Outline

- **Part 1: Neural Style Transfer**
- What is style transfer?
- Style transfer implementation
- Example: Generating images using style transfer

- **Part 2: Generative Adversarial Network (GAN)**
- What is GAN?
- GAN implementation
- Training a GAN

- **Lab Assignment**

Part 1: Neural Style Transfer

Style Transfer

Style Source

Content Source

Output Image

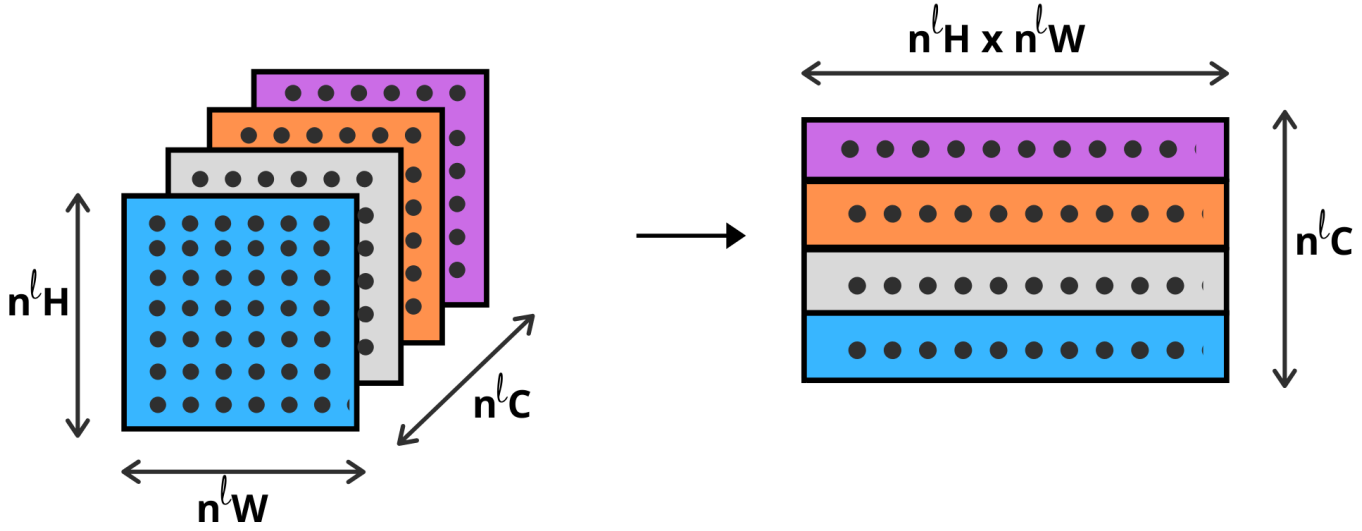
Style Transfer Implementation

- Computing Loss
  - Content loss
  - Style loss
  - Total-variation regularization

- Put everything together: Style Transfer

Content Loss

- Content loss measures how much the feature map of the **generated image** differs from the feature map of the **source image**.
- We only care about the **content representation** of one layer of the network
- We will work with **reshaped versions** of these feature maps.



Content Loss

- $F^{\ell} \in \mathbb{R}^{C_\ell \times M_\ell}$: the feature map for the **current image**
- $P^{\ell} \in \mathbb{R}^{C_\ell \times M_\ell}$: the feature map for the **content source image**
- $M_\ell = H_\ell \times W_\ell$ = # of elements in each feature map
- w_c = weight of the content loss term in the loss function

```python
def content_loss(content_weight, content_current, content_original):
    """
    Compute the content loss for style transfer.

    Inputs:
    - content_weight: Scalar giving the weighting for the content loss.
    - content_current: features of the current image; this is a PyTorch Tensor of shape
      (1, C_l, H_l, W_l).
    - content_target: features of the content image, Tensor with shape (1, C_l, H_l, W_l).

    Returns:
    - scalar content loss
    """
    loss = torch.sum( content_weight * (content_current - content_original)**2 )
    return loss
```

