University of Tehran
Electrical and Computer Engineering Department
Neural Networks and Deep Learning
Mini-Project 3

Name	Danial Saeedi
Student No	810198571
Date	Monday - 2022 31 January

Table of contents

1. Question 1: AC-GAN

Part A: What is AC-GAN?

The **Auxiliary Classifier GAN**, or **AC-GAN** for short, is an extension of the conditional GAN that changes the discriminator to predict the class label of a given image rather than receive it as input. By adding an auxiliary classifier to the discriminator of a GAN, the discriminator produces not only a probability distribution over sources but also probability distribution over the class labels. It has the effect of stabilizing the training process and allowing the generation of large high-quality images whilst learning a representation in the latent space that is independent of the class label.

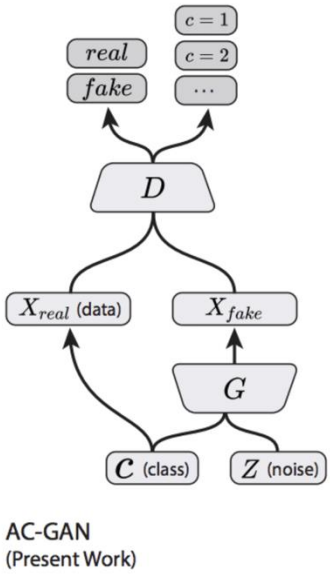


Figure 1 AC-GAN

Part B

In this part, Adam and BCE were used as optimizer and loss function respectively.

Importing the CIFAR-10 Dataset

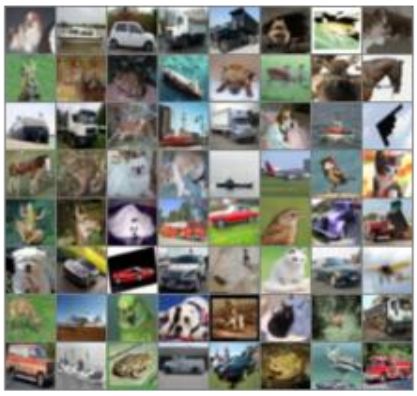


Figure 2 Sample Images of CIFAR-10

Defining Generator

In [ ]:
```
class Generator(nn.Module):

    def __init__(self):
        super(Generator,self).__init__()

        #input 100*1*1
        self.layer1 = nn.Sequential(nn.ConvTranspose2d(100,512,4,1,0,bias = False),
                                    nn.ReLU(True))

        #input 512*4*4
        self.layer2 = nn.Sequential(nn.ConvTranspose2d(512,256,4,2,1,bias = False),
                                    nn.BatchNorm2d(256),
                                    nn.ReLU(True))
        #input 256*8*8
        self.layer3 = nn.Sequential(nn.ConvTranspose2d(256,128,4,2,1,bias = False),
                                    nn.BatchNorm2d(128),
                                    nn.ReLU(True))
        #input 128*16*16
        self.layer4 = nn.Sequential(nn.ConvTranspose2d(128,64,4,2,1,bias = False),
                                    nn.BatchNorm2d(64),
                                    nn.ReLU(True))
        #input 64*32*32
        self.layer5 = nn.Sequential(nn.ConvTranspose2d(64,3,4,2,1,bias = False),
                                    nn.Tanh())
        #output 3*64*64

        self.embedding = nn.Embedding(10,100)


    def forward(self,noise,label):

        label_embedding = self.embedding(label)
        x = torch.mul(noise,label_embedding)
        x = x.view(-1,100,1,1)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        return x
```

Discriminator Definition

In [ ]:
```
class Discriminator(nn.Module):

    def __init__(self):
        super(Discriminator,self).__init__()

        #input 3*64*64
        self.layer1 = nn.Sequential(nn.Conv2d(3,64,4,2,1,bias = False),
                                    nn.BatchNorm2d(64),
                                   nn.LeakyReLU(0.2,True),
                                   nn.Dropout2d(0.5))

        #input 64*32*32
        self.layer2 = nn.Sequential(nn.Conv2d(64,128,4,2,1,bias = False),
                                    nn.BatchNorm2d(128),
                                   nn.LeakyReLU(0.2,True),
                                   nn.Dropout2d(0.5))
        #input 128*16*16
        self.layer3 = nn.Sequential(nn.Conv2d(128,256,4,2,1,bias = False),
                                    nn.BatchNorm2d(256),
                                   nn.LeakyReLU(0.2,True),
                                   nn.Dropout2d(0.5))
        #input 256*8*8
        self.layer4 = nn.Sequential(nn.Conv2d(256,512,4,2,1,bias = False),
                                    nn.BatchNorm2d(512),
                                   nn.LeakyReLU(0.2,True))
        #input 512*4*4
        self.validity_layer = nn.Sequential(nn.Conv2d(512,1,4,1,0,bias = False),
                                   nn.Sigmoid())

        self.label_layer = nn.Sequential(nn.Conv2d(512,11,4,1,0,bias = False),
                                   nn.LogSoftmax(dim = 1))

    def forward(self,x):

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        validity = self.validity_layer(x)
        plabel = self.label_layer(x)

        validity = validity.view(-1)
        plabel = plabel.view(-1,11)

        return validity,plabel
```

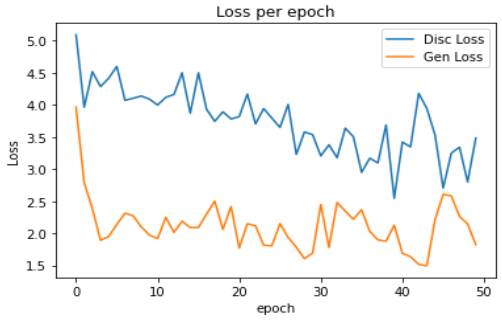


Figure 3 Loss per epoch

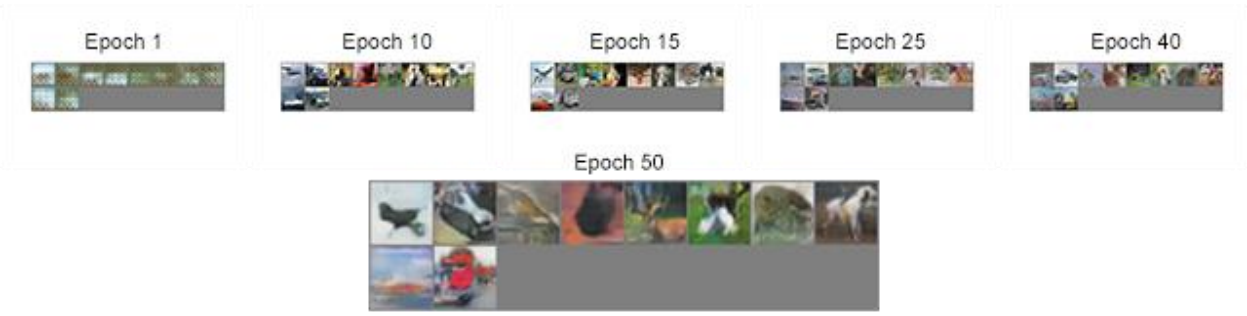


Figure 4 Result of training after 50 epochs

Part C: Trying different optimizer

While momentum accelerates our search in direction of minimal, RMSProp impedes our search in direction of oscillations. Adam Optimization algorithm combines the heuristics of both Momentum and RMSProp.

