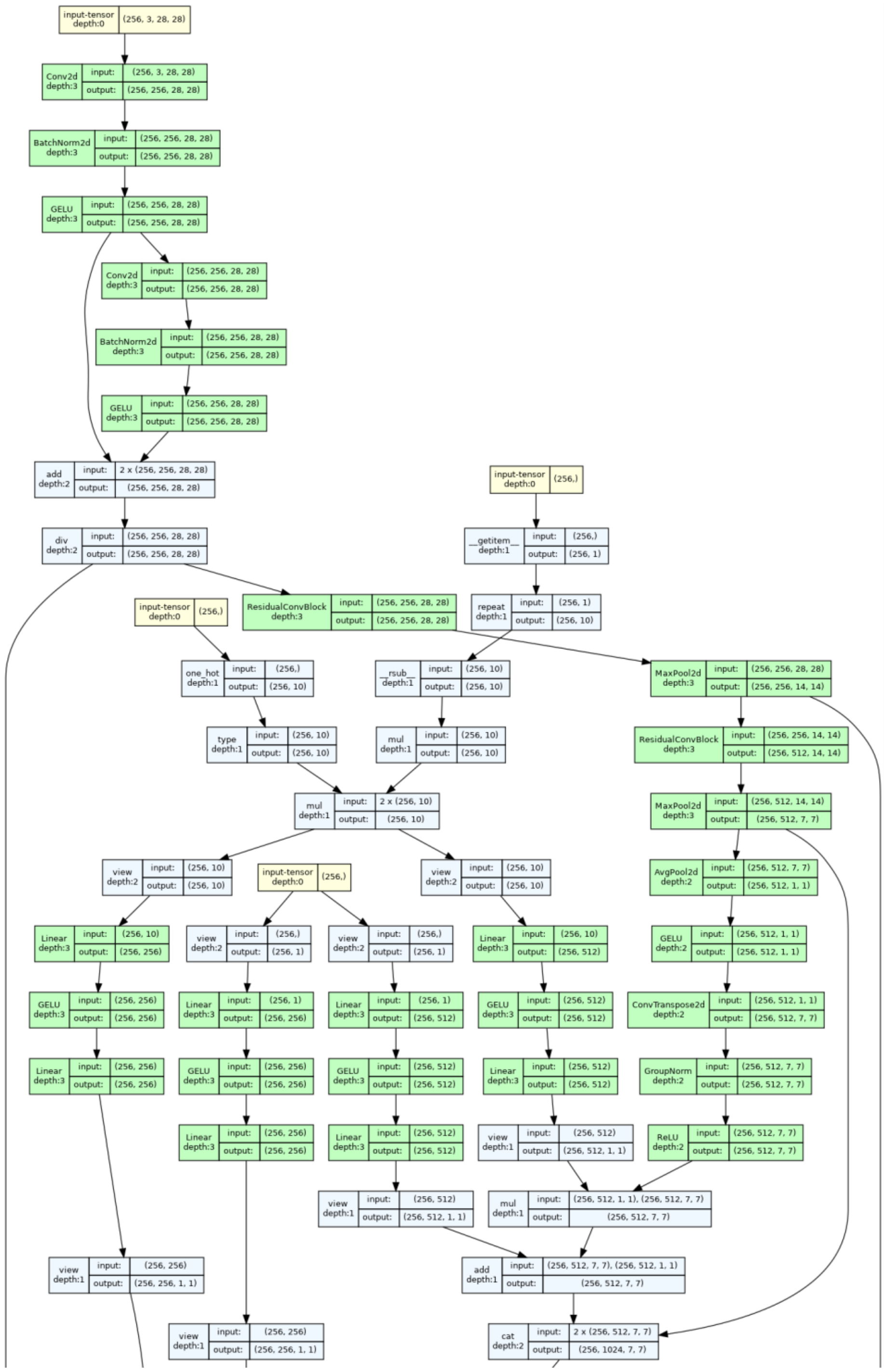
Deep Learning for Computer Vision

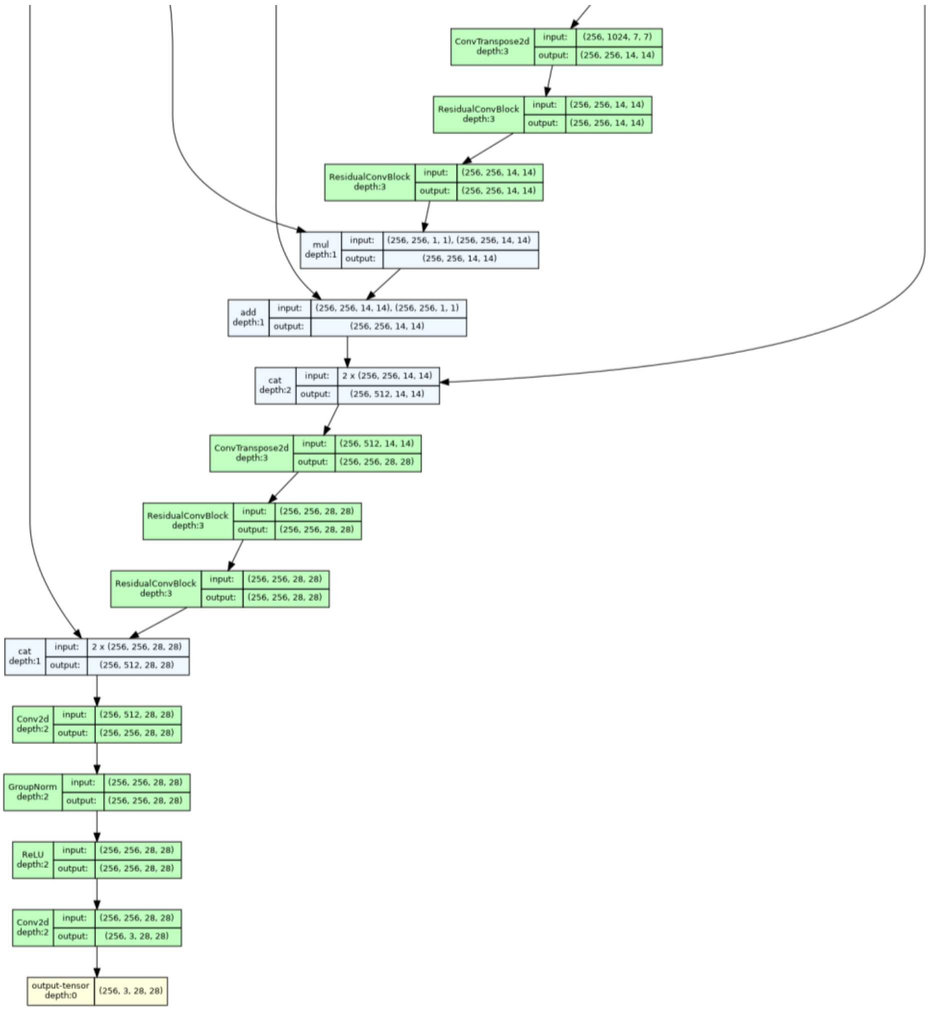
NTU, Fall 2023, homework2

電機所碩一 謝宗翰 r12921a10

- **Problem 1: Diffusion models**

1. *Follow the Github Example to draw your model architecture and describe your implementation details.*





Implement Detail - Parameters				
Epoch	Batch Size	Time Step	Feat	Loss
100	256	400	256	MSE Loss