The **content loss** is given by:

$$L_c = w_c \times \sum_{i,j} (F_{ij}^{\ell} - P_{ij}^{\ell})^2$$

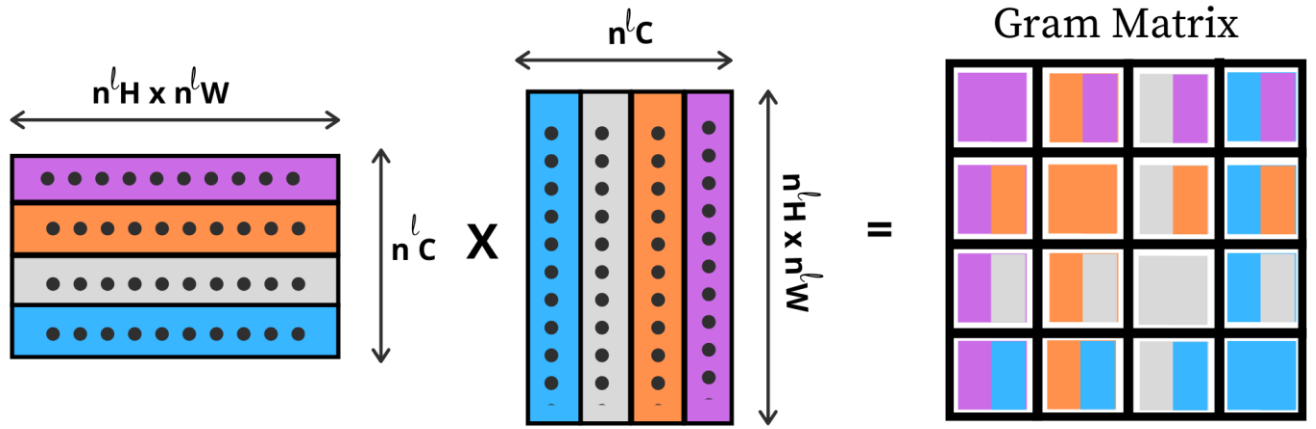
Style Transfer Implementation

- Computing Loss
  - Content loss
  - Style loss
  - Total-variation regularization
- Put everything together: Style Transfer

Style Loss

- For a given layer ℓ, the style loss is defined as follows:
  - First, compute the **Gram matrix G** which represents the correlations between the values in each channel of the feature map.
  - The **Gram matrix** is an approximation of the covariance matrix.
  - We want the **activation statistics** of our **generated image** to match the **activation statistics** of our **style image.**

Style Loss



The **feature map** is given by: $F^\ell \in \mathbb{R}^{C_\ell \times M_\ell}$

The **Gram matrix** is computed by: $G^\ell_{ij} = \sum_k F^\ell_{ik} F^\ell_{jk}$

The **Gram matrix** has shape: (C_ℓ, C_ℓ)

The **Gram matrix** from the feature map of the **current image**: G^ℓ

The **Gram matrix** from the feature map of the **source style image**: A^ℓ

Style Loss

The **style loss** for the layer ℓ is simply the weighted Euclidean distance between the two Gram matrices:

$$L_s^\ell = w_\ell \sum_{i,j} \left(G_{ij}^\ell - A_{ij}^\ell \right)^2$$

In practice we usually compute the style loss at a set of layers L rather just a single layer ℓ; then the **total style loss** is the sum of style losses at each layer:

$$L_s = \sum_{\ell \in \mathcal{L}} L_s^\ell$$

```python
def gram_matrix(features, normalize=True):
    """
    Compute the Gram matrix from features.

    Inputs:
    - features: PyTorch Tensor of shape (N, C, H, W) giving features for
      a batch of N images.
    - normalize: optional, whether to normalize the Gram matrix
        If True, divide the Gram matrix by the number of neurons (H * W * C)

    Returns:
    - gram: PyTorch Tensor of shape (N, C, C) giving the
      (optionally normalized) Gram matrices for the N input images.
    """
    N, C, H, W = features.size()
    features = features.view(N*C,H*W) #reshape it
    gram_matrix = torch.mm(features, features.t())
    if (normalize):
        gram_matrix /= float(H*W*C)
    return gram_matrix
```

```python
def style_loss(feats, style_layers, style_targets, style_weights):
    """
    Computes the style loss at a set of layers.
    Inputs:
    - feats: list of the features at every layer of the current image, as produced by
      the extract_features function.
    - style_layers: List of layer indices into feats giving the layers to include in the
      style loss.
    - style_targets: List of the same length as style_layers, where style_targets[i] is
      a PyTorch Tensor giving the Gram matrix of the source style image computed at
      layer style_layers[i].
    - style_weights: List of the same length as style_layers, where style_weights[i]
      is a scalar giving the weight for the style loss at layer style_layers[i].
    Returns:
    - style_loss: A PyTorch Tensor holding a scalar giving the style loss.
    """
    loss = 0
    for i,layer in enumerate(style_layers):
        current = gram_matrix(feats[layer])
        loss += (style_weights[i] * torch.sum((current-style_targets[i])**2))
    return loss
```