As you can see, the model is more stable with Adam optimizer than RMSprop. Also the generated images using Adam Optimizer seems to be **brighter** and **clearer** than RMSprop optimizer. RMSprop is often recommended for RNN.

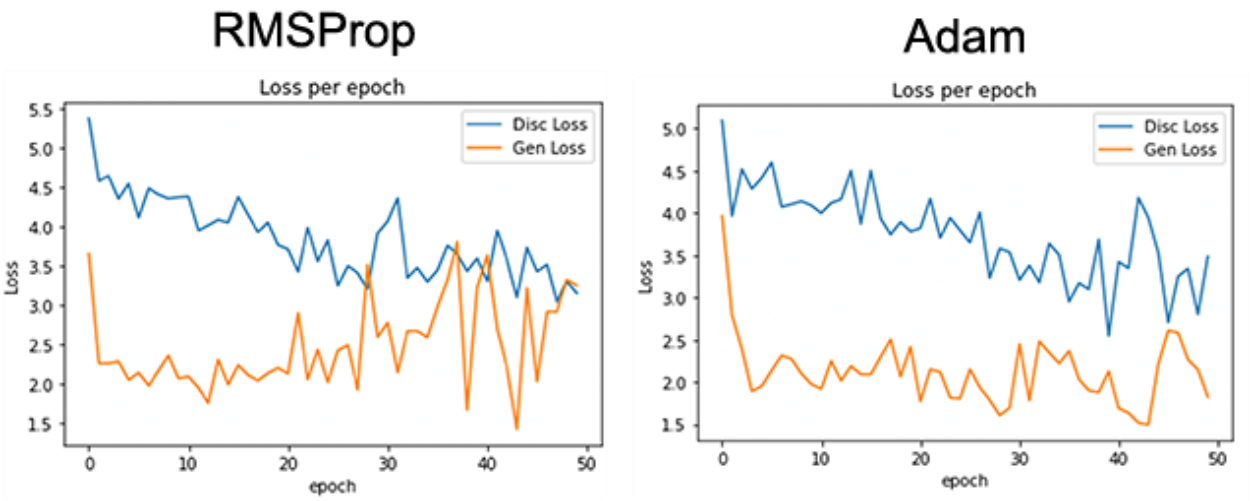


Figure 5 Comparing Adam and RMSProp

Figure 6 Generated images using RMSprop Optimizer

Part D(Bonus): U-Net as Generator

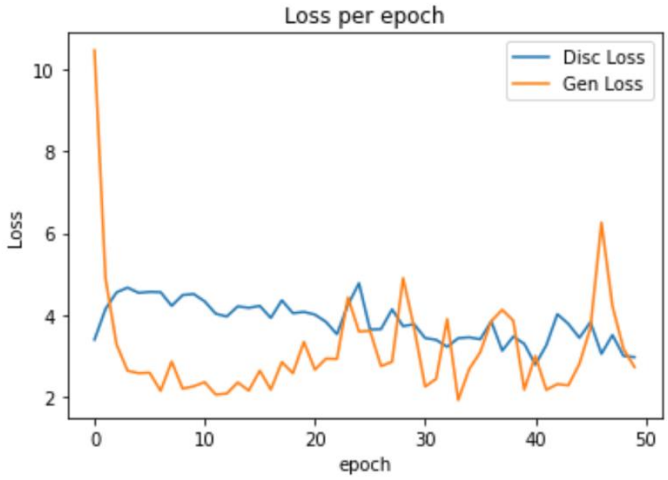


Figure 7 Loss per epoch

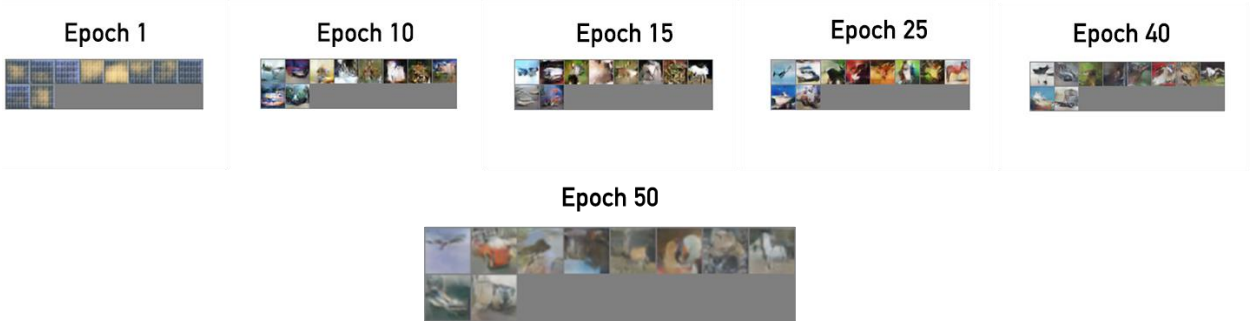


Figure 8 Generated Pictures per epoch

Block

The contractive path consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for down-sampling. At each down-sampling step we double the number of feature channels.

```python
class Block(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3)
        self.relu  = nn.ReLU()
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3)

    def forward(self, x):
        return self.relu(self.conv2(self.relu(self.conv1(x))))
```

Encoder

Each block is followed by a 2x2 max pooling operation with stride 2 for down-sampling.

```python
class Encoder(nn.Module):
    def __init__(self, chs=(3,64,128,256,512,1024)):
        super().__init__()
        self.enc_blocks = nn.ModuleList([Block(chs[i], chs[i+1]) for i in range(len(chs)-1)])
        self.pool       = nn.MaxPool2d(2)

    def forward(self, x):
        ftrs = []
        for block in self.enc_blocks:
            x = block(x)
            ftrs.append(x)
            x = self.pool(x)
        return ftrs
```

Decoder

Every step in the expansive path consists of an up-sampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution.

```python
class Decoder(nn.Module):
    def __init__(self, chs=(1024, 512, 256, 128, 64)):
        super().__init__()
        self.chs         = chs
        self.upconvs    = nn.ModuleList([nn.ConvTranspose2d(chs[i],
                                                            chs[i+1], 2, 2) for i in range(len(chs)-1)])
        self.dec_blocks = nn.ModuleList([Block(chs[i], chs[i+1]) for i in range(len(chs)-1)])

    def forward(self, x, encoder_features):
        for i in range(len(self.chs)-1):
            x        = self.upconvs[i](x)
            enc_ftrs = self.crop(encoder_features[i], x)
            x        = torch.cat([x, enc_ftrs], dim=1)
            x        = self.dec_blocks[i](x)
        return x

    def crop(self, enc_ftrs, x):
        _, _, H, W = x.shape
        enc_ftrs   = torchvision.transforms.CenterCrop([H, W])(enc_ftrs)
        return enc_ftrs
```

U-Net

```python
class UNet_Generator(nn.Module):
    def __init__(self, enc_chs=(3,64,128,256,512,1024), dec_chs=(1024, 512, 256, 128, 64),
                 num_class=1, retain_dim=False, out_sz=(572,572)):
        super().__init__()
        self.encoder     = Encoder(enc_chs)
        self.decoder     = Decoder(dec_chs)
        self.head        = nn.Conv2d(dec_chs[-1], num_class, 1)
        self.retain_dim  = retain_dim

    def forward(self, x):
        enc_ftrs = self.encoder(x)
        out      = self.decoder(enc_ftrs[::-1][0], enc_ftrs[::-1][1:])
        out      = self.head(out)
        if self.retain_dim:
            out = F.interpolate(out, (572,572))
        return out
```

