In [ ]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
 %reload_ext autoreload
```

Imports

In [ ]:
```
import pathlib
import random
import time

import matplotlib.pyplot as plt
import tqdm
import torch
from torch import nn, optim
from torch.nn.utils import clip_grad_norm_
from torch.nn.functional import interpolate
from torchvision.utils import save_image
from torch.utils.data import DataLoader
from torchvision.transforms.functional import adjust_brightness

from super_resolution.src.sen2venus_dataset import (
    create_train_test_split,
)
from super_resolution.src.visualization import plot_gallery
from super_resolution.src.srgan import SRResNet, Discriminator
```

In [ ]:
```
DATA_DIR = pathlib.Path("C:/Users/Mitch/stat3007_data")
SITES_DIR = DATA_DIR / "sites"
PREPROCESSING_DIR = DATA_DIR / "preprocessing"
RESULTS_DIR = DATA_DIR / "results"
```

Prepare Data

In [ ]:
```
sites = {
    "SO2",
    "FR-BIL",
    "NARYN",
}
train_patches, test_patches = create_train_test_split(
    str(SITES_DIR) + "\\", sites=sites
)
print(f"Num train {len(train_patches)}\n" f"Num test {len(test_patches)}")
```

```
Num train 8159
Num test 3498
```

In [ ]:
```
def image_transform(x, y):
    x = x[:3, :, :]
    y = y[:3, :, :]

    x = torch.clamp(x, 0, 1)
    y = torch.clamp(y, 0, 1)

    return x, y
```

In [ ]:
```
train_patches.set_transform(image_transform)
test_patches.set_transform(image_transform)
```

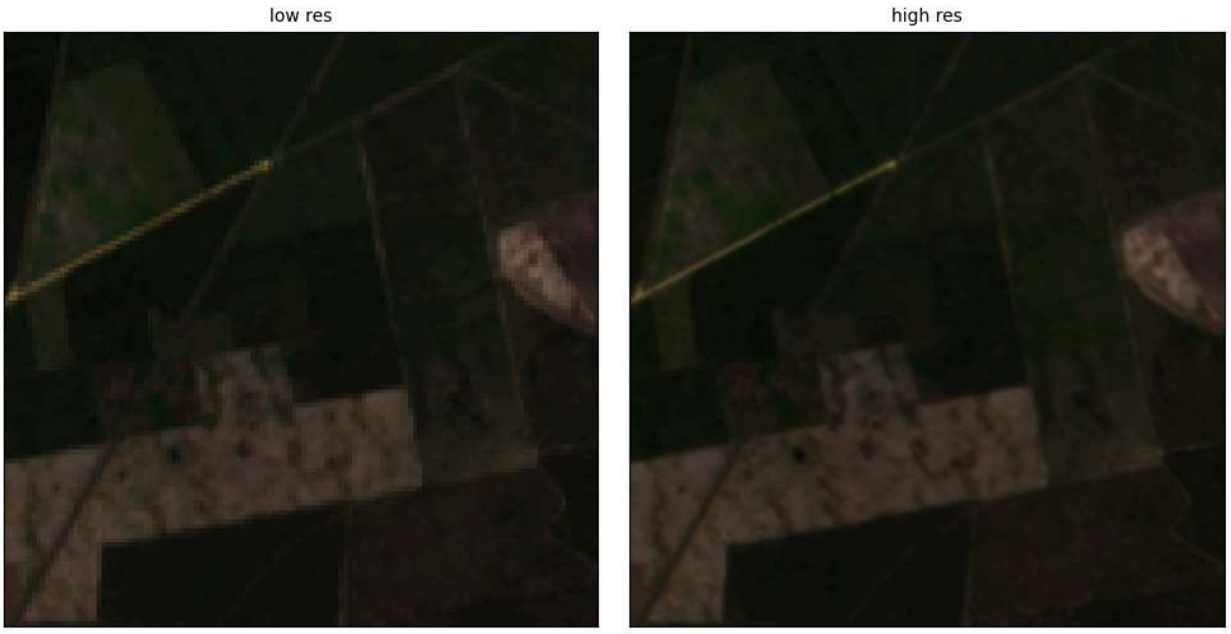
In [ ]:
```
train_loader = DataLoader(train_patches, batch_size=1)
```

In [ ]:
```
(low_res, high_res) = next(train_loader.__iter__())
```

In [ ]:
```
index = random.randint(0, len(low_res) - 1)
low_res_example = low_res[index]