Style Transfer Implementation

- Computing Loss
  - Content loss
  - Style loss
  - Total-variation regularization
- Put everything together: Style Transfer

Total-variation regularization

- Encourage **smoothness** in the image by adding "**total variation**"
- Computed as the sum of the squares of differences in the pixel values for all pairs of pixels that are next to each other (**horizontally or vertically**)

```python
def tv_loss(img, tv_weight):
    """
    Compute total variation loss.
    Inputs:
    - img: PyTorch Variable of shape (1, 3, H, W) holding an input image.
    - tv_weight: Scalar giving the weight w_t to use for the TV loss.
    Returns:
    - loss: PyTorch Variable holding a scalar giving the total variation loss
      for img weighted by tv_weight.
    """
    loss = 0
    #do the row one
    loss += torch.sum( ( img[:,:,1:,:] - img[:,:,:-1,:] )**2 )
    #on paper do for 2X2 and 3X3 then easy to see
    #do the column one
    loss += torch.sum( ( img[:,:,:,1:] - img[:,:,:,:-1] )**2 )
    #weighting
    loss *= tv_weight
    return loss
```

$$L_{tv} = w_t \times \left(\sum_{c=1}^{3} \sum_{i=1}^{H-1} \sum_{j=1}^{W} (x_{i+1,j,c} - x_{i,j,c})^2 + \sum_{c=1}^{3} \sum_{i=1}^{H} \sum_{j=1}^{W-1} (x_{i,j+1,c} - x_{i,j,c})^2 \right)$$

Style Transfer Implementation

- Computing Loss
  - Content loss
  - Style loss
  - Total-variation regularization
- Put everything together: Style Transfer

Put everything together

- Extract features for the content and style images separately
- Initialize output image to content or random noise
- Optimization setup
  - Hyperparameters
  - Optimizer choice
- Update loop
  - Compute losses
  - Update generated image

Example: Generating images with style transfer

Generating images with style transfer

- Use content image and style image to generate a new image
- Hyperparameters include:
  - The weights of content loss and style loss
  - The layer representations of content and style
  - Size of images

