IMAGE TRANSLATION AND UDA

Jamwal Snigdha, Ramanathan Thanya, Tricoire Timour

Matric. No.: G2304207H, G2303746K, G2304288L
Email: SNIGDHA001@e.ntu.edu.sg, THANYA001@e.ntu.edu.sg, TIMOURLO001@e.ntu.ed.sg

1 Introduction

Unsupervised Domain Translation Unsupervised Domain Adaptation (UDA) may sound complex, but it's all about making deep neural networks smarter in handling new types of data. Imagine it as a tool that helps these networks better understand and work with different sets of information. UDA involves transforming objects in a way that keeps the important details intact, all while learning from new data without direct examples.

In the research by Hoffman et al., UDA [1] is revealed as a game-changer for deep neural networks, teaching them to apply what they've learned to fresh datasets. This paper dives into UDA's details, focusing on practical methods like aligning features through adversarial discriminator accuracy—a technique explained by Hoffman et al. Our goal is to demystify UDA and showcase its potential for improving how neural networks adapt and perform in image classification.

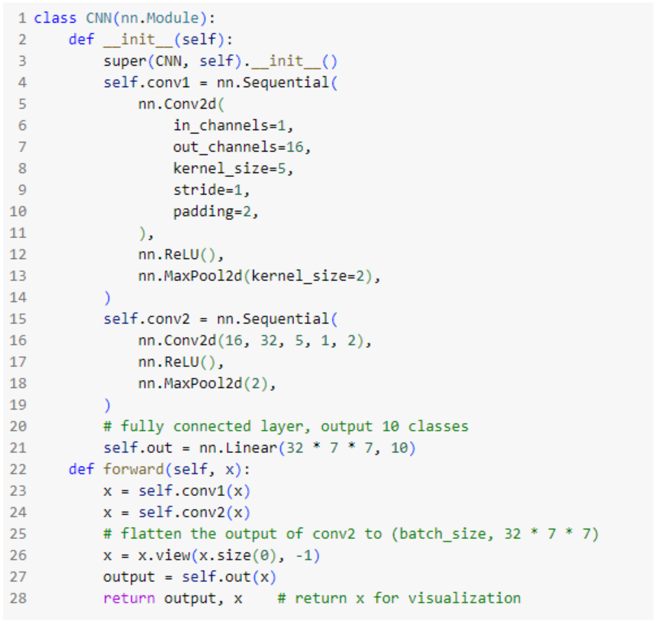
2 Methodology and Implementation

2.1 Dataset

For computing power reasons, we decided to use the MNIST and USPS datasets. We preprocessed all images in our dataset by applying a simple transform: first a resize to 28×28, for computing power reasons; next, a conversion to greyscale, discarding information that is irrelevant to the task; finally, a normalisation, bringing both the mean value and standard deviation of the image tensor to 0.5. The importance of that last step cannot be overstated; we empirically found that its simple addition changed the performance of our UDA model from 30% to 90% on our target dataset.

2.2 CNN - Convolutional Neural Network

We decided to perform image classification to check how the UDA helps to improve a model's performance. We approached this using a standard CNN. First, we trained a standard CNN to perform image classification on the MNIST dataset, using a network with two convolutions, each using the Relu activation function and followed by a pooling layer; then, a single fully connected layer, leading to ten output classes. This network was trained for only 10 epochs on the MNIST test dataset of 10000 images due to space and computational limitations. We also tested it on the USPS dataset, and it reached an accuracy of only 68%.



```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x    # return x for visualization
```

Fig. 1. Code for the CNN

2.3 CycleGAN

To generalise our method to the USPS dataset, we first train a cycleGAN. The structure of a CycleGAN Generator, broadly defined, is simple: First, an input is shrunk using convolutions. Afterward, residual blocks are applied. A residual block is a block consisting of the application of several convolutions to the input, and the summing of the result to the original input. Next, transpose convolutions are applied to increase the size of the image to the target size. Finally, a convolution is applied to the image. Our implementation follows this general structure, adding normalisation (batch in residual layers and instance otherwise) and Relu between each convo-

