Homework #4

Deep Learning for Computer Vision

Problem 1: Prototypical Network (50%)

1. (11%)

```
PrototypicalNet(
  (extractor): Encoder(
    (encoder): Sequential(
      (0): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
        )
      )
      (1): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
        )
      )
      (2): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
        )
```

```
        )
        (3): Conv_Block(
          (conv): Sequential(
            (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (2): ReLU(inplace=True)
            (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
          )
        )
      )
    )
    (mlp): MLP(
      (enhance): Sequential(
        (0): Linear(in_features=1600, out_features=400, bias=True)
        (1): ReLU(inplace=True)
        (2): Linear(in_features=400, out_features=1600, bias=True)
        (3): Dropout(p=0.5, inplace=False)
      )
      (transform): Sequential(
        (0): ReLU(inplace=True)
        (1): Linear(in_features=1600, out_features=1600, bias=True)
      )
    )
)
Total number of params = 3956688
```

Model implementation details:

在本題中使用的 Prototypical Net 包含了助教所規定的 CNN Feature Extractor (class Encoder) 以及自行設計的 Multilayer Perceptron (class MLP)。

Encoder 主要使用了四個 Conv_Block(Conv2d, BatchNorm2d, ReLU, MaxPool2d)，並有使用 Gaussian noise 來 initialize weights。

MLP 主要分成兩個 branch (enhance, transform)，enhance 部分包含了兩個 Linear module，之間有一個 ReLU module，最後還有一個 Dropout module (p = 0.5)。而 transform 部分只包含了一個 ReLU module 和一個 Linear module。兩個 branch 皆有利用 Gaussian noise 來 initialize weights。

先將原 Image 經過 Encoder 產生初步的 feature，然後經過 MLP 時，會先將初步的 feature 經過 enhance module (其中經過降維以降低參數、運算量)，產生 enhance feature，然後再將此 enhance feature 乘上 scale 係數，和原初步 feature 相加，藉以達到加重某些特定重要 feature weight 的效果，再將此 feature 經過 activation function 和最後的 Linear module，產生最後的 feature。

為助理解，架構流程如下圖所示：

```
class PrototypicalNet(nn.Module):
        def __init__(self):
                super(PrototypicalNet, self).__init__()
                self.extractor = Encoder()
                self.mlp = MLP(in_channels = 1600, out_channels = 1600)

        def forward(self, input):
                feature = self.extractor(input)
                feature = self.mlp(feature)
                return feature
```

```
class Conv_Block(nn.Module):
        def __init__(self, in_channels, out_channels, padding):
                super(Conv_Block, self).__init__()
                self.conv = nn.Sequential(
                                                nn.Conv2d(in_channels, out_channels, 3, padding = padding),
                                                nn.BatchNorm2d(out_channels),
                                                nn.ReLU(True),
                                                nn.MaxPool2d(2),
                                        )
        def forward(self, input):
                return self.conv(input)


class Encoder(nn.Module):
        def __init__(self, in_channels = 3, hidden = 64, out_channels = 64):
                super(Encoder, self).__init__()
                self.encoder = nn.Sequential(
                                                        Conv_Block(in_channels, hidden, 1),
                                                        Conv_Block(hidden, hidden, 1),
                                                        Conv_Block(hidden, hidden, 1),
                                                        Conv_Block(hidden, out_channels, 1),
                                                )
                self.__initialize_weights()

        def forward(self, input):
                output = self.encoder(input).view(input.size(0), -1)
                return output

        def __initialize_weights(self):
                for m in self.modules():
                        if isinstance(m, nn.Conv2d):
                                m.weight.data.normal_(0, 1)
                                if m.bias is not None:
                                        m.bias.data.zero_()
                        elif isinstance(m, nn.BatchNorm2d):
                                m.weight.data.fill_(1)
                                m.bias.data.zero_()


class MLP(nn.Module):
        def __init__(self, in_channels, out_channels, scale = 0.1):
                super(MLP, self).__init__()
                self.scale = scale
                self.enhance = nn.Sequential(
                                                        nn.Linear(in_channels, in_channels // 4),
                                                        nn.ReLU(True),
                                                        nn.Linear(in_channels // 4, in_channels),
                                                        nn.Dropout(0.5),
                                                )
                self.transform = nn.Sequential(
                                                        nn.ReLU(True),
                                                        nn.Linear(in_channels, out_channels),
                                                )
                weights_init_uniform(self.enhance)
                weights_init_uniform(self.transform)

        def forward(self, input):
                enhance = self.enhance(input)
                output = self.transform(input + self.scale * enhance)
                return output
```

```python
def weights_init_uniform(m):
        classname = m.__class__.__name__
        # for every Linear layer in a model..
        if classname.find('Linear') != -1:
                # apply a uniform distribution to the weights and a bias=0
                m.weight.data.uniform_(0.0, 1.0)
                m.bias.data.fill_(0)
```

