
















부록 D. 확산 모델 프로그래밍 실습

13.4절에서 소개한 확산 모델(diffusion model)은 DALL·E2와 Imagen을 구현하는 중요 기술이다. 여기서는 텐서플로를 이용하여 확산 모델을 구현한다. 데이터셋으로는 Oxford Flowers 102 데이터셋을 활용한다. 이 데이터셋은 영국에 서식하는 102종의 꽃 영상을 담고 있는데, 부류 별로 40~258장의 영상을 가진다[Nilsback2008]. 총 8189장의 영상이 있다. [그림 D-1]은 몇 종류의 샘플을 예시한다.

Category	#ims	Category	#ims	Category	#ims
 alpine sea holly	43	 buttercup	71	 fire lily	40
 anthurium	105	 californian poppy	102	 foxglove	162
 artichoke	78	 camellia	91	 frangipani	166
 azalea	96	 canna lily	82	 fritillary	91
 ball moss	46	 canterbury bells	40	 garden phlox	45

[그림 D-1] Oxford Flowers 102 데이터셋(<https://www.robots.ox.ac.uk/~vgg/data/flowers/102/>)

D.1 확산 모델의 학습

[프로그램 D-1]은 Oxford Flowers 102 데이터셋을 읽고, 확산 모델을 학습하고, 새로운 샘플을 생성해 보여준다. 이 프로그램은 케라스 공식 사이트가 제공하는 소스코드를 이 책의 스타일에 맞게 개조한 것이다. 이 소스코드는 13.4절에서 소개한 DDIM[Song2020]을 구현한다. DDIM은 [알고리즘 13-1]과 [알고리즘 13-2]가 설명하는 DDPM[Ho2020]의 속도를 개선한 모델이다. 소스코드에 대한 설명과 구현 요령에 대한 상세한 내용은 <https://keras.io/examples/generative/ddim/>을 참조한다.

[프로그램 D-1]은 학습에 많은 시간이 소요된다. 학습이 부담스런 독자는 사전 학습된 모델을 사용하는 [프로그램 D-2]를 활용하기 바란다.

08~11행은 데이터셋과 관련한 값을 설정한다. 영상 크기는 64*64이고 미니 배치 크기는 64이다. 33행의 prepare_dataset 함수는 데이터셋을 읽어오고, 25행의 preprocess_image 함수는 전처리를

수행한다. 39~40행은 `prepare_dataset` 함수를 이용하여 `train`, `validation`, `test`로 분할되어 있는 데이터셋을 하나로 합치고 그중 80%는 `train_dataset`, 나머지 20%는 `val_dataset`에 저장한다.

42~77행는 KID 클래스를 정의한다. 이 클래스는 13.5절이 소개한 커널 인셉션 거리를 측정하는 메트릭이다. 143~258행에 정의된 `DiffusionModel` 클래스에 있는 `compile` 함수(150~154행)의 154행과 `metrics` 함수(157~158행)의 158행을 통해 모델 성능을 측정하는 메트릭으로 등록한다. 이렇게 등록해두면, 267~268행의 `compile`과 `fit` 함수로 모델을 실제 학습할 때 세대가 끝날 때마다 이 메트릭으로 성능을 측정하여 결과를 출력한다.

Tip: 이 책에서 실습한 모든 프로그램에서는 별도로 `metrics` 함수를 정의하지 않았기 때문에 텐서플로에 미리 등록되어 있는 'accuracy'같은 메트릭을 사용한다.

79~84행의 `sinusoidal_embedding` 함수는 트랜스포머의 위치 인코딩과 유사한 정보를 생성하여 신경망에 분산에 관련한 정보를 제공한다. 122~141행의 `get_network` 함수는 86~98행의 `ResidualBlock`, 100~109행의 `DownBlock`, 111~120행의 `UpBlock` 함수를 이용하여 U-net 신경망을 구현한다. 126행은 `sinusoidal_embedding` 함수로 신경망에 분산 정보를 주입한다. 114행과 127행은 영상의 크기를 키우는 업 샘플링을 담당하는데, 9.4.2절에서 소개한 전치 컨볼루션 대신 3.5.3절에서 소개한 최근접 이웃과 양선형 보간을 사용한다. 전치 컨볼루션으로 대체하여 성능 향상이 있는 지 확인해보는 일은 좋은 연습문제다.