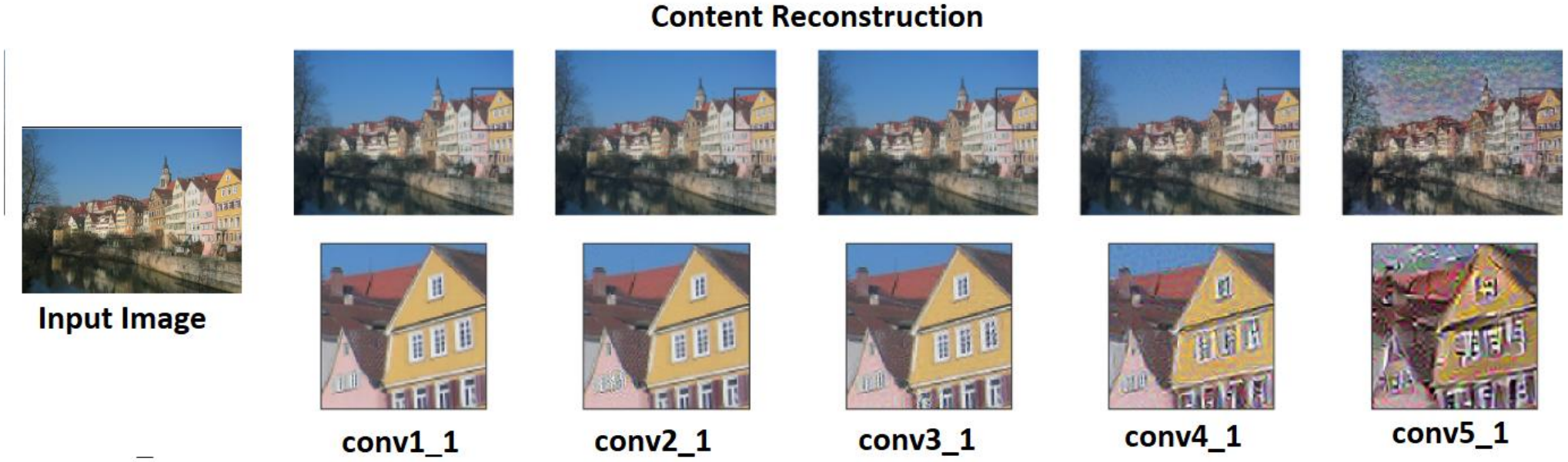
```
# Composition VII + Tubingen
params1 = {
    'content_image' : 'tubingen.jpg',
    'style_image' : 'composition_vii.jpg',
    'image_size' : 192,
    'style_size' : 512,
    'content_layer' : 'conv_2',
    'content_weight' : 5e-2,
    'style_layers' : ['conv_1', 'conv_3', 'conv_5', 'conv_7'],
    'style_weights' : (20000, 500, 1200, 1000),
    'tv_weight' : 5e-2
}

style_transfer(**params1)
```

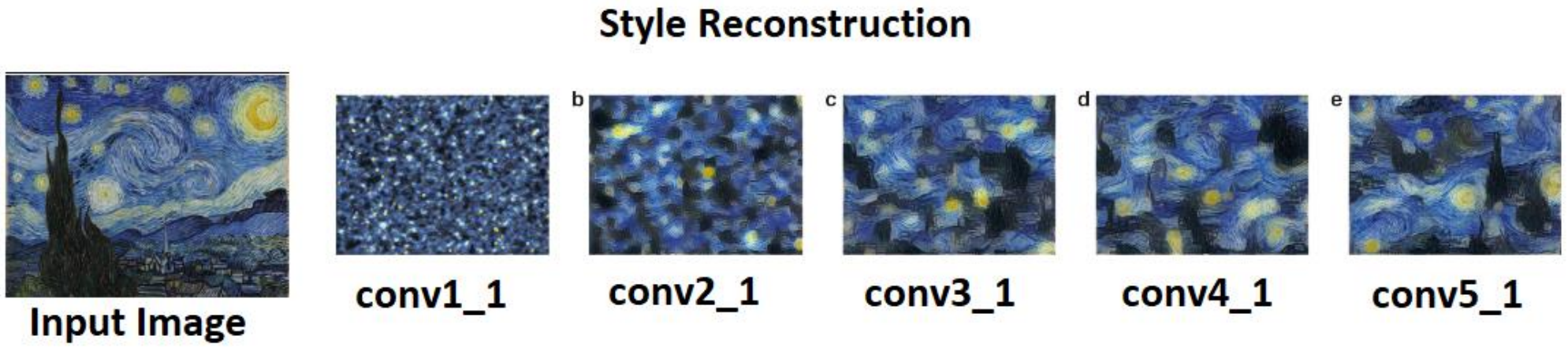
Generating images with style transfer

More weight to content loss

More weight to style loss

Generating images with style transfer





Generating images with style transfer

Resizing style image before running style transfer algorithm can transfer different types of features

Part 2: Generative Adversarial Networks (GANs)

What is a GAN?