Training implementation details:

本題 Prototypical Net 一共 train 了 30 個 epochs，當中每個 epoch 的 episode 數目為 2000。使用 SGD optimizer，initial learning rate 為 1e-3，每 20 個 epoch 會衰減為當前的一半。當中使用了 RandomHorizontalFlip 的 data augmentation 技巧，distance function 採用的是 Euclidean distance，Loss function 為 cross entropy loss。在進行 meta-training 時，採用了 30 way 1 shot 的設定，在 meta-testing 則為 5 way 1 shot。

5 way 1 shot 設定之下，於 validation set 進行測試時，Accuracy 為 47.64 %。

2. (12%)

在 meta-training 和 meta-testing 皆為相同的 5 way 1 shot 設定且訓練的 epoch, episode 數量皆相同時，不同的 distance 計算方式所獲得的 accuracy 分別呈現如下：

Euclidean distance: 37.43 %

Cosine similarity: 39.15 %

Parametric function: 39.74 %

此處的 Parametric function 主要是結合了上述的 Euclidean distance 和 Cosine similarity 各自的特點。因為 Euclidean distance 主要反應的是兩個 features 間各個維度綜合起來的距離大小，而 Cosine similarity 則主要反應兩個 features 之間角度、方向上的差異，所以將兩者結合後，Parametric function 式子如下：

-Euclidean distance (F(feature1), F(feature2)) / (Cosine similarity(F(feature1), F(feature2)) + 2)

(其中 F 為一經過 training 所訓練出來的空間轉換 function)

大致概念就是先利用 F function，將原 features 重組、轉換至較能夠共同表現 Euclidean distance 和 Cosine similarity 差距的空間。然後分別計算 Euclidean distance 和 Cosine similarity，Euclidean distance 值越大代表兩 feature 間差距越大，所以乘上負號 (因為最後預測的 candidate 會選擇 Parametric function 結果最大的)；而 Cosine similarity 的值則會介於 [-1, 1]，將其結果 +2 平移至 [1, 3] 之間。再將兩者結果相除，已知分子皆為負數，而分母必介於 [1, 3] 之間，

所以當分母越大時，Parametric function 結果也會越大(其值越大代表兩 features 越接近，最後選擇預測結果最大者)。

透過比較三種不同 distance 計算方式的 accuracy，可以發現在此次測試中，表現優至劣為：Parametric function, Cosine similarity, Euclidean distance。除此之外，在大部分相同的 iteration 次數的 model 表現上，三者也是呈現如此的優至劣表現，而 Euclidean distance 和其餘兩者存在著較大的差距，其在 early epoch 時的 accuracy 也上升較慢、loss下降較慢。

3. (12%)

在 meta-training 和 meta-testing 皆為 5 way K shot 設定且訓練的 epoch, episode 數量皆相同、皆使用 Euclidean distance 時，不同 K 所表現的 accuracy 分別呈現如下：

K = 1: 37.43 %

K = 5: 58.55 %

K = 10: 62.96 %

可以發現，使用不同 shot 數量時，對 model 的表現影響非常大，K 越大時，往往 model 的表現越好，越容易 train，彼此存在著極大的差距，比如：

K = 1 early epoch model accuracy: 24.25 %

K = 1 final epoch model accuracy: 37.43 %

K = 5 early epoch model accuracy: 45.17 %

K = 5 final epoch model accuracy: 58.55 %

K = 10 early epoch model accuracy: 50.37 %

K = 10 final epoch model accuracy: 62.96 %

可以看到不同 K 時，存在著很明顯的 startup 以及 final performance 差距，當然 middle stage 也是如此。

Problem 2: Data Hallucination for Few-shot Learning (50%)

