



دانشگاه تهران  
دانشکده مهندسی برق و  
کامپیوتر



درس شبکه‌های عصبی و یادگیری عمیق  
تمرین ششم

متین بذرافshan - 810100093  
شهریار عطار - 810100186

## فهرست

1.....	<b>Variational Auto-Encoder . 1 پرسش</b>
1.....	1-1. پیشپردازش دیتاست .....
2.....	شکل 1-1. تصاویر نمونه‌ای از دو دیتاست .....
3.....	3-1. ساخت VAE روی دیتاست‌ها .....
13.....	شکل 1-2. نمونه تولیدی مدل برای Anime Faces با نویز گاووسی .....
13.....	شکل 1-3. نمونه تولیدی مدل برای Anime Faces با نویز نمایی .....
14.....	شکل 1-4. نمونه تولیدی مدل برای Cartoon Faces با نویز گاووسی .....
14.....	شکل 1-5. نمونه تولیدی مدل برای Cartoon Faces با نویز نمایی .....
15.....	3-1. استفاده از یک مدل برای دو دیتاست .....
19.....	..... VQ-VAE . 4-1
24.....	..... VQ-VAE 2 . 5-1
25.....	<b>Image Translation - 2 پرسش</b>
25.....	1-2. آشنایی با Pix2Pix و معماری Image Translation .....
26.....	شکل 2-1. نحوه کار Discriminator .....
26.....	شکل 2-2. تفاوت U-Net و Encoder-Decoder ساده .....
28.....	لایه‌های مورد استفاده .....
29.....	شرح بلوک‌ها .....
29.....	شکل 2-3. معماری U-Net .....
32.....	2-2. پیاده‌سازی معماری Pix2Pix .....
33.....	شکل 2-4. چند نمونه تصویر از دیتاست .....
36.....	شکل 2-5. خروجی شبکه در ابتدای کار .....
37.....	شکل 2-6. خروجی شبکه در حالت عکس ماهواره‌ای به عنوان هدف .....
38.....	شکل 2-7. توابع هزینه در حالت عکس ماهواره‌ای به عنوان هدف .....
39.....	شکل 2-8. خروجی شبکه در حالت نقشه به عنوان هدف .....
40.....	شکل 2-9. مقادیر توابع هزینه در حالت نقشه به عنوان هدف .....

## پرسش 1. Variational Auto-Encoder .1

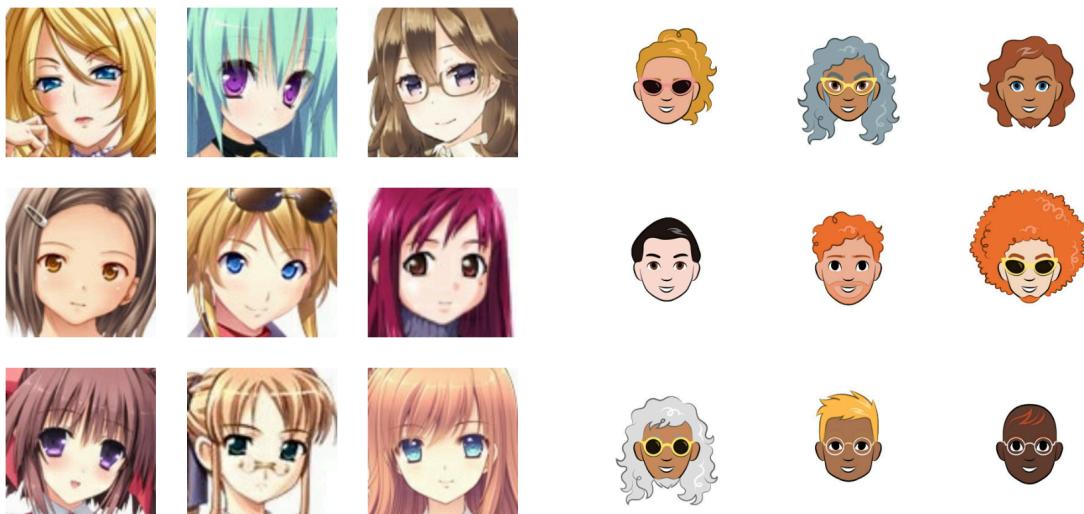
### 1-1. پیش‌پردازش دیتاست

ابتدا برای لود کردن دیتاست به شکل زیر عمل کردیم:

```
anime_train_ds =  
    tf.keras.preprocessing.image_dataset_from_directory(  
        "../data/anime_face/images",  
        image_size=(img_height, img_width),  
        batch_size=batch_size,  
        label_mode=None,  
    )
```

```
cartoon_train_ds = None  
  
for i in range(5):  
    cartoon_ds =  
        tf.keras.preprocessing.image_dataset_from_directory(  
            f"../data/cartoonset100k_jpg/{i}",  
            image_size=(img_height, img_width),  
            batch_size=batch_size,  
            label_mode=None,  
        )  
    if cartoon_train_ds is None:  
        cartoon_train_ds = cartoon_ds  
    else:  
        cartoon_train_ds = cartoon_train_ds.concatenate(cartoon_ds)
```

حال تعدادی از نمونه عکس‌های موجود در دیتاست را بررسی می‌کنیم:



شکل ۱-۱. تصاویر نمونه‌ای از دو دیتاست

سپس به پیش‌پردازش می‌پردازیم. در این بخش ما صرفا از یک لایه normalizer برای درست کردن محدوده پیکسل‌ها استفاده کردیم و تغییر سایز در هنگام لود شدن دیتاست (بخش بالا) انجام شده است و پیش‌پردازش دیگری انجام نشده.

```
normalization_layer = layers.Rescaling(scale=1.0 / 255)

anime_normalized_ds = anime_train_ds.map(lambda x:
normalization_layer(x))
cartoon_normalized_ds = cartoon_train_ds.map(lambda x:
normalization_layer(x))
```

## 2-1. ساخت VAE روی دیتاستها

:Variational Autoencoder نحوه کارکرد مدل

VAE یک نوع خاص از شبکه‌های عصبی است که برای تولید داده‌های جدید مشابه با داده‌های ورودی استفاده می‌شود. VAE یک مدل مولد است که با استفاده از مفاهیم احتمالاتی، داده‌ها را فشرده‌سازی و سپس بازسازی می‌کند.

:VAE ساختار کلی

VAE از دو بخش اصلی تشکیل شده است:

Encoder - داده‌های ورودی را به یک فضای فشرده (latent space) نگاشت می‌کند.  
Decoder - داده‌های فشرده را بازسازی می‌کند تا شبیه داده‌های ورودی اولیه باشد.

:VAE فرآیند کارکرد

Encoding : مرحله 1

1. ورودی داده‌ها: داده‌های ورودی (مثل تصاویر) به مدل وارد می‌شوند.
2. نگاشت به فضای نهفته: Encoder داده‌ها را به یک فضای فشرده (latent space) نگاشت می‌کند. این نگاشت به جای یک نقطه دقیق، یک توزیع احتمالاتی (معمولًا گاوی) تولید می‌کند که با پارامترهای میانگین ( $\mu$ ) و انحراف معیار ( $\sigma$ ) توصیف می‌شود.

Sampling : مرحله 2

1. توزیع گاوی: از توزیع گاوی با پارامترهای میانگین و انحراف معیار نمونه‌گیری می‌شود تا یک نقطه در فضای نهفته بدست آید.

Decoding : مرحله 3

1. بازسازی داده‌ها: نقطه نمونه‌گیری شده از فضای نهفته به Decoder داده می‌شود تا داده‌های ورودی بازسازی شوند.

#### 4. تابع هزینه VAE

VAE از دو بخش تشکیل شده است:

- تابع هزینه بازسازی: اختلاف بین داده‌های ورودی و داده‌های بازسازی شده توسط Decoder را محاسبه می‌کند.
  - تابع هزینه کلدبک-لیبلر (KL Divergence): اختلاف بین توزیع نهفته تولید شده توسط Encoder و یک توزیع گاوی استاندارد را اندازه‌گیری می‌کند. این تابع هزینه باعث می‌شود تا فضای نهفته یک ساختار منظم داشته باشد.
- تابع هزینه کلی مدل جمع این دو تابع هزینه می‌باشد.

حال مدل مورد نظر که ساختاری بر پایه Convolution دارد را طراحی می‌کنیم. ابتدا بخش که وظیفه dimensionality reduction encoder و ساخت میانگین و واریانس مورد نیاز برای توزیع گاوی را دارد می‌سازیم:

```
def encoder(input_encoder):  
    inputs = Input(shape=input_encoder, name="input_layer")  
  
    x = Conv2D(128, kernel_size=3, strides=2, padding="same",  
name="conv_1")(inputs)  
    x = BatchNormalization(name="bn_1")(x)  
    x = LeakyReLU(name="lrelu_1")(x)  
  
    x = Conv2D(128, kernel_size=3, strides=2, padding="same",  
name="conv_2")(x)  
    x = BatchNormalization(name="bn_2")(x)  
    x = LeakyReLU(name="lrelu_2")(x)  
    x = Dropout(0.2, name="dropout_1")(x)  
  
    x = Conv2D(64, kernel_size=3, strides=2, padding="same",  
name="conv_3")(x)  
    x = BatchNormalization(name="bn_3")(x)  
    x = LeakyReLU(name="lrelu_3")(x)  
    x = Dropout(0.2, name="dropout_2")(x)  
  
    x = Conv2D(32, kernel_size=3, strides=2, padding="same",  
name="conv_4")(x)  
    x = BatchNormalization(name="bn_4")(x)  
    x = LeakyReLU(name="lrelu_4")(x)
```

```

x = Dropout(0.2, name="dropout_3")(x)

flatten = Flatten()(x)

mean = Dense(200, name="mean")(flatten)
log_var = Dense(200, name="log_var")(flatten)

model = Model(inputs, (mean, log_var), name="Encoder")

return model

```

که اگر به طور خلاصه معماری را بخواهیم بررسی کنیم به شکل زیر خواهد بود:

Model: "Encoder"

