AIL 862

Lecture 26

Conditional generation

- Adaptive layer normalization

Conditional generation

- Cross-attention

Conditional generation

- Token concatentation

Classifier guidance

Algorithm 1 Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale s.

Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\,\nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
end for
return x_0

Diffusion Models Beat GANs on Image Synthesis, 2021

```python
import tqdm

def run_inference(unet, classifier, class_name, class_list, gradient_scale, num_row=10, num_col=10):
    unet.eval()
    classifier.eval()

    #####
    # preparation
    #####

    # generate array of sigma_t
    alpha_bars_prev = torch.cat((torch.ones(1).to(device), alpha_bars[:-1]))
    sigma_t_squared = (1.0 - alphas) * (1.0 - alpha_bars_prev) / (1.0 - alpha_bars)
    sigma_t = torch.sqrt(sigma_t_squared)

    # generate y (tensor batch array of class id)
    class_id_list = [i for i,v in enumerate(class_list) if v==class_name]
    if len(class_id_list) == 0:
        raise Exception("class name doesn't exist")
    y = class_id_list[0]
    y_batch = (torch.tensor(y).to(device)).repeat(num_row*num_col)

    #####
    # 1. make white noise
    #####
    x = torch.randn(num_row*num_col, 3, 32, 32).to(device)

    #####
    # 2. Loop
```

```python
#####
# 1. make white noise
#####
x = torch.randn(num_row*num_col, 3, 32, 32).to(device)

#####
# 2. loop
#####
for t in tqdm.tqdm(reversed(range(T)), total=T):
    # get mu
    t_batch = (torch.tensor(t).to(device)).repeat(num_row*num_col)
    with torch.no_grad():
        epsilon = unet(x, t_batch)
    mu = (1.0 / torch.sqrt(alphas[t])).float() * (x - ((1.0 - alphas[t]) / torch.sqrt(1.0 - alpha_bars[t])).float() * epsilon

    # get nabla_x(log(p(y|x))) at x_t
    x_in = x.detach().requires_grad_(True)
    logits = classifier(x_in, t_batch)
    log_probs = F.log_softmax(logits, dim=-1)
    selected = log_probs[range(len(logits)), y_batch.view(-1)]
    grad = torch.autograd.grad(selected.sum(), x_in)[0]

    # pick up x_{t-1}
    if t > 0:
        z = torch.randn_like(x).to(device)
    else:
        z = torch.zeros_like(x).to(device)
    x = mu + gradient_scale * sigma_t_squared[t].float() * grad + \
        sigma_t[t].float() * z

#####
# 3. get x_0
#####

# reshape to channels-last : (N,C,H,W) --> (N,H,W,C)
x = x.permute(0, 2, 3, 1)
# clip
x = torch.clamp(x, min=0.0, max=1.0)
```

Figure 3: Samples from an unconditional diffusion model with classifier guidance to condition on the class "Pembroke Welsh corgi". Using classifier scale 1.0 (left; FID: 33.0) does not produce convincing samples in this class, whereas classifier scale 10.0 (right; FID: 12.0) produces much more class-consistent images.

CLIP guidance

CLIP guidance

- Works similar to classifier guidance
- Perturb the reverse process mean with the gradient of the dot product of the image and text/caption encodings w.r.t. the image

GLIDE: Towards Photorealistic Image Generation and Editing with Text-Guided Diffusion Models, 2022

Classifier-free guidance

- A conditional denoising model and an unconditional denoising model

Classifier-Free Diffusion Guidance, 2022

Classifier-free guidance

- A conditional denoising model and an unconditional denoising model
- Single neural network is used to parametrize both models

Classifier-free guidance

- A conditional denoising model and an unconditional denoising model
- Single neural network is used to parametrize both models

- Perform sampling using a linear combination of the conditional and unconditional outputs

$$\tilde{\boldsymbol{\epsilon}}_\theta(\mathbf{z}_\lambda, \mathbf{c}) = (1 + w)\boldsymbol{\epsilon}_\theta(\mathbf{z}_\lambda, \mathbf{c}) - w\boldsymbol{\epsilon}_\theta(\mathbf{z}_\lambda)$$