1. (10%)

```
PrototypicalNet(
  (extractor): Encoder(
    (encoder): Sequential(
      (0): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
        )
      )
      (1): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
        )
      )
      (2): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
          (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU(inplace=True)
          (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
        )
      )
      (3): Conv_Block(
        (conv): Sequential(
          (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
                (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
                (2): ReLU(inplace=True)
                (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
              )
            )
          )
    )
    (hallucinate): HallucinationNet(
      (encoder): Sequential(
        (0): Linear(in_features=1728, out_features=400, bias=True)
        (1): Dropout(p=0.2, inplace=False)
        (2): ReLU(inplace=True)
        (3): Linear(in_features=400, out_features=1600, bias=True)
      )
    )
    (mlp): MLP(
      (enhance): Sequential(
        (0): Linear(in_features=1600, out_features=400, bias=True)
        (1): ReLU(inplace=True)
        (2): Linear(in_features=400, out_features=1600, bias=True)
        (3): Dropout(p=0.5, inplace=False)
      )
      (transform): Sequential(
        (0): ReLU(inplace=True)
        (1): Linear(in_features=1600, out_features=1600, bias=True)
      )
    )
)
Total number of params = 5289888
```

Model implementation details:

在本題中使用的 Prototypical Net 包含了助教所規定的 CNN Feature Extractor (class Encoder) 以及自行設計的 Multilayer Perceptron (class MLP)，和額外新增的 Hallucination Net (class HallucinationNet)。

Encoder 和 MLP 的部分皆和 Problem1 完全相同，所以在此就不再贅述，注重於 HallucinationNet 的說明。

此題 HallucinationNet 包含了兩個 Linear module，之間有一個 Dropout module (p = 0.2)，以及一個 ReLU module，並有利用 Gaussian noise 來 initialize weights。

先將原 Image 經過 Encoder 產生初步的 feature，和 Problem1 不同的是，經過 MLP 之前，會先將初步產生的 feature 和從 normal distribution 所 sample 出的 noise 做 concatenate，再經過 HallucinateNet，產生 hallucinated feature，用以推測、模擬出其他此相同 class 的 feature。最後將這些 hallucinated feature 和對應的初步 feature 取平均，再經過 MLP，產生最後的 feature。

為助理解，架構流程如下圖所示：

```
class HallucinationNet(nn.Module):
        def __init__(self, args, in_channels = 1600, noise_channel = 128, out_channels = 1600):
                super(HallucinationNet, self).__init__()
                self.args = args
                self.encoder = nn.Sequential(
                                                        nn.Linear(in_channels + noise_channel, out_channels // 4),
                                                        nn.Dropout(0.2),
                                                        nn.ReLU(True),
                                                        nn.Linear(out_channels // 4, out_channels),
                                        )
                weights_init_uniform(self.encoder)

        def forward(self, feauture_noise):
                fake_feature = self.encoder(feauture_noise)
                return fake_feature


class PrototypicalNet(nn.Module):
        def __init__(self, args):
                super(PrototypicalNet, self).__init__()
                self.args = args
                self.extractor = Encoder()
                self.hallucinate = HallucinationNet(args)
                self.mlp = MLP(in_channels = 1600, out_channels = 1600)

        def augment(self, support_feature):
                fake_support_feature = []
                batch_size, channels = support_feature.shape
                for select_index in range(batch_size):
                        noise = torch.randn(self.args.h_degree, 128).cuda()
                        feauture_noise = torch.cat([support_feature[select_index].unsqueeze(0).expand(self.args.h_degree, -1), noise], dim = 1)
                        fake_support_feature.append(self.hallucinate(feauture_noise).unsqueeze(0))
                support_feature = support_feature.unsqueeze(1)
                fake_support_feature = torch.cat(fake_support_feature, dim = 0)
                mix_feature = torch.mean(torch.cat([support_feature, fake_support_feature], dim = 1), dim = 1).squeeze(1)
                return mix_feature, support_feature, fake_support_feature

        def forward(self, input):
                feature = self.extractor(input)
                mix_feature, support_feature, fake_support_feature = self.augment(feature)
                mix_feature = self.mlp(mix_feature)
                support_feature = self.mlp(support_feature)
                fake_support_feature = self.mlp(fake_support_feature)
                return mix_feature, support_feature, fake_support_feature
```