DDPM Schedules

```python
def ddpm_schedules(beta1, beta2, T):
    """
    Returns pre-computed schedules for DDPM sampling, training process.
    """
    assert beta1 < beta2 < 1.0, "beta1 and beta2 must be in (0, 1)"

    beta_t = (beta2 - beta1) * torch.arange(0, T + 1, dtype=torch.float32) / T + beta1
    sqrt_beta_t = torch.sqrt(beta_t)
    alpha_t = 1 - beta_t
    log_alpha_t = torch.log(alpha_t)
    alphabar_t = torch.cumsum(log_alpha_t, dim=0).exp()

    sqrtab = torch.sqrt(alphabar_t)
    oneover_sqrta = 1 / torch.sqrt(alpha_t)

    sqrtmab = torch.sqrt(1 - alphabar_t)
    mab_over_sqrtmab_inv = (1 - alpha_t) / sqrtmab

    return {
        "alpha_t": alpha_t,  # \alpha_t
        "oneover_sqrta": oneover_sqrta,  # 1/\sqrt{\alpha_t}
        "sqrt_beta_t": sqrt_beta_t,  # \sqrt{\beta_t}
        "alphabar_t": alphabar_t,  # \bar{\alpha_t}
        "sqrtab": sqrtab,  # \sqrt{\bar{\alpha_t}}
        "sqrtmab": sqrtmab,  # \sqrt{1-\bar{\alpha_t}}
        "mab_over_sqrtmab": mab_over_sqrtmab_inv,  # (1-\alpha_t)/\sqrt{1-\bar{\alpha_t}}
    }
```

Implement Detail
DDPM – Training Part

```python
class DDPM(nn.Module):
    def __init__(self, nn_model, betas, n_T, device, drop_prob=0.1):
        super(DDPM, self).__init__()
        self.nn_model = nn_model.to(device)

        # register_buffer allows accessing dictionary produced by ddpm_schedules
        # e.g. can access self.sqrtab later
        for k, v in ddpm_schedules(betas[0], betas[1], n_T).items():
            self.register_buffer(k, v)

        self.n_T = n_T
        self.device = device
        self.drop_prob = drop_prob
        self.loss_mse = nn.MSELoss()

    def forward(self, x, c):
        """
        this method is used in training, so samples t and noise randomly
        """
        # t ~ Uniform(0, n_T)
        _ts = torch.randint(1, self.n_T + 1, (x.shape[0],)).to(self.device)
        # eps ~ N(0, 1)
        noise = torch.randn_like(x)
        # This is the x_t, which is sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
        # We should predict the "error term" from this x_t. Loss is what we return.
        x_t = (
            self.sqrtab[_ts, None, None, None] * x
            + self.sqrtmab[_ts, None, None, None] * noise
        )

        # dropout context with some probability
        context_mask = torch.bernoulli(torch.zeros_like(c) + self.drop_prob).to(
            self.device
        )

        # return MSE between added noise, and our predicted noise
        Unet_predict = self.nn_model(x_t, c, _ts / self.n_T, context_mask)

        return self.loss_mse(noise, Unet_predict)
```

DDPM 方法實現

Training Unet

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \mathrm{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

按照 Pseudo code 的方法實現 DDPM 的過程。實際在進行 Training 的其實是 ContextUnet Model，把加了 Noise 的圖片、Label 跟 TimeStep 丟進 ContextUnet 中進行訓練。

因為 Problem1 要我們做的是 Conditional 的 Diffusion model，為了實現 Conditional 的效果，因此在 Training 的過程中特別加入了 Label。

DDPM – Sampling Part

```python
def sample(self, n_sample, size, device, guide_w=0.0):
        # x_T ~ N(0, 1), sample initial noise
        x_i = torch.randn(n_sample, *size).to(device)
        # context for us just cycles throught the mnist labels
        c_i = torch.arange(0, 10).to(device)
        c_i = c_i.repeat(int(n_sample / c_i.shape[0]))

        # don't drop context at test time
        context_mask = torch.zeros_like(c_i).to(device)

        # double the batch
        c_i = c_i.repeat(2)
        context_mask = context_mask.repeat(2)
        context_mask[n_sample:] = 1.0  # makes second half of batch context free
        print()
        for i in range(self.n_T, 0, -1):
            print(f"sampling timestep {i}", end="\r")
            t_is = torch.tensor([i / self.n_T]).to(device)
            t_is = t_is.repeat(n_sample, 1, 1, 1)
            # double batch
            x_i = x_i.repeat(2, 1, 1, 1)
            t_is = t_is.repeat(2, 1, 1, 1)

            z = torch.randn(n_sample, *size).to(device) if i > 1 else 0

            # split predictions and compute weighting
            eps = self.nn_model(x_i, c_i, t_is, context_mask)
            eps1 = eps[:n_sample]  # 有 condition
            eps2 = eps[n_sample:]  # 無 condition
            eps = (1 + guide_w) * eps1 - guide_w * eps2
            x_i = x_i[:n_sample]
            x_i = (
                self.oneover_sqrta[i] * (x_i - eps * self.mab_over_sqrtmab[i])
                + self.sqrt_beta_t[i] * z
            )

        # retrun noise+pic
        return x_i
```

Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

依照 ddpm 論文中的 pseudo code 去實作 sampling 過程。在每次的跌代中都去掉一點 noise。

2. *Please show 10 generated images for each digit (0-9) in your report. You can put all 100 outputs in one image with columns indicating different noise inputs and rows indicating different digits.*

3. *Visualize total six images in the reverse process of the first "0" in your grid in (2) with different time steps.*

4. *Please discuss what you've observed and learned from implementing conditional diffusion model.*

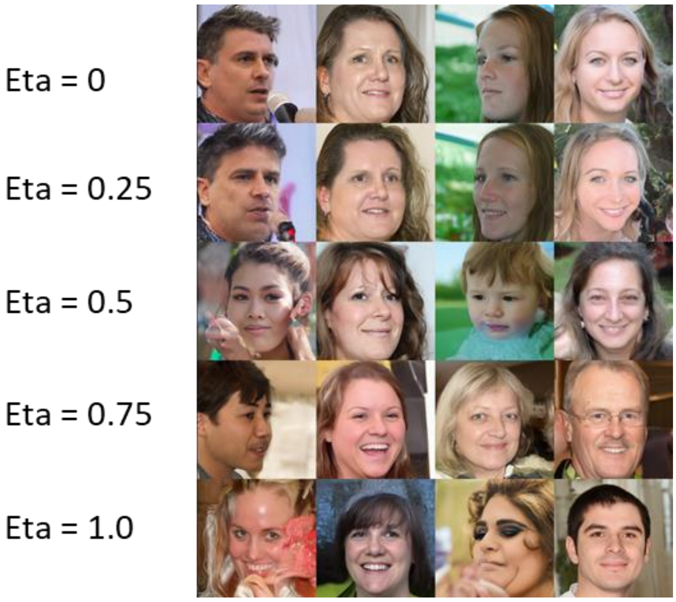
這是我第一次接觸到 Diffusion Model，Model Sampling 的過程很像在雕刻一個大理石，讓我想到海綿寶寶雕刻大理石的那集，其實圖片都已經藏在 Noise 內，將 Noise 丟進 Model 後讓他把原本就在 Noise 內的圖片雕刻出來。

在訓練 Diffusion model 時，我發現相較於我之前碰過的 GAN，Diffusion model 更穩定。但我發現他有個比 GAN 差的地方，就是因為

他迭代的特性，Diffusion Model 的速度比 GAN 慢很多，但是速度與穩定性的權衡是值得的，用速度換取穩定度。在實作 Conditional Diffusion model 後，我熟悉了有關 Training Diffusion model 和如何加入 Conditional 的條件方式，這是個寶貴的經驗，我以後可以根據我想要的 Prompt 來做些酷酷的模型了。

- **Problem 2:** *DDIM*

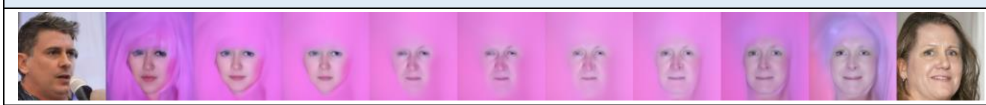
1. *Please generate face images of noise 00.pt ~ 03.pt with different eta in one grid. Report and explain your observation in this experiment.*



如果 eta 越大，加入的隨機 Noise 就會對原本的 Noise 的影響越大。如果 eta 為 0，所有人用同一個 Unet weight 對同一個 Noise Sampling 出來的圖片都會一樣。如果加了 eta，圖片的隨機性就會越大，如圖中 eta 越大就越不像原本的 eta = 0 的圖片。