143~258행은 확산 모델을 구현하는 `DiffusionModel` 클래스다. 생성자 `__init__` 함수는 `get_network` 함수로 U-net을 두 개 만든다. 147행이 만드는 `network` 객체는 확산 모델의 디노이징을 담당하는데 쓰고 148행이 만드는 `ema_network` 객체는 KID를 측정하는데 쓴다. 164~171행의 `diffusion_schedule` 함수는 삼각함수에 기반한 분산 스케줄을 구현한다. 173~179행의 `denoise` 함수는 `network`이라는 신경망을 통해 잡음을 분리하여 `pred_noises`에 저장하고(177행), 잡음을 분리한 영상을 만들어 `pred_images`에 저장하고(178행), `pred_noises`와 `pred_images`를 반환한다. 199~203행의 `generate` 함수는 `reverse_diffusion` 함수와 `denoise` 함수를 활용하여 `num_images` 매개변수가 지정한 개수의 영상을 생성한다.

205~226행의 `train_step` 함수는 268행의 `fit` 함수를 위해 실제 학습을 수행한다. 이 책에서 실습한 모든 프로그램에서는 별도로 `train_step` 함수를 정의하지 않았기 때문에 텐서플로 안에 미리 등록되어 있는 함수를 사용했지만 여기서는 명시적으로 정의되어 있는 205~226행의 `train_step`을 사용한다. 228~244행의 `test_step` 함수도 명시적으로 정의되어 있기 때문에 내부 함수 대신 사용한다. `test_step` 함수는 학습 도중에 세대가 끝날 때마다 성능을 측정하여 반환하는 기능을 수행한다. 246~258행의 `plot_images` 함수는 지정한 개수만큼 영상을 생성하여 그려준다.

260~271행은 메인에 해당한다. 260행은 `DiffusionModel` 클래스의 객체를 생성하여 `model`에 저장한다. 262~263행은 체크포인트에 쓸 정보를 설정한다. 'val_kid'를 기준으로 새로운 최소를 달성한

모델의 가중치를 지정한 폴더에 저장하라는 뜻이다. 학습하는데 시간이 많이 걸리는 경우 매우 편리한 기능이다. 265행은 훈련 집합에서 데이터 정규화에 쓸 값을 추정하고 신경망 모델에 기록 해두는 역할을 한다.

267~268행의 compile과 fit 함수는 실제 학습을 수행한다. 옵티마이저로 Adam 대신 AdamW를 사용한다. AdamW에 대해서는 <https://keras.io/api/optimizers/adamw/>를 참조한다. 손실 함수로는 $MSE_{\text{Mean Squared Error}}$ 대신 $MAE_{\text{Mean Absolute Error}}$ 를 사용한다. <https://keras.io/examples/generative/ddim/>에서는 실험 결과 AdamW와 MAE가 더 우수하기 때문이라고 설명한다.

프로그램 실행 결과를 보면, 1~2 세대에서는 잡음만 생성되다가 5세대에서 패턴이 나타나기 시작 한다. 10세대에서 꽃 모양이 나타나며, 이후 꾸준히 영상 품질이 개선된다. 검증 집합 val_dataset 에 대한 KID는 1세대에서 2.0365로 출발하여 꾸준히 줄어 50세대에서 0.1976이 되었다.