high_res_example = high_res[index]


plot_gallery(
    [
        adjust_brightness(low_res_example, 2).permute(1, 2, 0),
        adjust_brightness(high_res_example, 2).permute(1, 2, 0),
    ],
    titles=["low res", "high res"],
    xscale=5,
    yscale=5,
)
```



In [ ]:
```
del train_loader
```

Training

In [ ]:
```
# For clearing GPU memory
# 1 / 0
```

In [ ]:
```
# For clearing GPU memory
import gc

gc.collect()
torch.cuda.empty_cache()
```

In [ ]:
```
BATCH_SIZE = 16
GEN_LEARNING_RATE = 1e-5
DISCRIM_LEARNING_RATE = 1e-5
GEN_WEIGHT_DECAY = 1e-8
DISCRIM_WEIGHT_DECAY = 1e-8
DISCRIM_WEIGHT = 1e-6
```

In [ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.cuda.empty_cache()
should_pin_memory = {"cuda": True, "cpu": False}

train_loader = DataLoader(
    train_patches,
    shuffle=True,
    batch_size=BATCH_SIZE,
    drop_last=True,
    pin_memory=should_pin_memory[device.type],
    num_workers=0,
)
```

In [ ]:
```
device
```

Out[ ]:
```
device(type='cuda')
```

In [ ]:
```
generator = SRResNet(scaling_factor=2, n_blocks=8)
```

In [ ]:
```
discriminator = Discriminator(n_blocks=2, fc_size=128)
```

In [ ]:
```
generator = generator.to(device)
generator = generator.train()
discriminator = discriminator.to(device)
discriminator = discriminator.train()
```

In [ ]:
```
gen_optimizer = optim.Adam(
    generator.parameters(), lr=GEN_LEARNING_RATE, weight_decay=GEN_WEIGHT_DECAY
)

discrim_optimizer = optim.Adam(
    discriminator.parameters(),
    lr=DISCRIM_LEARNING_RATE,
```

```
    weight_decay=DISCRIM_WEIGHT_DECAY,
)
```

In [ ]:
```
# save_file = RESULTS_DIR / "continuingbestgen_374epoch.pkl"
# loaded_experiment = torch.load(save_file, map_location=device)
# # generator = SRResNet(scaling_factor=2, n_blocks=16).to(device)
# # discriminator = Discriminator(n_blocks=3, fc_size=128)
# generator.load_state_dict(loaded_experiment["gen_state"])
# discriminator.load_state_dict(loaded_experiment["discrim_state"])
# gen_optimizer.load_state_dict(loaded_experiment["gen_optimizer_state"])
# discrim_optimizer.load_state_dict(loaded_experiment["discrim_optimizer_state"])
# gen_losses = loaded_experiment["gen_losses"]
# discrim_losses = loaded_experiment["discrim_losses"]
# train_time = loaded_experiment["train_time"]
```

In [ ]:
```
pixel_criterion = nn.MSELoss()
discrim_criterion = nn.BCEWithLogitsLoss()
```