---

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
<hr/>			
input_layer (InputLayer)	[(None, 256, 256, 3 0 )]	[ ]	
conv_1 (Conv2D) ['input_layer[0][0]']	(None, 128, 128, 12 3584 8)		
bn_1 (BatchNormalization) ['conv_1[0][0]']	(None, 128, 128, 12 512 8)		
lrelu_1 (LeakyReLU) ['bn_1[0][0]']	(None, 128, 128, 12 0 8)		
conv_2 (Conv2D)	(None, 64, 64, 128) 147584		

```

['lrelu_1[0][0]']

bn_2 (BatchNormalization)      (None, 64, 64, 128)  512
['conv_2[0][0]']

lrelu_2 (LeakyReLU)          (None, 64, 64, 128)  0
['bn_2[0][0]']

dropout_1 (Dropout)          (None, 64, 64, 128)  0
['lrelu_2[0][0]']

conv_3 (Conv2D)              (None, 32, 32, 64)   73792
['dropout_1[0][0]']

bn_3 (BatchNormalization)    (None, 32, 32, 64)   256
['conv_3[0][0]']

lrelu_3 (LeakyReLU)          (None, 32, 32, 64)  0
['bn_3[0][0]']

dropout_2 (Dropout)          (None, 32, 32, 64)  0
['lrelu_3[0][0]']

conv_4 (Conv2D)              (None, 16, 16, 32)   18464
['dropout_2[0][0]']

bn_4 (BatchNormalization)    (None, 16, 16, 32)   128
['conv_4[0][0]']

lrelu_4 (LeakyReLU)          (None, 16, 16, 32)  0
['bn_4[0][0]']

dropout_3 (Dropout)          (None, 16, 16, 32)  0
['lrelu_4[0][0]']

flatten_2 (Flatten)          (None, 8192)        0
['dropout_3[0][0]']

```

```

mean (Dense)           (None, 200)      1638600
['flatten_2[0][0]']

log_var (Dense)        (None, 200)      1638600
['flatten_2[0][0]']

=====
=====
Total params: 3,522,032
Trainable params: 3,521,328
Non-trainable params: 704

```

حال برای ساخت بخش sampling که باید با کمک میانگین و واریانس بدست آمده خروجی را تعیین کند را نیز به شکل زیر تعریف کردیم:

```

def sampling_model(distribution_params):
    mean, log_var = distribution_params
    epsilon = K.random_normal(shape=K.shape(mean), mean=0.0,
stddev=1.0)
    return mean + K.exp(log_var / 2) * epsilon

def sampling(input_1, input_2):
    mean = Input(shape=input_1, name="input_layer1")
    log_var = Input(shape=input_2, name="input_layer2")
    out = layers.Lambda(sampling_model,
name="encoder_output")([mean, log_var])
    enc_2 = tf.keras.Model([mean, log_var], out,
name="Encoder_2")
    return enc_2

```

در انتها نیز برای ساخت بخش reconstruct که وظیفه decoder کردن ورودی را دارد را طراحی کردیم، باید توجه کرد که معماری این بخش باید معکوس معماری بخش encoder باشد برای همین از لایه‌هایی مانند Conv2DTranspose استفاده شده.

```

def decoder(input_decoder):
    inputs = Input(shape=input_decoder, name="input_layer")

```

```

x = Dense(4096, name="dense_1")(inputs)
x = Reshape((8, 8, 64), name="Reshape")(x)

x = Conv2DTranspose(32, kernel_size=3, strides=2,
padding="same", name="conv_transpose_1")(x)
x = BatchNormalization(name="bn_1")(x)
x = LeakyReLU(name="lrelu_1")(x)

x = Conv2DTranspose(64, kernel_size=3, strides=2,
padding="same", name="conv_transpose_2")(x)
x = BatchNormalization(name="bn_2")(x)
x = LeakyReLU(name="lrelu_2")(x)
x = Dropout(0.2, name="dropout_1")(x)

x = Conv2DTranspose(128, kernel_size=3, strides=2,
padding="same", name="conv_transpose_3")(x)
x = BatchNormalization(name="bn_3")(x)
x = LeakyReLU(name="lrelu_3")(x)
x = Dropout(0.2, name="dropout_2")(x)

x = Conv2DTranspose(128, kernel_size=3, strides=2,
padding="same", name="conv_transpose_4")(x)
x = BatchNormalization(name="bn_4")(x)
x = LeakyReLU(name="lrelu_4")(x)
x = Dropout(0.2, name="dropout_3")(x)

output = Conv2DTranspose(3, kernel_size=3, strides=2,
padding="same", activation="sigmoid",
name="conv_transpose_5")(x)

model = Model(inputs, output, name="Decoder")

return model

```

Model: "Decoder"

Layer (type)	Output Shape	Param #
<hr/>		
input_layer (InputLayer)	[(None, 200)]	0
dense_1 (Dense)	(None, 4096)	823296

Reshape (Reshape)	(None, 8, 8, 64)	0
conv_transpose_1 (Conv2DTranspose)	(None, 16, 16, 32)	18464
bn_1 (BatchNormalization)	(None, 16, 16, 32)	128
lrelu_1 (LeakyReLU)	(None, 16, 16, 32)	0
conv_transpose_2 (Conv2DTranspose)	(None, 32, 32, 64)	18496
bn_2 (BatchNormalization)	(None, 32, 32, 64)	256
lrelu_2 (LeakyReLU)	(None, 32, 32, 64)	0
dropout_1 (Dropout)	(None, 32, 32, 64)	0
conv_transpose_3 (Conv2DTranspose)	(None, 64, 64, 128)	73856
bn_3 (BatchNormalization)	(None, 64, 64, 128)	512
lrelu_3 (LeakyReLU)	(None, 64, 64, 128)	0
dropout_2 (Dropout)	(None, 64, 64, 128)	0
conv_transpose_4 (Conv2DTranspose)	(None, 128, 128, 128)	147584
bn_4 (BatchNormalization)	(None, 128, 128, 128)	512
lrelu_4 (LeakyReLU)	(None, 128, 128, 128)	0
dropout_3 (Dropout)	(None, 128, 128, 128)	0
conv_transpose_5 (Conv2DTranspose)	(None, 256, 256, 3)	3459

=====

Total params: 1,086,563

```
Trainable params: 1,085,859  
Non-trainable params: 704
```

---

پس از آن توابع هزینه و optimizer را تعریف کردیم که از فرمول و روش‌های معمول در این بخش استفاده شده.

```
optimizer = Adam(learning_rate=0.0005)

def mse_loss(y_true, y_pred):
    r_loss = K.mean(K.square(y_true - y_pred), axis=[1, 2, 3])
    return 1000 * r_loss

def kl_loss(mean, log_var):
    kl_loss_ = -0.5 * K.sum(1 + log_var - K.square(mean) -
K.exp(log_var), axis=1)
    return kl_loss_

def vae_loss(y_true, y_pred, mean, log_var):
    r_loss = mse_loss(y_true, y_pred)
    kl_loss_ = kl_loss(mean, log_var)
    return r_loss + kl_loss_
```

پس هر مرحله در آموزش مدل به شکل زیر خواهد بود:

```
@tf.function
def train_step(images):

    with tf.GradientTape() as encoder, tf.GradientTape() as
decoder:

        mean, log_var = enc(images, training=True)
        z = sample([mean, log_var])
        generated_images = dec(z, training=True)

        reconstruct_loss = mse_loss(images, generated_images)
        loss = vae_loss(images, generated_images, mean, log_var)
```

```

        gradients_of_enc = encoder.gradient(loss,
enc.trainable_variables)
        gradients_of_dec = decoder.gradient(loss,
dec.trainable_variables)

        optimizer.apply_gradients(zip(gradients_of_enc,
enc.trainable_variables))
        optimizer.apply_gradients(zip(gradients_of_dec,
dec.trainable_variables))

    return reconstruct_loss, loss

```

به این دلیل از `tf.function` نیز استفاده شده که تابع را کامپایل کند و توسط Tensorflow Graph قابل صدا زدن باشد.

سپس مدل را به شکل زیر آموزش دادیم:

```

def train(dataset, epochs, name):
    for epoch in range(epochs):
        start = time.time()
        losses = []
        reconstruct_losses = []

        for image_batch in dataset:
            reconstruct_loss, loss = train_step(image_batch)

            loss = np.mean(loss.numpy())
            reconstruct_loss = np.mean(reconstruct_loss.numpy())

            losses.append(loss)
            reconstruct_losses.append(reconstruct_loss)

        seed = image_batch[:25]
        save_generated_images([enc, sample, dec], epoch + 1, seed,
name)

        print("Epoch: ", epoch + 1, " Loss: ", losses[-1], "
Reconstruct Loss: ", reconstruct_losses[-1], " Time: ",
time.time() - start)

```

```

    save_generated_images([enc, sample, dec], epochs, seed,
name)

    enc.save_weights(f"../models/{name}/enc.h5")
    dec.save_weights(f"../models/{name}/dec.h5")

```

که دارای نتایج زیر بود: (ابتدا برای دیتاست Anime Face و سپس برای دیتاست  
(Cartoon100K

```

Epoch: 1 Loss: 45.113102 Reconstruct Loss: 37.526035 Time:
375.5771789550781
Epoch: 2 Loss: 42.10288 Reconstruct Loss: 35.256493 Time:
388.4650020599365
Epoch: 3 Loss: 40.888332 Reconstruct Loss: 33.999104 Time:
391.7428877353668
Epoch: 4 Loss: 42.977787 Reconstruct Loss: 36.14412 Time:
383.18104577064514
Epoch: 5 Loss: 43.67771 Reconstruct Loss: 36.325558 Time:
369.0791726112366

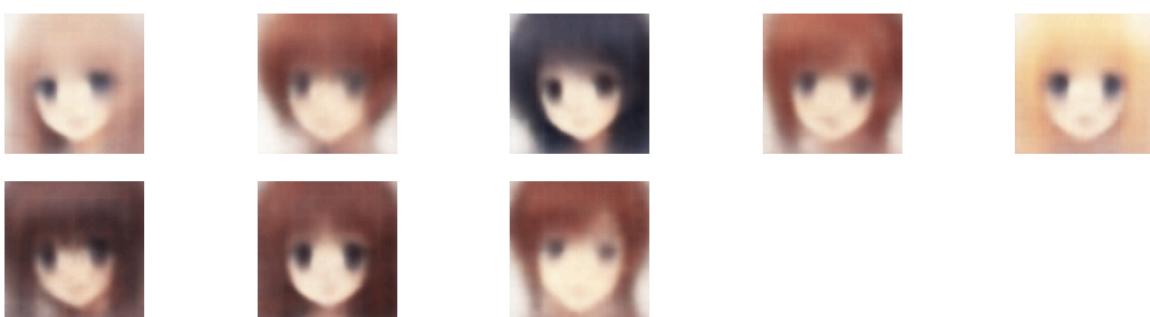
```

```

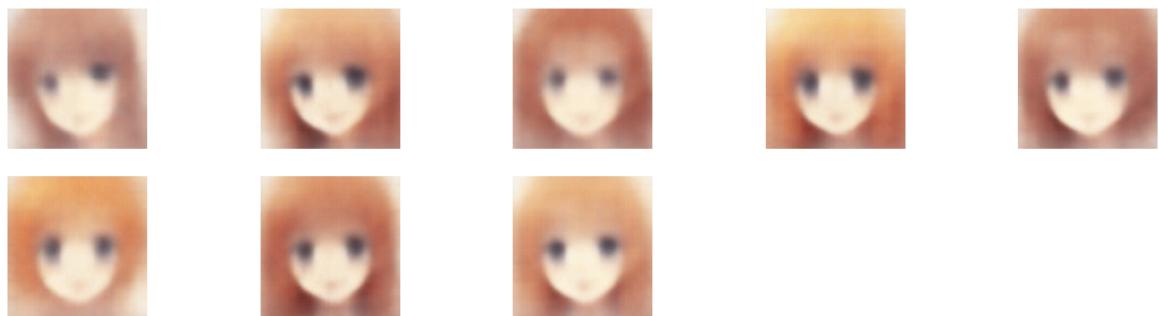
Epoch: 1 Loss: 22.641958 Reconstruct Loss: 17.313942 Time:
300.17361307144165
Epoch: 2 Loss: 20.999107 Reconstruct Loss: 16.009174 Time:
306.5335018634796
Epoch: 3 Loss: 20.68335 Reconstruct Loss: 15.09233 Time:
310.165322303772
Epoch: 4 Loss: 17.218472 Reconstruct Loss: 11.97661 Time:
320.37937593460083
Epoch: 5 Loss: 16.53772 Reconstruct Loss: 11.898354 Time:
327.02315759658813