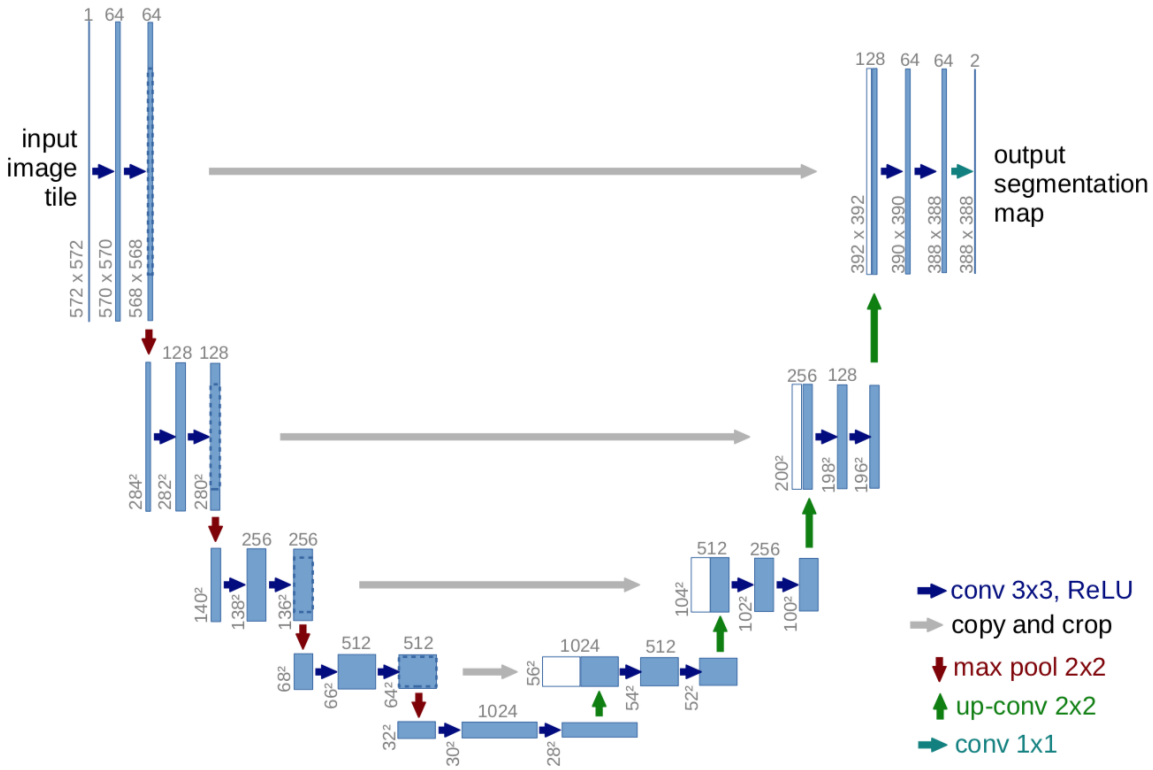


Figure 9 U-Net Architecture

2. Question 2: DC-GAN

The dataset for training in this question can be found in here.

Part A

It is immediately obvious that the late training sample is of much **higher quality** than the early training sample. The images are less blurry and almost appear to resemble emojis.

Early on the GAN seems to generate a lot of rectangular shapes, while later in training it learns that emojis usually have a rounder shape. It also learned to draw emoji faces fairly well (e.g. the image in the lower left corner of the late training sample).

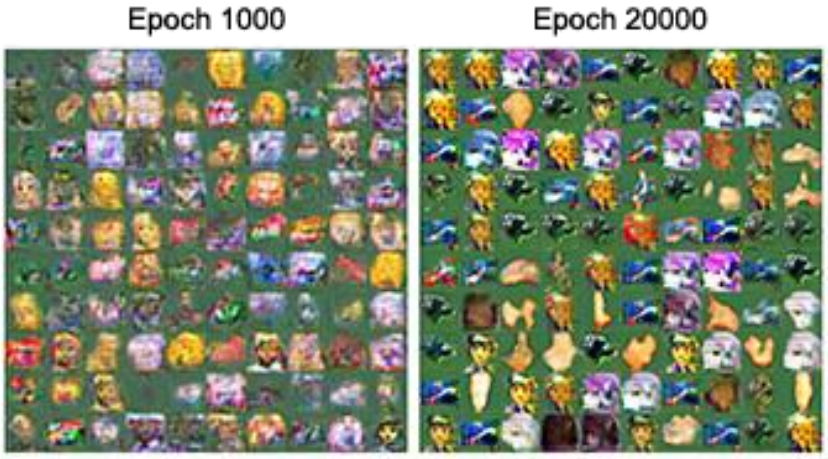


Figure 10 Generated Emojis

```
args_dict = {
                'image_size':32,
                'g_conv_dim':32,
                'd_conv_dim':64,
                'noise_size':100,
                'num_workers': 0,
                'train_iters':20000,
                'X':'Windows',  # options: 'Windows' / 'Apple'
                'Y': 'Deconvolution',
                # Optimize Options
                'lr':0.0003,
                'beta1':0.5,
                'beta2':0.999,

                'batch_size':32,
                'checkpoint_dir': 'results/checkpoints_gan',
                'sample_dir': 'results/samples_gan',
                'load': None,
                'log_step':200,
                'sample_every':200,
                'checkpoint_every':1000,
                'spectral_norm': False,
                'gradient_penalty': False,
                'd_train_iters': 1
}
```

Figure 11 Model's Parameters

Defining Generator

In [ ]:
```
class DCGenerator_Deconvolution(nn.Module):
    def __init__(self, noise_size, conv_dim, spectral_norm=False):
        super(DCGenerator_Deconvolution, self).__init__()
        self.main = nn.Sequential(
            # Input is 1 x 64 x 64
            nn.Conv2d(1, 64, (4, 4), (2, 2), (1, 1), bias=True),
            nn.LeakyReLU(0.2, True),
            # State size. 64 x 32 x 32
            nn.Conv2d(64, 128, (4, 4), (2, 2), (1, 1), bias=False),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2, True),
            # State size. 128 x 16 x 16
            nn.Conv2d(128, 256, (4, 4), (2, 2), (1, 1), bias=False),
            nn.BatchNorm2d(256),
            nn.LeakyReLU(0.2, True),
            # State size. 256 x 8 x 8
            nn.Conv2d(256, 512, (4, 4), (2, 2), (1, 1), bias=False),
            nn.BatchNorm2d(512),
            nn.LeakyReLU(0.2, True),
            # State size. 512 x 4 x 4
            nn.Conv2d(512, 1, (4, 4), (1, 1), (0, 0), bias=True),
            nn.Sigmoid()
        )

    def forward(self, x):
        out = self.main(x)
        out = torch.flatten(out, 1)

        return out
```