[프로그램 D-1] Oxford Flowers 102 데이터셋을 이용한 확산 모델 프로그래밍

```
01 import math
02 import matplotlib.pyplot as plt
03 import tensorflow as tf
04 import tensorflow_datasets as tfds
05 from tensorflow import keras
06 from keras import layers
07
08 dataset_name="oxford_flowers102" # 옥스퍼드 꽃 데이터셋
09 dataset_repetitions=5
10 img_siz=64
11 batch_siz=64
12
13 kid_img_siz=75 # KID
14 kid_diffusion_steps=5
15 plot_diffusion_steps=20
16
17 min_signal_rate=0.02 # 샘플링
18 max_signal_rate=0.95
19
20 zdim=32 # 신경망 구조
21 embed_max_freq=1000.0
22 widths=[32,64,96,128]
23 block_depth=2
24
```

```

25 def preprocess_image(data):
26     height=tf.shape(data["image"])[0] # 중앙 잘라내기(center cropping)
27     width=tf.shape(data["image"])[1]
28     crop_siz=tf.minimum(height, width)
29     image=tf.image.crop_to_bounding_box(data["image"],(height-crop_siz)//2,(width-
        crop_siz)//2,crop_siz,crop_siz)
30     image=tf.image.resize(image,size=[img_siz,img_siz],antialias=True) # antialias=True 설정
        중요
31     return tf.clip_by_value(image/255.0,0.0,1.0)
32
33 def prepare_dataset(split):
34     return (tfds.load(dataset_name,split=split,shuffle_files=True)
35             .map(preprocess_image, num_parallel_calls=tf.data.AUTOTUNE).cache()
36             .repeat(dataset_repetitions).shuffle(10*batch_siz) # 셔플링은 KID에 중요
37             .batch(batch_siz,drop_remainder=True).prefetch(buffer_size=tf.data.AUTOTUNE))
38
39 train_dataset=prepare_dataset("train[:80%]+validation[:80%]+test[:80%]") # 데이터셋
40 val_dataset=prepare_dataset("train[80%:]+validation[80%:]+test[80%:]")
41
42 class KID(keras.metrics.Metric): # KID 측정을 위한 클래스
43     def __init__(self,name,**kwargs):
44         super().__init__(name=name,**kwargs)
45         self.kid_tracker=keras.metrics.Mean(name="kid_tracker")
46         self.encoder=keras.Sequential( # InceptionV3 사용
47             [keras.Input(shape=(img_siz,img_siz,3)),layers.Rescaling(255.0),
48              layers.Resizing(height=kid_img_siz,width=kid_img_siz),
49              layers.Lambda(keras.applications.inception_v3.preprocess_input),
50              keras.applications.InceptionV3(include_top=False,input_shape=(kid_img_siz,kid_img_siz,3),weights="imagenet"),
51              layers.GlobalAveragePooling2D()],name="inception_encoder")
52
53     def polynomial_kernel(self,features_1,features_2):
54         feature_dimensions=tf.cast(tf.shape(features_1)[1],dtype=tf.float32)
55         return (features_1 @ tf.transpose(features_2)/feature_dimensions+1.0)**3.0
56
57     def update_state(self,real_images,generated_images,sample_weight=None):
58         real_features=self.encoder(real_images,training=False)
59         generated_features=self.encoder(generated_images,training=False)

```

```

60
61         kernel_real=self.polynomial_kernel(real_features,real_features) # 두 특징으로 다항식
        커널 계산
62         kernel_generated=self.polynomial_kernel(generated_features,generated_features)
63         kernel_cross = self.polynomial_kernel(real_features, generated_features)
64
65         batch_siz=tf.shape(real_features)[0] # 평균 커널값으로 squared maximum mean
        discrepancy 측정
66         batch_sizf=tf.cast(batch_siz,dtype=tf.float32)
67         mean_kernel_real=tf.reduce_sum(kernel_real*(1.0-
        tf.eye(batch_siz)))/(batch_sizf*(batch_sizf-1.0))
68         mean_kernel_generated=tf.reduce_sum(kernel_generated*(1.0-
        tf.eye(batch_siz)))/(batch_sizf*(batch_sizf-1.0))
69         mean_kernel_cross=tf.reduce_mean(kernel_cross)
70         kid=mean_kernel_real+mean_kernel_generated-2.0*mean_kernel_cross
71         self.kid_tracker.update_state(kid) # 평균 KID 측정을 갱신
72
73     def result(self):
74         return self.kid_tracker.result()
75
76     def reset_state(self):
77         self.kid_tracker.reset_state()
78
79     def sinusoidal_embedding(x):
80         embed_min_freq=1.0
81         freq=tf.exp(tf.linspace(tf.math.log(embed_min_freq),tf.math.log(embed_max_freq),zdim//2))
82         angular_speeds=2.0*math.pi*freq
83         embeddings=tf.concat([tf.sin(angular_speeds*x),tf.cos(angular_speeds*x)],axis=3)
84         return embeddings
85
86     def ResidualBlock(width):
87         def apply(x):
88             input_width=x.shape[3]
89             if input_width==width: residual=x
90             else: residual=layers.Conv2D(width,kernel_size=1)(x)
91
92             x=layers.BatchNormalization(center=False,scale=False)(x)
93
94             x=layers.Conv2D(width,kernel_size=3,padding="same",activation=keras.activations.swish)(x)