```

سپس با کمک دو نوع نویز ساخته شده به ترتیب با توزیع نرمال و توزیع نمایی تعدادی عکس جدید ساختیم که خروجی به شرح زیر بود: (خروجی نویزها به عنوان ورودی به decoder داده شده و عکس‌های زیر ساخته شده)



شکل 1-2. نمونه تولیدی مدل برای Anime Faces با نویز گاوی



شکل 1-3. نمونه تولیدی مدل برای Anime Faces با نویز نمایی



شکل ۱-۴. نمونه تولیدی مدل برای Cartoon Faces با نویز گاووسی



شکل ۱-۵. نمونه تولیدی مدل برای Cartoon Faces با نویز نمایی

### 3-1. استفاده از یک مدل برای دو دیتاست

:CVAE (Conditional Variational Autoencoder)

CVAE نیز یک مدل مولد است اما هدف آن تولید نمونه‌های جدید با توجه به شرایط خاص (شرط) است.

ساختار CVAE مشابه VAE است، اما با این تفاوت که علاوه بر داده‌های ورودی، یک شرط (label) یا هر اطلاعات اضافی دیگر) نیز به encoder و decoder داده می‌شود.

این مدل برای کاربردهایی مثل تولید تصاویر با ویژگی‌های خاص یا ترجمه متون با شرایط خاص کاربرد دارد.

:latent space

در CVAE latent space با توجه به شرط موجود یادگیری می‌شود. به عبارت دیگر، پارامترهای توزیع گاووسی encoder علاوه بر داده‌های ورودی، به شرط نیز وابسته هستند.

در فرآیند آموزش، encoder سعی می‌کند پارامترهای توزیع گاووسی را با در نظر گرفتن هر دو ورودی و شرط یاد بگیرد.

تفاوت‌های کلیدی این مدل با مدل VAE خالی در وجود شرط است:

در VAE هیچ شرط اضافی وجود ندارد و تنها داده‌های ورودی به مدل داده می‌شوند.

در CVAE علاوه بر داده‌های ورودی، یک شرط اضافی نیز به مدل داده می‌شود که باعث می‌شود مدل بتواند نمونه‌های جدید با توجه به شرایط خاص تولید کند.

پس در این بخش باید یک مدل واحد بسازیم که با کمک آن بتوانیم با توجه به لیبل یک نمونه از هر دو کلاس مورد نظر بسازیم. برای این کار به مراحل زیر را پیش بردیم:

ابتدا دیتاست‌ها را با یکدیگر ترکیب کردیم و به آن‌ها لیبل زدیم

```
anime_label = to_categorical([0] * batch_size, num_classes=2)
cartoon_label = to_categorical([1] * batch_size, num_classes=2)
anime_train_ds = anime_train_ds.map(lambda x: (x, anime_label))
cartoon_train_ds = cartoon_train_ds.map(lambda x: (x,
cartoon_label))
combined_ds =
tf.data.Dataset.sample_from_datasets([anime_train_ds,
cartoon_train_ds], [0.5, 0.5])
```

سپس encoder و decoder را به نحوی تعریف کردیم که بتواند از لیبل‌ها برای ساخت در واقع مدل به صورت شرطی با توجه به ورودی خروجی مورد نظر را می‌سازد.

```
def encoder(input_shape, label_dim, filters, latent_dim):
    x = Input(shape=input_shape)
    y = Input(shape=(label_dim,))

    y_reshaped = Dense(input_shape[0] * input_shape[1] *
input_shape[2])(y)
    y_reshaped = Reshape((input_shape[0], input_shape[1],
input_shape[2]))(y_reshaped)

    inputs = Concatenate(axis=-1)([x, y_reshaped])

    conv1 = Conv2D(filters, kernel_size=3, strides=2,
activation="relu", padding="same")(inputs)
    conv2 = Conv2D(filters * 2, kernel_size=3, strides=2,
activation="relu", padding="same")(conv1)
    conv3 = Conv2D(filters * 4, kernel_size=3, strides=2,
activation="relu", padding="same")(conv2)
    conv4 = Conv2D(filters * 8, kernel_size=3, strides=2,
activation="relu", padding="same")(conv3)

    flattened = Flatten()(conv4)
    z_mean = Dense(latent_dim)(flattened)
    z_log_var = Dense(latent_dim)(flattened)

    return Model([x, y], [z_mean, z_log_var], name="encoder")
```

```
def decoder(input_shape, label_dim, filters, latent_dim):
    z = Input(shape=(latent_dim,))
    y = Input(shape=(label_dim,))

    inputs = Concatenate()([z, y])

    hidden = Dense(16 * 16 * filters * 8,
activation="relu")(inputs)
    reshaped = Reshape((16, 16, filters * 8))(hidden)

    deconv1 = Conv2DTranspose(filters * 8, kernel_size=3,
strides=2, activation="relu", padding="same")(reshaped)
```

```

        deconv2 = Conv2DTranspose(filters * 4, kernel_size=3,
strides=2, activation="relu", padding="same")(deconv1)
        deconv3 = Conv2DTranspose(filters * 2, kernel_size=3,
strides=2, activation="relu", padding="same")(deconv2)
        deconv4 = Conv2DTranspose(filters, kernel_size=3,
strides=2, activation="relu", padding="same")(deconv3)
        x_decoded = Conv2DTranspose(3, kernel_size=3,
activation="sigmoid", padding="same")(deconv4)

    return Model([z, y], x_decoded, name="decoder")

```

برای نیز از این تابع جدید استفاده کردیم:

```

def sampling(args):
    z_mean, z_log_var = args
    epsilon = tf.random.normal(shape=(tf.shape(z_mean)[0],
latent_dim))
    return z_mean + tf.exp(0.5 * z_log_var) * epsilon

```

سپس مدل CVAE را به این شکل ساختیم که بتوانیم با کمک آن نمونه مناسب برای هر دو دیتاست بسازیم

```

enc = encoder((img_height, img_width, 3), label_dim, filters,
latent_dim)
dec = decoder((img_height, img_width, 3), label_dim, filters,
latent_dim)

x = Input(shape=(img_height, img_width, 3))
y = Input(shape=(label_dim,))
z_mean, z_log_var = enc([x, y])
z = layers.Lambda(sampling)([z_mean, z_log_var])
x_decoded = dec([z, y])

cvae = Model([x, y], x_decoded, name="vae")
reconstruction_loss = MeanSquaredError()(x, x_decoded)
kl_loss = -0.5 * tf.reduce_mean(1 + z_log_var -
tf.square(z_mean) - tf.exp(z_log_var))
cvae_loss = tf.reduce_mean(reconstruction_loss + kl_loss)

cvae.add_loss(cvae_loss)

```

```
cvae.compile(optimizer=Adam(learning_rate=0.001))
```

سپس مدل را آموزش دادیم که نتایج به شکل زیر بود:

این بخش به دلیل عدم زمان کافی برای دیباگ کردن و آموزش مدل انجام نشده.

## VQ-VAE .4-1

مدل VQ-VAE که توسط DeepMind معرفی شده است، یک نوع مدل تولیدی است که برای یادگیری نمایش‌های گستته از داده طراحی شده است. این مدل به برطرف کردن محدودیت‌هایی که در چارچوب واریانسی‌ای برداری (VAE) سنتی وجود دارد، مانند "فروپاشی پسین"، با استفاده از کوانتیزاسیون برداری پاسخ می‌دهد.

### اجزاء کلیدی:

1. شبکه کدگذار: خروجی‌های گستته را به جای متغیرهای پیوسته تولید می‌کند.
2. کوانتیزاسیون برداری (VQ): خروجی‌های پیوسته کدگذار را به کدهای گستته تبدیل می‌کند، با یافتن نزدیکترین همسایه در یک کتابخانه کدهای تعریف شده.
3. شبکه کدگشایی: داده ورودی را از این کدهای گستته بازسازی می‌کند.
4. پیشین یادگرفته‌شده: برخلاف VAE های با پیشینه‌های ثابت، VQ-VAE پیشینه توزیع بر روی متغیرهای گستته را یاد می‌گیرد.

### کارکرد:

1. فضای تعبیر جاسازی: مدل یک فضای تعبیر جاسازی را حفظ می‌کند که شامل  $K$  بردار جاسازی است، هر کدام با ابعاد  $D$ .
2. کدگذاری: کدگذار ورودی  $x$  را پردازش می‌کند تا  $(x)_z e$  تولید کند.
3. کوانتیزاسیون برداری: خروجی کدگذار  $(x)_z e$  به بردار جاسازی نزدیکترین  $e_k$  از فضای جاسازی تبدیل می‌شود با استفاده از جستجوی همسایگی نزدیکترین.
4. کدگشا: بردار جاسازی انتخاب شده  $e_k$  سپس به کدگشا منتقل می‌شود تا ورودی را بازسازی کند.