Discriminator Definition

In [ ]:
```
class DCDiscriminator(nn.Module):
    def __init__(self, conv_dim=64, spectral_norm=False):
        super(DCDiscriminator, self).__init__()

        self.conv1 = conv(in_channels=3, out_channels=conv_dim, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.conv2 = conv(in_channels=conv_dim, out_channels=conv_dim*2, kernel_size=5, stride=2,
                          spectral_norm=spectral_norm)
        self.conv3 = conv(in_channels=conv_dim*2, out_channels=conv_dim*4, kernel_size=5,
                          stride=2, spectral_norm=spectral_norm)
        self.conv4 = conv(in_channels=conv_dim*4, out_channels=1, kernel_size=5, stride=2, padding=1,
                          batch_norm=False, spectral_norm=spectral_norm)

    def forward(self, x):
        batch_size = x.size(0)

        out = F.relu(self.conv1(x))
        out = F.relu(self.conv2(out))
        out = F.relu(self.conv3(out))

        out = self.conv4(out).squeeze()
        out_size = out.size()
        if out_size != torch.Size([batch_size,]):
            raise ValueError("expect {} x 1, but get {}".format(batch_size, out_size))
        return out
```

Part B: Gradient Penalty

Ways to for train stable GANs

1. **Use Strided Convolutions:** In GANs, it is recommended to not use pooling layers, and instead use the stride in convolutional layers to perform down-sampling in the discriminator model. Similarly, fractional stride (deconvolutional layers) can be used in the generator for up-sampling.
2. **Remove Fully-Connected Layers:** In GANs, fully-connected layers are not used, in the discriminator and the convolutional layers are flattened and passed directly to the output layer.
3. **Use Batch Normalization:** Batch norm layers are recommended in both the discriminator and generator models, except the output of the generator and input to the discriminator.
4. **Use ReLU, Leaky ReLU, and Tanh:** ReLU is recommended for the generator, but not for the discriminator model. Instead, a variation of ReLU that allows values less than zero, called Leaky ReLU, is preferred in the discriminator.
5. **Gradient Penalty:** The idea of Gradient Penalty is to **enforce a constraint** such that the gradients of the critic's output with respect to the inputs to have unit norm.

   In Figure 9. the terms to the left of the sum is the original critic loss and the term to the right of the sum is the **gradient penalty**.

$$\mathcal{L} = \mathop{\mathbb{E}}_{\tilde{\mathbf{x}} \sim \mathbb{P}_g} [f(\tilde{\mathbf{x}})] - \mathop{\mathbb{E}}_{\mathbf{x} \sim \mathbb{P}_r} [f(\mathbf{x})] + \lambda \mathop{\mathbb{E}}_{\hat{\mathbf{x}} \sim \mathbb{P}_{\hat{x}}} [(||\nabla_{\hat{\mathbf{x}}} f(\hat{\mathbf{x}})||_2 - 1)^2]$$

Figure 12 Critic Loss Function

Looking at samples from different stages of training, the training does indeed appear more stable in terms of oscillation and mode collapse. However, the results appear to be of much lower quality than the previous model. This might simply be due to insufficiently tuned hyperparameters and too few training iterations.

Adding a gradient penalty to the loss function is an alternative way of a Lipschitz constraint, the simplest being gradient clipping, which requires careful hyperparameter tuning. The gradient penalty instead penalizes gradients with a norm largely different from 1 directly in the cost function itself. This might in some cases reduce or eliminate the need for hyperparameter tuning to enforce the Lipschitz constraint.

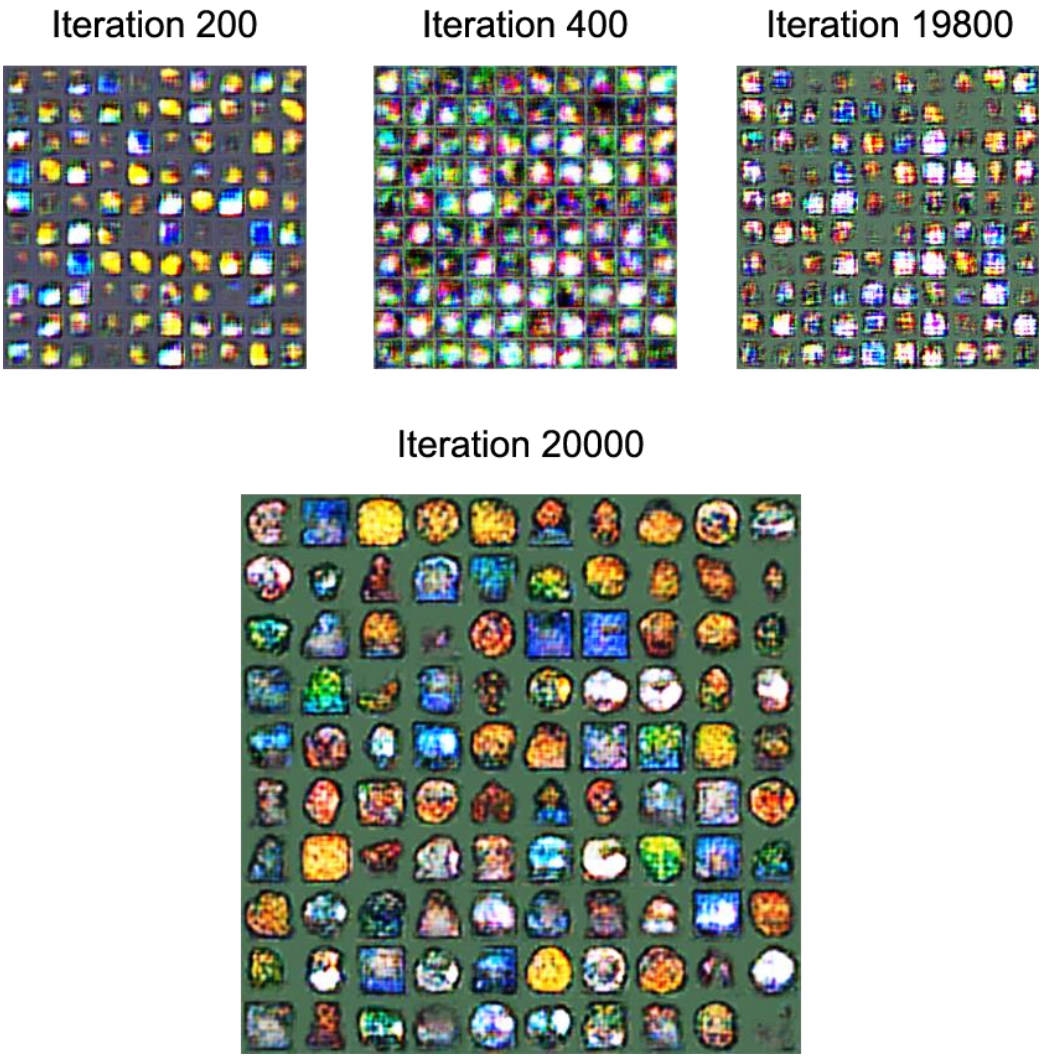


Figure 13 Generated Emojis using Gradient Penalty

Part C: Changing other Hyperparameters

For all of the following, only the stated parameter has been changed from the default settings used in part 1 of this section. For all of them a sample from iteration 200 is shown on the left, and a sample from iteration 20000 on the right.

