Conditional Variational Autoencoder vs Conditional Generative Adversarial Network

Introduction

CVAE

Conditional Variational Autoencoder (CVAE) is an extension of Variational Autoencoder (VAE).

On VAE, the objective is:

$$logP(X) - D_{KL}[Q(z \mid X) \,||\, P(z \mid X)] = E[logP(X \mid z)] - D_{KL}[Q(z \mid X) \,||\, P(z)]$$

The original VAE model has two parts: the encoder $Q(z \mid X)$ and the decoder $P(X \mid z)$.The reason why VAE can't generate specific data is because the encoder models the latent variable Z directly based on X, it doesn't care about the different type of X.

Hence, VAE could be improved by conditioning the encoder and decoder to C, so the encoder is now conditioned to two variables X and C: $Q(z \mid X, c)$. The same with the decoder, it's now conditioned to two variables Z and C: $P(X \mid z, c)$.

Finally, the variational lower bound objective is in this following form:

$$logP(X \mid c) - D_{KL}[Q(z \mid X, c) \| P(z \mid X, c)] = E[logP(X \mid z, c)] - D_{KL}[Q(z \mid X, c) \| P(z \mid c)]$$

Now, the real latent variable is distributed under $P(z \mid c)$ and for each possible value of C, we would have a $P(z)$. We could also use this form of thinking for the decoder.

CGAN

Similarly, Conditional Generative Adversarial Network (CGAN) is an extension of Generative Adversarial Network (GAN)

GAN consists of two 'adversarial' models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. Both G and D could be a non-linear mapping function, such as a multi-layer perceptron.

To learn a generator distribution p_g over data x, the generator builds a mapping function from a prior noise distribution $p_z(z)$ to data space as $G(z;\theta_g)$. And the discriminator, $D(x;\theta_d)$, outputs a single scalar representing the probability that x came form training

data rather than p_g. G and D are both trained simultaneously: we adjust parameters for G to minimize $log(1-D(G(z))$ and adjust parameters for D to minimize $logD(X)$, as if they are following the two-player min-max game with value function $V(G, D)$:

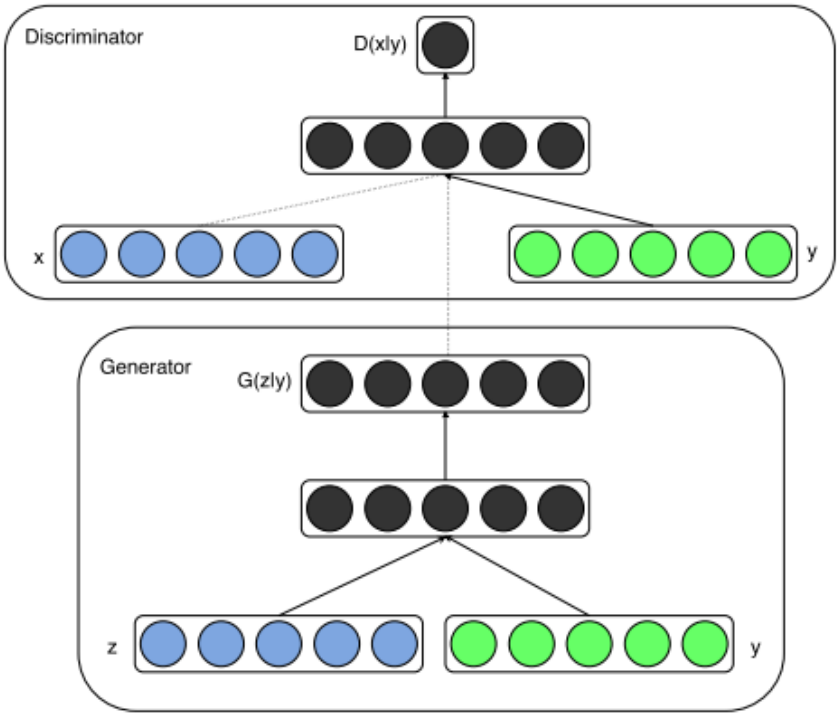
$$\min_G \max_D V(D, G) = E_{x\sim P_{data}(x)}\left[log\ D(x)\right] + E_{z\sim p_z(z)}\left[log\left(1 - D(G(z))\right)\right].$$

Generative adversarial nets can be extended to a conditional model if both the generator and discriminator are conditioned on some extra information y. y could be any kind of auxiliary information, such as class labels or data from other modalities.

We can perform the conditioning by feeding y into the both the discriminator and generator as additional input layer. In the generator the prior input noise $p_z(z)$, and y are combined in joint hidden representation, and the adversarial training framework allows for considerable flexibility in how this hidden representation is composed. In the discriminator x and y are presented as inputs and to a discriminative function (embodied again by a MLP in this case).