2. *Please generate the face images of the interpolation of noise 00.pt ~ 01.pt. The interpolation formula is **spherical linear interpolation**, which is also known as **slerp**.*

Slerp

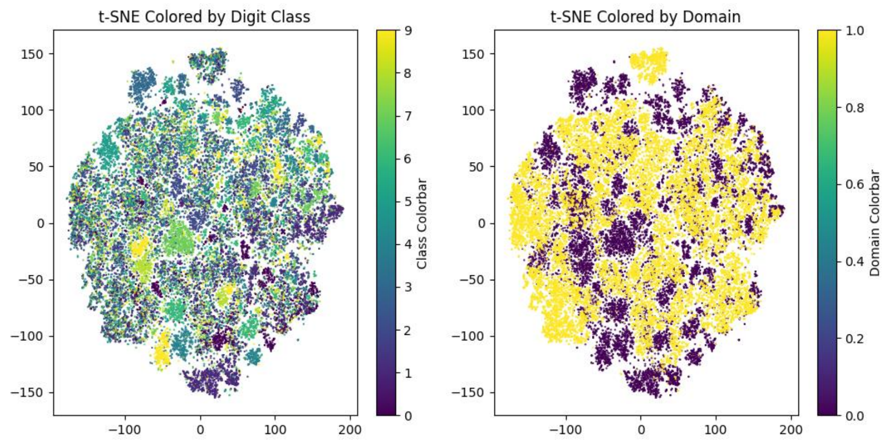

Linear


- **Problem 3: DANN**

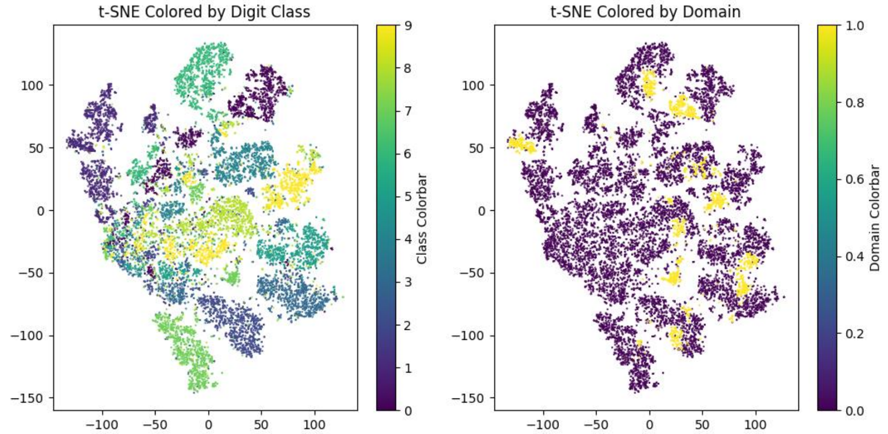
1. *Please create and fill the table with the following format in your report:*

	MNIST-M → SVHN	MNIST-M → USPS
Trained on source	31.25%	68.27%
Adaptation (DANN)	44.313%	84.95%
Trained on target	91.8%	97.2%

2. *Please visualize the latent space (output of CNN layers) of DANN by mapping the validation images to 2D space with t-SNE. For each scenario, you need to plot two figures which are colored by digit class (0-9) and by domain, respectively.*

MNIST → SVHN

從 Class 的角度可以看到，MNIST 有很多明顯結塊的地方，代表分 Class 分的還行，但在 SVHN 就沒這麼明顯的結塊。但從 Domain 來看的話，MNIST 與 SVHN 交疊的地方變多的，應該有達到混淆資料集的效果。

MNIST → USPS



Class 的分群效果還不錯，MNIST 跟 USPS 都有很多明顯結塊的地方，在 Domain 也可以看出 USPS 的分群效果很好，每塊間的間隙都蠻大的。

3. *Please describe the implementation details of your model and discuss what you've observed and learned from implementing DANN.*

從圖表中可以觀察到：MNIST → SVHN 跟 MNIST → USPS 的 features 融合在一起的情況還不錯。總體來看，我們可以清楚地看到，與 MNIST-M 和 USPS 之間的域間隙相比，MNIST-M 和 SVHN 之間的域間隙要大得多。