Spectral Norm Enabled

Learning Rate set to 0.00015 (half the default)

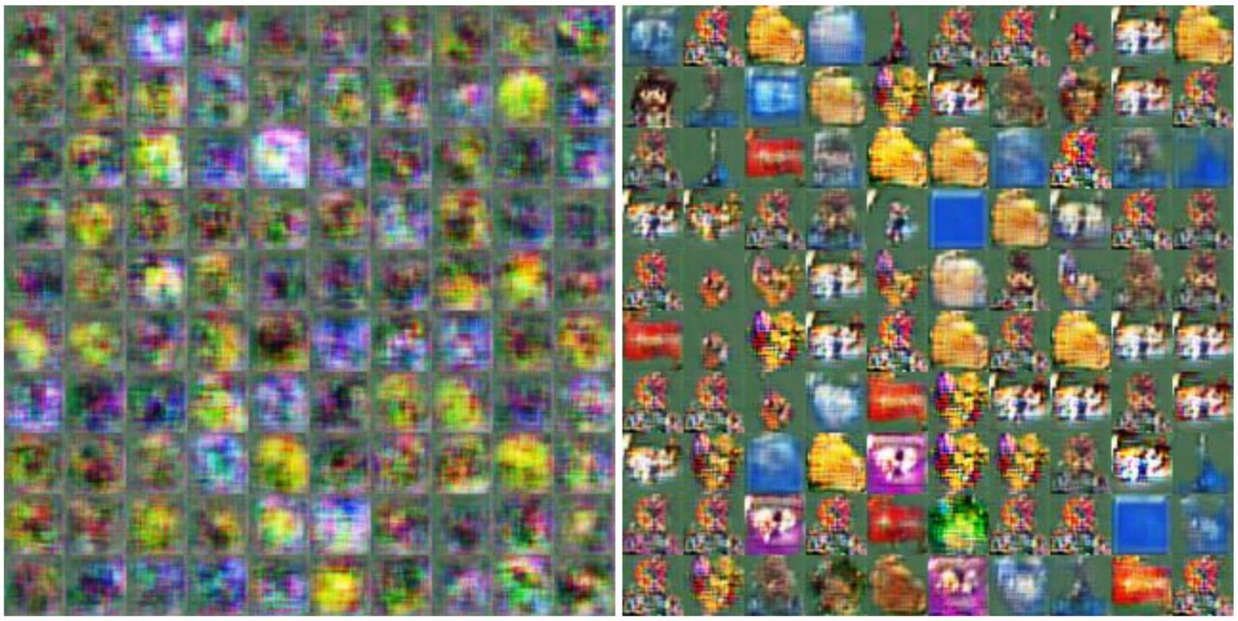


Figure 14 Generated Emojis

d_train_iters set to 5

Figure 15 Generated Emojis

None of the measures seem to significantly help with stabilizing the training and the final results are far from perfect. Especially mode collapse seems to be a severe issue with this architecture for all sets of hyperparameters I have tried. However, in theory those measures can help, when the parameters are finely tuned. Applying the spectral norm technique can help with stabilizing the discriminator by normalizing its weight matrices. Lowering the learning rate can generally be helpful for almost any machine learning model, albeit very computationally expensive. Increasing the number of discriminator updates per generator update can be helpful in stabilizing the training by reducing the generators tendency towards oscillating between generating different samples from step to step.

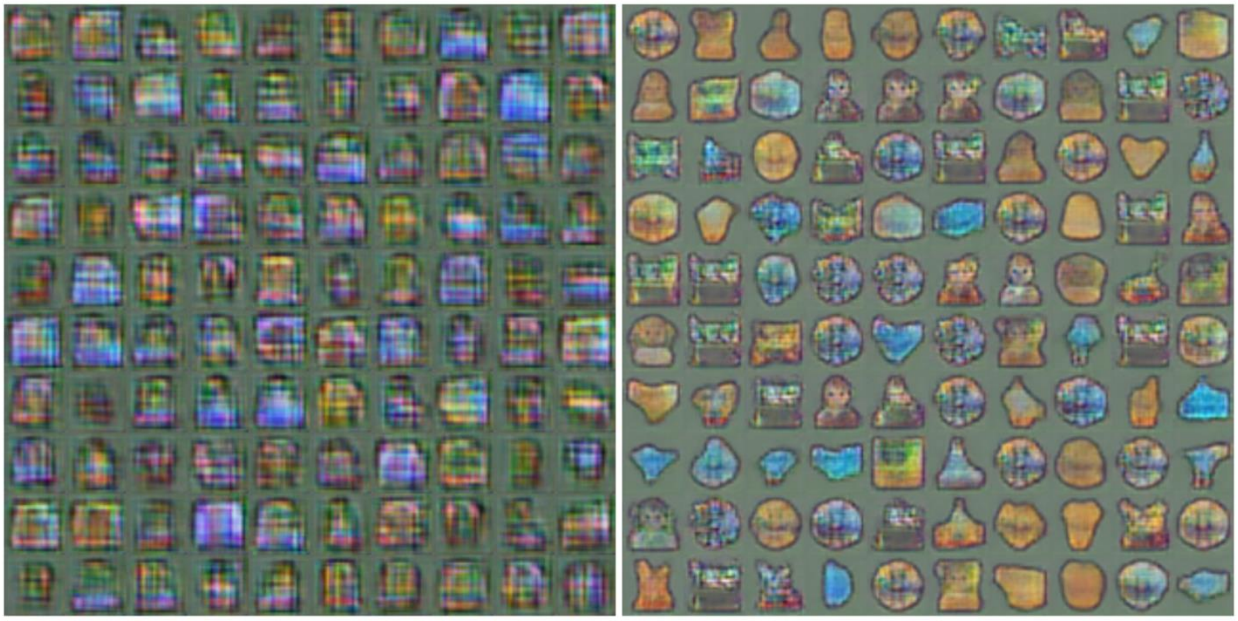
Part D: Using up-sampling instead of deconvolution

Up-sampling layer has been implemented in the following manner:

In [ ]:
```
# Generates a PyTorch Tensor of uniform random noise.
def sample_noise(batch_size, dim):
    return to_var(torch.rand(batch_size, dim) * 2 - 1).unsqueeze(2).unsqueeze(3)


def upsampling(in_channels, out_channels, kernel_size, stride=2, padding=2, batch_norm=True, spectral_norm=False):
    layers = []
    if stride>1:
        layers.append(nn.Upsample(scale_factor=stride))
    conv_layer = nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                           kernel_size=kernel_size, stride=1, padding=padding, bias=False)
    if spectral_norm:
        layers.append(SpectralNorm(conv_layer))
    else:
        layers.append(conv_layer)
    if batch_norm:
        layers.append(nn.BatchNorm2d(out_channels))
    return nn.Sequential(*layers)
```

