable, enabling us to better see the shared characteristics of the images in-between different epochs. Though we can conclude that with more epochs, the generated images are demonstrated with more recognizable results, the final image after 10 epochs still appear to be blurry, indicating the dataset requires more training steps to get satisfactory results.



(a) (b) (c)

Figure 48: Generated images of CIFAR-10 after (a) 3 epochs, (b) 5 epochs (c) 10 epochs.

From Figure 49 we observe that with higher resolution, the discriminator are more proficient at distinguishing between the generated images and the real ones. But the results are still not ideal for the discriminator, since with the progress of training, the generator appears to start producing images of high quality, making it difficult for the discriminator to identify. But the fluctuations still exhibits instabilty for the generator to challenge the discriminator in the process.



(a) (b) (c)

Figure 49: Training Curves for CIFAR-10 with higher resolution : (a) loss(D) and loss(G), (b) $D(G(z))$ (c) $D(x)$

3 Conditional Generative Adversarial Networks

In this section, we explore the concept and implementation of Conditional Generative Adversarial Networks (cGANs), which enhance traditional GANs by incorporating additional conditional information. This section covers the general principles of cGANs and their specific application to datasets such as MNIST.

3.1 General Principle

Conditional Generative Adversarial Networks (cGANs) extend traditional GANs by introducing conditional information y into both the generator and discriminator. This

conditional information can take various forms, such as class labels, text descriptions, or additional data features, which guide the generation process.

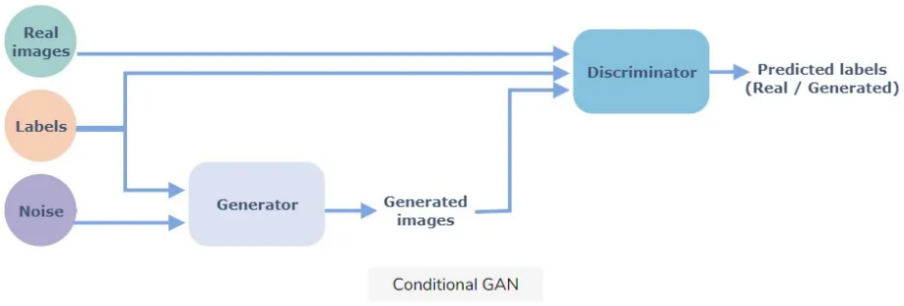


Figure 50: Visualization of the impact of z on generation

In a cGAN, the generator $G(z, y)$ produces a sample $\tilde{x}$ based on a noise vector z and the condition y (extracted from real data). Simultaneously, the discriminator $D(x, y)$ evaluates both the authenticity of the input x and whether it matches the condition y. Mathematically, we have:

$$x^* \in \text{Data}, \quad \tilde{x} = cG(z, y), \quad z \sim P(z), \quad y = attribute(x^*)$$

$$D(x, y) \in [0, 1], \quad \text{ideally}, \quad \begin{cases} D(\tilde{x}, y) = 0, & \tilde{x} = G(z, y) \\ D(x^*, y) = 1, & x^* \in \text{Data} \end{cases}$$

Finally, similar to traditional GAN model, the adversarial optimization objective for a cGAN is as follows:

$$\min_G \max_D \mathbb{E}_{(x^*,y)\sim P_{\text{data}}(x,y)}[\log D(x^*, y)] + \mathbb{E}_{z\sim P(z), y\sim P_{\text{data}}(y)}[\log(1 - D(G(z, y), y))] \quad (1)$$

By including this additional conditional input y, cGANs gain the ability to generate samples that are not only realistic but also aligned with specific conditions, improving both diversity and control over the generated outputs.

3.2 cDCGAN Architectures for MNIST

We follow the previous DCGAN architecture since the behavior of the generator and discriminator remains unchanged. The only modification is that both now take an additional input: the label y. For MNIST, the label y is represented as a one-hot vector of size 10.