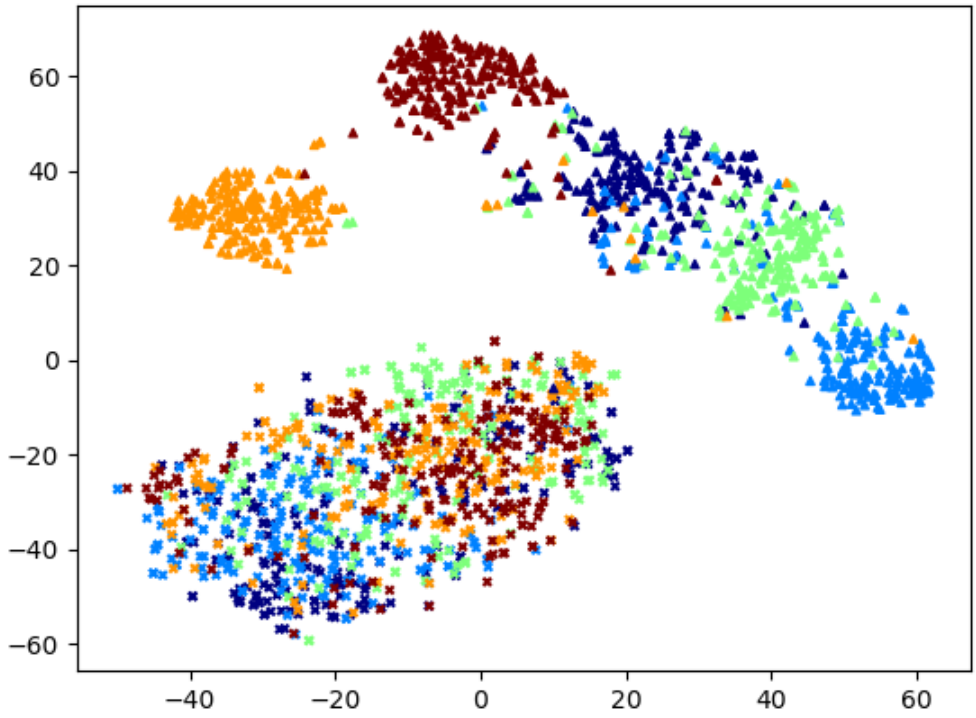
Training implementation details:

本題 Prototypical Net 一共 train 了 30 個 epochs，當中每個 epoch 的 episode 數目為 2000。使用 SGD optimizer，initial learning rate 為 1e-3，每 20 個 epoch 會衰減為當前的一半。當中使用了 RandomHorizontalFlip 的 data augmentation 技巧，distance function 採用的是 Euclidean distance，Loss function 為 cross entropy loss。在進行 meta-training 時，採用了 30 way 1 shot 200 augmentation 的設定，在 meta-testing 則為 5 way 1 shot 200 augmentation。

其實上述的設定之中，除了新增 M augmentation setting 之外，其餘部分和 Problem1 的部分完全相同，主要是想透過控制這些因素不變，主要來探討 training 方法不同之下的 model 表現優劣。

5 way 1 shot 200 augmentation 設定之下，於 validation set 進行測試時，Accuracy 為 49.48 %。

2. (10%)



透過觀察上方的 tsne 結果圖可以發現，在左下角 real data feature (x標示) 的分布上，雖然有些 class 之間有些微聚集的傾向，稍微可以用肉眼進行分堆，但不夠明顯且非常勉強。而在 hallucinated data feature (triangle標示) 的分布上，五種 class 能以肉眼輕易分堆，且不同 class 間都有著一定的距離，甚至相同 class 之間，有著些許朦朧、類似 normal distribution 的分佈，雖然不是那麼明確，但說和 sample 出來的 noise 有些關聯也不過分。

當 inference 的時候，因為利用了大量的 M augmentation hallucinated data feature 和 real data feature 做平均，可以達到將該 real data feature 轉換、拉移至好進行分類的空間中做分類，所以 accuracy 得以上升。

3. (10%)

在 meta-training 和 meta-testing 皆為 5 way 1 shot M augmentation 設定且訓練的 epoch, episode 數量皆相同、皆使用 Euclidean distance 時，不同 M 所表現的 accuracy 分別呈現如下：