In [ ]:
```
NUM_EPOCHS = 2000
```

In [ ]:
```
train_time = 0.0
gen_losses = []
discrim_losses = []
```

In [ ]:
```
generator = generator.to(device)
generator = generator.train()
discriminator = discriminator.to(device)
discriminator = discriminator.train()

for epoch in range(NUM_EPOCHS):
    progress_bar = tqdm.tqdm(train_loader, total=len(train_loader), ncols=100)
    gen_epoch_loss = 0.0
    discrim_epoch_loss = 0.0
    num_batches = 0
    for i, (low_res_batch, high_res_batch) in enumerate(progress_bar):
        num_batches += 1
        start_time = time.time()

        # Push to GPU
        low_res_batch = low_res_batch.to(device)
        high_res_batch = high_res_batch.to(device)

        # Update generator
        ##########################################################################################
        super_resolved = generator(low_res_batch)
        natural_probs = discriminator(super_resolved)

        pixel_loss = pixel_criterion(super_resolved, high_res_batch)
        adversarial_loss = discrim_criterion(
            natural_probs, torch.ones_like(natural_probs)
        )
        gen_loss = pixel_loss

        gen_loss += DISCRIM_WEIGHT * adversarial_loss

        gen_optimizer.zero_grad()
        gen_loss.backward()
        gen_optimizer.step()

        # Update discriminator
        ##########################################################################################
        true_natural_probs = discriminator(high_res_batch)

        # Detach to skip generator computations
        fake_natural_probs = discriminator(super_resolved.detach())

        true_natural_loss = discrim_criterion(
            true_natural_probs, torch.ones_like(true_natural_probs)
        )
        fake_natural_loss = discrim_criterion(
            fake_natural_probs, torch.zeros_like(fake_natural_probs)
        )
        discrim_loss = true_natural_loss + fake_natural_loss

        discrim_optimizer.zero_grad()
        discrim_loss.backward()
        discrim_optimizer.step()

        # Collect data
        gen_epoch_loss += gen_loss.item()
        discrim_epoch_loss += discrim_loss.item()
        progress_bar.set_postfix(
            epoch=epoch,
            gen_loss=f"{gen_epoch_loss/num_batches:.8f}",
```

```
            discrim_loss=f"{discrim_epoch_loss/num_batches:.8f}",
        )

        end_time = time.time()
        train_time += end_time - start_time

    gen_epoch_loss /= len(train_loader)
    discrim_epoch_loss /= len(train_loader)
    gen_losses.append(gen_epoch_loss)
    discrim_losses.append(discrim_epoch_loss)
    print(
        f"Epoch: {epoch} / gen_loss: {gen_epoch_loss:.8f} / discrim_loss: {discrim_epoch_loss:.8f}"
    )
```