The primary difference lies in how the label y is fused with either the noise vector z (for the generator) or the image x (for the discriminator). Additionally, we adjust the number of input channels in the first layer to accommodate the inclusion of y.

For the generator, the label y is concatenated with the noise vector z directly before being processed by the network. For the discriminator, y is expanded spatially along the height and width dimensions to match the dimensions of x, and the two are stacked along the channel dimension as input to the first convolutional layer. These changes ensure that the conditional information is incorporated effectively into the generation and discrimination processes.

3.3 Questions & Answers

Q1 Comment on your experiences with the conditional DCGAN.

Based on our experiments, the conditional DCGAN generates clear and well-formed digits more quickly compared to the standard DCGAN. However, its training time is longer, which is understandable given the added complexity of incorporating conditional information. Additionally, the loss function of the cDCGAN converges more smoothly, indicating a more stable training process.

Q2 Could we remove the vector y from the input of the discriminator (so having $cD(x)$ instead of $cD(x, y)$?

Here are the digits created by the model after modifying the discriminator's input. Some samples look like a mix of different classes, while others don't resemble any recognizable digit.

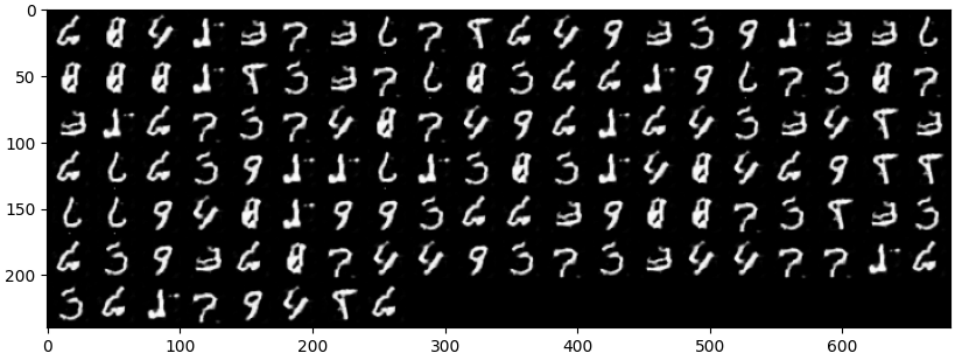
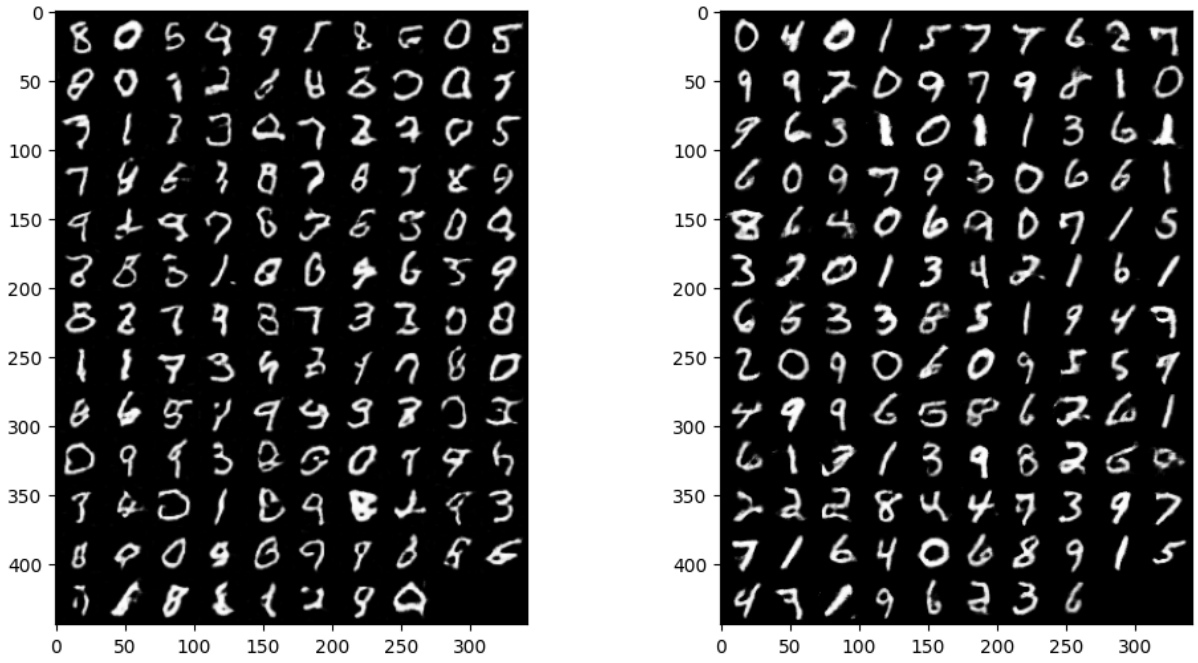