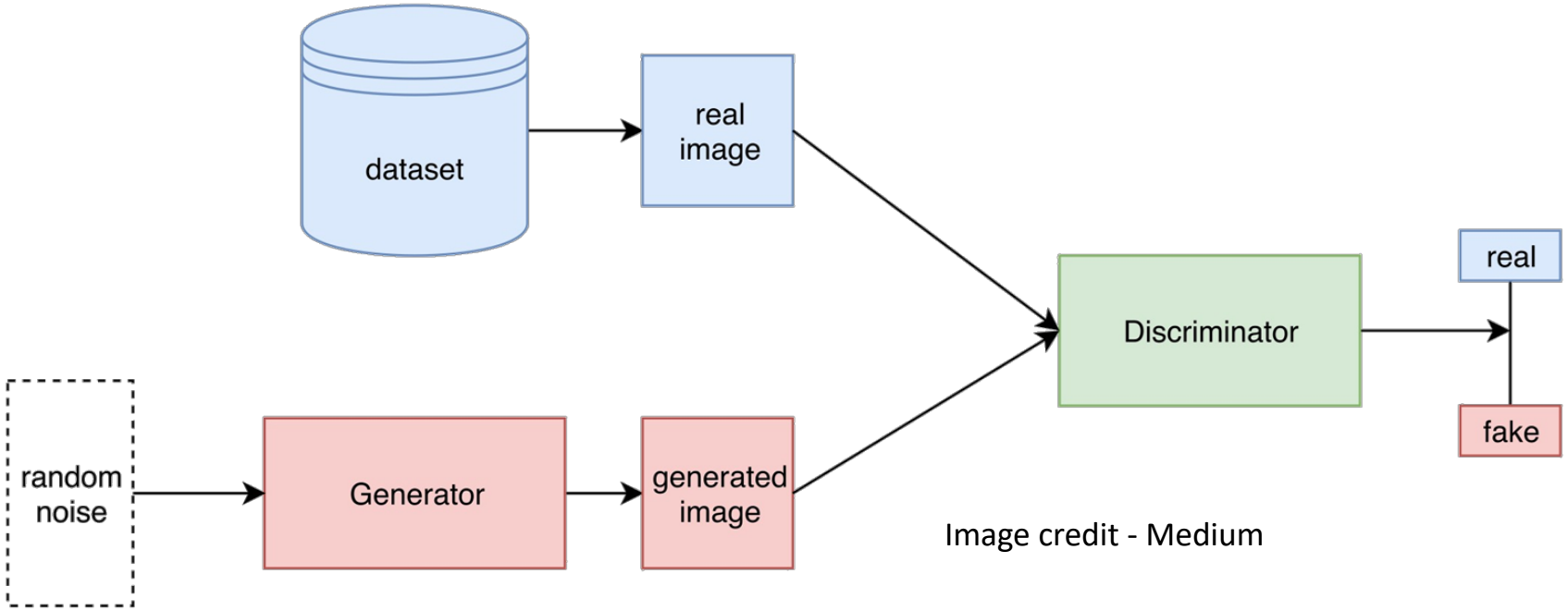


Image credit - Medium

What is a GAN?

- Cost Function

$$\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

- In practice, we alternate the following updates:
  - Update the **generator** (G) to maximize the probability of the discriminator making the incorrect choice on generated data:

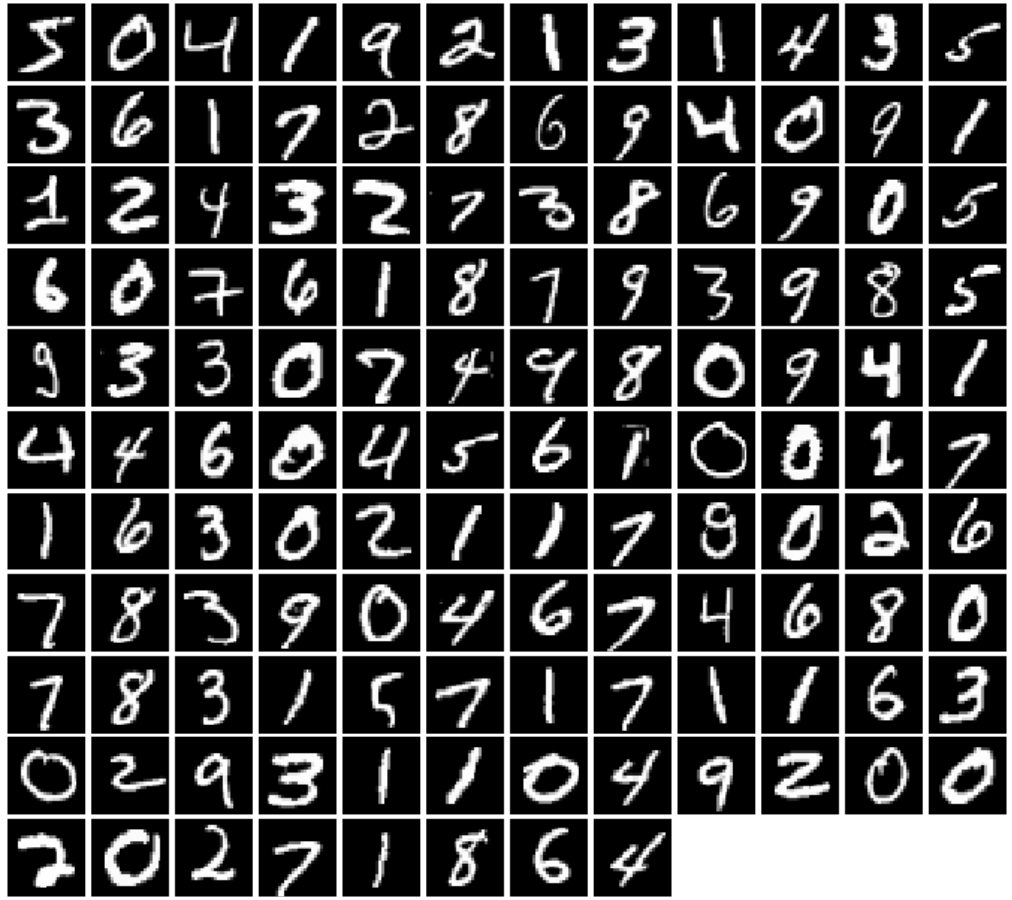
$$\underset{G}{\text{maximize}}\ \mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

  - Update the **discriminator** (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

$$\underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

GAN Implementation

- Dataset: MNIST
- Random Noise
- Discriminator
- Generator
- GAN Loss
- Training a GAN



GAN Implementation

- Dataset: MNIST
- Random Noise
  - Generate uniform noise from -1 to 1 with shape [batch_size, dim].
  - You can also use torch.rand
- Discriminator
- Generator
- GAN Loss
- Training a GAN

```python
def sample_noise(batch_size, dim):
    """
    Generate a PyTorch Tensor of uniform random noise.

    Input:
    - batch_size: Integer giving the batch size of noise to generate.
    - dim: Integer giving the dimension of noise to generate.

    Output:
    - A PyTorch Tensor of shape (batch_size, dim) containing uniform
      random noise in the range (-1, 1).
    """
    return torch.FloatTensor(batch_size, dim).uniform_(-1, 1)
```

GAN Implementation

- Dataset: MNIST
- Random Noise
- Discriminator
  - FC layer (784 -> 256)
  - LeakyRelu with alpha 0.01
  - FC layer (256 -> 256)
  - LeakyRelu with alpha 0.01
  - FC layer (256 -> 1)
- Generator
- GAN Loss
- Training a GAN

```
def discriminator():
    """
    Build and return a PyTorch model implementing the architecture above.
    """
    model = nn.Sequential( Flatten(),
                           nn.Linear(784, 256),
                           nn.LeakyReLU(inplace=True),
                           nn.Linear(256,256),
                           nn.LeakyReLU(inplace=True),
                           nn.Linear(256,1)
                         )
    return model
```

GAN Implementation

- Dataset: MNIST
- Random Noise
- Discriminator
- Generator
  - FC layer (noise_dim -> 1024)
  - Relu
  - FC layer(1024 -> 1024)
  - Relu
  - FC layer (1024 -> 784)
  - Tanh (to clip the image to be in the range [-1, 1]
- GAN Loss
- Training a GAN

```
def generator(noise_dim=NOISE_DIM):
    """
    Build and return a PyTorch model implementing the architecture above.
    """
    model = nn.Sequential( nn.Linear(noise_dim,1024),
                           nn.ReLU(inplace=True),
                           nn.Linear(1024,1024),
                           nn.ReLU(inplace=True),
                           nn.Linear(1024,784),
                           nn.Tanh()
                         )
    return model
```