In [ ]:
```
train_time / 60
```

Out[ ]:
```
1053.3067023237547
```

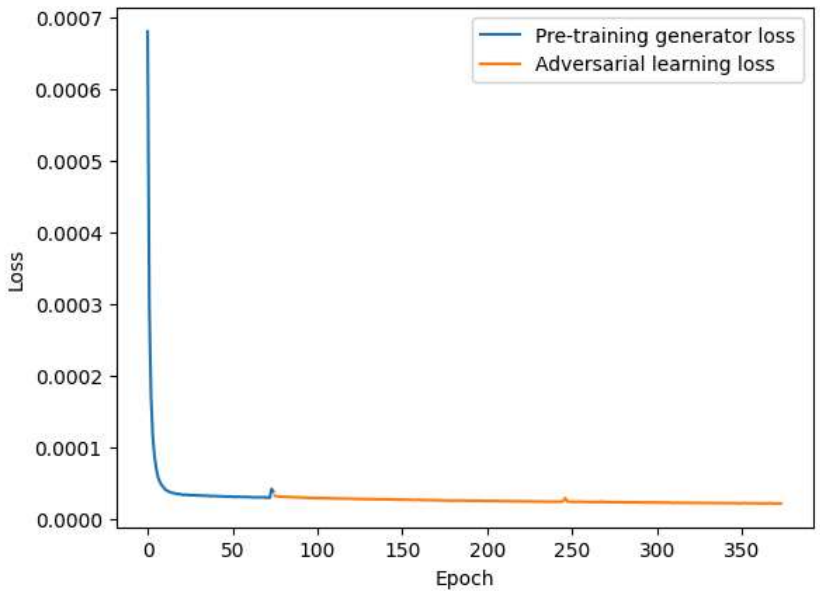
In [ ]:
```
len(gen_losses)
```

Out[ ]:
```
374
```

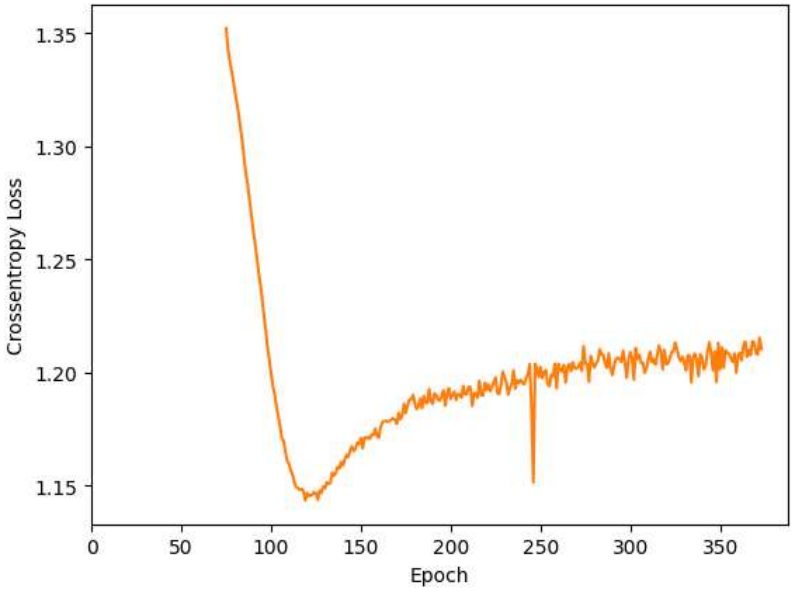
In [ ]:
```
gen_losses[-5:]
```

Out[ ]:
```
[2.1545394862930416e-05,
 2.1526079207431426e-05,
 2.174403593542232e-05,
 2.1463911347496594e-05,
 2.1551264829290158e-05]
```

In [ ]:
```
plt.plot(gen_losses[:75], label="Pre-training generator loss")
plt.plot(range(75, len(gen_losses)), gen_losses[75:], label="Adversarial learning loss")
# plt.title("Generator losses")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



In [ ]:
```
plt.plot([])
plt.plot(range(75, len(discrim_losses)), discrim_losses[75:])
plt.xlim(left=0)
# plt.title("Discriminator losses")
plt.xlabel("Epoch")
plt.ylabel("Crossentropy Loss")
plt.show()
```



Saving

In [ ]:
```
experiment = {
    "gen_losses": gen_losses,
    "discrim_losses": discrim_losses,
    "gen_state": generator.state_dict(),
    "discrim_state": discriminator.state_dict(),
    "gen_optimizer_state": gen_optimizer.state_dict(),
    "discrim_optimizer_state": discrim_optimizer.state_dict(),
    "gen_learning_rate": GEN_LEARNING_RATE,
    "discrim_learning_rate": DISCRIM_LEARNING_RATE,
    "gen_weight_decay": GEN_WEIGHT_DECAY,
    "discrim_weight_decay": DISCRIM_WEIGHT_DECAY,
    "discrim_weight": DISCRIM_WEIGHT,
    "train_time": train_time,
}
```

In [ ]:
```
save_file = RESULTS_DIR / "continuingbestgen_378epoch.pkl"
```

In [ ]:
```
if not save_file.exists():
    torch.save(experiment, save_file)
    print(f"Saved to {save_file}")
```

```
Saved to C:\Users\Mitch\stat3007_data\results\continuingbestgen_378epoch.pkl
```

Generated Examples

In [ ]:
```
BRIGHT_FACTOR = 3.5
```

In [ ]:
```
generator = generator.to("cpu")
```

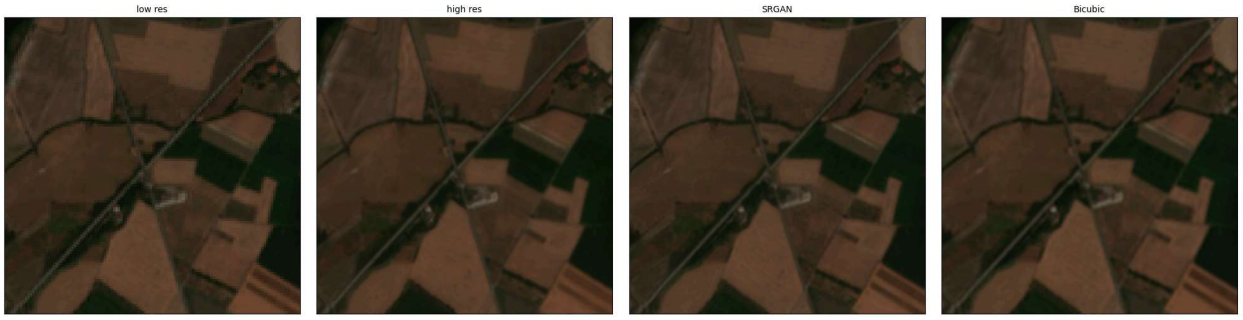
In [ ]:
```
loader = DataLoader(train_patches, batch_size=100)
(low_res, high_res) = next(loader.__iter__())
```

In [ ]:
```
import random

