Deep Learning and Practice Lab 7

309552007

袁鈺勛

A. Introduction

這次 lab 是要實作 conditional GAN 和 conditional normalizing flow。在 cGAN 中，利用 condition 以及 latent code 讓 generator 產生 fake image，並利用 discriminator 來區分收到的 image 是否是真實的，藉此來訓練 generator 以及 discriminator。在 cNF 中，image 經由 flow 轉成 latent code，如果要生成 image 的話，會將 latent code 經由 inverse flow 來還原。

B. Implementation details

1. GAN

在 gan 的選擇我是使用了最簡單的 dcgan，progressive growing of gan 是因為時間上所以沒有去考慮他，super resolution gan 是因為他的設計功能是將低解析度或模糊的照片輸入 generator 來產生高解析度的照片，所以不太適合這次的目的，sagan 則是有實作但不知道是不是哪裡實作錯誤導致效果不理想，所以沒有採納。在 condition 處理方面在 discriminator 裡面是將 label 經過 neural network 變成 1 channel 的 image 模樣的 condition，並且將他 concatenate 到 image 中當成第四個 channel，至於在 generator 裡面是在產生 noise 的時候就將他當成了 noise 的一部份讓 generator 產生 image。上述的方式便是下面兩張 code。

```python
class DCGenerator(nn.Module):
    def __init__(self, noise_size: int, label_size: int): ...

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Generator forwarding
        :param x: Batched data
        :return: Batched image
        """
        return self.net(x)
```

```python
class DCDiscriminator(nn.Module):
    def __init__(self, num_classes: int, image_size: int): ...

    def forward(self, x: torch.Tensor, label: torch.Tensor) -> torch.Tensor:
        """
        Discriminator forwarding
        :param x: Batched data
        :param label: Batched labels
        :return: Discrimination results
        """
        batch_size, num_classes = label.size()
        label = label.view(batch_size, num_classes, 1, 1)
        condition = self.label_to_condition(label).view(batch_size, 1, -1)
        condition = self.linear(condition).view(-1, 1, self.image_size, self.image_size)
        inputs = torch.cat([x, condition], 1)
        return self.net(inputs).view(-1, 1)
```

2. NF

在 normaling flow 的選擇我是使用了 glow，考量到 glow 的效果比 realnvp 好，不使用效果比 glow 好的 flow++是因為 flow++用到了 variational dequantization，所以他的架構上就像是有兩個 glow，在工作站的顯卡記憶體沒那麼大，連 glow 都不能用到常見的大小和深度，所以只有採用 glow，但在 prerpocess 是採用 realnvp 的 logits 方式而不是 glow 原先的單純加 uniform noise。在 condition 處理方面是將 label 經過 neural network 轉成 1 channel 的 image 模樣的 condition，當成參數和 image 一起往 glow 裡面計算。

```python
class ActNorm(nn.Module):
    """
    Activation normalization for 2D inputs.
    The bias and scale get initialized using the mean and variance of the first mini-batch.
    After the init, bias and scale are trainable parameters.
    Adapted from: https://github.com/openai/glow
    :arg in_channels: Number of channels in the input
    :arg scale: Scale factor for initial logs
    """

    def __init__(self, in_channels: int, scale: float = 1.): ...

    def initialize_parameters(self, x: torch.Tensor) -> None: ...

    def _center(self, x: torch.Tensor, reverse: bool = False) -> torch.Tensor: ...

    def _scale(self, x: torch.Tensor, sld: torch.Tensor, reverse: bool = False) -> Tuple[torch.Tensor,

    def forward(self, x: torch.Tensor, sld: torch.Tensor = None, reverse: bool = False) -> Tuple[torch.
```

```python
class InvConv(nn.Module):
    """
    Invertible 1×1 Convolution for 2D inputs.
    Originally described in Glow (https://arxiv.org/abs/1807.03039).
    :arg num_channels: Number of channels in the input and output
    """

    def __init__(self, num_channels: int): ...

    def forward(self, x: torch.Tensor, sld: torch.tensor, reverse: bool = False) -> Tuple[torch.Tensor,
```

```python
class Coupling(nn.Module):
    """
    Affine coupling layer
    :arg in_channels: Number of channels in the input
    :arg mid_channels: Number of channels in the intermediate activation in NN
    """

    def __init__(self, in_channels: int, cond_channels: int, mid_channels: int): ...

    def forward(self,
                x: torch.Tensor,
                x_cond: torch.Tensor,
                sld: torch.Tensor,
                reverse: bool = False) -> Tuple[torch.Tensor, ...]: ...
```