### :train

- تابع خطای آموزش شامل سه مولفه است:

1. خطای بازسازی: اندازه‌گیری می‌کند که چقدر کدگشا ورودی را به خوبی بازسازی می‌کند.
2. خطای جاسازی: اطمینان حاصل می‌شود که بردارهای جاسازی به خروجی‌های کدگذار نزدیک باشند و به روزرسانی شوند.
3. خطای تعهد: جلوگیری می‌کند که خروجی کدگذار به اندازه بزرگ شدن دست بزند، با اطمینان از این که به یک بردار جاسازی خاص تعهد کند.

## تفاوت‌ها با VAE

1. متغیرهای گسسته در مقابل پیوسته: VAE های سنتی از متغیرهای پیوسته استفاده می‌کنند، در حالی که VQ-VAE از متغیرهای گسسته استفاده می‌کند.
2. فروپاشی پسین: VAE ها اغلب با "فروپاشی پسین" مواجه می‌شوند که در آن متغیرهای گسسته نادیده گرفته می‌شوند. این مشکل را با استفاده از نمایش‌های گسسته و کوانتیزاسیون برداری رفع می‌کند.
3. پیشین یادگرفته: در VAE های سنتی، پیشینه اغلب یک توزیع ثابت است (مانند یک گاوی). در VQ-VAE، پیشینه یادگرفته می‌شود که امکان انعطاف‌پذیری بیشتر و بهتر مدل کردن توزیع داده را فراهم می‌کند.

حال که با مدل آشنای شدیم بپیاده‌سازی آن می‌پردازیم. برای خوانایی بیشتر از یک تابع برای ساخت تعدادی لایه convolution پشت سر هم استفاده کردیم که به شکل زیر هست:

```
def residual_stack(h, num_hiddens, num_residual_layers,
num_residual_hiddens):  
  
    for i in range(num_residual_layers):  
        h_i = tf.nn.relu(h)  
  
        h_i = Conv2D(  
            output_channels=num_residual_hiddens,  
            kernel_shape=(3, 3),  
            name="res3x3_%d" % i,  
        )(h_i)  
        h_i = tf.nn.relu(h_i)  
  
        h_i = Conv2D(  
            output_channels=num_hiddens,  
            kernel_shape=(1, 1),  
            name="res1x1_%d" % i,  
        )(h_i)  
  
        h += h_i  
  
    return tf.nn.relu(h)
```

سپس دو بخش encoder و decoder را می‌سازیم:

```

class Encoder(Model):
    def __init__(self, num_hiddens, num_residual_layers,
num_residual_hiddens, name="encoder"):
        super(Encoder, self).__init__(name=name)
        self._num_hiddens = num_hiddens
        self._num_residual_layers = num_residual_layers
        self._num_residual_hiddens = num_residual_hiddens

    def _build(self, x):

        h = Conv2D(
            output_channels=self._num_hiddens / 2,
            kernel_shape=(4, 4),
            stride=(2, 2),
        )(x)
        h = relu(h)

        h = Conv2D(
            output_channels=self._num_hiddens,
            kernel_shape=(4, 4),
            stride=(2, 2),
        )(h)
        h = relu(h)

        h = Conv2D(
            output_channels=self._num_hiddens,
            kernel_shape=(3, 3),
            stride=(1, 1),
        )(h)

        h = residual_stack(
            h, self._num_hiddens, self._num_residual_layers,
            self._num_residual_hiddens
        )

    return h

```

```

class Decoder(Model):
    def __init__(self, num_hiddens, num_residual_layers,
num_residual_hiddens, name="decoder"):
        super(Decoder, self).__init__(name=name)
        self._num_hiddens = num_hiddens
        self._num_residual_layers = num_residual_layers
        self._num_residual_hiddens = num_residual_hiddens

    def _build(self, x):
        h = Conv2D(
            output_channels=self._num_hiddens,
            kernel_shape=(3, 3),
        )(x)

        h = residual_stack(
            h, self._num_hiddens, self._num_residual_layers,
            self._num_residual_hiddens
        )

        h = Conv2DTranspose(
            output_channels=int(self._num_hiddens / 2),
            output_shape=None,
            kernel_shape=(4, 4),
            stride=(2, 2),
        )(h)
        h = relu(h)

        x_recon = Conv2DTranspose(
            output_channels=3,
            output_shape=None,
            kernel_shape=(3, 3),
        )(h)

    return x_recon

```

تفاوت پیاده‌سازی این دو بخش در این است که در این مدل خروجی encoder به صورت گسسته است. سپس برای پیش‌پردازش داده‌ها به شکل زیر عمل کردیم:

```

def cast_and_normalise_images(data):
    images = data
    data = (tf.cast(images, tf.float32) / 255.0)
    return data

train_dataset_iterator = (
    tf.data.Dataset.from_tensor_slices(anime_train_ds)
    .map(cast_and_normalise_images)
    .shuffle(10000)
    .repeat(-1)
    .batch(batch_size)).make_one_shot_iterator()

train_dataset_batch = train_dataset_iterator.get_next()

valid_dataset_iterator = (
    tf.data.Dataset.from_tensor_slices(anime_val_ds)
    .map(cast_and_normalise_images)
    .repeat(1)
    .batch(batch_size)).make_initializable_iterator()

valid_dataset_batch = valid_dataset_iterator.get_next()

```

سپس مدل نهایی را به با کمک کتابخانه sonnet به شکل زیر تعریف کردیم:

```

encoder = Encoder(num_hiddens, num_residual_layers,
num_residual_hiddens)
decoder = Decoder(num_hiddens, num_residual_layers,
num_residual_hiddens)
pre_vq_conv1 = Conv2D(output_channels=embedding_dim,
kernel_shape=(1, 1), stride=(1, 1), name="to_vq")
vq_vae = snt.nets.VectorQuantizer(
    embedding_dim=embedding_dim,
    num_embeddings=num_embeddings,
    commitment_cost=commitment_cost,
)

```

پس در نهایت به شکل زیر مدل را آموزش دادیم و خروجی‌های زیر را گرفتیم:  
این بخش نیز به دلیل زمان کم به مرحله دیباگ و ران کردن نرسید.

## VQ-VAE 2 .5-1

VQ-VAE-2 (Vector Quantized Variational AutoEncoder-2) یک مدل نسل جدید تصاویر است که بر مبنای مدل VQ-VAE ساخته شده است. هدف اصلی این مدل، تولید تصاویر با کیفیت بالا و تنوع بیشتر است. ابتدا با اصول VQ-VAE آشنا شدیم و حال، تفاوت‌های VQ-VAE-2 با نسخه اولیه را بررسی کنیم.

مدل 2 VQ-VAE بهبودهایی بر روی مدل اصلی ارائه می‌دهد تا کیفیت تصاویر تولیدی و کارایی را افزایش دهد. این بهبودها شامل موارد زیر است:

۱. **Hierarchical Organization**: به جای استفاده از یک سطح، از چند سطح کوانتیزاسیون

استفاده می‌شود. این ساختار به مدل اجازه می‌دهد تا اطلاعات محلی مانند بافت‌ها و اطلاعات جهانی مانند شکل‌ها و هندسه‌ها را به صورت جداگانه مدل‌سازی کند.

۲. **Powerful Priors**: از مدل‌های پیش‌بینی قدرتمند مانند PixelCNN استفاده می‌شود تا کدهای نهفته را مدل‌سازی کند. این مدل‌ها می‌توانند توزیع‌های پیچیده را بهتر یاد بگیرند و تصاویر واقعی‌تری تولید کنند.

۳. سرعت بالا در نمونه‌گیری: به دلیل استفاده از فضای نهفته فشرده، نمونه‌گیری در این مدل بسیار سریع‌تر از نمونه‌گیری مستقیم از پیکسل‌ها است.

مدل VQ-VAE-2 با استفاده از بهبودهای ساختاری و مدل‌های پیش‌بینی قدرتمندتر، تصاویر با کیفیت و تنوع بیشتری تولید می‌کند. این مدل نسبت به نسخه اولیه خود سریع‌تر و کارآمدتر است و مشکلاتی مانند کمبود تنوع در تصاویر تولیدی را برطرف کرده است.

## پرسش 2 - Image Translation

### 1- آشنایی با Pix2Pix و معماری Image Translation

#### 1. تفاوت مدل Pix2Pix با یک GAN ساده

مدل Pix2Pix یک مدل Image to Image Translation می‌باشد. این مدل یک Conditional GAN است که با دریافت یک تصویر ورودی و یک نویز تلاش می‌کند تا نویز را به تصویر خروجی Map کند. برای این کار از یک U-Net استفاده می‌کند. تفاوت عمدۀ این مدل وجود یک شرط است که همان تصویر ورودی است. این شبکه تحت وجود این شرط، نگاشت تصویر به تصویر را یاد می‌گیرد. همچنین ممکن است پرسیده شود که یک encoder-decoder هم می‌تواند نگاشت را یاد بگیرد و چه نیازی به یک GAN است. پاسخ این سوال این است که با وجود discriminator شبکه تابع هزینه را نیز خودش یاد می‌گیرد.

برخی از کاربردهای این معماری در حوزه پزشکی:

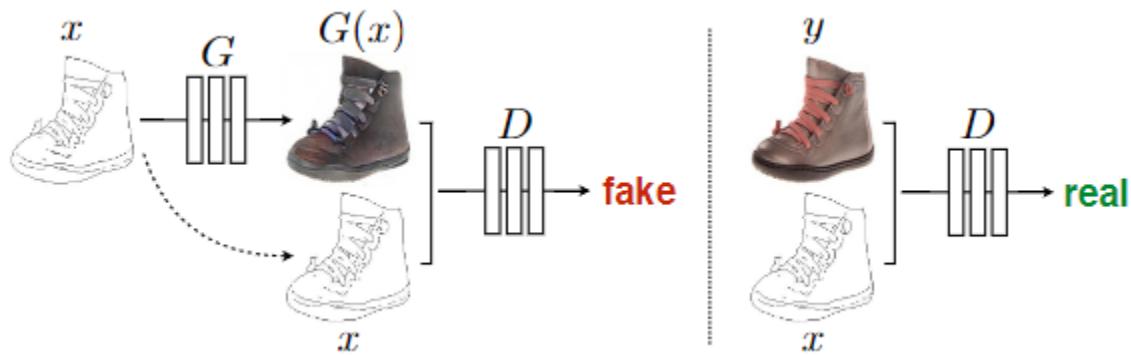
- A. تبدیل تصاویر سیاه و سفید به تصاویر رنگی
- B. تبدیل تصاویر از یک مDALIte به م DALIte دیگر

در پزشکی، مDALIte‌های تصویربرداری مختلفی مانند CT، MRI و PET وجود دارند که هر کدام اطلاعات مختلفی ارائه می‌دهند. Pix2Pix می‌تواند تصاویر یک M DALIte را به M DALIte دیگر تبدیل کند.