Latent Diffusion

- Instead of pixel space, operate in latent space

Latent Diffusion

- Instead of pixel space, operate in latent space

- Pixel space to latent space conversion

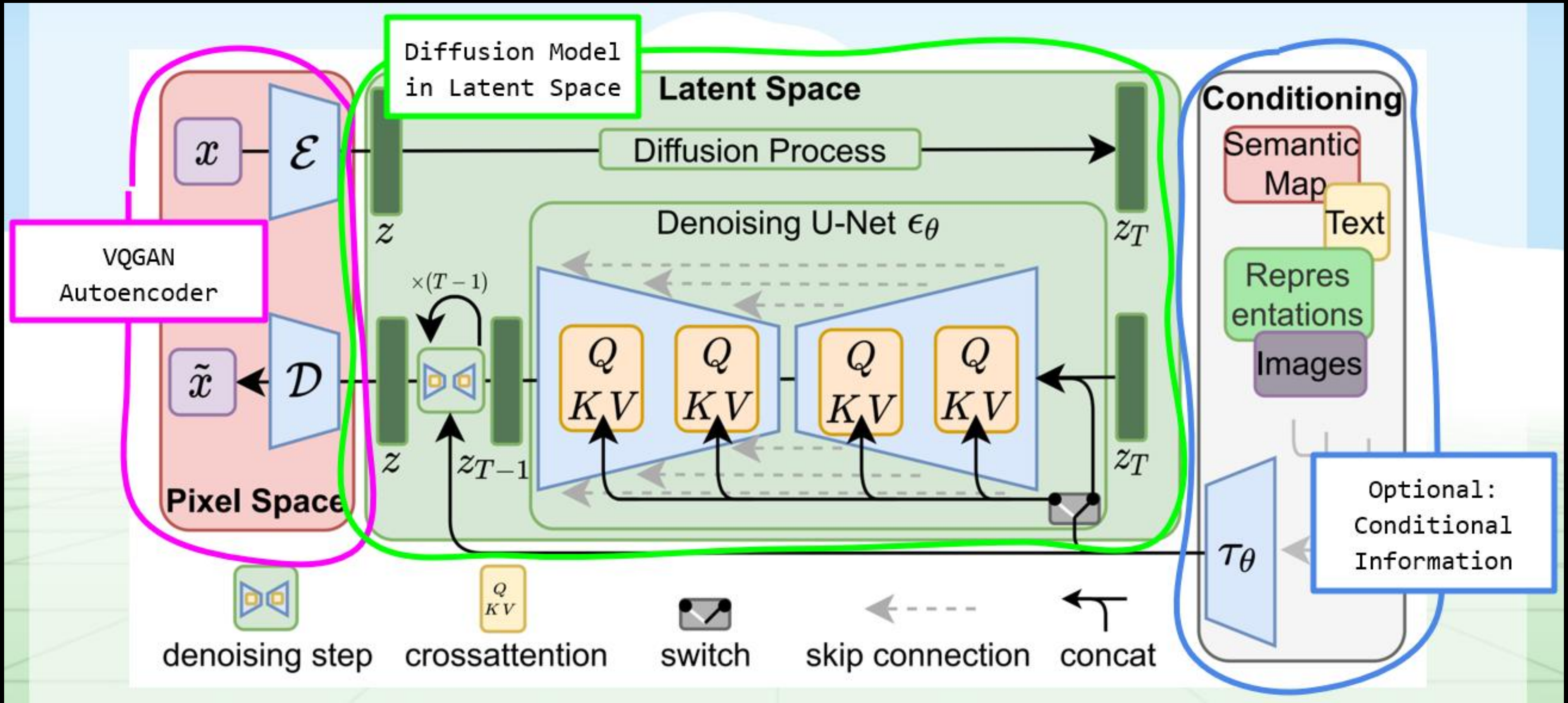
Latent Diffusion

- Instead of pixel space, operate in latent space
- Pixel space to latent space conversion
- U-Net denoising

Latent Diffusion

- Instead of pixel space, operate in latent space
- Pixel space to latent space conversion
- U-Net denoising
- Conditional information – cross-attention