M = 10: 36.53 %

M = 50: 38.92 %

M = 100: 39.72 %

觀察整個訓練過程的 model 表現後，發現在 M 比較小的 early epoch model，其 accuracy 上升較快、loss 下降較快，而在 M 比較大的 early epoch model，其 accuracy 上升較慢、loss 下降較慢，但在 middle, final epoch 時，情況則相反。

4. (5%)

可以發現，使用不同 M 數量時，對 model 的表現有產生影響，但不如前述 shot 數量的影響大，且在訓練途中，相同 epoch、不同 M 的 model 表現，也各自有高有低，沒有相對 shot 不同時，那麼穩定的結果。這部分可能源自於 HallucinationNet 的表現，當 HallucinationNet 表現比較不好或較不穩定時，可能因為取平均方法，連帶影響到 PrototypicalNet 的預測結果好壞。

當 HallucinationNet 表現不好時，M 越大，連帶影響到預測結果的程度也越大，而當 HallucinationNet 表現較好時，M 越大，連帶預測結果也會越好，實作起來有種雙面刃的感覺。

Problem 3: Improved Data Hallucination

Model (20%)

1. (9%)

```
Extractor(
  (encoder): Sequential(
    (0): Conv_Block(
      (conv): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      )
    )
    (1): Conv_Block(
      (conv): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      )
    )
    (2): Conv_Block(
      (conv): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): ReLU(inplace=True)
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
      )
    )
    (3): Conv_Block(
      (conv): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
            (2): ReLU(inplace=True)
            (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
          )
        )
    )
    (mlp): MLP(
      (enhance): Sequential(
        (0): Linear(in_features=1600, out_features=400, bias=True)
        (1): ReLU(inplace=True)
        (2): Linear(in_features=400, out_features=1600, bias=True)
        (3): Dropout(p=0.5, inplace=False)
      )
      (transform): Sequential(
        (0): ReLU(inplace=True)
        (1): Linear(in_features=1600, out_features=1600, bias=True)
      )
    )
)
Total number of params =    3956688
Generator(
    (generate): Sequential(
      (0): Linear(in_features=1728, out_features=400, bias=True)
      (1): ReLU(inplace=True)
      (2): Linear(in_features=400, out_features=1600, bias=True)
    )
)
Total number of params =    1333200
Discriminator(
    (judge): Sequential(
      (0): Linear(in_features=1600, out_features=400, bias=True)
      (1): LeakyReLU(negative_slope=0.2)
      (2): Linear(in_features=400, out_features=1, bias=True)
      (3): Sigmoid()
    )
)
Total number of params =    640801
```

Model implementation details:

在本題中使用的 Extractor 即是 Problem1 所使用的 Prototypical Net，兩者完全相同，所以在此就不再贅述，注重於 Generator 和 Discriminator 的說明。

此題 Generator 包含了兩個 Linear module，之間有一個 ReLU module，並有利用 Gaussian noise 來 initialize weights。

而 Discriminator 也是包含了兩個 Linear module，兩個 Linear module 之間有一個 LeakyReLU module，且最後還有個 Sigmoid module，也有利用 Gaussian noise 來 initialize weights。

Generator 會 sample 兩組 normal distribution noise 和 real feature 合併，以產生兩組 fake feature，而 Discriminator 則會做出真假的判斷，如同一般的 GAN model architecture。