#### 2. تفاوت Discriminator

در GAN ساده، discriminator یک تصویر را دریافت می‌کند و احتمال واقعی بودن آن را به عنوان خروجی می‌دهد، در حالی که در معماری Pix2Pix، علاوه بر تصویر خروجی Generator یا تصویر واقعی، تصویر ورودی را نیز دریافت می‌کند. این باعث می‌شود که اطلاعات ساختاری و محتوایی را بتواند بهتر تشخیص دهد و تصویر واقعی را از جعلی بهتر تشخیص دهد. همچنین باعث حفظ اطلاعات زمینه‌ای نیز می‌شود. به عنوان مثال، اگر تصویر ورودی یک طرح خطی است، متمایزکننده می‌تواند تشخیص دهد که آیا تصویر تولید شده به درستی رنگ‌ها و بافت‌های مناسب را به این طرح افزوده است یا خیر.

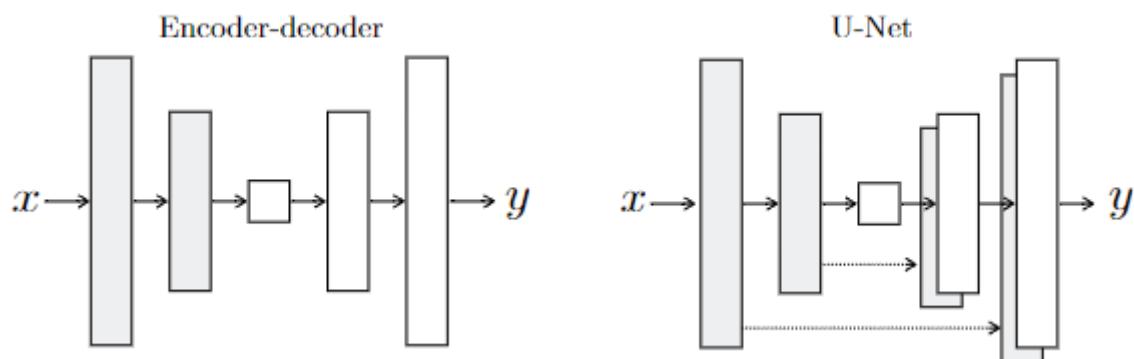
این Discriminator په جای یک خروجی بین صفر و یک که احتمال واقعی بودن را نشان می‌دهد، یک ماتریس  $N$  در  $N$  را به عنوان خروجی می‌دهد. این تابع، تصویر را به  $N \times N$  بخش (یا همان Patch) تقسیم می‌کند و برای هر Patch به طور جداگانه تصمیم می‌گیرد و در آخر میانگین آن‌ها را به طور اجمع شده خروجی می‌دهد.



شکل 2-1. نحوه کار Discriminator

### 3. Unet معماری

شبکه مولد این معماری یک U-Net می‌باشد. تفاوت اصلی یک U-Net با یک Encoder-Decoder عادی وجود اتصال‌های بین بخش‌های encoding و decoding می‌باشد که به حفظ محتوای زمینه‌ای<sup>1</sup> عکس کمک بسیاری می‌کند. در ادامه شبکه مورد استفاده به طور کامل توضیح داده شده است.



شکل 2-2. تفاوت Encoder-Decoder و U-Net ساده

<sup>1</sup> Context

## شبکه Unet

یک معماری شبکه عصبی کانولوشنی (CNN) است که برای وظایف بخش‌بندی در حوزه بینایی ماشین طراحی شده است. نام "UNet" از ساختار لـ شکل شبکه گرفته است که شامل یک مسیر رمزگذار و یک مسیر رمزگشای است. مسیر رمزگذار، متناظر با تصویر ورودی، ابعاد فضایی را کاهش داده ویژگی‌ها را از تصویر استخراج می‌کند، در حالی که مسیر رمزگشای، ابعاد فضایی را بازیابی کرده و نقشه بخش‌بندی را تولید می‌کند.

در ادامه، یک نمای کلی از معماری UNet آورده شده است:

۱. **مسیر رمزگذار<sup>۲</sup>**: مسیر رمزگذار تصویر ورودی را دریافت می‌کند و با استفاده از لایه‌های کانولوشنی و لایه‌های پولینگ، ابعاد فضایی آن را به تدریج کاهش می‌دهد. این لایه‌ها ویژگی‌های تصویر را با حفظ اطلاعات فضایی استخراج می‌کنند.

۲. **پل<sup>۳</sup>**: در پایین معماری شکل لـ، یک اتصال پل وجود دارد که ویژگی‌های با کیفیت بالا یادگرفته شده توسط رمزگذار را نگه می‌دارد. این پل به رمزگشای اجازه می‌دهد در فرایند باز نمودن، به هر دو ویژگی نقشه‌های با کیفیت پایین و با کیفیت بالا دسترسی داشته باشد.

۳. **مسیر رمزگشای<sup>۴</sup>**: مسیر رمزگشای ویژگی‌های کاهش یافته از رمزگذار را دریافت می‌کند و آنها را به اندازه اولیه تصویر ورودی بزرگ می‌کند. هر مرحله از باز نمودن شامل ترکیبی از افزایش اندازه (به عنوان مثال با استفاده از تراکم خط) و لایه‌های کانولوشنی است. هدف از رمزگشای بازیابی اطلاعات فضایی از دست رفته در فرایند رمزگذاری و تولید نقشه نهایی بخش‌بندی است.

۴. **اتصال‌های پرش<sup>۵</sup>**: UNet از اتصال‌های پرش برای اتصال لایه‌های متناظر رمزگذار و رمزگشای استفاده می‌کند. این اتصالات ویژگی‌های رمزگذار را با ویژگی‌های رمزگشای در همان رزولوشن ارتباط می‌دهند. اتصالات پرش کمک می‌کنند تا جزئیات ریز را حفظ کنند و به شبکه اطلاعات محلی را ارائه دهند تا دقیق‌تر بخشدند.

۵. **خروجی**: خروجی نهایی UNet یک نقشه بخش‌بندی است با همان ابعاد فضایی تصویر ورودی. هر پیکسل در نقشه بخش‌بندی یک برچسب کلاس را نشان می‌دهد که نشان‌دهنده دسته‌بندی پیش‌بینی شده برای پیکسل متناظر در تصویر ورودی است.

UNet به دلیل قابلیت مدیریت همزمان اطلاعات محلی و سراسری به محبوبیت رسیده است. اتصالات پرش به شبکه امکان استفاده از ویژگی‌های با کیفیت پایین و با کیفیت بالا را

<sup>2</sup> Encoder

<sup>3</sup> Bridge

<sup>4</sup> Decoder

<sup>5</sup> Skip Connections

فراهم می‌کنند، که بهبود جزئیات ریز را حفظ کرده و آگاهی از متناظر های سراسری را حفظ می‌کنند. این معماری با موفقیت در برنامه‌های مختلف بخش‌بندی مانند تشخیص تصاویر پزشکی، تشخیص سلول‌ها و تشخیص اشیاء در صحنه‌های طبیعی مورد استفاده قرار گرفته است.

## لایه‌های مورد استفاده

1. **لایه ورودی (Input)**: این لایه ورودی تصویر را به عنوان ورودی شبکه دریافت می‌کند. ابعاد ورودی به اندازه  $128 \times 128 \times 3$  است که نشان‌دهنده ابعاد تصویر و عمق رنگ آن است.
2. **لایه‌های کانولوشن (Conv2D)**: این لایه‌ها عملیات کانولوشن را روی ورودی انجام می‌دهند. هر لایه کانولوشن یک تعدادی فیلتر با اندازه ویژگی خاص خود دارد و با استفاده از تابع فعال‌سازی ReLU، نقاط قوی تصویر را برجسته می‌کند.
3. **لایه حذف تصادفی (Dropout)**: این لایه‌ها با احتمال مشخص شده اتصالات موجود در شبکه را به صورت تصادفی غیرفعال می‌کنند. این کار باعث جلوگیری از بیش‌برازش<sup>6</sup> می‌شود.
4. **لایه‌های ادغام حداقل گیر (MaxPooling2D)**: این لایه‌ها با استفاده از عملیات حذف نمونه‌ها حداقل گیری ابعاد تصویر را کاهش می‌دهند. این کار باعث می‌شود که ویژگی‌های مهم تصویر حفظ شده و تعداد پارامترها و محاسبات در شبکه کاهش یابد.
5. **لایه‌های افزایش اندازه (UpSampling2D)**: این لایه‌ها با استفاده از عملیات ترکیب و افزایش اندازه (UpSampling) ابعاد تصویر را افزایش می‌دهند. این کار باعث می‌شود که اطلاعات دقیق‌تری در مقیاس بزرگ‌تر در دسترس باشد.
6. **لایه‌های الحاق (Concatenate)**: این لایه‌ها از دو ورودی مختلف (لایه‌های قبلی) استفاده کرده و آنها را در یک محور مشخص (محور سوم) ترکیب می‌کنند.
7. **لایه خروجی (Sigmoid و Conv2D)**: در انتهای، با استفاده از لایه Conv2D با یک فیلتر به ابعاد  $1 \times 1$  و تابع فعال‌سازی Sigmoid، خروجی نهایی شبکه تولید می‌شود. این خروجی یک تصویر به ابعاد ورودی است که مقادیر آن بین 0 و 1 قرار دارد و نشان می‌دهد که هر پیکسل از تصویر ورودی به چه احتمالی متعلق به دسته‌بندی مورد نظر است.

---

<sup>6</sup> Overfitting

## شرح بلوکها

### 1. بلوکهای Encoder

این بلوکها شامل چهار لایه (شش لایه، اگر تابع فعال‌ساز را جداگانه در نظر بگیریم) می‌باشند. یک لایه کانولوشنی که ابعاد فیلترهای آن  $3 \times 3$  می‌باشد. با فعال‌ساز ReLU که در ادامه‌ی آن لایه Dropout می‌آید. این لایه‌ها دو بار تکرار می‌شوند. دقیق‌تر دقت کنید تعداد فیلترهای لایه‌های کانولوشنی در بلوکهای مختلف، متفاوت است. همچنین ضربی برابر 0.2 در نظر گرفته شده است.