```

```

94         x=layers.Conv2D(width,kernel_size=3,padding="same")(x)
95         x=layers.Add()([x,residual])
96         return x
97
98     return apply
99
100 def DownBlock(width,block_depth):
101     def apply(x):
102         x,skips=x
103         for _ in range(block_depth):
104             x=ResidualBlock(width)(x)
105             skips.append(x)
106         x=layers.AveragePooling2D(pool_size=2)(x)
107         return x
108
109     return apply
110
111 def UpBlock(width,block_depth):
112     def apply(x):
113         x,skips=x
114         x=layers.UpSampling2D(size=2,interpolation="bilinear")(x)
115         for _ in range(block_depth):
116             x=layers.Concatenate()([x,skips.pop()])
117             x=ResidualBlock(width)(x)
118         return x
119
120     return apply
121
122 def get_network(image_size,widths,block_depth):
123     noisy_images=keras.Input(shape=(image_size,image_size,3))
124     noise_variances=keras.Input(shape=(1,1,1))
125
126     e=layers.Lambda(sinusoidal_embedding)(noise_variances)
127     e=layers.UpSampling2D(size=image_size,interpolation="nearest")(e)
128
129     x=layers.Conv2D(widths[0],kernel_size=1)(noisy_images)
130     x=layers.Concatenate()([x,e])
131
132     skips=[]

```

```

133     for width in widths[:-1]:
134         x=DownBlock(width,block_depth)([x,skips])
135     for _ in range(block_depth):
136         x=ResidualBlock(widths[-1])(x)
137     for width in reversed(widths[:-1]):
138         x=UpBlock(width,block_depth)([x,skips])
139     x=layers.Conv2D(3,kernel_size=1,kernel_initializer="zeros")(x)
140
141     return keras.Model([noisy_images,noise_variances],x,name="residual_unet")
142
143 class DiffusionModel(keras.Model): # 확산 모델을 위한 클래스
144     def __init__(self,image_size,widths,block_depth):
145         super().__init__()
146         self.normalizer=layers.Normalization()
147         self.network=get_network(image_size,widths,block_depth) # denoise용 U-net
148         self.ema_network=keras.models.clone_model(self.network) # KID용 U-net
149
150     def compile(self, **kwargs):
151         super().compile(**kwargs)
152         self.noise_loss_tracker = keras.metrics.Mean(name="n_loss")
153         self.image_loss_tracker = keras.metrics.Mean(name="i_loss")
154         self.kid = KID(name="kid")
155
156     @property
157     def metrics(self):
158         return [self.noise_loss_tracker, self.image_loss_tracker, self.kid]
159
160     def denormalize(self, images): # 화소 값을 [0,1] 사이로 역변환
161         images=self.normalizer.mean+images*self.normalizer.variance**0.5
162         return tf.clip_by_value(images,0.0,1.0)
163
164     def diffusion_schedule(self, diffusion_times):
165         start_angle=tf.acos(max_signal_rate) # 확산 시간을 각도로 변환
166         end_angle=tf.acos(min_signal_rate)
167         diffusion_angles=start_angle+diffusion_times*(end_angle-start_angle)
168
169         signal_rates=tf.cos(diffusion_angles) # signal_rates와
170         noise_rates=tf.sin(diffusion_angles) # noise_rates의 제곱 합은 1
171         return noise_rates,signal_rates