為助理解，架構流程如下圖所示：

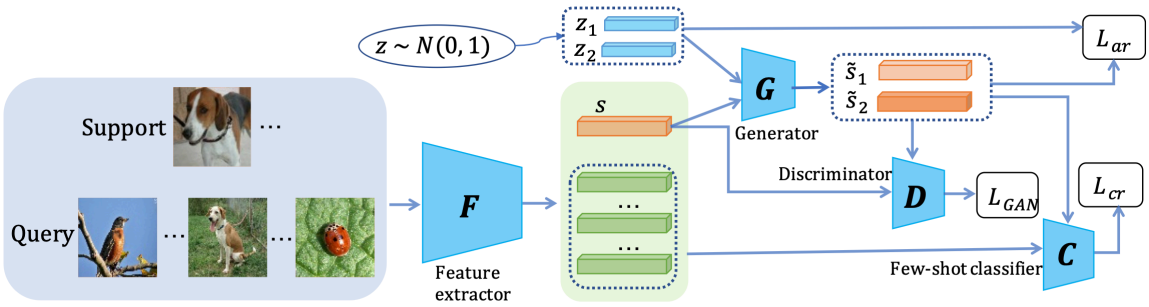
```
class Generator(nn.Module):
        def __init__(self, in_channels = 1600, noise_channel = 128, out_channels = 1600):
                super(Generator, self).__init__()
                self.generate = nn.Sequential(
                                                        nn.Linear(in_channels + noise_channel, in_channels // 4),
                                                        nn.ReLU(True),
                                                        nn.Linear(in_channels // 4, out_channels),
                                        )
                weights_init_uniform(self.generate)

        def forward(self, feautures):
                generate_size = 2
                noises = torch.randn(feautures.shape[0], generate_size, 128).cuda()
                fake_features = [self.generate(torch.cat([feautures[index].unsqueeze(0).expand(generate_size, -1), noises[index]], dim = 1)) for index in range(
                fake_features = torch.cat(fake_features, dim = 0)
                return fake_features, noises


class Discriminator(nn.Module):
        def __init__(self, in_channels = 1600, out_channels = 1):
                super(Discriminator, self).__init__()
                self.judge = nn.Sequential(
                                                nn.Linear(in_channels, in_channels // 4),
                                                nn.LeakyReLU(0.2),
                                                nn.Linear(in_channels // 4, out_channels),
                                                nn.Sigmoid()
                                        )
                weights_init_uniform(self.judge)

        def forward(self, input):
                output = self.judge(input)
                return output
```

Training implementation details:



本題的 improve 方法是參考 AFHN 的 paper，訓練的大致流程如上圖所示，不過有些細節部分有做修改，會於本小題最後處進行說明。

本題 Extractor 一共 train 了 40 個 epochs，當中每個 epoch 的 episode 數目為 2000。Extractor 使用 SGD optimizer，initial learning rate 為 1e-3，每 20 個 epoch 會衰減為當前的一半。而 Generator 和 Discriminator 使用 Adam optimizer，fixed learning rate 為 1e-4。當中使用了 RandomHorizontalFlip 的 data augmentation 技巧，distance function 採用的是 Euclidean distance，Loss function 為 cross entropy loss 和 Binary cross entropy。在進行 meta-training 時，採用了 30 way 1 shot 的設定，在 meta-testing 則為 5 way 1 shot。

其實上述的設定之中，除了和 Generator 和 Discriminator 有相關聯的部分之外，其餘部分和 Problem1 的部分大致相同，主要是想透過控制其餘因素不變，主要來探討 training 方法不同之下的 model 表現優劣。

5 way 1 shot 設定之下，於 validation set 進行測試時，Accuracy 為 50.22 %。

以下為主要不同或有做額外調整之處：
原 paper 在進行 Generator, Discriminator 訓練前，有先對 Extractor 做 pretrain 的動作，本次實作沒有對 Extractor 進行 pretrain，而是把 Extractor, Generator, Discriminator 三個 model 同時進行訓練，不過在利用 Generator 所產生的 fake feature 所做的預測計算出 loss，進行 backpropagation 更新 Extractor 之前，有利用隨之變動的 lambda 給以權重，即是希望 Generator 品質越差時，訓練所隨之調整的幅度越小。而原 paper 在每個 iteration 中，先訓練 Discriminator，再訓練 Generator，本次實作是依照 Extractor, Generator, Discriminator 的順序進行每個 iteration 的訓練。

