

HW4.

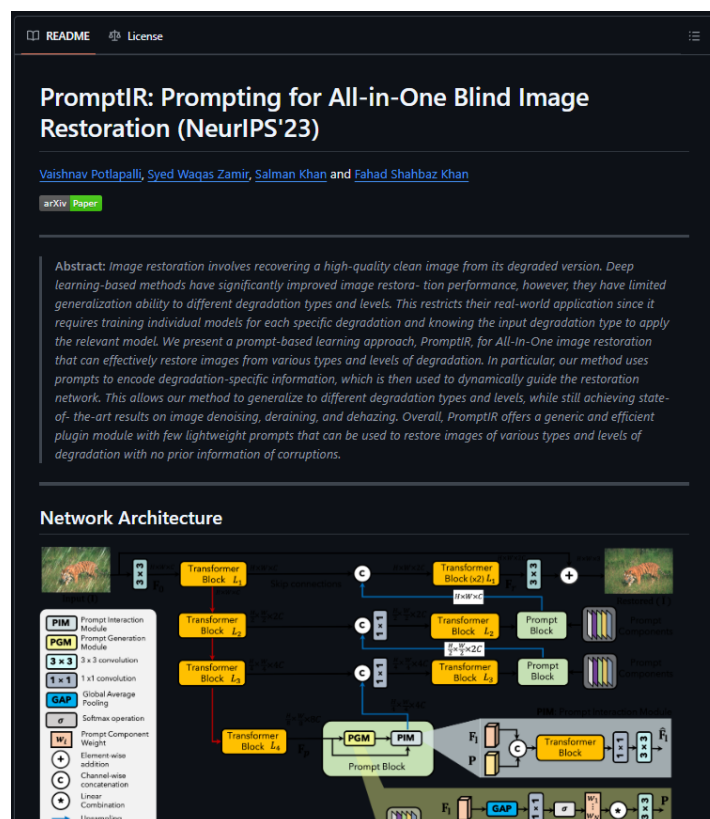
110550128 蔡耀霆

Github: https://github.com/WhiteOuO/VRDL_HW4

Introduction

The task of this project is image restoration. The training and testing data have been specially designed, with noise uniformly distributed across the entire image. Each image contains only one type of noise, and there are only two types of noise: rain and snow. The deep learning model used in this project is based on the concept of PromptIR. The core objective of the model is to handle multiple types of noise within a unified model architecture. It introduces the concept of "prompts," where each prompt corresponds to a specific noise removal task. During the learning process, the model becomes increasingly accurate in interpreting these prompts, enabling better restoration of images affected by specific types of noise.

The model architecture in this project is built upon the framework provided in this GitHub repository: <https://github.com/va1shn9v/PromptIR>



All programming and improvements will be based on the designs and concepts implemented in this article, adapted to suit the image restoration and noise removal tasks addressed in this project. The following sections will describe how the program was modified.

Method.(modification on framework)

Model modification.

```
class PromptGenBlock(nn.Module):
    def __init__(self, prompt_dim=128, prompt_len=5, prompt_size=96, lin_dim=192):
        super(PromptGenBlock, self).__init__()
        # Learnable parameter for rain prompt, initialized with random values
        self.prompt_rain = nn.Parameter(torch.rand(1, prompt_len, prompt_dim, prompt_size, prompt_size))
        # Learnable parameter for snow prompt, initialized with random values
        self.prompt_snow = nn.Parameter(torch.rand(1, prompt_len, prompt_dim, prompt_size, prompt_size))
        # Linear layer to compute prompt weights based on input embedding
        self.linear_layer = nn.Linear(lin_dim, prompt_len * 2)
        self.conv3x3 = nn.Conv2d(prompt_dim, prompt_dim, kernel_size=3, stride=1, padding=1, bias=False)
```