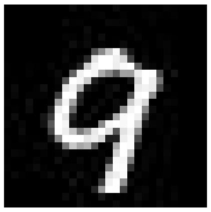
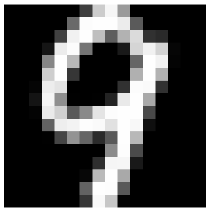
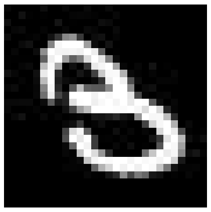
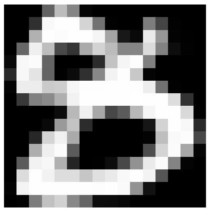
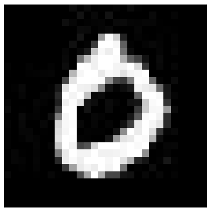
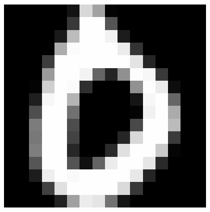
Inputs	Outputs
	
	
	

Table 1. Inputs and outputs of the CycleGAN. Note that the second row includes an incorrect translation.

lution. It counts three convolutions in the first step, six residual blocks, and two transpose convolutions in the last stage.

The network is then trained using a discriminator, another neural network simultaneously trained to discriminate (hence the name) images from the target dataset to images generated by the CycleGAN Generator. Thus, as one model improves, so does the other. Our discriminator consists simply of four convolution layers, with instance norm after the second and third convolutions, and a leaky relu before each convolution except the first. We thus train the the CycleGAN with MNIST images as source, and USPS images as target. The general shape of the numbers is largely preserved by the fact that all layers in the generator are convolutional, enforcing the preservation of local patterns. Once trained, the generator creates images as in table 1.

2.4 UDA

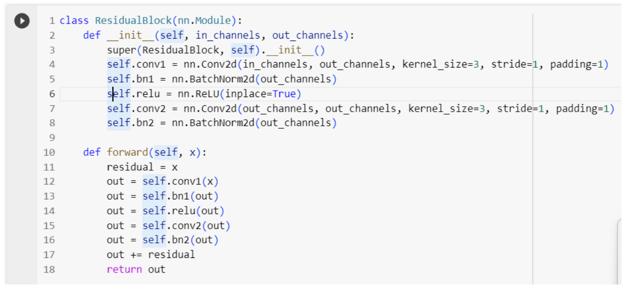
The CycleGAN allows us to train an image classification model for use on the USPS dataset without having labels for the USPS dataset. We first generate a new dataset by applying the previously trained CycleGAN to the MNIST dataset. We can then train a CNN with two convolutions (each followed by a Relu and a max-pooling. After training, we can once again test the model's performance on the USPS dataset. Despite being trained only on the MNIST dataset, the new CNN has at test accuracy of 90% – a far cry from 68% without using UDA.

3 Implementation

This CycleGAN model is designed for image-to-image translation, specifically for generating realistic images in a cyclical manner between two domains (e.g., translating images from domain A to domain B and vice versa). The model consists of three main components: the ResidualBlock, CycleGANDiscriminator, and CycleGANGenerator.

3.1 Residual Block

A key component of deep neural networks that aids in solving the vanishing gradient issue is the residual block. It has two convolutional layers using rectified linear unit (ReLU) activation functions and batch normalization. The residual connection improves the network's training and convergence by enabling the model to understand the distinction between the input and output.



```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out += residual
        return out
```

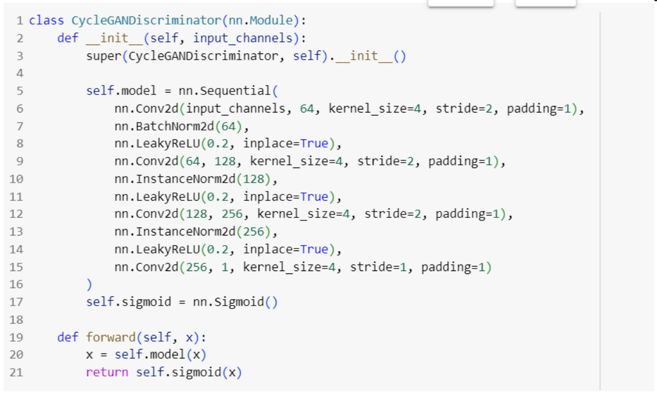
Fig. 2. Code for the Residual Block

3.2 Discriminator

The task of differentiating between authentic and fraudulent images falls to the CycleGANDiscriminator. It is a four-layer convolutional neural network with a leaky ReLU activation and either batch or instance normalization after each convolutional layer. The last layer generates a single-channel output that shows how likely it is that the input image is real. Utilizing a sigmoid activation function, the output is compressed to lie between 0 and 1.