上面三張圖是 glow 的基本架構，分別是 ActNorm、Invertible 1x1 Convolution 和 Affine Coupling。

```python
class NN(nn.Module):
    """
    Small convolutional network used to compute scale and translate factors.
    :arg in_channels: Number of channels in the input
    :arg cond_channels: Number of channels in the condition
    :arg mid_channels: Number of channels in the hidden activations
    :arg out_channels: Number of channels in the output
    """

    def __init__(self, in_channels: int, cond_channels: int, mid_channels: int, out_channels: int): ...

    def forward(self, x: torch.Tensor, x_cond: torch.Tensor) -> torch.Tensor: ...
```

上圖中的 neural network 是用在 affine coupling 裡面，藉由 condition 和一半 channel 的 image 產生 scale 和 translate。

```python
class FlowStep(nn.Module):
    """
    Single flow step
    Forward: ActNorm -> InvConv -> Coupling
    Reverse: Coupling -> InvConv -> ActNorm
    :arg in_channels: Number of channels in the input
    :arg cond_channels: Number of channels in the condition
    :arg mid_channels: Number of hidden channels in the coupling layer
    """

    def __init__(self, in_channels: int, cond_channels: int, mid_channels: int): ...

    def forward(self,
                x: torch.Tensor,
                x_cond: torch.Tensor,
                sld: torch.Tensor = None,
                reverse: bool = False) -> Tuple[torch.Tensor, ...]: ...
```

上圖中便是 glow 中的一個 flow step，會經過上面提到的三個基本架構。

```python
class CGlow(nn.Module):
    """
    Conditional Glow model
    :arg num_channels: Number of channels in the hidden layers
    :arg num_levels: Number of levels in the model (number of _CGlow classes)
    :arg num_steps: Number of flow steps in each level
    :arg num_classes: Number of classes in the condition
    :arg image_size: Image size
    """

    def __init__(self, num_channels: int, num_levels: int, num_steps: int, num_classes: int, image_size:

    def forward(self,
                x_label: torch.Tensor,
                x: torch.Tensor = None,
                reverse: bool = False) -> Optional[Tuple[torch.Tensor, ...] or torch.Tensor]:...

    def get_mean_and_logs(self, data: torch.Tensor, label: torch.Tensor) -> Tuple[torch.Tensor, ...]:...

    def _pre_process(self, x: torch.Tensor) -> Tuple[torch.Tensor, ...]: ...
```

最後會用這個 cGlow class 來遞迴式的建立每一層的多個 flow steps，

同時也是在這將拿到的 label 轉成 condition 的。

3. Hyperparameter

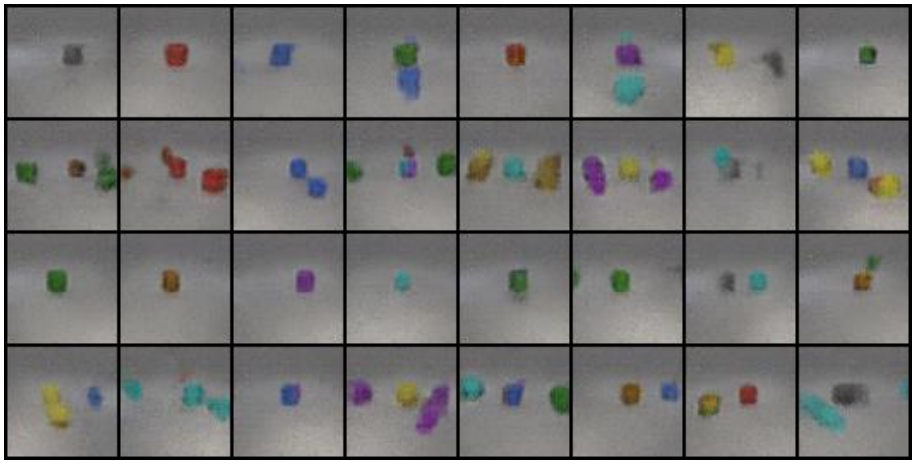
Argument	Description	Default
'-b', '--batch_size'	Batch size	64
'-i', '--image_size'	Image size	64
'-w', '--width'	Dimension of the hidden layers in normalizing flow	128
'-d', '--depth'	Depth of the normalizing flow	8
'-n', '--num_levels'	Number of levels in normalizing flow	3
'-gv', '--grad_value_clip'	Clip gradients at specific value	0
'-gn', '--grad_norm_clip'	Clip gradients' norm at specific value	0
'-lrd', '--learning_rate_discriminator'	Learning rate of discriminator	0.0002
'-lrg', '--learning_rate_generator'	Learning rate of generator	0.0002
'-lrnf', '--learning_rate_normalizing_flow'	Learning rate of normalizing flow	0.0005
'-e', '--epochs'	Number of epochs	300
'-wu', '--warmup'	Number of warmup epochs	10
'-t', '--task'	Task 1 or task 2	1 (1-2)
'-m', '--model'	cGAN or cNF	'dcgan'
'-inf', '--inference'	Only infer or not	False
'-v', '--verbosity'	Verbosity level	0 (0-2)