The task includes two types: rain removal and snow removal. A set of vectors is randomly initialized for each of these tasks. During training, these vectors are continuously updated to more accurately describe the two types of noise. The model uses a linear layer to project the embedding vector, which represents the spatial features of the input image, into a weight vector. This weight vector is then used to compute the final prompt through a weighted calculation.

```
def forward(self, x, de_id=None):
    # Extract batch size, channels, height, and width from input tensor
    B, C, H, W = x.shape
    # Compute spatial mean embedding from input for weight calculation
    emb = x.mean(dim=(-2, -1))

    # Compute softmax-normalized weights for rain and snow prompts
    prompt_weights = F.softmax(self.linear_layer(emb), dim=1)
    rain_weights = prompt_weights[:, :self.prompt_rain.size(1)]
    snow_weights = prompt_weights[:, self.prompt_rain.size(1):]

    # Weighted sum of rain prompts based on rain weights
    prompt_rain = (rain_weights.unsqueeze(-1).unsqueeze(-1).unsqueeze(-1) *
                  self.prompt_rain.unsqueeze(0).repeat(8, 1, 1, 1, 1, 1)).squeeze(1)
    # Weighted sum of snow prompts based on snow weights
    prompt_snow = (snow_weights.unsqueeze(-1).unsqueeze(-1).unsqueeze(-1) *
                  self.prompt_snow.unsqueeze(0).repeat(8, 1, 1, 1, 1, 1)).squeeze(1)

    # Training phase: Use de_id to select prompt
    if de_id is not None:
        prompt = torch.zeros_like(prompt_rain, device=x.device)
        task = torch.empty(8, dtype=torch.long, device=x.device)
        for b in range(8):
            if de_id[b] == 3:
                prompt[b] = torch.sum(prompt_rain[b], dim=0)
                task[b] = 0 # 0 indicates "derain"
            elif de_id[b] == 4:
                prompt[b] = torch.sum(prompt_snow[b], dim=0)
                task[b] = 1 # 1 indicates "desnow"
        # Map task indices to string labels
        task_str = ["derain" if t == 0 else "desnow" for t in task.tolist()]
    else: # Testing phase: Determine task based on weights
        rain_score = torch.sum(rain_weights, dim=1, keepdim=True)
        snow_score = torch.sum(snow_weights, dim=1, keepdim=True)
        # Select task based on higher score (0 for derain, 1 for desnow)
        task = torch.where(rain_score > snow_score, 0, 1)
        task = task.squeeze(-1)
        # Select prompt based on score comparison
        prompt = torch.where((rain_score > snow_score).unsqueeze(-1).unsqueeze(-1).unsqueeze(-1),
                           prompt_rain, prompt_snow)
        prompt = torch.sum(prompt, dim=1)
        # Map task indices to string labels
        task_str = ["derain" if t == 0 else "desnow" for t in task.tolist()]

    # Interpolate prompt to match input height and width
    prompt = F.interpolate(prompt, (H, W), mode="bilinear")
    # Apply 3x3 convolution to refine the prompt
    prompt = self.conv3x3(prompt)
    return prompt, task_str
```

Some designs were added here. During the training phase, if a `de_id` is provided, the program will enter the if branch above. If `de_id=3`, it indicates a rain removal task. The portion of the final vector representing rain is extracted from the vector computed by the model. Subsequently, the prompt representing rain is provided to the model, and the information about the rain removal task is also passed to the subsequent model. This ensures that when calculating the loss, only the corresponding prompt is updated. During the testing phase, the program will enter the else branch below. At this stage, the prompts for rain and snow have been fully trained and stabilized. Therefore, based on the weights computed earlier, the model determines whether the input image contains rain or snow noise, thereby deciding which prompt to use for processing the current image.

Train_data_loader modification.

```
class PromptTrainDataset(Dataset):
    def __init__(self, args, transform=None):
        super(PromptTrainDataset, self).__init__()
        self.args = args
        self.rs_ids = []
        self.snow_ids = []
        self.D = Degradation(args)
        self.de_temp = 0
        self.de_type = self.args.de_type
        self.transform = transform # for data augmentations
        print(self.de_type)

        self.de_dict = {'derain': 3, 'desnow': 4} # we use 3 and 4 for derain and desnow

        self._init_ids()
        self._merge_ids()

        self.crop_transform = Compose([
            ToPILImage(),
            RandomCrop(args.patch_size),
        ])

        self.toTensor = ToTensor()

    def _init_ids(self):
        if 'derain' in self.de_type:
            self._init_rs_ids()
        if 'desnow' in self.de_type:
            self._init_snow_ids()

        random.shuffle(self.de_type)
```

```
def _get_gt_name(self, rainy_name):
    return rainy_name.replace('rain-', 'rain_clean-')

def _get_snow_clean_name(self, snow_name):
    return snow_name.replace('snow-', 'snow_clean-')
```

Experiments 1.