Latent diffusion



Full fine tuning

- Need to store and deploy separate model weights for each task

Full fine tuning

- Foundations models – huge number of parameters

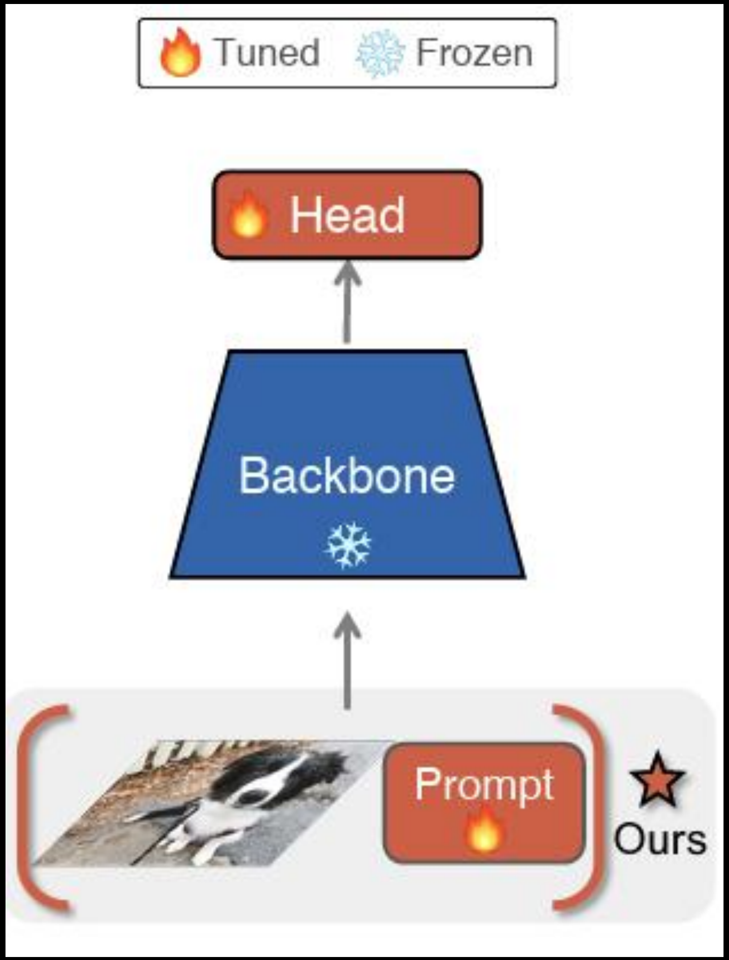
- Can overfit on / memorize few thousand examples

- Also chance of catastrophic forgetting

Fine tuning with linear layer

- Underperform full fine tuning in performance

Visual-prompt tuning



VPT key idea

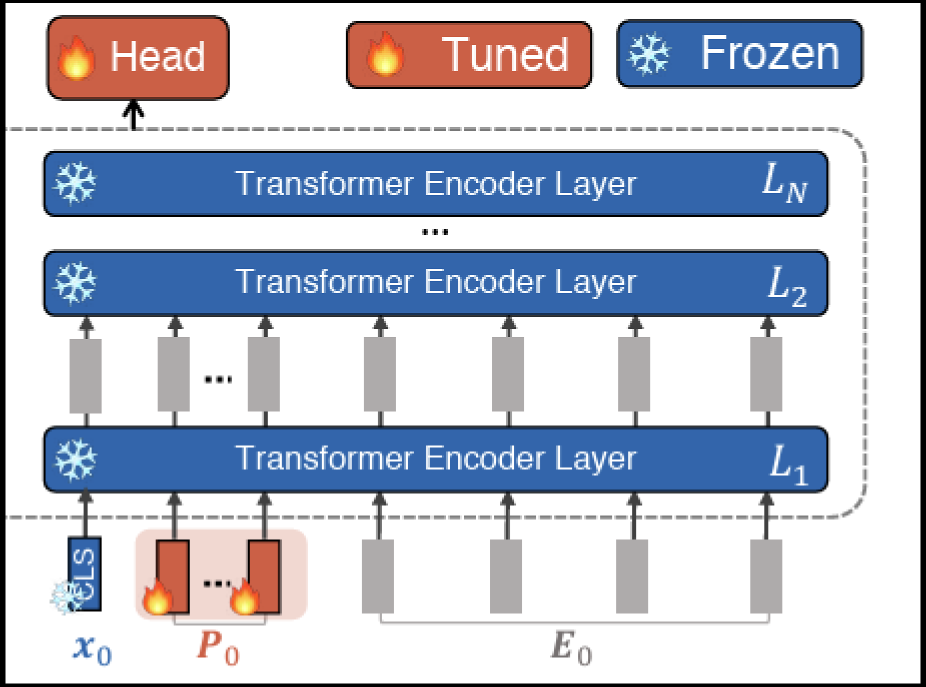
Instead of altering or fine-tuning the pre-trained Transformer itself, modify the input to the Transformer