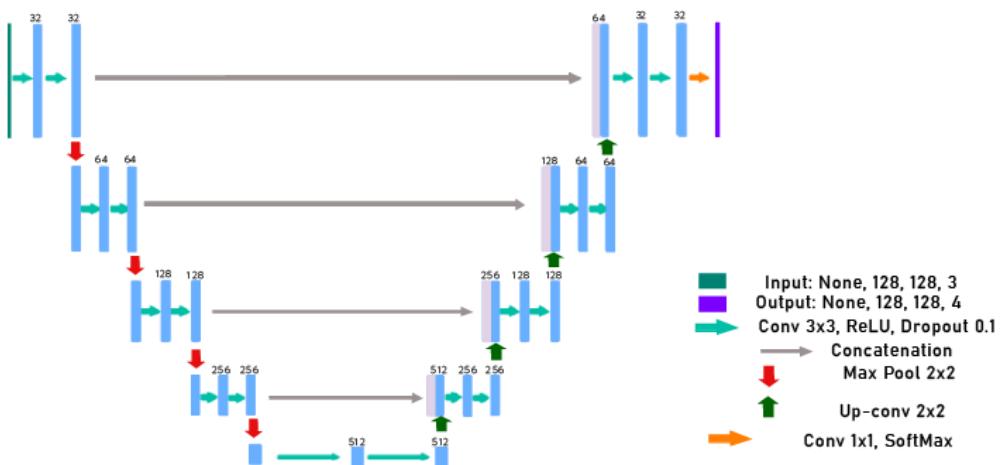
علت تفاوت نرخ dropout با مقاله، جلوگیری از بیش‌برازش به علت ساده‌تر بودن وظیفه شبکه مورد استفاده می‌باشد.

```
Conv2D -> ReLU -> Dropout -> Conv2D -> ReLU -> Dropout
```

### 2. اتصال بلوکهای Encoder با یک MaxPooling2D با اندازه $2 \times 2$ انجام می‌شود. 3. بلوکهای Decoder

این بلوکها در ابتدا با فیچر مپ هم‌بعادشان در بخش encoding الحاق می‌شوند و سپس تعدادی لایه مانند بخش encoder دارند و در انتهای لایه UpSample2D مورد استفاده قرار می‌گیرند.

```
Concatenate Conv2D -> ReLU -> Dropout -> Conv2D -> ReLU ->  
Dropout -> UpSample2D
```



شکل 2-3. معماری U-Net

در این شبکه از لایه‌های Dropout استفاده شده است، علت استفاده از این لایه تنوع بخشی به عکس‌های تولید شده است. علت این پدیده به طور کلی استفاده نکردن از بخشی از ویژگی‌های یادگرفته شده است. (زیرا آن نورون‌ها در ساخت خروجی بی‌اثر می‌شوند)

- Dropout به شبکه اجازه می‌دهد که با همان ورودی، تصاویر خروجی متفاوت و متنوعی تولید شود.
- با خاموش کردن تصادفی نورون‌ها، شبکه مجبور می‌شود که به جای تکیه بر مسیرهای خاص، ویژگی‌های مختلف را در داده‌های ورودی کشف و استفاده کند. این فرآیند موجب می‌شود که مدل به صورت کلی‌تری آموزش ببیند و ویژگی‌های متنوع‌تری را بیاموزد.
- باعث می‌شود که شبکه نسبت به تغییرات و نویزهای موجود در داده‌های Dropout ورودی مقاوم‌تر باشد. این مقاومت به شبکه کمک می‌کند تا در شرایط مختلف و با داده‌های مختلف نتایج بهتری ارائه دهد.

## 4. تابع هزینه

در مقاله Pix2Pix، تابع هزینه نهایی از دو بخش اصلی تشکیل شده است: تابع هزینه تقابلی (Adversarial Loss) و تابع هزینه بازسازی (L1 Loss). هر کدام از این بخش‌ها نقش مهمی در آموزش و بهبود کیفیت مدل ایفا می‌کنند.

### 1. تابع هزینه تقابلی (Adversarial Loss)

تابع هزینه تقابلی به مولد (Generator) و متمایزکننده (Discriminator) کمک می‌کند تا به طور همزمان با یکدیگر رقابت کنند و کیفیت تصاویر تولیدی را بهبود بخشنند. این تابع هزینه به صورت زیر تعریف می‌شود:

$$\mathcal{L}_{GAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z)))]$$

هدف مولد این است که متمایزکننده را فریب دهد، در حالی که هدف متمایزکننده این است که بتواند جفت‌های واقعی و جعلی را به درستی تشخیص دهد.

### 2. تابع هزینه بازسازی (L1 Loss)

برای اطمینان از اینکه تصاویر تولید شده توسط مولد به تصاویر هدف نزدیک باشند، از تابع هزینه بازسازی L1 استفاده می‌شود:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1]$$

این بخش از تابع هزینه مولد را تشویق می‌کند تا تصاویر تولیدی شباهت بیشتری به تصاویر واقعی داشته باشند.

### تابع هزینه نهایی

تابع هزینه نهایی به صورت ترکیبی از دو بخش فوق تعریف می‌شود:

$$\mathcal{L}(G, D) = \mathcal{L}_{GAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

در اینجا  $\lambda$  وزنی است که اهمیت هزینه بازسازی را نسبت به هزینه تقابلی تنظیم می‌کند.

### بروزرسانی وزن‌های Discriminator

وزن‌های متمایزکننده بر اساس تابع هزینه تقابلی به روزرسانی می‌شوند. هدف این است که متمایزکننده بتواند جفت‌های واقعی و جعلی را به درستی تشخیص دهد و وزن‌های مولد بر اساس ترکیبی از تابع هزینه تقابلی و هزینه بازسازی به روزرسانی می‌شوند. هدف مولد این است که متمایزکننده را فریب دهد و تصاویر تولیدی را به تصاویر واقعی نزدیک‌تر کند. به روزرسانی وزن‌ها نیز با استفاده از روش‌های بهینه‌سازی مانند Adam انجام می‌شود.

## 5. پژوهش‌های مرتبط

### Pix2PixHD: High-Resolution Image Synthesis and Semantic A. مقاله: Manipulation with Conditional GANs

به عنوان یک نسخه بهبود یافته از Pix2Pix طراحی شده است که به تولید تصاویر با وضوح بالا و دستکاری معنایی تصاویر می‌پردازد. ایده اصلی این مقاله استفاده از معماری چند مقیاسی برای مولد و متمایزکننده است. مولد از یک شبکه U-Net با معماری سلسله مراتبی تشکیل شده است که تصاویر را در چندین سطح مقیاس تولید می‌کند. این ساختار اجازه می‌دهد تا جزئیات بیشتری در سطوح مختلف تصویر حفظ شود و تصاویر با وضوح بالا و کیفیت بهتر تولید شوند. همچنین، متمایزکننده نیز به صورت چند مقیاسی عمل می‌کند که کمک می‌کند تا تصاویر تولیدی به دقت ارزیابی شوند و مدل بهبود یابد.

## B. مقاله: CycleGAN: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

CycleGAN بهبود دیگری بر پایه ایده‌های Pix2Pix است، با این تفاوت که نیاز به داده‌های جفت شده ندارد. ایده اصلی این مقاله استفاده از یک چرخه هماهنگ (Cycle) برای ترجمه تصویر به تصویر است. در این روش، یک تصویر از حوزه A به حوزه B ترجمه می‌شود و سپس دوباره به حوزه A بازگردانده می‌شود. این فرآیند تضمین می‌کند که تصویر ورودی و خروجی اولیه باید به یکدیگر شباهت داشته باشند. استفاده از چرخه هماهنگ باعث می‌شود که مدل بدون نیاز به داده‌های جفت شده (paired data) قادر به یادگیری ترجمه‌های معنایی بین دو حوزه مختلف باشد. این ایده به طور خاص برای کاربردهایی مفید است که داده‌های جفت شده به سختی قابل دسترسی هستند.

### 2-2. پیاده‌سازی معماری Pix2Pix

دیتاست مورد استفاده، حاوی تصاویر نقشه‌ای و ماهواره‌ای می‌باشد. همچنین دیتاست داده شده، خود شامل تصاویر آموزش و آزمون می‌باشد.

پس از خواندن عکس‌ها، ابتدا آن‌ها را به اندازه  $256 \times 256$  تغییر اندازه می‌دهیم و سپس آن‌ها را نرمالایز می‌کنیم و تمامی مقادیر پیکسل‌ها را بین 1- تا 1 می‌کنیم.

```
def read_data(image_file):
    resize = lambda x: tf.image.resize(x, [256, 256],
method=tf.image.ResizeMethod.NEAREST_NEIGHBOR)
    normalize = lambda x: (x / 127.5) - 1

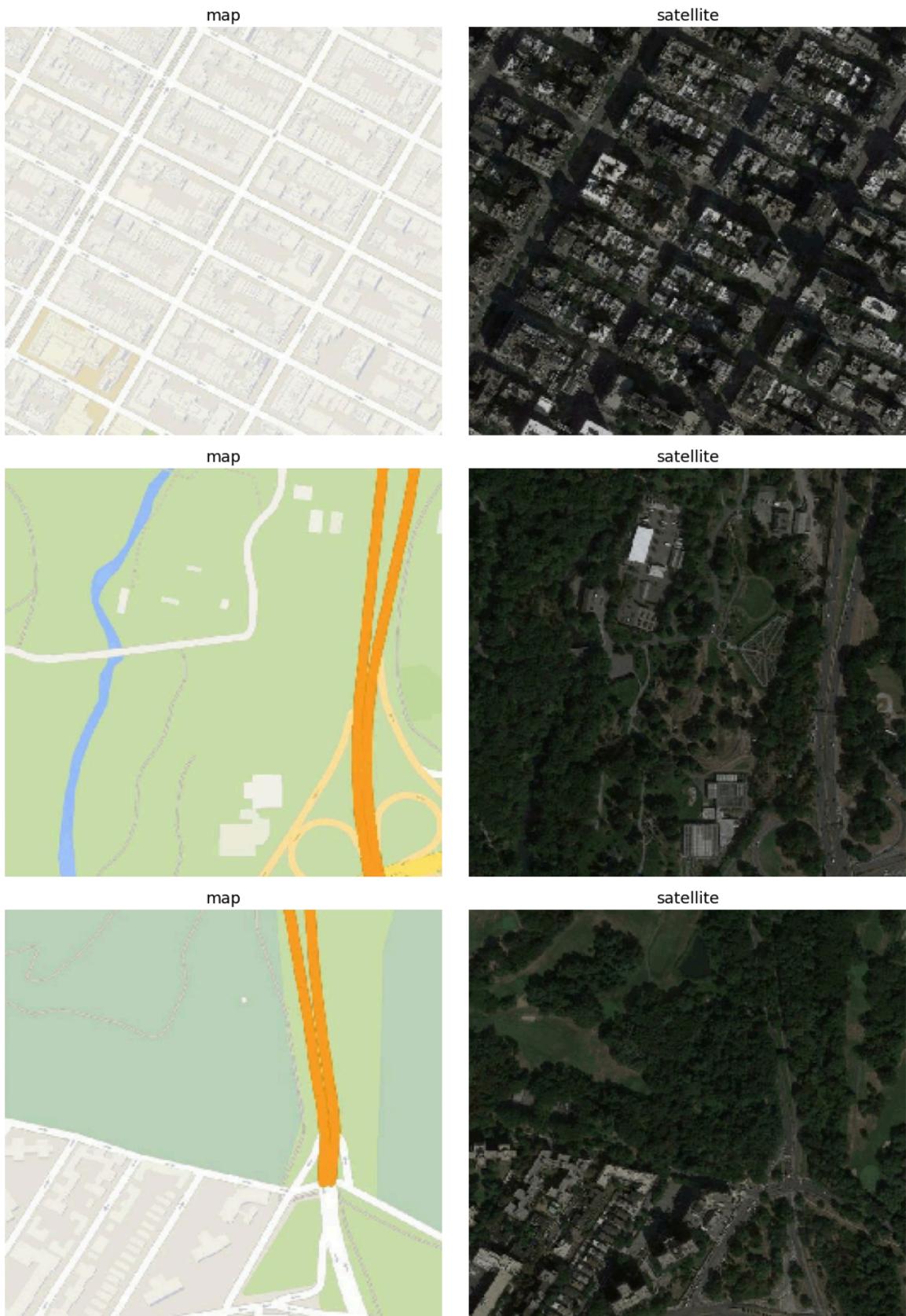
    image = tf.io.read_file(image_file)
    image = tf.image.decode_jpeg(image, channels=3)

    w = tf.shape(image)[1] // 2
    satellite = image[:, :w, :]
    map = image[:, w:, :]

    satellite = tf.cast(satellite, tf.float32)
    map = tf.cast(map, tf.float32)

    map = normalize(resize(map))
    satellite = normalize(resize(satellite))

    return map, satellite
```