GAN Implementation

- Dataset: MNIST
- Random Noise
- Discriminator
- Generator
- GAN Loss
- Training a GAN

GAN Loss

- The generator loss:

$$\ell_G = -\mathbb{E}_{z\sim p(z)}\left[\log D(G(z))\right]$$

- The discriminator loss:

$$\ell_D = -\mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] - \mathbb{E}_{z\sim p(z)}\left[\log(1 - D(G(z)))\right]$$

- We will use the binary cross entropy loss - Given a score ***s*** and a label ***y***, the binary cross entropy loss is:

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

GAN loss

- A naïve implementation of the binary cross entropy loss formula can be numerically unstable.

$$bce(s, y) = -y * \log(s) - (1 - y) * \log(1 - s)$$

```python
def bce_loss(input, target):
    """
    Numerically stable version of the binary cross-entropy loss function.

    As per https://github.com/pytorch/pytorch/issues/751
    See the TensorFlow docs for a derivation of this formula:
    https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits

    Inputs:
    - input: PyTorch Tensor of shape (N, ) giving scores.
    - target: PyTorch Tensor of shape (N,) containing 0 and 1 giving targets.

    Returns:
    - A PyTorch Tensor containing the mean BCE loss over the minibatch of input data.
    """
    neg_abs = - input.abs()
    loss = input.clamp(min=0) - input * target + (1 + neg_abs.exp()).log()
    return loss.mean()
```

GAN Loss

- Discriminator loss implementation

```python
def discriminator_loss(logits_real, logits_fake):
    """
    Computes the discriminator loss described above.
    Inputs:
    - logits_real: PyTorch Tensor of shape (N,) giving scores for the real data.
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.
    Returns:
    - loss: PyTorch Tensor containing (scalar) the loss for the discriminator.
    """
    N, _ = logits_real.size()
    loss = (bce_loss(logits_real, torch.ones(N).type(dtype)))+(bce_loss(logits_fake, torch.zeros(N).type(dtype)))
    return loss
```

- Generator loss implementation

```python
def generator_loss(logits_fake):
    """
    Computes the generator loss described above.
    Inputs:
    - logits_fake: PyTorch Tensor of shape (N,) giving scores for the fake data.
    Returns:
    - loss: PyTorch Tensor containing the (scalar) loss for the generator.
    """
    N, _ = logits_fake.size()
    loss = (bce_loss(logits_fake, torch.ones(N).type(dtype)))
    return loss
```

Training a GAN

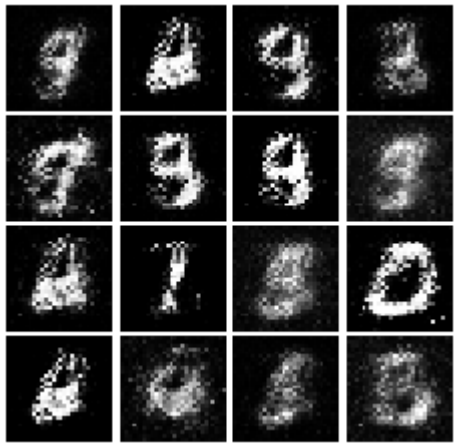
- **Training Discriminator**
  - Pass the **sample noise** to generator and get the fake images
  - Use **.detach()** to break gradient connection
  - Pass the fake images to discriminator, compute loss, and update discriminator
- **Training Generator**
  - Pass the sample noise to generator and get the fake images
  - Pass the fake images to discriminator
  - Compute generator loss and update generator

```python
def run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss, show_every=250,
              batch_size=128, noise_size=96, num_epochs=10):
    """
    Train a GAN!

    Inputs:
    - D, G: PyTorch models for the discriminator and generator
    - D_solver, G_solver: torch.optim Optimizers to use for training the
      discriminator and generator.
    - discriminator_loss, generator_loss: Functions to use for computing the generator and
      discriminator loss, respectively.
    - show_every: Show samples after every show_every iterations.
    - batch_size: Batch size to use for training.
    - noise_size: Dimension of the noise to use as input to the generator.
    - num_epochs: Number of epochs over the training dataset to use for training.
    """
    iter_count = 0
    for epoch in range(num_epochs):
        for x, _ in loader_train:
            if len(x) != batch_size:
                continue
            D_solver.zero_grad()
            real_data = x.type(dtype)
            logits_real = D(2* (real_data - 0.5)).type(dtype)

            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed).detach()
            logits_fake = D(fake_images.view(batch_size, 1, 28, 28))

            d_total_error = discriminator_loss(logits_real, logits_fake)
            d_total_error.backward()
            D_solver.step()

            G_solver.zero_grad()
            g_fake_seed = sample_noise(batch_size, noise_size).type(dtype)
            fake_images = G(g_fake_seed)

            gen_logits_fake = D(fake_images.view(batch_size, 1, 28, 28))
            g_error = generator_loss(gen_logits_fake)
            g_error.backward()
            G_solver.step()
```