上圖為 hyperparameter 的設定，可以透過 argument 來調整。

C. Results and discussion

1. Task 1

a. cGAN

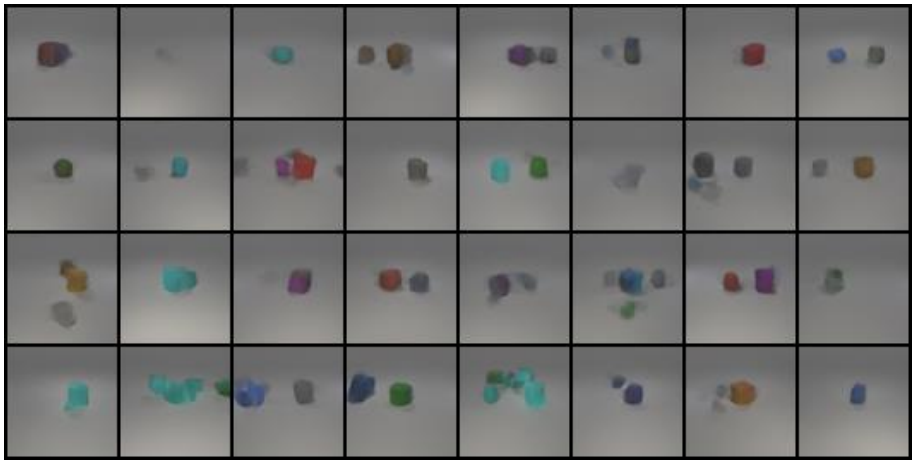


```
Average accuracy: 0.47
New Average accuracy: 0.51
```

上兩張圖分別是用 dcgan 產生的 test image 以及 test 和 new test 的 accuracy。相較於 normalizing flow 他的 latent code 的維度是比

image 還小的，所以他沒辦法如同 normalizing flow 有一對一關係，所以經由 latent code 產生的圖片有可能會 map 到別張，所以 accuracy 上下浮動比較大，但也因此他的 model 比 normalizing flow 小，也比較快收斂。

b. cNF



```
Average accuracy: 0.19
New Average accuracy: 0.19
```

上兩張圖分別是用 glow 產生的 test image 以及 test 和 new test 的 accuracy。Normalizing flow 的好處是他設定了 latent code 和 image 各自所在的空間維度是一樣大的，所以他可以保持一對一，讓產生的圖片更精準，但我上面的圖和 accuracy 都不太理想，可能是因為顯卡的記憶體不夠讓 glow 塞下夠大的 model，或是因為我只有跑 200 epochs，可能跑得不夠久。

2. Task 2

以下的實驗成果是用預設超參數的 glow，只有跑 250 epochs。

a. Conditional face generation

上圖的做法是從 training data 中挑出 32 個 label，將 label 丟入 glow 中做 inverse flow 產生的 fake images，就是下圖的 code。但

fake images 看起來成效不是很好，有可能是 glow 的超參數設太小或是訓練的不夠久

```
# Get labels for inference
labels = torch.rand(0, num_classes)
for idx in range(32):
    _, label = train_dataset[idx]
    labels = torch.cat([labels, torch.from_numpy(label).view(1, 40)], 0)
labels = labels.to(training_device).type(torch.float)

# Produce fake images
with torch.no_grad():
    fake_images, _, _ = normalizing_flow.forward(x=None, x_label=labels, reverse=True)
```