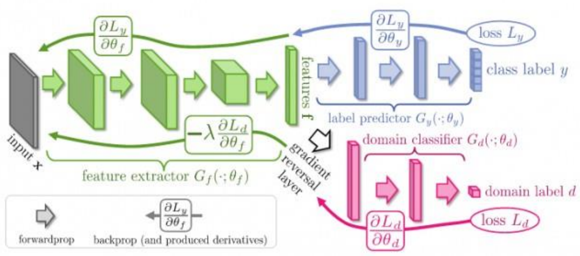
DANN Model Implementation

```python
class DANN(nn.Module):
    def __init__(self, feature_extractor, domain_regressor, classifier):
        super(DANN, self).__init__()

        self.feature_extractor = feature_extractor
        self.domain_regressor = domain_regressor
        self.classifier = classifier

    def forward(self, x):
        x = self.feature_extractor(x)
        y = GradReverse.grad_reverse(x)
        return self.classifier(x), self.domain_regressor(y)

```



依照 DANN 的架構實現 Feature Extractor、Domain Regressor、Classifier

1. 用 Feature Extractor 提取特徵
2. 將提取的特徵做 Gradient Reverse
3. 將 Reverse 的結果丟進 Classifier 與 Domain Regressor

Training Implementation

```python
for epoch in range(NUM_EPOCH):
    print(f"Training epoch {epoch}...")
    for src_data, target_data in tqdm(
        zip(src_train_loader, target_train_loader),
        desc=f"Epoch {epoch}",
        total=min(len(src_train_loader), len(target_train_loader)),
    ):
        # Update progress
        p += 1 / total_steps

        # Compute the regularization term
        gamma = 10
        lambda_p = 2 / (1 + np.exp(-gamma * p)) - 1

        # Split and transfer to GPU
        src_imgs, src_labels = src_data[0].to(device), src_data[1].to(device)
        target_imgs, target_labels = target_data[0].to(device), target_data[1].to(
            device
        )

        # Source forward pass
        src_class, src_domain = dann(src_imgs)    拿到 source domain 的 loss

        # Classifier loss
        class_loss = criterion_classifier(src_class, src_labels)
        # Target forward pass
        _, target_domain = dann(target_imgs)    拿到 target domain 的 loss

        # Domain Loss
        preds_domain = torch.cat((src_domain, target_domain))    計算兩個 domain loss
        domain_loss = criterion_domain_regressor(preds_domain, labels_domain)

        # Total loss
        loss = class_loss.cpu() + lambda_p * domain_loss.cpu()

        # Backward and Optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # # Scheduler step
```

Implement Detail - Parameters		
Epoch	Batch Size	Loss
100	256	CrossEntropyLoss
Schedules		

```python
# SGD optimizer
optimizer = optim.SGD(
    [
        {"params": dann.feature_extractor.parameters()},
        {"params": dann.classifier.parameters()},
        {"params": dann.domain_regressor.parameters()},
    ],
    lr=0.01,
    momentum=0.9,
)
```

```python
# Learning rate scheduler
def mu_p(step):
    alpha = 10
    beta = 0.75
    mu_p = 1 / (1 + alpha * step / total_steps) ** beta
    return mu_p

# Virtual learning rate for the domain regressor
def domain_regressor_lr_scheduler(step):
    gamma = 10

    # If step=0, just returns mu_p to avoid division by zero
    if step == 0:
        lambda_p = 1
    else:
        p = step / total_steps
        lambda_p = 2 / (1 + np.exp(-gamma * p)) - 1

    return mu_p(step) / lambda_p


# Learning rate scheduler
scheduler = torch.optim.lr_scheduler.LambdaLR(
    optimizer, [mu_p, mu_p, domain_regressor_lr_scheduler]
)
```

- **Reference**

DDPM:

https://github.com/TeaPearce/Conditional_Diffusion_MNIST

DDIM:

https://zhuanlan.zhihu.com/p/565698027

DANN:

https://zhuanlan.zhihu.com/p/565698027

https://github.com/vcoyette/DANN