```

```

172
173     def denoise(self, noisy_images, noise_rates, signal_rates, training):
174         if training: network=self.network # 학습할 때 쓰는 신경망
175         else: network=self.ema_network # KID 평가할 때 쓰는 신경망
176
177         pred_noises=network([noisy_images,noise_rates**2],training=training)
178         pred_images=(noisy_images-noise_rates*pred_noises)/signal_rates
179         return pred_noises,pred_images
180
181     def reverse_diffusion(self,initial_noise,diffusion_steps):
182         num_images=initial_noise.shape[0]
183         step_size=1.0/diffusion_steps
184
185         next_noisy_images=initial_noise
186         for step in range(diffusion_steps):
187             noisy_images = next_noisy_images
188
189             diffusion_times=tf.ones((num_images,1,1,1))-step*step_size
190             noise_rates,signal_rates=self.diffusion_schedule(diffusion_times)
191
192             pred_noises,pred_images=self.denoise(noisy_images,noise_rates,signal_rates,training=False)
193
194             next_diffusion_times=diffusion_times-step_size
195             next_noise_rates,next_signal_rates=self.diffusion_schedule(next_diffusion_times)
196
197             next_noisy_images=(next_signal_rates*pred_images+next_noise_rates*pred_noises)
198
199         return pred_images
200
201     def generate(self,num_images,diffusion_steps):
202         initial_noise=tf.random.normal(shape=(num_images,img_siz,img_siz,3))
203         generated_images=self.reverse_diffusion(initial_noise,diffusion_steps) # 역확산
204         generated_images=self.denormalize(generated_images) # 역정규화
205         return generated_images
206
207     def train_step(self, images):
208         images=self.normalizer(images,training=True) # 정규화
209         noises=tf.random.normal(shape=(batch_siz,img_siz,img_siz,3)) # 잡음

```



```

209         diffusion_times=tf.random.uniform(shape=(batch_siz,1,1,1),minval=0.0,maxval=1.0)
210         noise_rates,signal_rates=self.diffusion_schedule(diffusion_times)
211         noisy_images=signal_rates*images+noise_rates*noises # 확산 스케줄에 따라 잡음과
영상 혼합
212
213         with tf.GradientTape() as tape: # denoise로 잡음과 영상 분리하고 손실 계산
214
215             pred_noises,pred_images=self.denoise(noisy_images,noise_rates,signal_rates,training=True)
216             noise_loss=self.loss(noises,pred_noises) # 학습에 사용하는 손실
217             image_loss=self.loss(images,pred_images) # 평가에 사용하는 손실
218
219             gradients=tape.gradient(noise_loss,self.network.trainable_weights)
220             self.optimizer.apply_gradients(zip(gradients,self.network.trainable_weights))
221             self.noise_loss_tracker.update_state(noise_loss)
222             self.image_loss_tracker.update_state(image_loss)
223
224             for weight,ema_weight in zip(self.network.weights,self.ema_network.weights):
225                 ema_weight.assign(0.999*ema_weight+(1-0.999)*weight) # 가중치의 EMA 추적
226
227         return {m.name:m.result() for m in self.metrics[:-1]}
228
229     def test_step(self, images):
230         images=self.normalizer(images,training=False)
231         noises=tf.random.normal(shape=(batch_siz,img_siz,img_siz,3))
232         diffusion_times=tf.random.uniform(shape=(batch_siz,1,1,1),minval=0.0,maxval=1.0)
233         noise_rates,signal_rates=self.diffusion_schedule(diffusion_times)
234         noisy_images=signal_rates*images+noise_rates*noises
235
236         pred_noises,pred_images=self.denoise(noisy_images,noise_rates,signal_rates,training=False)
237         noise_loss=self.loss(noises,pred_noises)
238         image_loss=self.loss(images,pred_images)
239         self.image_loss_tracker.update_state(image_loss)
240         self.noise_loss_tracker.update_state(noise_loss)
241
242         images=self.denormalize(images)
243
244         generated_images=self.generate(num_images=batch_siz,diffusion_steps=kid_diffusion_steps)
245         self.kid.update_state(images, generated_images)

```

```

244         return {m.name: m.result() for m in self.metrics}
245
246     def plot_images(self, epoch=None, logs=None, num_rows=1, num_cols=8): # 영상 생성하고
        그리기
247
        generated_images=self.generate(num_images=num_rows*num_cols,diffusion_steps=plot_diffusi
        on_steps)
248
249         plt.figure(figsize=(num_cols*2.0,num_rows*2.0))
250         for row in range(num_rows):
251             for col in range(num_cols):
252                 index=row*num_cols+col
253                 plt.subplot(num_rows,num_cols,index+1)
254                 plt.imshow(generated_images[index])
255                 plt.axis("off")
256         plt.tight_layout()
257         plt.show()
258         plt.close()
259
260 model=DiffusionModel(img_siz, widths, block_depth) # 모델 생성
261
262 cp_path="checkpoints/diffusion_model" # 체크포인트: 최고 모델 저장(KID 메트릭 사용)
263 cp_callback=tf.keras.callbacks.ModelCheckpoint(filepath=cp_path,save_weights_only=True,moni
        tor="val_kid",mode="min",save_best_only=True)
264
265 model.normalizer.adapt(train_dataset) # 데이터 정규화에 쓸 값 추정하고 저장
266
267 model.compile(optimizer=tf.keras.optimizers.experimental.AdamW(learning_rate=1e-
        3,weight_decay=1e-4),loss=keras.losses.mean_absolute_error)
268 model.fit(train_dataset,epochs=50,validation_data=val_dataset,callbacks=[keras.callbacks.Lambd
        aCallback(on_epoch_end=model.plot_images),cp_callback])
269
270 model.load_weights(cp_path) # 추론:체크포인트로 저장해둔 모델 불러와 영상 생성
271 model.plot_images()

