M2 M2A

Report for RDFIA Homework

Section 2
Deep learning applications

Students :
Kai MA, Ruyi TANG, Jinyi WU

December 7, 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.

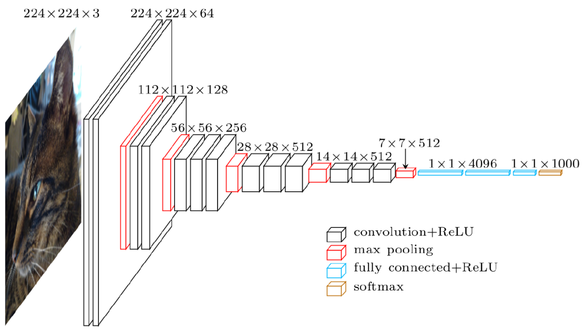


Figure 1: VGG16 Networks

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.

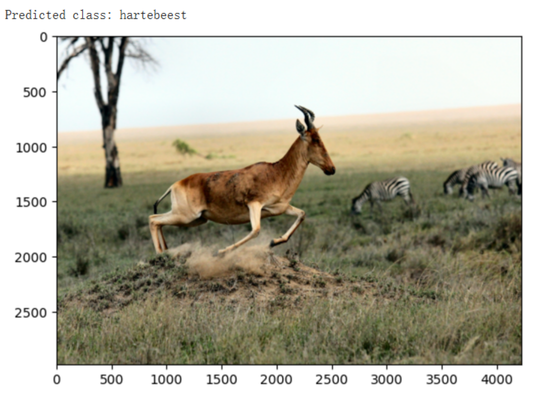
Q3* Apply the network on several images of your choice and comment on the results.

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

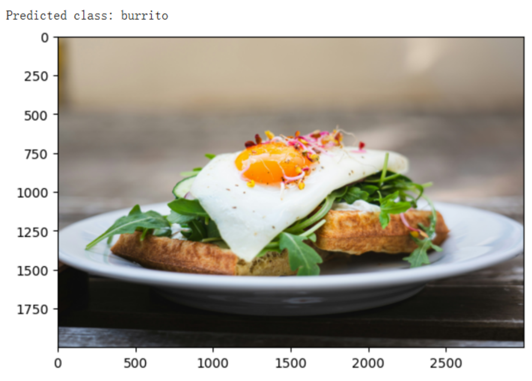
We applied the VGG16 network on several images, the results is as follows:

(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?

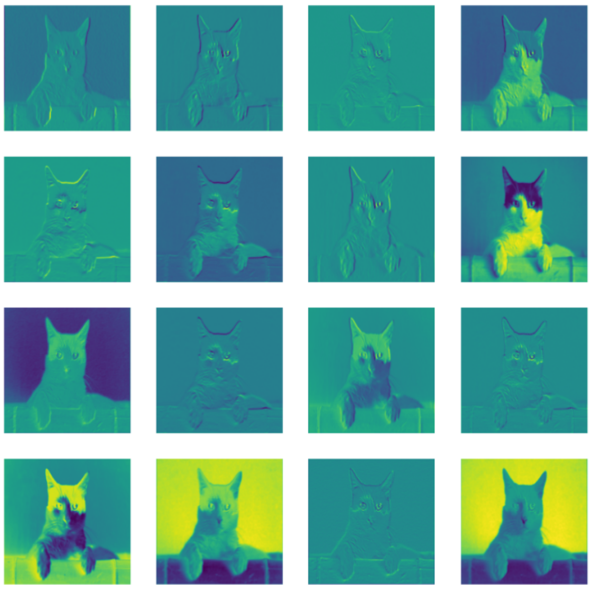


Figure 3: Activation Maps

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.

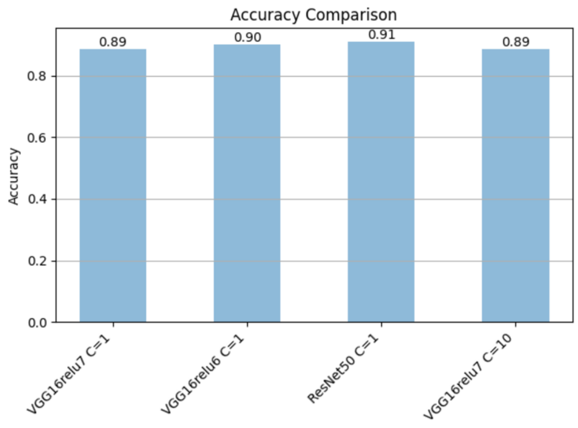


Figure 4: Comparison of accuracy

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:

2-c:

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., $y_i = 1$:

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

- 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:

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

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

- fixed D, optimize G

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

- fixed G, optimize D

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

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, $n_z = 1000$, $ngf = 32$, $ndf = 32$.

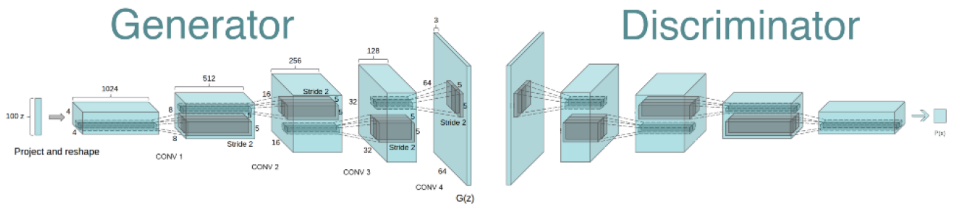


Figure 5: Architecture of the classic DCGAN

- **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 $outputsize : (128, 1, 1, 1)$

3 Conditional Generative Adversarial Networks