In [ ]:
```
class DCGenerator_Upsampling(nn.Module):
    def __init__(self, noise_size, conv_dim, spectral_norm=False):
        super(DCGenerator_Upsampling, self).__init__()

        self.conv_dim = conv_dim

        self.linear_bn = upsampling(in_channels=100, out_channels=self.conv_dim*4,
                                    kernel_size=3, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upsampling1 = upsampling(in_channels=self.conv_dim*4, out_channels=self.conv_dim*2,
                                      kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upsampling2 = upsampling(in_channels=self.conv_dim*2, out_channels=self.conv_dim,
                                      kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upsampling3 = upsampling(in_channels=self.conv_dim, out_channels=3,
                                      kernel_size=5, stride=2, spectral_norm=spectral_norm)

    def forward(self, z):
        batch_size = z.size(0)

        out = F.relu(self.linear_bn(z)).view(-1, self.conv_dim*4, 4, 4)    # BS x 128 x 4 x 4
        out = F.relu(self.upsampling1(out))  # BS x 64 x 8 x 8
        out = F.relu(self.upsampling2(out))  # BS x 32 x 16 x 16
        out = F.tanh(self.upsampling3(out))  # BS x 3 x 32 x 32

        out_size = out.size()
        if out_size != torch.Size([batch_size, 3, 32, 32]):
            raise ValueError("expect {} x 3 x 32 x 32, but get {}".format(batch_size, out_size))
        return out
```

Using up-sampling layers instead of deconvolution layers made the results much better. The following samples are from iteration 200 and iteration 4000. The quality for both images are not very ideal. An human can easily identify this emojis as generated since they have serious artifacts. However, the quality does improve a lot through the training process. Compare to the one from iteration 200, where all the generated emojis look like some random pixels, the one from iteration 4000 have some emojis that are recognizable such as a face or a human. Both the shape and the color of the emojis improved a lot.



(a) Iteration 200 (b) Iteration 4000

Figure 16 Generated Emojis

Part E: Info-GAN

Info-GAN Concept

Info-GAN is an information-theoretic extension to the GAN that is able to learn disentangled representations in an unsupervised manner. Info-GANs are used when your dataset is very complex, when you'd like to train a cGAN and the dataset is not labelled, and when you'd like to see the most important features of your images.

In information theory, mutual information between X and Y, I(X, Y), measures the "amount of information" learned from knowledge of random variable Y about the other random variable X.

The mutual information can be expressed as the difference of two entropy terms:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

- I(X; Y) is the reduction of uncertainty in X when Y is observed.
- If X and Y are independent, then I(X; Y) = 0, because knowing one variable reveals nothing about the other.

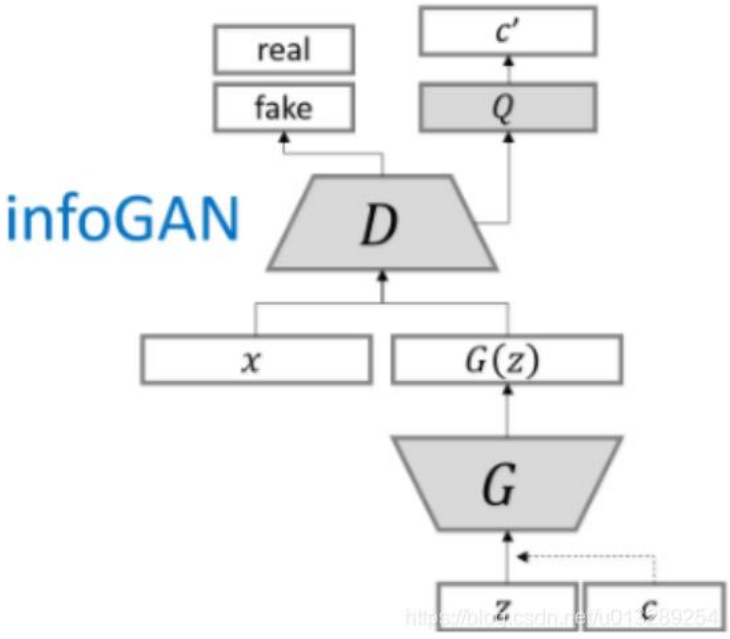


Figure 17 Info-GAN Structure Source: https://sh-tsang.medium.com/review-infogan-interpretable-representation-learning-by-information-maximizing-generative-4344b9dc45d0

The generated emojis by Info-GAN model:

Figure 18 400 Epochs

Figure 19 4000 epochs

As you can see, the generated emoji by Info-GAN is much better even though some of the emojis are not clear. If we continue training this model, the result could be much clearer.

3. Question 3: WGAN

Part I

One-sided label smoothing

Deep networks may suffer from overconfidence. For example, it uses very few features to classify an object. To mitigate the problem, deep learning uses regulation and dropout to avoid overconfidence.

In GAN, if the discriminator depends on a small set of features to detect real images, the generator may just produce these features only to exploit the discriminator. The optimization may turn too greedy and produces no long term benefit. In GAN, overconfidence hurts badly. To avoid the problem, we penalize the discriminator when the prediction for any real images go beyond 0.9 (D(real image)>0.9). This is done by setting our target label value to be 0.9 instead of 1.0.

Add Noise

The recent work by here proposes to extend the support of the distributions by adding noise to both generated and real images before they are fed as input to the discriminator. This procedure results in a smoothing of both data and model probability distributions, which indeed increases their support extent. We generate random numbers(noise) from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution).

$$out_i \sim N(0,1)$$

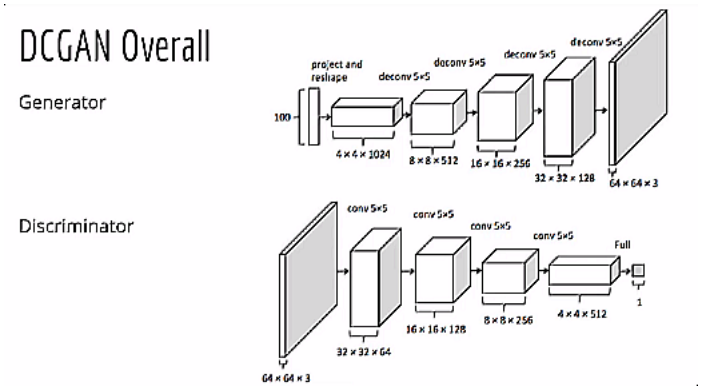


Figure 20 DC-GAN Architecture

Generator

In [ ]:
```
# Generator
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d( ngf * 2, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, input):
        return self.main(input)
```