```

Epoch 1/50

511/511 [=====] - 166s 293ms/step - n_loss: 0.2129 -
i_loss: 0.4175 - val_n_loss: 0.7921 - val_i_loss: 2.4438 - val_kid: 2.0365

Epoch 2/50

511/511 [=====] - 143s 280ms/step - n_loss: 0.1692 -
i_loss: 0.2878 - val_n_loss: 0.7296 - val_i_loss: 2.2532 - val_kid: 1.9091

... ..

Epoch 5/50

511/511 [=====] - 141s 276ms/step - n_loss: 0.1559 -
i_loss: 0.2596 - val_n_loss: 0.3400 - val_i_loss: 0.9053 - val_kid: 1.5204

... ..

Epoch 10/50

511/511 [=====] - 142s 279ms/step - n_loss: 0.1501 -
i_loss: 0.2492 - val_n_loss: 0.1518 - val_i_loss: 0.2588 - val_kid: 0.8640

... ..

Epoch 20/50

511/511 [=====] - 141s 276ms/step - n_loss: 0.1462 -
i_loss: 0.2408 - val_n_loss: 0.1429 - val_i_loss: 0.2340 - val_kid: 0.2382

... ..

Epoch 30/50

511/511 [=====] - 142s 278ms/step - n_loss: 0.1446 -
i_loss: 0.2364 - val_n_loss: 0.1413 - val_i_loss: 0.2345 - val_kid: 0.2136

... ..

Epoch 40/50

511/511 [=====] - 141s 276ms/step - n_loss: 0.1432 -
i_loss: 0.2353 - val_n_loss: 0.1418 - val_i_loss: 0.2303 - val_kid: 0.1918

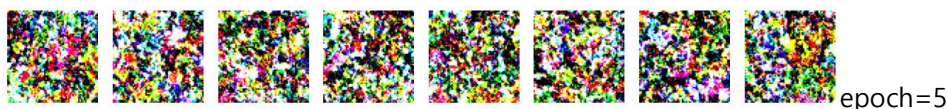
... ..

Epoch 50/50

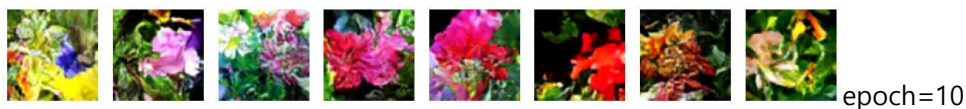
511/511 [=====] - 141s 276ms/step - n_loss: 0.1435 -
i_loss: 0.2324 - val_n_loss: 0.1421 - val_i_loss: 0.2287 - val_kid: 0.1976