3.3 Generator

The purpose of the CycleGANGenerator is to convert input images between different domains. It is made up of a decoder, an encoder, and several residual blocks. To create the output image, the encoder down-samples the input image, the residual blocks improve the feature representation, and the decoder up-samples the features. To guarantee stable training, the generator uses batch normalization, ReLU activation functions, and instance normalization. A hyperbolic tangent



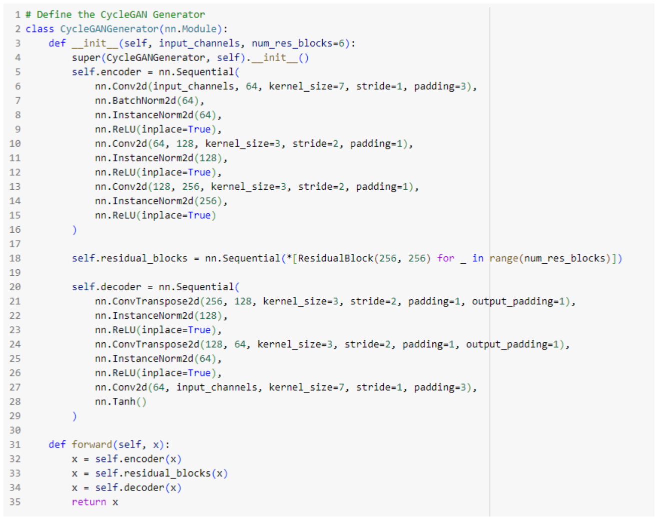
```python
class CycleGANDiscriminator(nn.Module):
    def __init__(self, input_channels):
        super(CycleGANDiscriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(input_channels, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(256, 1, kernel_size=4, stride=1, padding=1)
        )
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.model(x)
        return self.sigmoid(x)
```

Fig. 3. Code for the Discriminator

(Tanh) activation is applied to the output to guarantee that the pixel values fall between [-1, 1].



```python
# Define the CycleGAN Generator
class CycleGANGenerator(nn.Module):
    def __init__(self, input_channels, num_res_blocks=6):
        super(CycleGANGenerator, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(input_channels, 64, kernel_size=7, stride=1, padding=3),
            nn.BatchNorm2d(64),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1),
            nn.InstanceNorm2d(256),
            nn.ReLU(inplace=True)
        )

        self.residual_blocks = nn.Sequential(*[ResidualBlock(256, 256) for _ in range(num_res_blocks)])

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.InstanceNorm2d(128),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, input_channels, kernel_size=7, stride=1, padding=3),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.residual_blocks(x)
        x = self.decoder(x)
        return x
```

Fig. 4. Code for the Generator

3.4 Model Workflow

During the forward pass, an input image is first processed by the encoder of the CycleGANGenerator to extract features. These features go through a stack of ResidualBlocks, capturing high-level representations. The decoder then transforms these features back into the original domain. The CycleGAN's unique characteristic lies in its cycle consistency, where images translated from domain A to B and back to A (and vice versa) should resemble the original input, helping the model learn meaningful mappings between the domains without paired training data. The discriminators play a crucial role in adversarial training, providing feedback to improve the realism of the generated images.

4 Evaluation and Constraints

The application of UDA successfully allowed for a model to perform well applying image recognition on a different dataset than the one it was trained on, without that latter dataset being labelled; in that sense, UDA using a CycleGAN is a very successful technique. However, it still has constraints that limit its effectiveness. First, the very factor that makes the strength of the CycleGAN, the focus on locality that allows it to run unsupervised, also means that the source and target dataset must maintain mostly the same shape. If, for example, part of the difference between the two datasets is a certain homomorphy, such as a rotation and a skew, this technique would no longer be sufficient. Another flaw of the technique is that any model can only generalise to one dataset; all other target datasets require the entire process to be run anew, or at the very least, for a CycleGAN to be trained from them to the source dataset, from which they can be once more translated to the target dataset – a process that introduces potential infidelity. Apart from these two foibles, however, the UDA technique remains extremely useful for using an easily labelled dataset as a proxy for a harder to label one, allowing more productive use to be made of unlabelled data.

5 References

[1] Hoffman, Judy, et al. ”Cycada: Cycle-consistent adversarial domain adaptation.” International conference on machine learning. Pmlr, 2018.

[2] Nutan. “PyTorch Convolutional Neural Network With MNIST Dataset.” Medium, 6 Jan. 2022, medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118.

[3] “Text of prompt” prompt. ChatGPT, Day Month version, OpenAI, Day Month Year, chat.openai.com/chat.