b. Linear interpolation

上圖的做法是從 training data 中選出 10 張照片，把他們兩兩經過 glow 得到各自的 latent code 後，得到 latent code 的 interval 和 label 的 interval，藉此便可以將新的 latent code 和 label 給 glow 做 inverse flow 得到 interpolation，就是下面的 code。可以看到 interpolation 在靠近中間的地方的 image 大多不是很清晰，原因應該就是在 face generation 提到的。

```
# Generate latent code
with torch.no_grad():
    first_z, _, _ = normalizing_flow.forward(x=first_image, x_label=first_label)
    second_z, _, _ = normalizing_flow.forward(x=second_image, x_label=second_label)

# Compute interval
interval_z = (second_z - first_z) / 8.0
interval_label = (second_label - first_label) / 8.0

# Generate linear images
for num_of_intervals in range(9):
    with torch.no_grad():
        image, _, _ = normalizing_flow.forward(x=first_z + num_of_intervals * interval_z,
                                               x_label=first_label + num_of_intervals * interval_label,
                                               reverse=True)
    linear_images = torch.cat([linear_images,
                               image.cpu().detach().view(1, 3, args.image_size, args.image_size)], 0)
```

c. Attribute manipulation

上圖的做法是先從 training data 中取得一個 image，將他經過 glow 取得 latent，並取得 negtive label 和 positive label 之間的 interval，再來是將所有的 image 經過 glow 取得 latent，並且將目標 attribute 是 positive 和 negative 的分別做平均並且得到他們的 interval，以這個 interval_z 和 image latent 以及 label 的 interval 便可以經過 glow 做 reverse flow 得到 images，就是下面的 code。上面圖中的第一個 row 是做 smiling，第二個 row 是做 bald，可以看到幾乎沒有什麼變化，原因應該就是在 face generation 中提到的。

```
with torch.no_grad():
    latent, _, _ = normalizing_flow.forward(x=image, x_label=label)

# Get negative labels
neg_smiling_label = torch.clone(label)
neg_bald_label = torch.clone(label)
neg_smiling_label[0, 31] = -1.
neg_bald_label[0, 4] = -1.

# Get positive labels
pos_smiling_label = torch.clone(label)
pos_bald_label = torch.clone(label)
pos_smiling_label[0, 31] = 1.
pos_bald_label[0, 4] = 1.

# Compute conditional interval
interval_smiling_label = (pos_smiling_label - neg_smiling_label) / 4.0
interval_bald_label = (pos_bald_label - neg_bald_label) / 4.0
```

```
pos_z_mean = torch.zeros(*(latent.size()), dtype=torch.float)
neg_z_mean = torch.zeros(*(latent.size()), dtype=torch.float)
num_pos, num_neg = 0, 0
for images, labels in data_loader:
    images = images.to(training_device)
    labels = labels.to(training_device).type(torch.float)
    pos_indices = (labels[:, idx] == 1).nonzero(as_tuple=True)[0]
    neg_indices = (labels[:, idx] == -1).nonzero(as_tuple=True)[0]

    with torch.no_grad():
        z, _, _ = normalizing_flow.forward(x=images, x_label=labels)
    z = z.cpu().detach()

    if len(pos_indices) > 0:
        num_pos += len(pos_indices)
        pos_z_mean = (num_pos - len(pos_indices)) / num_pos * pos_z_mean + z[pos_indices].sum(dim=0)
    if len(neg_indices) > 0:
        num_neg += len(neg_indices)
        neg_z_mean = (num_neg - len(neg_indices)) / num_neg * neg_z_mean + z[neg_indices].sum(dim=0)
interval_z = 1.6 * (pos_z_mean - neg_z_mean)
interval_z = interval_z.to(training_device)
```

```
alphas = [-1.0, -0.5, 0.0, 0.5, 1.0]
for num_of_intervals, alpha in enumerate(alphas):
    with torch.no_grad():
        image, _, _ = normalizing_flow.forward(x=latent + alpha * interval_z,
                                               x_label=neg_label + num_of_intervals * interval_label,
                                               reverse=True)
    manipulated_images = torch.cat([manipulated_images,
                                    image.cpu().detach().view(1, 3, args.image_size, args.image_size)],
```