Pixio

Example implementation of the Pixio method. Pixio builds on the Masked Autoencoder (MAE) and adapts it for dense prediction through three changes: a much deeper decoder (32 blocks) that takes over pixel-level detail modeling, a larger masking granularity that masks whole blocks of patches on a regular grid instead of individual patches, and multiple class tokens whose mean is used as the global image representation.

Key Components

- **Data Augmentations**: Like MAE, Pixio relies only on random resized cropping.
- **Masking**: Pixio masks 75% of the patches, but at a coarser granularity: whole `grid_size` x `grid_size` blocks of patches are masked together (4x4 by default), which prevents trivial reconstruction from neighboring patches.
- **Backbone**: A standard ViT with multiple class tokens (8 by default, realized via `reg_tokens`).
- **Decoder**: A deep (32-block) decoder that reconstructs the masked pixels.
- **Reconstruction Loss**: A Mean Squared Error (MSE) loss between the predicted and the normalized pixel values of the masked patches.

Good to Know

- **Masking granularity**: The paper's headline configuration uses a 4x4 grid and 8 class tokens. The dense-prediction-optimal ablation uses a 2x2 grid and 4 class tokens.
- **Input resolution**: The reference model is trained at 256x256 with patch size 16 so that the 16x16 patch grid divides evenly into 4x4 blocks.

Reference:

In Pursuit of Pixel Supervision for Visual Pre-training, 2025

Note

Pixio requires TIMM to be installed

```
pip install "lightly[timm]"
```

PyTorch | Lightning | Lightning Distributed

This example can be run from the command line with:

```
python lightly/examples/pytorch/pixio.py
```

```python
# This example requires the following dependencies to be installed:
# pip install "lightly[timm]"

# Note: The model and training settings do not follow the reference settings
# from the paper. The settings are chosen such that the example can easily be
# run on a small dataset with a single GPU.
import torch
import torchvision
from timm.models.vision_transformer import vit_small_patch16_224
from torch import nn

from lightly.models import utils
from lightly.models.modules import MaskedVisionTransformerTIMM, PixioDecoderTIMM
from lightly.transforms import MAETransform


class Pixio(nn.Module):
    def __init__(self):
        super().__init__()

        decoder_dim = 512
        vit = vit_small_patch16_224(img_size=256, reg_tokens=7)
        self.mask_ratio = 0.75
        self.grid_size = 4
        self.patch_size = vit.patch_embed.patch_size[0]
        self.num_prefix_tokens = vit.num_prefix_tokens

        self.backbone = MaskedVisionTransformerTIMM(vit=vit)
        self.sequence_length = self.backbone.sequence_length
        self.decoder = PixioDecoderTIMM(
            num_patches=vit.patch_embed.num_patches,
            patch_size=self.patch_size,
            embed_dim=vit.embed_dim,
            decoder_embed_dim=decoder_dim,
            decoder_depth=32,
            decoder_num_heads=16,
            num_prefix_tokens=self.num_prefix_tokens,
            mlp_ratio=4.0,
            proj_drop_rate=0.0,
            attn_drop_rate=0.0,
        )

    def forward_encoder(self, images, idx_keep=None):
        return self.backbone.encode(images=images, idx_keep=idx_keep)

    def forward_decoder(self, x_encoded, idx_keep, idx_mask):
        # build decoder input
        batch_size = x_encoded.shape[0]
        x_decode = self.decoder.embed(x_encoded)
        x_masked = utils.repeat_token(
            self.decoder.mask_token, (batch_size, self.sequence_length)
        )
        x_masked = utils.set_at_index(x_masked, idx_keep, x_decode.type_as(x_masked))

        # decoder forward pass
        x_decoded = self.decoder.decode(x_masked)

        # predict pixel values for masked tokens
        x_pred = utils.get_at_index(x_decoded, idx_mask)
```

```
        x_pred = self.decoder.predict(x_pred)
        return x_pred

    def forward(self, images):
        batch_size = images.shape[0]
        idx_keep, idx_mask = utils.random_grid_token_mask(
            size=(batch_size, self.sequence_length),
            mask_ratio=self.mask_ratio,
            grid_size=self.grid_size,
            num_prefix_tokens=self.num_prefix_tokens,
            device=images.device,
        )
        x_encoded = self.forward_encoder(images=images, idx_keep=idx_keep)
        x_pred = self.forward_decoder(
            x_encoded=x_encoded, idx_keep=idx_keep, idx_mask=idx_mask
        )

        # get image patches for masked tokens
        patches = utils.patchify(images, self.patch_size)
        # must adjust idx_mask for the prefix tokens
        target = utils.get_at_index(patches, idx_mask - self.num_prefix_tokens)
        target = utils.normalize_mean_var(target)
        return x_pred, target


model = Pixio()

device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)

transform = MAETransform(input_size=256)
# we ignore object detection annotations by setting target_transform to return 0


def target_transform(t):
    return 0


dataset = torchvision.datasets.VOCDetection(
    "datasets/pascal_voc",
    download=True,
    transform=transform,
    target_transform=target_transform,
)
# or create a dataset from a folder containing images or videos:
# dataset = LightlyDataset("path/to/folder")

dataloader = torch.utils.data.DataLoader(
    dataset,
    batch_size=64,
    shuffle=True,
    drop_last=True,
    num_workers=8,
)

criterion = nn.MSELoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=1.5e-4)

print("Starting Training")
for epoch in range(10):
```

```
    total_loss = 0
    for batch in dataloader:
        views = batch[0]
        images = views[0].to(device)  # views contains only a single view
        predictions, targets = model(images)
        loss = criterion(predictions, targets)
        total_loss += loss.detach()
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    avg_loss = total_loss / len(dataloader)
    print(f"epoch: {epoch:>02}, loss: {avg_loss:.5f}")
```