Increase number of transformers on different layers.

```

class PromptIR(nn.Module):
    def __init__(self,
        inp_channels=3,
        out_channels=3,
        dim=48,
        num_blocks=[6,8,8,10],
        num_refinement_blocks=6,
        heads=[1,2,4,8],
        ffn_expansion_factor=2.66,
        bias=False,
        LayerNorm_type='WithBias',
        decoder=False,
    ):
        super(PromptIR, self).__init__()

```

Modification: num_blocks =[6,8,8,10], num_refinement_blocks =6, which used to be [4,6,6,8], 4

The number of layers in the Encoder/Decoder Transformer is described as follows: the architecture includes three Encoder layers and three Decoder layers, with a Latent layer positioned between the three Encoder and three Decoder layers. Additionally, a Refinement layer is incorporated, receiving the output of the topmost Decoder layer to perform corrections, such as removing artifacts, reducing excessive smoothing, and ensuring consistency with the original image structure. The Patch Embed is initially fed into the Encoder layers for feature extraction, where each layer extracts features at different levels of detail—higher layers capture finer details, while lower layers represent more abstract features. The Latent layer extracts highly abstract features, such as the distribution or density of noise. Finally, the Decoder performs image restoration, integrating prompts previously generated by three PromptGenBlocks.

Expectations:

It is expected that the model can more accurately extract the characteristics of rain and snow noise, while improving its image restoration capability.

Results:

Based:

```

C: > Users > Yao Ting Tsai > Downloads > scoring_result (1) > scores.json > ...
1 [{"private_psnr": 27.99015360578561, "public_psnr": 28.558020685950788}]

```

More transformer blocks:

```
C: > Users > Yao Ting Tsai > AppData > Local > Temp > e64927bd-2a66-4f55-ada5-7c10090ee0d3_scoring
1 [{"private_psnr": 29.494266517471655, "public_psnr": 30.302327018811276}]
```

Gigantic progression on PSNR.

Experiments 2

Perceptual Loss

```
class VGGPerceptualLoss(nn.Module):
    def __init__(self, layer_indices=[3, 8, 15, 22]):
        super(VGGPerceptualLoss, self).__init__()
        vgg = vgg16(weights=VGG16_Weights.DEFAULT).features
        self.vgg = nn.Sequential(*list(vgg.children())[:23]).eval()
        self.layer_indices = layer_indices
        for param in self.vgg.parameters():
            param.requires_grad = False

    def forward(self, pred, target):
        pred = (pred - 0.5) * 2
        target = (target - 0.5) * 2
        pred_features = self.vgg(pred)
        target_features = self.vgg(target)
        loss = F.mse_loss(pred_features, target_features)
        return loss
```

Compared to the original L1 Loss, which simply calculates the average pixel-level difference, Perceptual Loss evaluates the output of feature layers, focusing on deeper features such as edges, textures, structures, and more. The goal is to make the generated images appear more natural visually.

```
class PromptIRModel(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.net = PromptIR(decoder=True)
        self.l1_loss = nn.L1Loss()
        self.perceptual_loss = VGGPerceptualLoss()
        self.lambda_l1 = 0.5
        self.lambda_perceptual = 0.5
        self.psnr_values = []

    def forward(self, x, de_id=None):
        return self.net(x)

    def calculate_psnr(self, pred, target, data_range=1.0):
        mse = F.mse_loss(pred, target, reduction='mean')
        if mse == 0:
            return float('inf')
        max_value = torch.tensor(data_range, device=pred.device)
        psnr = 20 * torch.log10(max_value) - 10 * torch.log10(mse)
        return psnr

    def training_step(self, batch, batch_idx):
        ([clean_name, de_id], degrad_patch, clean_patch) = batch
        restored = self.net(degrad_patch)

        # 计算损失
        l1_loss = self.l1_loss(restored, clean_patch)
        perceptual_loss = self.perceptual_loss(restored, clean_patch)
        total_loss = self.lambda_l1 * l1_loss + self.lambda_perceptual * perceptual_loss

        psnr = self.calculate_psnr(restored, clean_patch, data_range=1.0)
        self.psnr_values.append(psnr.item())
```

Here, a hybrid loss function is used, with the weights equally split (50% each), and the model is updated using this hybrid Loss.

Results:



Version 27 only added Transformer layers and used solely L1 Loss.

Version 16 incorporated additional Transformer Blocks and introduced Perceptual Loss, combined with L1 Loss in a 1:1 ratio to compute the Combine

The performance was notably poor. The speculated reason may be that mixing the two Loss functions caused the direction of model updates to become ambiguous or inconsistent—one focuses on pixel-level differences, while the other emphasizes overall structural outlines. Additionally, the evaluation metric used in this study is PSNR, which is also calculated based on pixel-level differences, making it more compatible with L1 Loss.