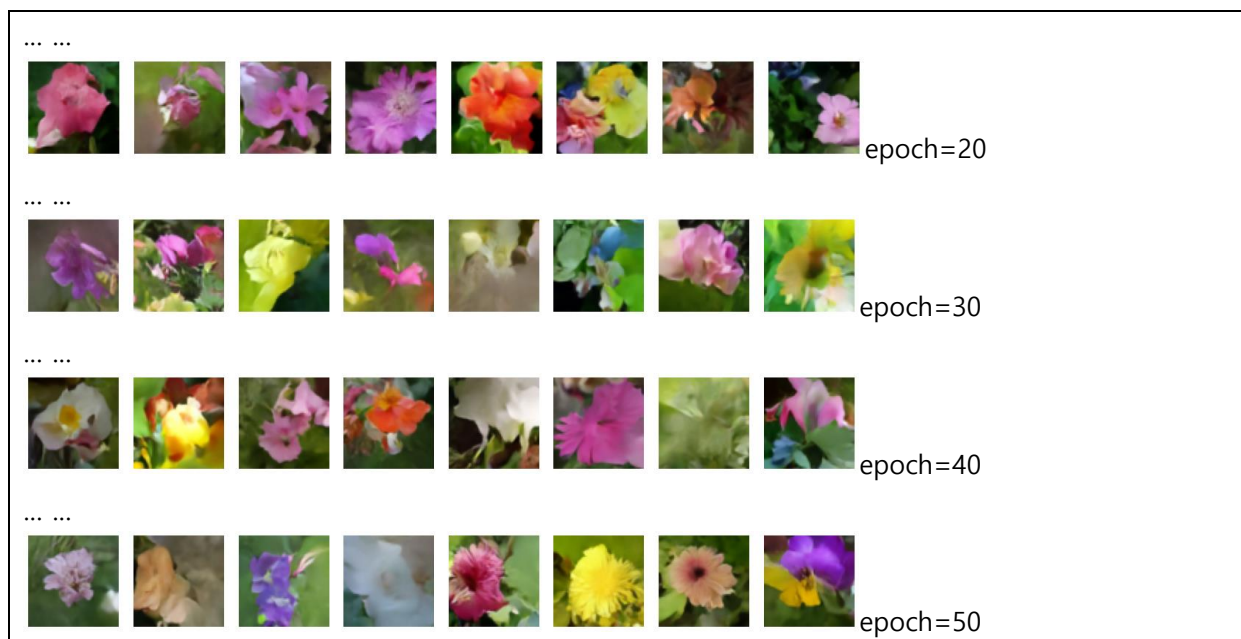


... ..



... ..





D.2 사전 학습 모델로 추론

[프로그램 D-1]은 학습하는데 많은 시간이 소요된다. NVIDIA GeForce RTX 3060의 GPU가 장착된 PC에서 세대 당 280초 가량 소요되므로 50세대를 학습하는데 4시간가량 걸린다. CPU만 사용하는 경우 훨씬 많은 시간이 걸린다. [프로그램 D-1]은 학습하는 도중에 발생한 가장 좋은 모델을 체크포인트 폴더에 저장해 두었다. [프로그램 D-2]는 이렇게 사전 학습된 모델을 불러다가 추론, 즉 새로운 영상을 생성한다.

01~258행은 [프로그램 D-1]과 같다. 메인에 해당하는 260~266행을 살펴보면, 학습을 담당하는 compile과 fit 함수를 수행하는 행을 삭제했다. 260행은 DiffusionModel 클래스로 새로운 모델을 만들어 model 객체에 저장한다. 262행은 체크포인트 경로를 설정하고, 263행은 훈련 집합에서 데이터 정규화에 쓸 값을 추정하고 신경망 모델에 기록해둔다.

265행은 체크포인트 경로에서 모델의 가중치를 읽어온다. 266행은 plot_image 함수를 이용하여 새로운 영상을 생성한다. 이때 num_rows와 num_cols를 10과 8로 설정하여 80개 영상을 생성하여 10*8 배열로 출력한다.

프로그램 실행 결과를 보면, 80개 영상을 확인할 수 있다. 실제 꽃 영상과 구별하기 어려울 정도의 품질이 높은 영상이 여럿 생성되었다.

[프로그램 D-2] 확산 모델을 이용한 추론
01~258행은 [프로그램 D-1]과 같음
259

```

260 model=DiffusionModel(img_siz, widths, block_depth) # 모델 생성
261
262 cp_path="checkpoints/diffusion_model" # 체크포인트: 최고 모델 저장(KID 메트릭 사용)
263 model.normalizer.adapt(train_dataset) # 데이터 정규화에 쓸 값 추정하고 저장
264
265 model.load_weights(cp_path) # 추론:체크포인트로 저장해둔 모델 불러와 영상 생성
266 model.plot_images(num_rows=10,num_cols=8)

```



참고 문헌

[Ho2020] J. Ho, A. Jain, and P. Abbeed, "Denoising diffusion probabilistic models," arXiv:2006.11239v2 (NeurIPS 2020).

[Nilsback2008] M Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," Indian Conference on Computer Vision, Graphics, and Image Processing.

[Song2020] J. Song, C. Meng, and S. Ermon, "Denoising diffusion implicit models," arXiv:2010.02502v4.