Training a GAN

```
# Make the discriminator
D = discriminator().type(dtype)

# Make the generator
G = generator().type(dtype)

# Use the function you wrote earlier to get optimizers for the Discriminator and the Generator
D_solver = get_optimizer(D)
G_solver = get_optimizer(G)
# Run it!
run_a_gan(D, G, D_solver, G_solver, discriminator_loss, generator_loss)
```

Iteration 0



Iteration 1000

Iteration 2000

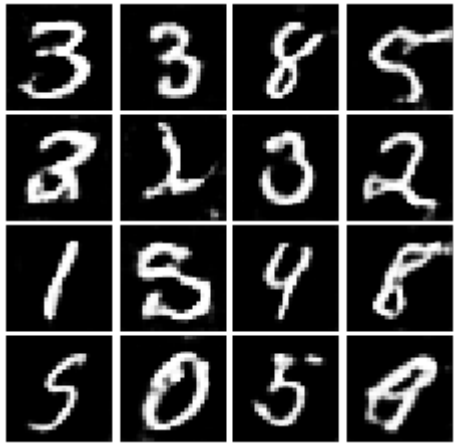
Iteration 3000

Lab Assignment: Deep Convolutional GANs

Assignment Details

- Implement a deep convolutional GAN (DCGAN) to generate much better fake images
- Implement the **discriminator** with:
  - Conv2D: 32 filters, 5x5, Stride 1
  - Leaky Relu (alpha = 0.01)
  - Max Pool 2x2, Stride 2
  - Conv2D: 64 filters, 5x5, Stride 1
  - Leaky ReLu (alpha = 0.01)
  - Max Pool 2x2 Stride 2
  - Flatten
  - Fully Connected with output size 4x4x64
  - Leaky Relu (alpha = 0.01)
  - Fully Connected with output size 1

Results of GAN

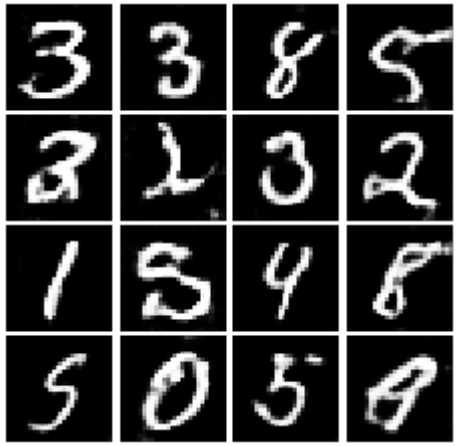


Results of DCGAN

Assignment Details

- Implement the **generator** with:
  - Fully connected with output size 1024
  - Relu
  - BatchNorm
  - Fully connected with output size 7x7x128
  - ReLu
  - BatchNorm
  - Reshape into image Tensor of shape 7,7,128
  - Conv2D Transpose: 64 filters of 4x4, stride 2, ‘same’ padding (use padding = 1)
  - BatchNorm
  - Conv2D Transpose: 1 filter of 4x4, stride 2, ‘same’ padding (use padding = 1)
  - Tanh
  - Should have a 28x28x1 image, reshape back into 784 vector

Results of GAN



Results of DCGAN

Evaluation:

Train DCGAN on MNIST dataset and plot out the generated images every 150 training iterations for 5 epochs