Figure 51: Digits Generated by the Model with Modified Discriminator Input

Obviously, if the vector y is removed from the discriminator's input, it would significantly impact the training and performance of the cDCGAN. This is because the discriminator's role is not only to differentiate between real and fake images but also to ensure that the generated images align with the given condition y. Without y, the discriminator can no longer check if the generated images match the specified class. As a result, the generator struggles to create digits that match the target classes. Also, removing y makes training harder. The generator has to learn how to produce all kinds of images at the same time without any guidance for specific classes, leading to blurry and unclear images with mixed or inconsistent categories.

Q3 Was your training more or less successful than the unconditional case ? Why ?

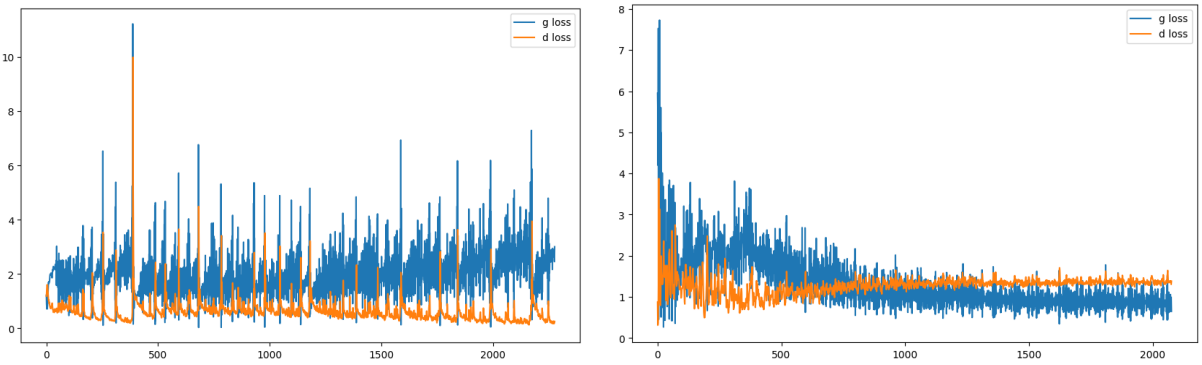


(a) DCGAN (b) cDCGAN

Figure 52: Digits Generated by GAN and cGAN Models

The training of the cDCGAN was significantly more successful compared to the unconditional DCGAN. As shown in Figure 52, the samples generated by the cGAN are much clearer and easier to distinguish.

The cDCGAN leverages the conditional label y, which provides explicit guidance for generating images corresponding to specific classes. This additional information helps the generator focus on creating digits consistent with the class labels, resulting in samples that are not only higher quality but also more diverse within each class. In contrast, the unconditional DCGAN struggles with consistency due to the lack of structured guidance. As a result, many of its samples are distorted or noisy, with some digits being difficult to recognize.