Advantages

Only introduces a small amount of task-specific learnable parameters into the input space while freezing the entire pre-trained Transformer backbone during downstream training.

In practice, these additional parameters are simply prepended into the input sequence of each Transformer layer and learned together with a linear head during fine-tuning.

VPT-shallow


Head
Tuned
Frozen
Transformer Encoder Layer L_N
...
Transformer Encoder Layer L_2
Transformer Encoder Layer L_1
CLS
x_0
P_0
E_0

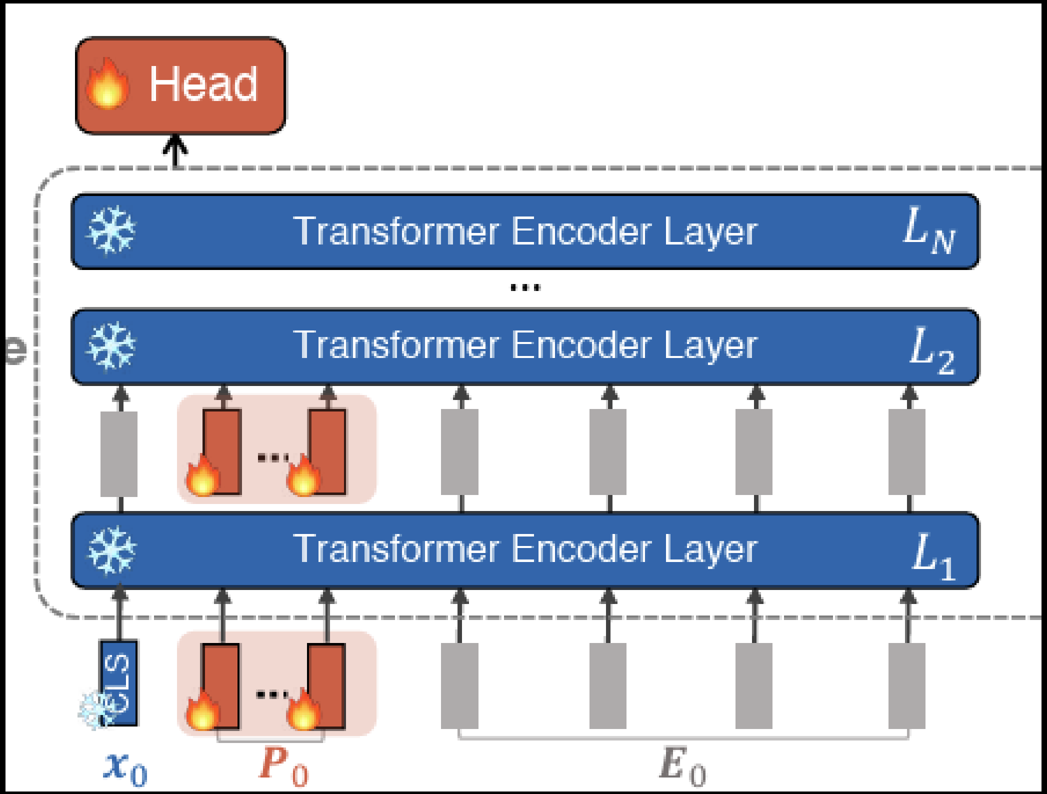
```python
# Config
NUM_CLASSES = 2          # positive / negative
PROMPT_LEN = 5           # number of prompt tokens
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
BATCH_SIZE = 64
NUM_WORKERS = 16          # adjust to your machine
```

```python
# Load pretrained ViT and freeze it
vit = create_model('vit_base_patch16_224', pretrained=True)
vit.head = nn.Identity()  # remove original head
for p in vit.parameters():
    p.requires_grad = False

embedDim = vit.embed_dim
```

```python
class VPTWrapper(nn.Module):
    def __init__(self, vit_model, prompt_len, num_classes):
        super().__init__()
        self.vit = vit_model
        self.prompt_len = prompt_len
        self.prompt = nn.Parameter(torch.randn(1, prompt_len, embedDim))
        # self._initial_prompt_saved = False
        self.classifier = nn.Linear(embedDim, num_classes)

    def forward(self, x):
        B = x.size(0)
        x = self.vit.patch_embed(x)             # (B, N, D)
        cls_token = self.vit.cls_token.expand(B, -1, -1)
        pos_embed  = self.vit.pos_embed[:, 1:1 + x.size(1), :]
        x = x + pos_embed

        # # debug: track prompt shift
        # if not self._initial_prompt_saved:
        #     self.initial_prompt = self.prompt[0, 0].detach().clone()
        #     self._initial_prompt_saved = True
        #     print("Initial prompt saved.")
        # curr = self.prompt[0, 0].detach()
        # print(f"Prompt Δ: {(curr - self.initial_prompt).norm().item():.6f}")

        prompt_tokens = self.prompt.expand(B, -1, -1)
        x = torch.cat([cls_token, prompt_tokens, x], dim=1)
        x = self.vit.pos_drop(x)
        for blk in self.vit.blocks:
            x = blk(x)
        x = self.vit.norm(x)
        return self.classifier(x[:, 0])
```

```python
model = VPTWrapper(vit, PROMPT_LEN, NUM_CLASSES).to(DEVICE)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()
```