The objective function of a two-player minimax game would be as

$$\min_G \max_D V(D, G) = E_{x \sim P_{data}(x)} \left[log\ D(x|y) \right] + E_{z \sim p_z(z)} \left[log\left(1 - D(G(z|y))\right) \right].$$



Conditional adversarial net

Implementation

CVAE

Use MNIST for example. We could use the label as our conditional variable c. In this case, c is categorically distributed.

```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
X_train, y_train = mnist.train.images, mnist.train.labels
X_test, y_test = mnist.test.images, mnist.test.labels

m = 50
n_x = X_train.shape[1]
n_y = y_train.shape[1]
n_z = 2
n_epoch = 20
```

```
# Q(z|X,y) -- encoder
X = Input(batch_shape=(m, n_x))
cond = Input(batch_shape=(m, n_y))
```

Concatenation

```
inputs = merge([X, cond], mode='concat', concat_axis=1)

h_q = Dense(512, activation='relu')(inputs)
mu = Dense(n_z, activation='linear')(h_q)
log_sigma = Dense(n_z, activation='linear')(h_q)

def sample_z(args):
    mu, log_sigma = args
    eps = K.random_normal(shape=(m, n_z), mean=0., std=1.)
    return mu + K.exp(log_sigma / 2) * eps

# Sample z ~ Q(z|X,y)
z = Lambda(sample_z)([mu, log_sigma])
z_cond = merge([z, cond], mode='concat', concat_axis=1) # <--- NEW!

# P(X|z,y) -- decoder
decoder_hidden = Dense(512, activation='relu')
decoder_out = Dense(784, activation='sigmoid')

h_p = decoder_hidden(z_cond)
outputs = decoder_out(h_p)
```

Loss function

```
def vae_loss(y_true, y_pred):
    """ Calculate loss = reconstruction loss + KL loss for each data in mini batch """
    # E[log P(X|z,y)]
    recon = K.sum(K.binary_crossentropy(y_pred, y_true), axis=1)
    # D_KL(Q(z|X,y) || P(z|X)); calculate in closed form as both dist. are Gaussian
    kl = 0.5 * K.sum(K.exp(log_sigma) + K.square(mu) - 1. - log_sigma, axis=1)

    return recon + kl
```

CGAN

Use MNIST for example as well.

```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

if not os.path.exists("logdir"):
os.makedirs("logdir")

LOGDIR = "logdir"
real_img_size = mnist.train.images[0].shape[0]
noise_size = 100
noise = 'normal0-1'
alpha = 0.1
learning_rate = 0.001
smooth = 0.05

batch_size = 100
k = 10
epochs = 120
```

Generator

```
def get_generator(digit, noise_img,  reuse = False):
    with tf.variable_scope("generator", reuse = reuse):
        concatenated_img_digit = tf.concat([digit, noise_img], 1)

#         output = tf.layers.dense(concatenated_img_digit, 256)

        output = fully_connected('gf1',concatenated_img_digit,128)
        output = leakyRelu(output)
        output = tf.layers.dropout(output, rate = 0.5)

#         output = tf.layers.dense(output, 128)

        output = fully_connected('gf2',output, 128)
        output = leakyRelu(output)
        output = tf.layers.dropout(output, rate = 0.5)

#         logits = tf.layers.dense(output, 784)
        logits = fully_connected('gf3',output,784)
        outputs = tf.tanh(logits)
        return logits, outputs
```

Discriminator

```
def get_discriminator(digit, img,  reuse = False):
    with tf.variable_scope("discriminator", reuse=reuse):
        concatenated_img_digit = tf.concat([digit, img], 1)

#         output = tf.layers.dense(concatenated_img_digit, 256)
        output = fully_connected('df1',concatenated_img_digit,128)
        output = leakyRelu(output)
        output = tf.layers.dropout(output, rate = 0.5)

#         output = tf.layers.dense(concatenated_img_digit, 128)
        output = fully_connected('df2',output, 128)
        output = leakyRelu(output)
        output = tf.layers.dropout(output, rate = 0.5)

#         logits = tf.layers.dense(output, 1)
        logits = fully_connected('df3',output, 1)
        outputs = tf.sigmoid(logits)

        return logits, outputs
```

Conditional MNIST

Test our CVAE model and CGAN model to generate MNIST data.

After reconstructing the images, the results are as followed.

Name	Epoch 1	Epoch 10	Epoch 25
CVAE			
CGAN			

We can find that, in general, for CVAE, the generated image is relatively stable, but the generated image diversity is bad, but for CGAN, the generated image diversity is better, but the generation process is random .

Conclusion

In this post, we compared the CVAE to the CGAN. In CVAE, we could generate data with specific attribute, an operation that can't be done with the vanilla VAE. We showed this by applying CVAE on MNIST data and conditioned the model to the images' labels. The resulting model allows us to sample data under specific label. While in CGAN, compared with the original GAN network, the CGAN network has not changed, only the input data of the generator G and the discriminator D are changed, which allows CGAN to be embedded in other GAN networks as a general strategy.

All in all, we can know that the CVAE optimization process forcibly fits the data to a finite-dimensional Gaussian mixture or other distributions, resulting in: information loss during the mapping process, especially the loss of secondary information, and information that does not conform to the preset distribution The encoding and recovery effect of the system is poor. In contrast, there is no such strong preset in the training of CGAN. It is optimized through the

discriminator network, so that the distribution of the data generated by the generator directly fits the distribution of the training data, and there is nothing special about the specific distribution.