Discriminator

In [ ]:
```
# Discriminator
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

CIFAR-10 Dataset

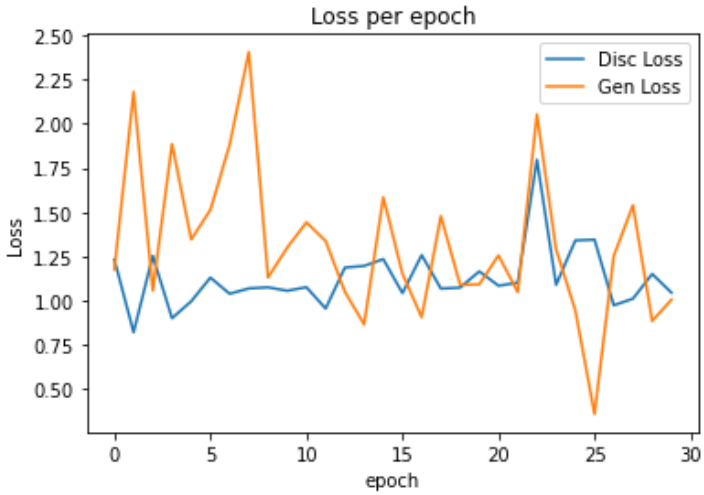


Figure 21 Loss per epoch

Figure 22 Generated Images After 30 Epochs

Fashion-MNIST Dataset

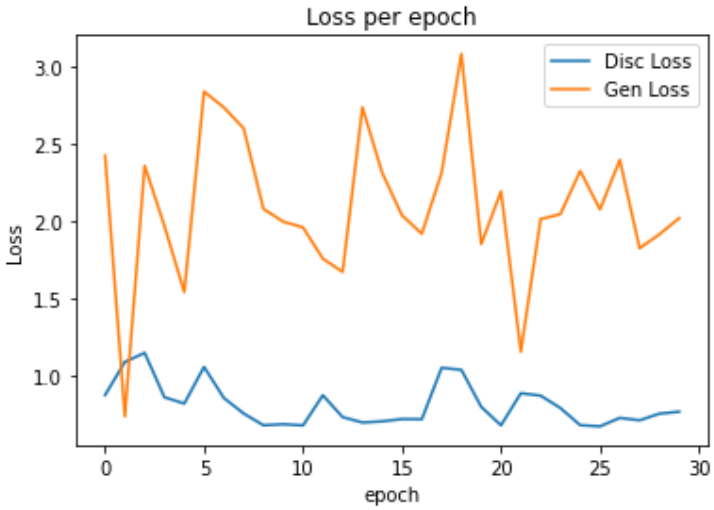


Figure 23 Loss per epoch

Figure 24 Generated Images After 30 Epochs

Part II: WGAN

The Wasserstein Generative Adversarial Network, or Wasserstein GAN, is an extension to the generative adversarial network that both improves the **stability** when training the model and provides a **loss function** that correlates with the quality of generated images.

1. Use a linear activation function in the output layer of the critic model (instead of sigmoid).
2. Use -1 labels for real images and 1 labels for fake images (instead of 1 and 0).
3. Use Wasserstein loss to train the critic and generator models.
4. Constrain critic model weights to a limited range after each mini batch update.
5. Update the critic model more times than the generator each iteration.
6. Use the RMSProp version of gradient descent with a small learning rate and no momentum.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 25 Algorithm for the Wasserstein Generative Adversarial Networks.

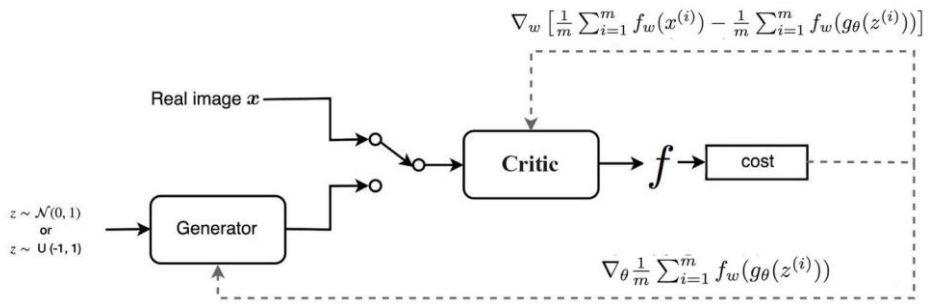


Figure 26 WGAN Architecture

Solution for Vanishing Gradient and Mode-collapse problems

One Solution for the issues discussed above is to use Wasserstein loss that approximates Earth Mover's Distance (EMD is the amount of effort needed to make one distribution to another distribution. In our case we want to make the generated image distribution equal to the real image distribution). WGAN uses **Wasserstein loss**.

	Discriminator/Critic	Generator
GAN	$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$	$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$
WGAN	$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f(x^{(i)}) - f(G(z^{(i)}))\right]$	$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f(G(z^{(i)}))$

Figure 27 GAN vs. WGAN

1. The loss is the difference b/w expected value of discriminator's output to real images and the expected value of discriminator's output to fake images that were generated.
2. The discriminator's objective is to maximize this difference while the generator's goal is to minimize this difference.
3. To use Wasserstein loss, our discriminator needs to be 1-L (1-Lipschitz) continuous i.e norm of gradient must be at most 1 on every point.

Implementing Generator and Discriminator of WGAN

```
netg = nn.Sequential(
            nn.ConvTranspose2d(opt.nz,opt.ngf*8,4,1,0,bias=False),
            nn.BatchNorm2d(opt.ngf*8),
            nn.ReLU(True),

            nn.ConvTranspose2d(opt.ngf*8,opt.ngf*4,4,2,1,bias=False),
            nn.BatchNorm2d(opt.ngf*4),
            nn.ReLU(True),

            nn.ConvTranspose2d(opt.ngf*4,opt.ngf*2,4,2,1,bias=False),
            nn.BatchNorm2d(opt.ngf*2),
            nn.ReLU(True),

            nn.ConvTranspose2d(opt.ngf*2,opt.ngf,4,2,1,bias=False),
            nn.BatchNorm2d(opt.ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(opt.ngf,opt.nc,4,2,1,bias=False),
            nn.Tanh()
        )

netd = nn.Sequential(
            nn.Conv2d(opt.nc,opt.ndf,4,2,1,bias=False),
            nn.LeakyReLU(0.2,inplace=True),

            nn.Conv2d(opt.ndf,opt.ndf*2,4,2,1,bias=False),
            nn.BatchNorm2d(opt.ndf*2),
            nn.LeakyReLU(0.2,inplace=True),

            nn.Conv2d(opt.ndf*2,opt.ndf*4,4,2,1,bias=False),
            nn.BatchNorm2d(opt.ndf*4),
            nn.LeakyReLU(0.2,inplace=True),

            nn.Conv2d(opt.ndf*4,opt.ndf*8,4,2,1,bias=False),
            nn.BatchNorm2d(opt.ndf*8),
            nn.LeakyReLU(0.2,inplace=True),

            nn.Conv2d(opt.ndf*8,1,4,1,0,bias=False),
            # Modification 1: remove sigmoid
            # nn.Sigmoid()
        )
```

Figure 28 Implementation of WGAN

Noise Generation

We generate random numbers(noise) from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution).

$$out_i \sim N(0,1)$$

CIFAR-10 Dataset

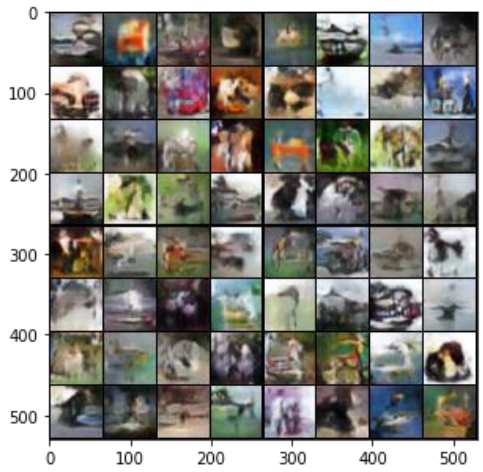


Figure 29 Generated Pictures

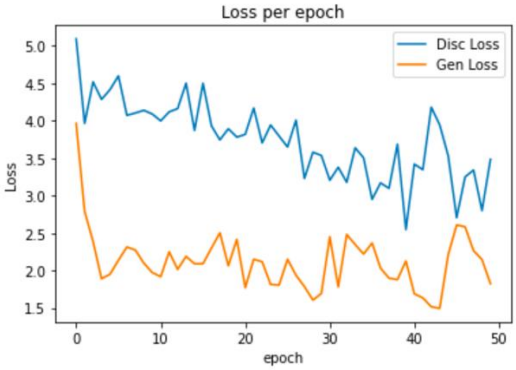


Figure 30 Loss per epoch

Part III: Image/Video Colorization

Part A: Extracting frames from Dragonball Animation

ExtractFrame.py can be found in Code/Q3/Bonus/. If you run this script, 120 frames are going to be extracted from DragonBall.mkv. The size of training and test dataset are 90 and 30 respectively.

In [ ]:
```
import cv2
vidcap = cv2.VideoCapture('DragonBall.mkv')
success,image = vidcap.read()
count = 0
frame_count = 0
while success:
    if count%200 == 0:
        if frame_count <= 90:
            cv2.imwrite("frames/train/frame%d.jpg" % frame_count, image)     # save frame as JPEG file
        else:
            cv2.imwrite("frames/test/frame%d.jpg" % frame_count, image)     # save frame as JPEG file
        frame_count += 1
    success,image = vidcap.read()
    count += 1