VPT-deep


Head
Transformer Encoder Layer L_N
Transformer Encoder Layer L_2
Transformer Encoder Layer L_1
CLS
x_0
P_0
E_0

Storage cost

VPT is beneficial in presence of multiple downstream tasks. We only need to store the learned prompts and classification head for each task and re-use the original copy of the pre-trained Transformer model, significantly reducing the storage cost.

Key observations

- Significant performance gap between VPT and finetuning using linear layer

Sounds good.

Key observations

- Significant performance gap between VPT and finetuning using MLP with 3 layers (instead of just a linear layer)

Sounds better?

Key observations

- VPT-deep outperforms full finetuning on most datasets/tasks

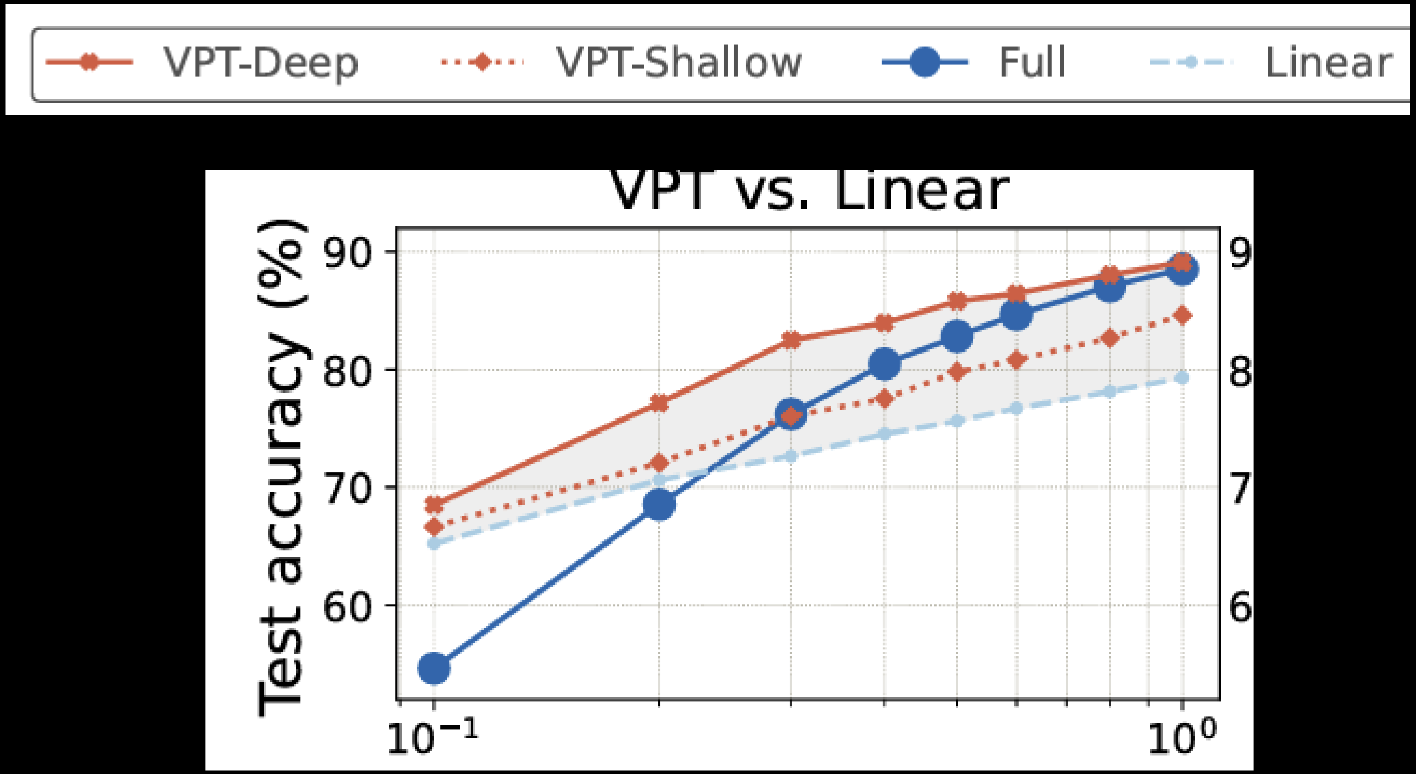
Exceeds expectation?

Key observations

- VPT-Deep outperforms all the other parameter-efficient tuning protocols (as of when the VPT paper was written)

- Although sub-optimal than VPT-deep, VPT-shallow still offers significant performance gain

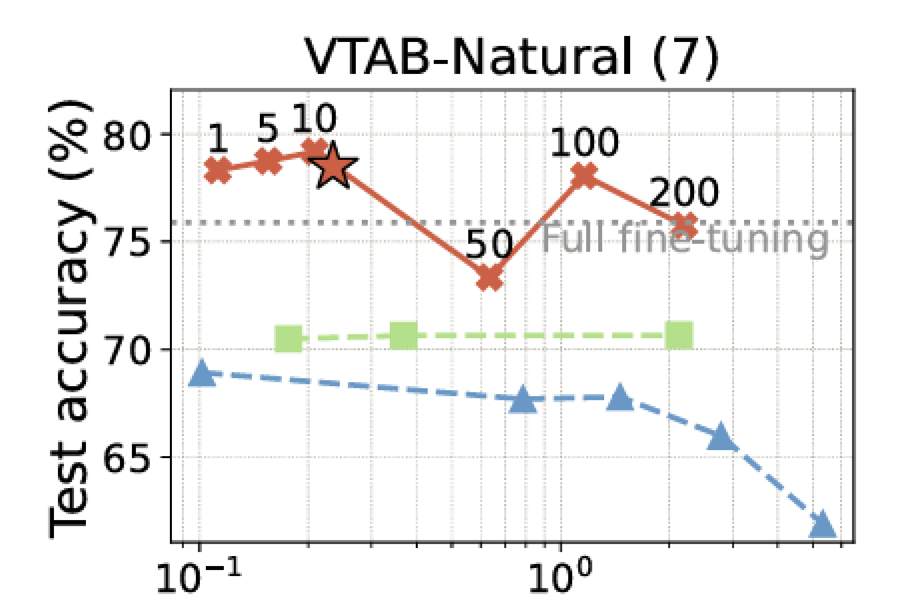
Fraction of downstream training dataset



Consistency across different model scales

- VPT performance is consistently better than other tuning strategies for ViT-Base/Large/Huge

Prompt length



VPT for adaptation between domain A and B

- Idea with SSL (based on pre-text task)

Four season analysis