شکل 2-4. چند نمونه تصویر از دیتاست

ابتدا Discriminator را تعریف می‌کنیم:

```
def Discriminator():
    initializer = tf.random_normal_initializer(0., 0.02)

    input = Layers.Input(shape=[256, 256, 3],
name='input_image')
    target = Layers.Input(shape=[256, 256, 3],
name='target_image')

    x = Layers.concatenate([input, target])
    x = downsample(64, 4, False)(x)
    x = downsample(128, 4)(x)
    x = downsample(256, 4)(x)
    x = Layers.ZeroPadding2D()(x)
    x = Layers.Conv2D(512, 4, strides=1,
kernel_initializer=initializer, use_bias=False)(x)
    x = Layers.BatchNormalization()(x)
    x = Layers.LeakyReLU()(x)
    x = Layers.ZeroPadding2D()(x)
    last = Layers.Conv2D(1, 4, strides=1,
kernel_initializer=initializer)(x)

    return keras.Model(inputs=[input, target], outputs=last)
```

:Downsample سپس

```
def downsample(filters, size, batchnorm=True):
    initializer = tf.random_normal_initializer(0., 0.02)
    block = keras.Sequential()
    block.add(
        Layers.Conv2D(
            filters,
            size,
            strides=2,
            padding='same',
            kernel_initializer = initializer,
            use_bias=False
        )
    )
    if batchnorm:
```

```

    block.add(Layers.BatchNormalization())
block.add(Layers.LeakyReLU())
return block

```

:upsample و سپس

```

def upsample(filters, size, dropout=False):
    initializer = tf.random_normal_initializer(0., 0.02)
    block = keras.Sequential()
    block.add(
        Layers.Conv2DTranspose(
            filters,
            size,
            strides=2,
            padding='same',
            kernel_initializer=initializer,
            use_bias=False
        )
    )
    block.add(Layers.BatchNormalization())
    if dropout:
        block.add(Layers.Dropout(0.5))
    block.add(Layers.ReLU())

    return block

```

برای آموزش یک بار هدف را عکس نقشه‌ای و بار دیگر عکس ماهواره‌ای در نظر گرفتیم.  
همچنین هایپر پارامترهای مورد استفاده را در جدول زیر آورده‌یم.

Epochs	15
Loss	L1 + GAN
Metrics	Losses
Optimizer	Adam(lr = 2e-5, beta = 0.5)
Batch Size	1

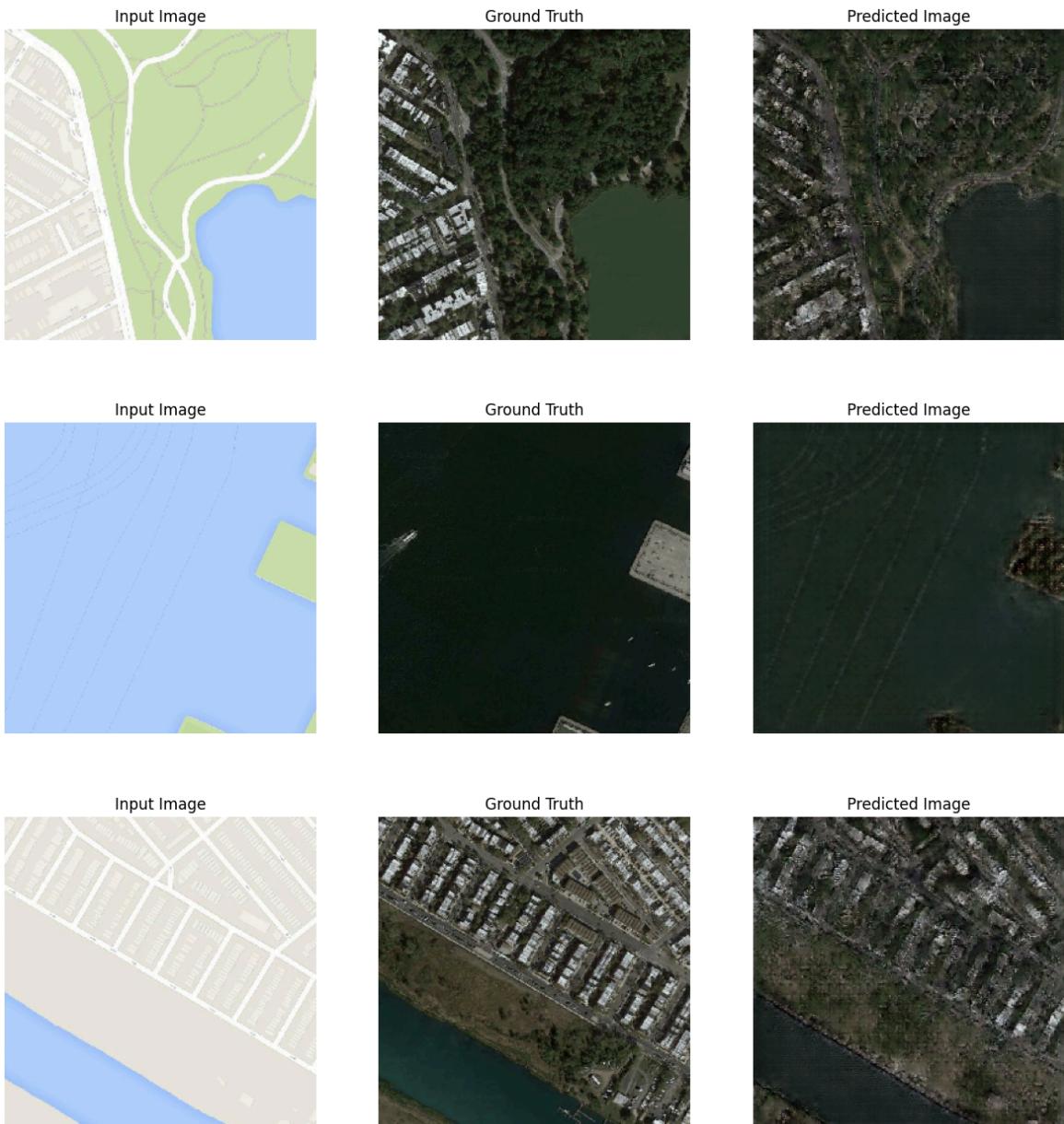
**الف. نقشه به عنوان هدف**

در ابتدا تصاویر تولید شده به صورت نویز می‌باشند.