Experiments 3

Data augmentation:

```
def main():
    train_transforms = A.Compose([
        A.HorizontalFlip(p=0.5),
        A.VerticalFlip(p=0.5),

        A.ColorJitter(
            brightness=0.2,
            contrast=0.2,
            saturation=0.2,
            hue=0.1,
            p=0.5
        ),

        A.RandomGamma(gamma_limit=(80, 120), p=0.5),
    ], additional_targets={'clean_patch': 'image'})
    torch.set_float32_matmul_precision('medium')
    print("Options")
    print(opt)

    trainset = PromptTrainDataset(opt, transform=train_transforms)
    checkpoint_callback = ModelCheckpoint(dirpath=opt.ckpt_dir, every_n_epochs=1, save_top_k=1)
    trainloader = DataLoader(
        trainset,
        batch_size=opt.batch_size,
        pin_memory=True,
        shuffle=True,
        drop_last=True,
        num_workers=opt.num_workers,
        persistent_workers=True
    )
```

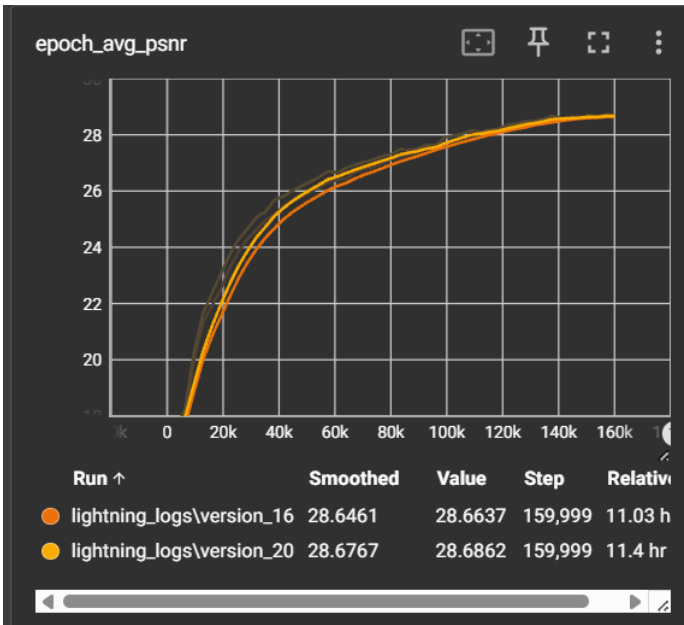
The data augmentation includes two types of horizontal and vertical flipping, as

well as primarily color-based fine-tuning. Augmentations such as Gaussian noise or random cropping are not introduced, to prevent the model from mistakenly identifying missing regions or other noise in the images as part of rain or snow noise.

Expectations:

By moderately incorporating some data augmentation, the Loss during training is expected to be more challenging to reduce, as the noise characteristics become more varied. However, during the testing phase, the model should better adapt to samples under different conditions.

Results.









Version 16 incorporated additional Transformer Blocks and introduced Perceptual Loss, combined with L1 Loss in a 1:1 ratio to compute the Combine Loss.

Version 20 incorporated additional Transformer Blocks and introduced Perceptual Loss, combined with L1 Loss in a 1:1 ratio to compute the Combine Loss, and included a version with Data Augmentation.

During the training phase, the PSNR appeared to improve slightly; however, significant issues arose during the Test phase. The model automatically detects and determines whether the current image belongs to Snow or Rain, but it classified all images as Snow.

It was observed that the Snow Score for each Test sample was consistently close to 98-99. The reason for the significant impact of Data Augmentation on this aspect remains unclear. A preliminary hypothesis suggests that after incorporating Data Augmentation, the features of Rain were largely unable to be learned effectively. Upon examining the model’s output images, it was evident that traces of Rain noise remained quite noticeable.

296675	pred_dataaug.zip	2025-05-23 07:45	Finished	18.21	  
296265	pred_test.zip	2025-05-22 16:57	Finished	29.41	  

The upper submission represents the version with data augmentation, while the lower submission serves as the control group.

Results:

More transformer blocks, L1 loss only, no data augmentation.

25	110550128	1	2025-05-24 15:20	297404	110550128	30.3
----	-----------	---	------------------	--------	-----------	------

The model's classification performance during the Test phase is suboptimal. The scores when determining whether the noise is Rain or Snow are extremely close, typically falling within the 49%-51% range. However, the noise appears to have been effectively removed.





References: <https://github.com/va1shn9v/PromptIR>