    if frame_count >= 121:
        break
```

In [14]:
```
import numpy as np
import PIL
# Ground Truth
train = []
test = []

for i in range(0,90):
  I = np.asarray(PIL.Image.open('frames/train/frame'+str(i)+'.jpg'))
  train.append(I)
for i in range(91,121):
  I = np.asarray(PIL.Image.open('frames/test/frame'+str(i)+'.jpg'))
  test.append(I)

train = np.array(train)
test = np.array(test)
```

Part B: Preprocessing

For extracting line-arts, Canny Edge Detection with Gaussian Filter is used.

In [11]:
```
import cv2 as cv
from matplotlib import pyplot as plt

train_lineart = []
test_lineart = []

for i in range(0,90):
  I = cv.imread('frames/train/frame'+str(i)+'.jpg')
  edges = cv.Canny(I,100,200)
  train_lineart.append(edges)

for i in range(91,121):
  I = cv.imread('frames/train/frame'+str(i)+'.jpg')
  edges = cv.Canny(I,100,200)
  test_lineart.append(edges)

train_lineart = np.array(train_lineart)
test_lineart = np.array(test_lineart)
```

Part C: Colorization

In this part, we're going to use 3 loss functions:

1. **Adversarial Loss:** The standard GAN loss function, also known as the min-max loss, was first described in a 2014 paper by Ian Goodfellow et al., titled "Generative Adversarial Networks".

$$E_x[log(D(x))] + E_z[log(1 - D(G(z)))]$$

2. **Style Loss:**

$$L_{style} = \sum_{l} \sum_{i,j} (\ G_{i,j}^{s,l} - \ G_{i,j}^{p,l})^2$$

$G_{i,j}^{s,l}$ is the Gram Matrix of the style image

$G_{i,j}^{p,l}$ is the Gram Matrix of the generated image

3. **Perceptual loss:** Perceptual loss functions are used when comparing two different images that look similar, like the same photo but shifted by one pixel. The function is used to compare high level differences, like content and style discrepancies, between images. A perceptual loss function is very similar to the per-pixel loss function, as both are used for training feed-forward neural networks for image transformation tasks. The perceptual loss function is a more commonly used component as it often provides more accurate results regarding style transfer.

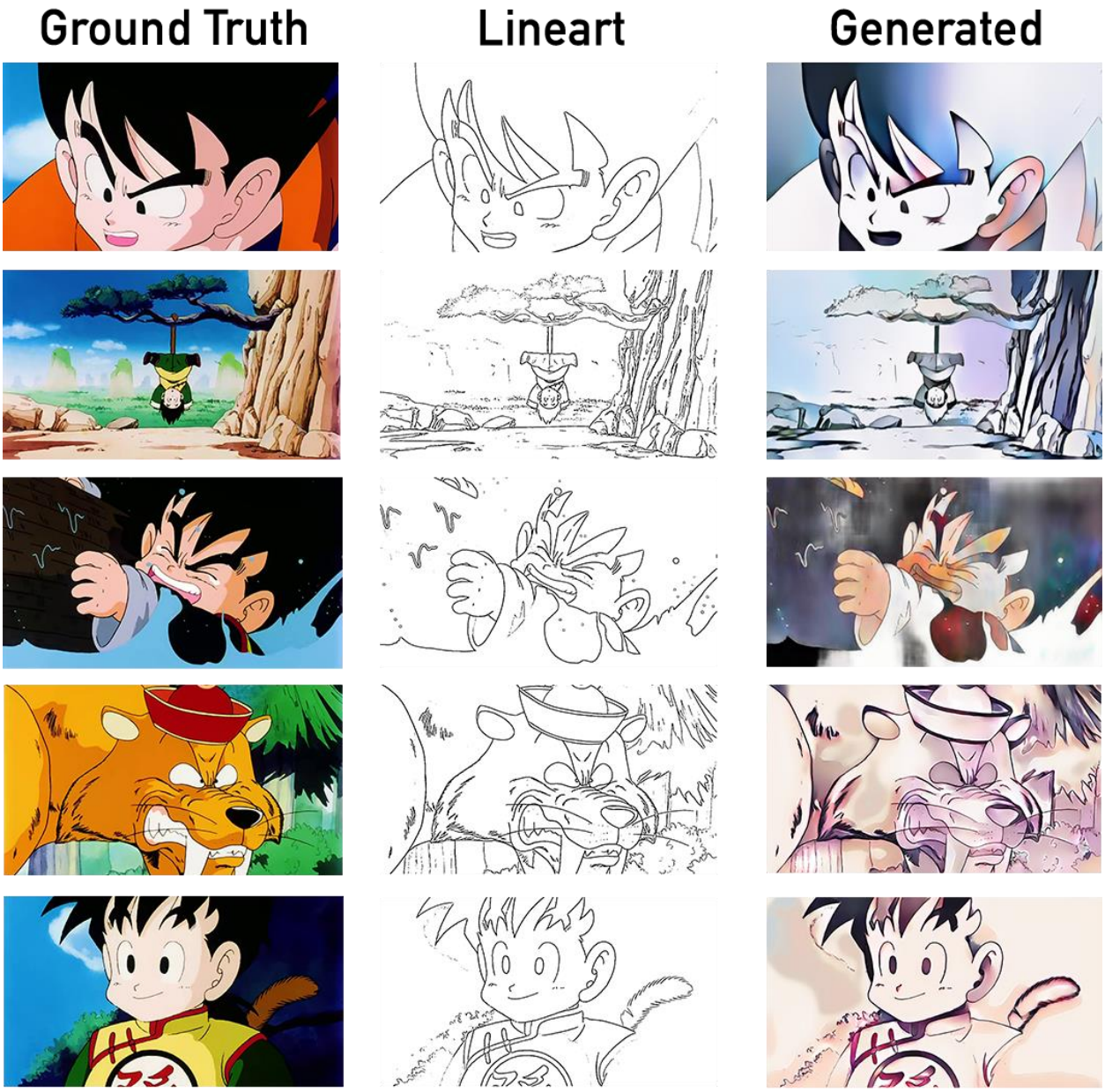


Figure 31 The colorized frames by our model