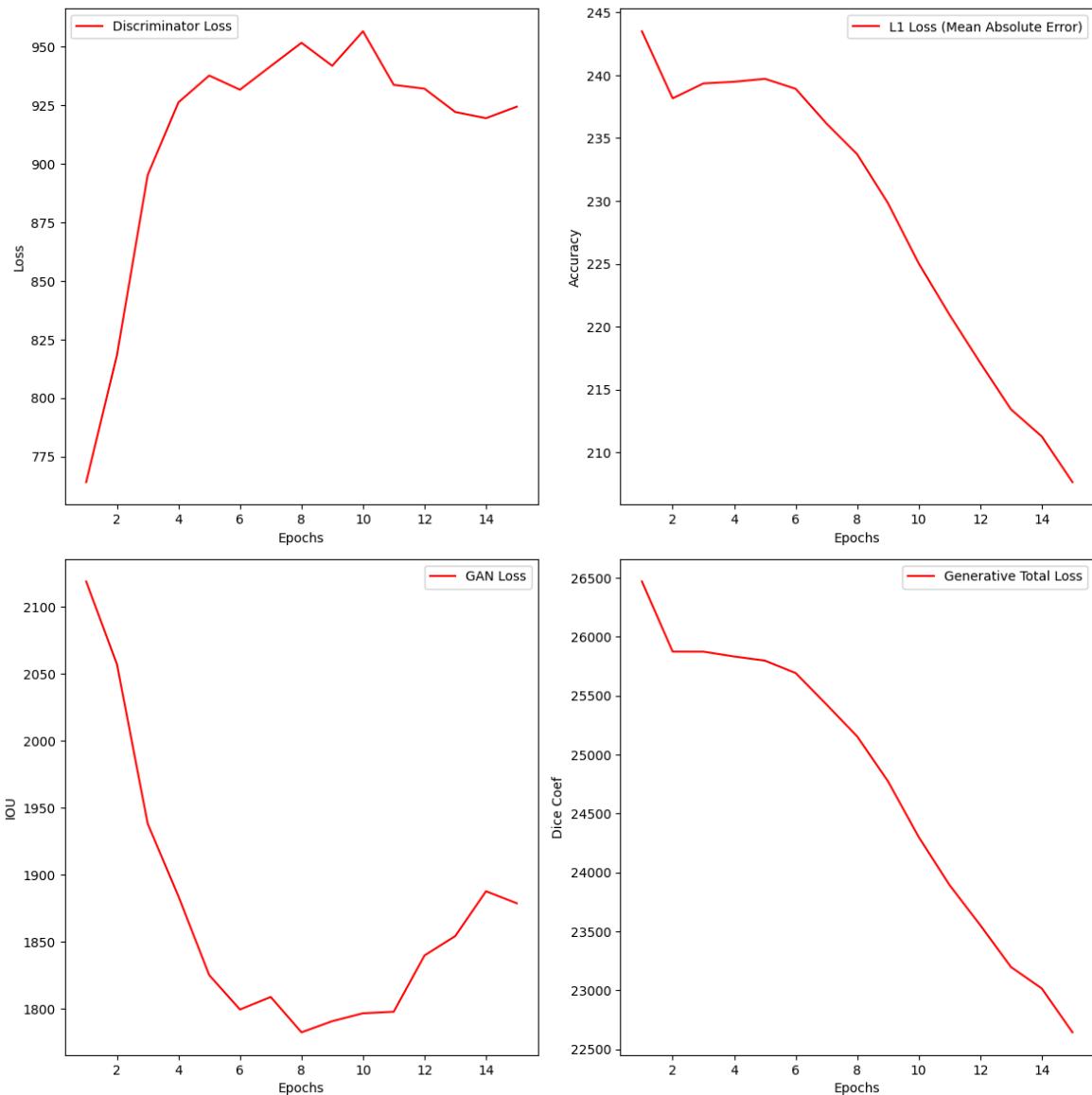
**شکل 2-5. خروجی شبکه در ابتدای کار**

اما در آخر می‌توان خروجی‌ها را به صورت زیر مشاهده کرد:



شکل 2-6. خروجی شبکه در حالت عکس ماهواره‌ای به عنوان هدف

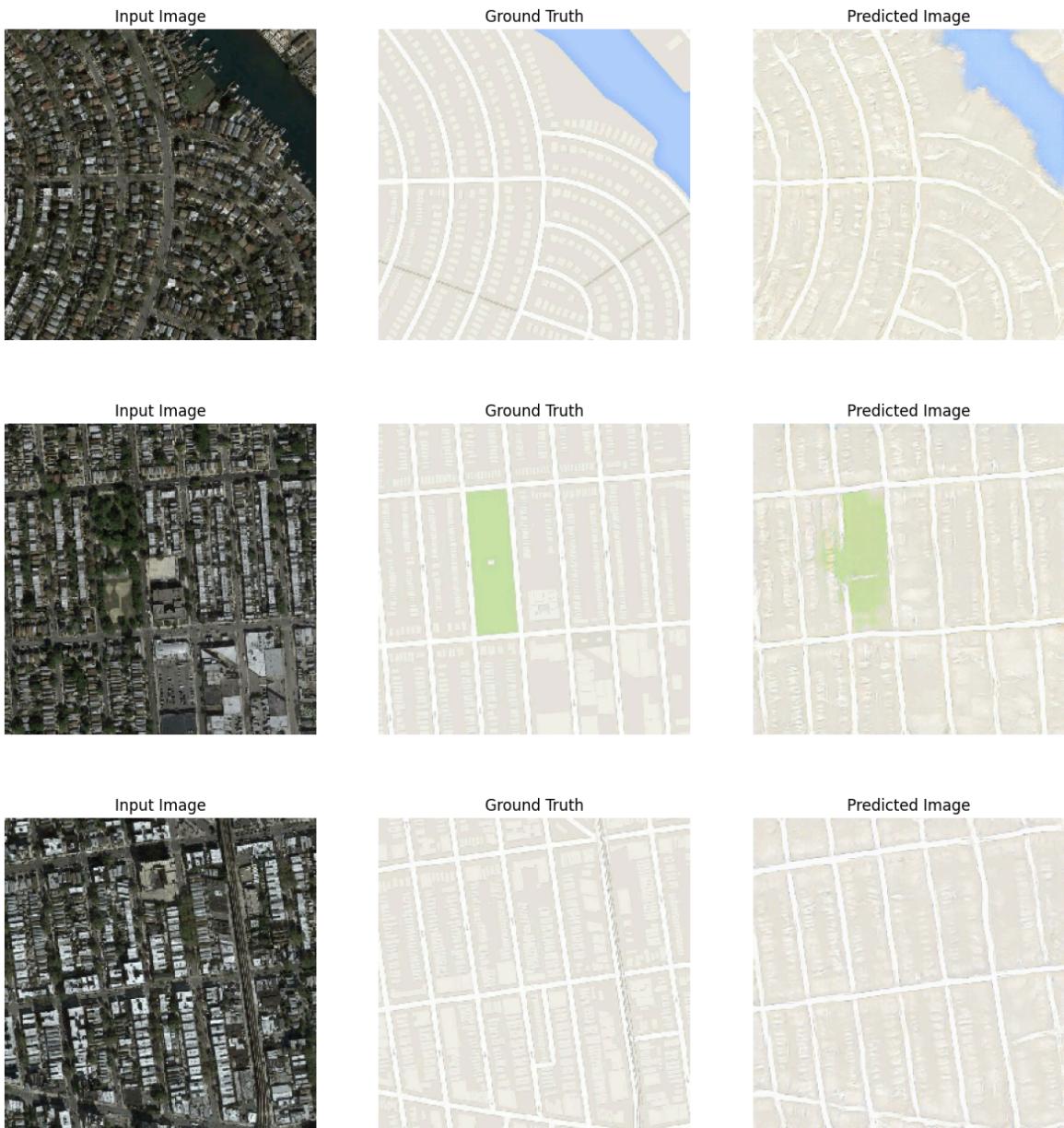
همچنین توابع هزینه در ادامه آمده است.



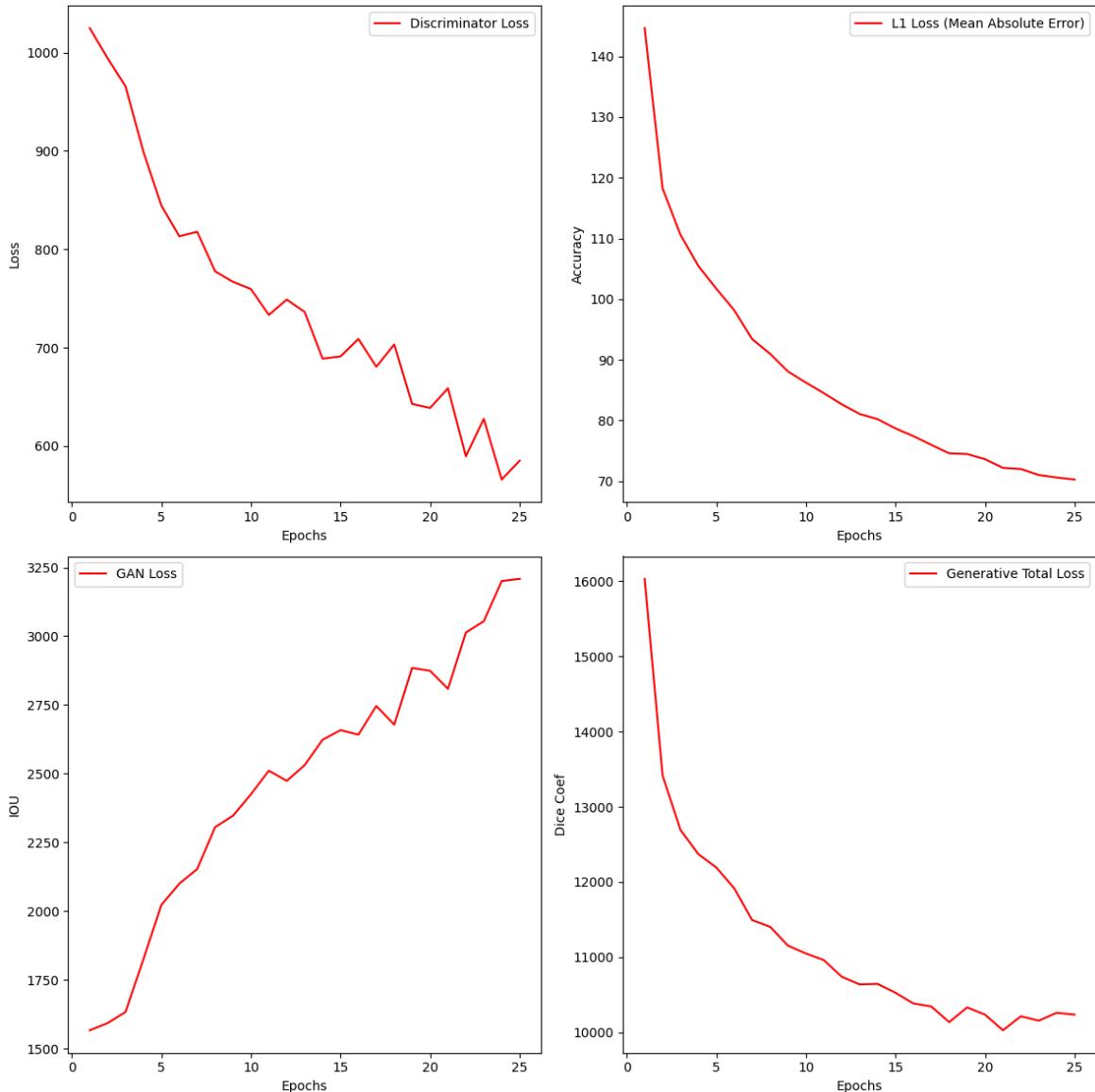
شکل 2-7. توابع هزینه در حالت عکس ماهواره‌ای به عنوان هدف

می‌توانیم مشاهده کنیم که شبکه مولد به خوبی در حال کاهش توابع هزینه خود می‌باشد و شبکه Discriminator را دچار اشتباه می‌کند در نتیجه این شبکه مقدار Loss آن افزایش می‌یابد. دقت کنید مقادیر آمده شده مجموع هزینه برای تمامی عکس‌های موجود در داده‌ی تست می‌باشد.

## ب. ماهواره به عنوان هدف



شکل 2-8. خروجی شبکه در حالت نقشه به عنوان هدف



شکل 2-9. مقادیر توابع هزینه در حالت نقشه به عنوان هدف

این شبکه به خوبی نتوانسته است که خروجی را تولید کند در نتیجه به راحتی عکس‌های تولید شده توسط آن بوسیله Discriminator تشخیص داده می‌شود.