index = random.randint(0, len(low_res) - 1)

low_res_example = low_res[index]
high_res_example = high_res[index]
out = generator(low_res_example.unsqueeze(0)).detach().clamp(0, 1)
bicubic_out = interpolate(
    low_res_example.unsqueeze(0),
    size=(256, 256),
    mode="bicubic",
).float()
plot_gallery(
    [
        adjust_brightness(low_res_example, BRIGHT_FACTOR).permute(1, 2, 0),
        adjust_brightness(high_res_example, BRIGHT_FACTOR).permute(1, 2, 0),
        adjust_brightness(out[0], BRIGHT_FACTOR).permute(1, 2, 0),
        adjust_brightness(bicubic_out[0], BRIGHT_FACTOR).permute(1, 2, 0),
    ],
    titles=["low res", "high res", "SRGAN", "Bicubic"],
```

```
    xscale=5,
    yscale=5,
)
```



In [ ]:
```
loader = DataLoader(test_patches, batch_size=300)
(low_res, high_res) = next(loader.__iter__())
```

In [ ]:
```
import random

index = random.randint(0, len(low_res) - 1)

low_res_example = low_res[index]
high_res_example = high_res[index]

out = generator(low_res_example.unsqueeze(0)).detach().clamp(0, 1)[0]
out = adjust_brightness(out, BRIGHT_FACTOR)

bicubic_out = (
    interpolate(low_res_example.unsqueeze(0), size=(256, 256), mode="bicubic")
    .float()
    .clamp(0, 1)
)[0]
bicubic_out = adjust_brightness(bicubic_out, BRIGHT_FACTOR)

low_res_example = adjust_brightness(low_res_example, BRIGHT_FACTOR)
high_res_example = adjust_brightness(high_res_example, BRIGHT_FACTOR)

plot_gallery(
    [
        low_res_example.permute(1, 2, 0),
        high_res_example.permute(1, 2, 0),
        out.permute(1, 2, 0),
        bicubic_out.permute(1, 2, 0),
    ],
    titles=["low res", "high res", "SRGAN", "Bicubic"],
    xscale=5,
    yscale=5,
)
```

Save images

In [ ]:
```
IMAGE_DIR = RESULTS_DIR / "images/final3_5bright"
```

In [ ]:
```
index
```

In [ ]:
```
srgan_file = IMAGE_DIR / f"{index}srgan.png"
bicubic_file = IMAGE_DIR / f"{index}bicubic.png"
low_res_file = IMAGE_DIR / f"{index}lowres.png"
high_res_file = IMAGE_DIR / f"{index}highres.png"
```

In [ ]:
```
save_image(low_res_example, low_res_file)
save_image(high_res_example, high_res_file)
save_image(out, srgan_file)
save_image(bicubic_out, bicubic_file)
```

Metrics

In [ ]:
```
del loader
```

In [ ]:
```
from super_resolution.src.testing import compute_metrics
```

In [ ]:
```
# 256 is largest I can handle on cpu
metric_loader = DataLoader(test_patches, batch_size=min(len(test_patches), 256))
```

In [ ]:
```
metrics = compute_metrics(lambda x: generator(x).clamp(0, 1), metric_loader)
```

```
100%|██████████| 14/14 [14:58<00:00, 64.16s/it]
```

In [ ]:
```
metrics
```

Out[ ]:
```
Metrics(mse=2.522575544259 91e-05, psnr=45.9937013898577, ssim=0.9888156950473785, fid=0.21431732177734375)
```

In [ ]:
```
bicubic_metrics = compute_metrics(
    lambda x: interpolate(x, size=(256, 256), mode="bicubic"), metric_loader
)
```

```
100%|██████████| 14/14 [09:37<00:00, 41.28s/it]
```

In [ ]:
```
bicubic_metrics
```

Out[ ]:
```
Metrics(mse=4.024813668885534e-05, psnr=43.96280016217913, ssim=0.9836511441639492, fid=1.3964385986328125)
```