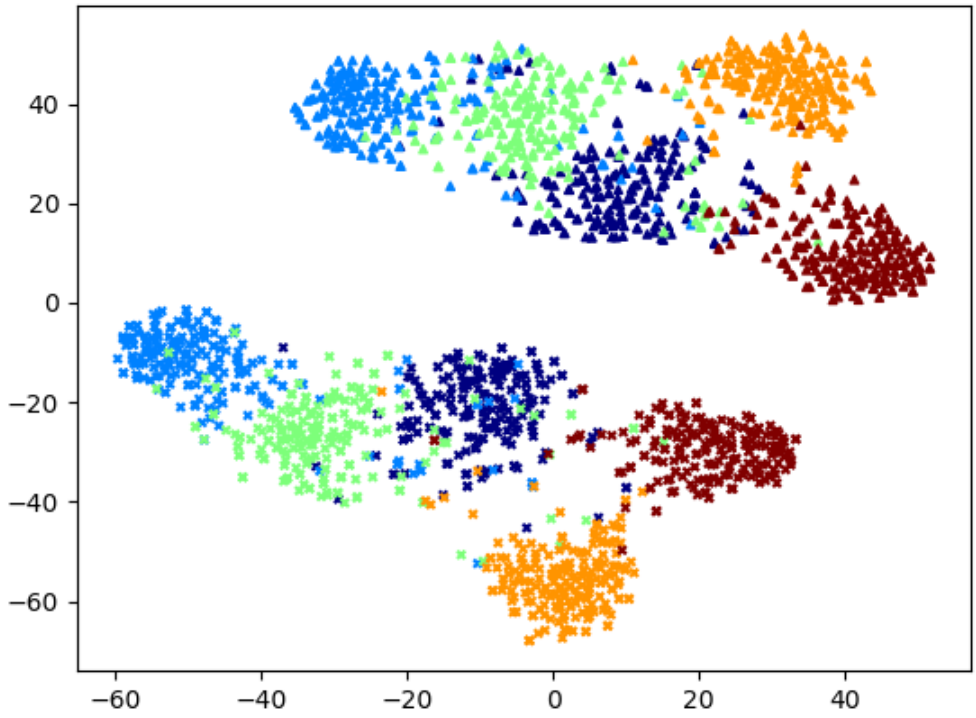
在 Loss function 的部分也有作出一些修改，classify loss 和 GAN loss 的部分並沒有更動，但把 anti-collapse loss 更改為自行設計的 Cosine similarity loss + Euclidean distance loss，公式分別如下：

Cosine similarity loss: (參考自原 paper 的 anti-collapse loss)
abs(mean((1 - Cosine similarity(feature1, feature2)) / (1 - Cosine similarity(noise1, noise2))) – 1.0)

Euclidean distance loss:
abs(mean(softmax(Euclidean distance (noise1, noise2)) / softmax(Euclidean distance(feature1, feature2))) – 1.0)

上述兩個 loss function 的主要概念即是去計算 noise1, noise2 兩者的差距(角度方向、大小) 和 feature1, feature2 兩者的差距(角度方向、大小)，並將兩者差距的相對值相除再取平均，並透過平移，將其收斂在 1 的位置 (兩者的差異比值為1代表分子分母的差異值相同，即是原 anti-collapse loss 想要達到的目的)，所以高於 1, 低於 1 皆會有相對應的 loss 計算產生。

2. (2%)



3. (4%)

透過觀察上方的 tsne 結果圖可以發現，不管是在左下角 real data feature (x標示) 還是 fake data feature (triangle標示) 的分布上，五種 class 皆能以肉眼輕易分堆，且不同 class 間都有著一定的距離。

這點跟 Problem2 的差異，可能跟 training 的方式有關，因為在本題的方法之中，也會針對 fake data feature 去進行 Extractor 的參數更新，且 Generator 所產生的 fake data feature 也會越來越像 real data feature，所以漸漸 Extractor 所產出的 real feature 也越來越容易被分類；而前一題中，主要是利用到 M augmentation 所產生的 data feature，去對原

Extractor 所產生的 feature 進行平均、校正，對其 Extractor 的更新程度相對較小，所以在原來的空間中，還是沒有辦法做出好的分辨，主要是依靠 HallucinationNet 所產生的 hallucinated data feature。

另外一個 improve 點是在 Problem3 時，有 Discriminator 負責維持 Generator 所產生的 fake feature 真實與否的品質，也有各種不同層面上的 loss 來確保 fake feature 的豐富性，所以額外產生的 feature 品質較高，對於 Extractor 的訓練來說比較有益處。而 Problem2 時，HallucinateNet 只求自行所產生的 feature 能夠被分類，有種用暴力、feature 數量來解決分類問題的感覺，額外的 feature 品質較低。

除上述之外，Problem3 的 Extractor model 訓練時間花費較少、model 參數量較低，有著較高的 accuracy，而 Problem2 的 Extractor model 訓練時間花費較多、model 參數量較高，有著相對較低的 accuracy。

URL:

https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

https://arxiv.org/pdf/2003.13193.pdf

https://arxiv.org/pdf/1801.05401.pdf

https://www.youtube.com/watch?v=rHGPfl0pvLY