(a) DCGAN (b) cDCGAN

Figure 53: Generator and Discriminator Loss Dynamics in DCGAN and cDCGAN Models over 5 Epochs

Additionally, as shown in Figure 53, the generator and discriminator losses in the DCGAN fluctuate significantly throughout the training process. This reflects the instability commonly associated with adversarial training. In contrast, the cDCGAN exhibits much more stable and convergent loss curves. The conditional label y reduces the complexity of the adversarial game by narrowing the generator's focus to a specific class for each input. This stability not only improves training dynamics but also accelerates convergence.

In conclusion, the cDCGAN achieves more successful training by utilizing the conditional label y, which effectively guides both the generator and discriminator. This guidance enables the cDCGAN to produce higher-quality samples that are consistent with class labels and stabilizes the training process, making it more robust and efficient compared to the unconditional DCGAN.

Q4 Test the code at the end. Each column corresponds to a unique noise vector z. What could z be interpreted as here ?

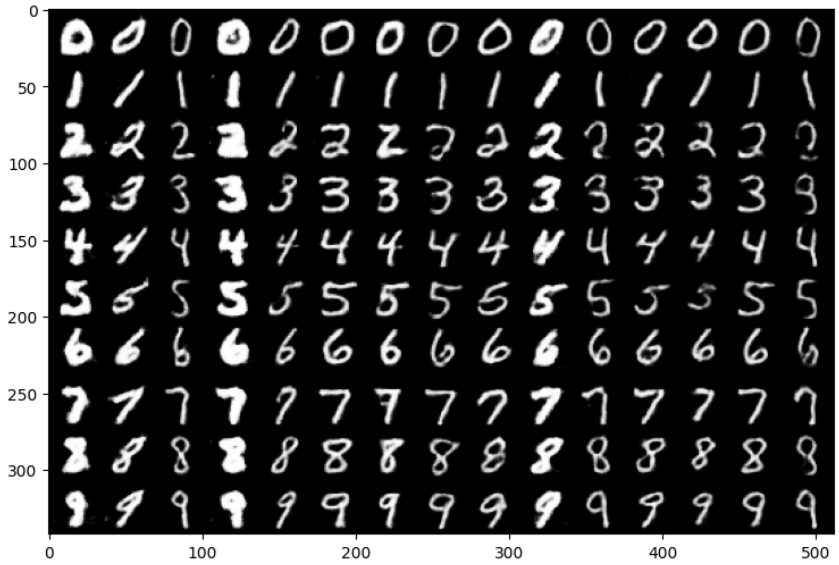


Figure 54: Visualization of the impact of z on generation

From this generated image, we can see that although the class of digits changes across rows, the overall style remains consistent within a column.

In that case, z can be interpreted as a latent vector that controls the style or global characteristics of the generated images. This latent vector captures abstract features such as the thickness of strokes, the slant, or the curvature of the handwritten digits.

In simple terms, z decides the "style," and y decides the "class" of the digit. This separation allows the cDCGAN to create a wide range of images where the digits are the same type but have different styles.

4 Conclusion

In this work, we explored both Generative Adversarial Networks (GANs) and Conditional Generative Adversarial Networks (cGANs) through various experiments and eval-

uations. Traditional GANs, despite generating realistic data, faced challenges such as training instability, mode collapse, and difficulties in balancing the adversarial game between the generator and discriminator. These issues were evident in experiments involving changes in noise vector size, training epochs, and dataset resolution.

On the other hand, cGANs significantly improved generation quality and stability by incorporating conditional information. By leveraging explicit guidance through labels, cGANs produced clearer, more diverse outputs aligned with specific conditions, and demonstrated smoother training dynamics. The conditional discriminator further ensured that generated samples matched the desired attributes, resulting in a more robust and efficient training process compared to the unconditional GAN.

Overall, our findings underscore the importance of conditional information in enhancing the capabilities of GANs, particularly for tasks requiring precise and interpretable outputs, such as class-specific image generation.