# Convolutional GAN과 Recurrent VAE를 이용한 편곡 프로그램

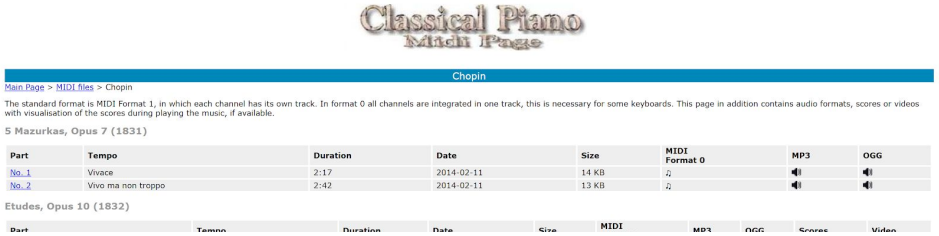
IoT · 인공지능 · 빅데이터의 실무응용
전기정보공학부 김진호

## I. 프로젝트 개요

인공지능은 다양한 분야에서 사람들이 직접 해오던 일을 대체하고 있지만, 창의성이 요구되는 일은 인공지능이 흉내낼 수 없는 고유한 영역으로 여겨져왔다. 하지만 컴퓨팅과 알고리즘 이론의 발전은 컴퓨터가 예술작품의 창작도 가능하게 만들었다. 특히 GAN의 등장은 컴퓨터가 학습된 데이터를 기반으로 입력 데이터를 원하는 형태로 바꾸거나 새로운 데이터를 생성하는 것을 가능하게 하였다. 이중에서 음악과 관련한 모델들도 많이 제시되어 왔는데[1], 본 프로젝트는 창작임무를 수행하는 인공지능을 직접 구현해보며 일반적으로 쓰이는 구조들이 창의적인 임무를 수행하는 데에도 문제가 없는지 알아보고자 하였다. 모델은 크게 두 부분으로 나누어 구성하였으며, MIDI 데이터를 학습하여 특정 음악가의 특성을 모방하는 GAN 모델과 두 가지의 멜로디를 합성하는 VAE 모델을 적절하게 사용하여 음악을 편곡하는 프로그램을 만들고자 하였다. GAN은 2D convolution을 이용하였고, VAE에서는 LSTM을 이용하였다. 곡의 멜로디가 모델의 인풋으로 주어지며, 이를 편곡한 멜로디가 아웃풋으로 주어진다. 학습에는 쇼팽 피아노곡들의 오른손 악보가 이용됐다.

## II. 데이터

학습데이터로는 Classical Piano Midi 사이트에서 제공하는 쇼팽의 음악들을 이용하였으며, 단기간의 프로젝트임을 고려하여 멜로디만을 효과적으로 학습할 수 있도록 이중에서 피아노 오른손 악보만 구분하여 학습시켰다.




〈데이터 출처 페이지 화면[2]〉

[1] Fernández, J. D., Vico, F. “AI Methods in Algorithmic Composition: A Comprehensive Survey”. Journal of Artificial Intelligence Research 48: 513-582, 2013

[2] Classical Piano Midi Page - Chopin (piano-midi.de)

MIDI 파일을 텐서플로우 모델이 학습할 수 있는 형태로 변환하는 데에는 Pretty Midi 라이브러리[3]의 get_piano_roll 함수를 사용하였다. 이 함수는 100분의 1초 마다 MIDI 형식에서 message 데이터를 추출하여 numpy array로 변환하여 준다. 여기서 각 원소들의 값은 'velocity'라고 하는데, 이는 메시지가 의미하는 음정이나 페달 등이 연주되는 세기를 나타낸다.

```
def get_piano_roll(self, fs=100, times=None, pedal_threshold=64):
    """Compute a piano roll matrix of the MIDI data.

    Parameters
    ----------
    fs : int
        Sampling frequency of the columns, i.e. each column is spaced apart
        by ``1./fs`` seconds.
    times : np.ndarray
        Times of the start of each column in the piano roll.
        Default ``None`` which is ``np.arange(0, get_end_time(), 1./fs)``.
    pedal_threshold : int
        Value of control change 64 (sustain pedal) message that is less
        than this value is reflected as pedal-off.  Pedals will be
        reflected as elongation of notes in the piano roll.
        If None, then CC64 message is ignored.
        Default is 64.

    Returns
    -------
    piano_roll : np.ndarray, shape=(128,times.shape[0])
        Piano roll of MIDI data, flattened across instruments.

    # If there are no instruments, return an empty array
    if len(self.instruments) == 0:
        return np.zeros((128, 0))

    # Get piano rolls for each instrument
    piano_rolls = [i.get_piano_roll(fs=fs, times=times,
                                    pedal_threshold=pedal_threshold)
                   for i in self.instruments]
```

[3] Pretty-MIDI documentation (craffel.github.io)

```
    # Allocate piano roll,
    # number of columns is max of # of columns in all piano rolls
    piano_roll = np.zeros((128, np.max([p.shape[1] for p in piano_rolls])))
    # Sum each piano roll into the aggregate piano roll
    for roll in piano_rolls:
        piano_roll[:, :roll.shape[1]] += roll
    return piano_roll
```

<get_piano_roll 소스코드[4]>

```
def roll_to_array(roll, size = 400): # 2차원 자료로 만들어준다
    roll = np.transpose(roll)
    ary = []
    for i in range (0,len(roll), size):
        tmp = []
        for k in range(size):
            if i+k < len(roll):
                tmp.append(roll[i+k])
            else:
                tmp.append([0] * len(roll[0]))
        ary.append(tmp)

    ary = np.asarray(ary)
    ary = np.expand_dims(ary, axis = 3)
    return ary
```

```
for i in tqdm(range(num_files)):
    train_mid[i] = pretty_midi.PrettyMIDI(trainset[i])
    num_inst = len(train_mid[i].instruments)
    inst_0 = train_mid[i].instruments[0]
    roll_0 = inst_0.get_piano_roll(fs = 24)
    merged_0 = np.append(merged_0, roll_0, axis = 1)
    data_0 = roll_to_array(merged_0[0:520], size = 92) #예시 코드에서는

data_norm = data_0/np.max(data_0)
```

<데이터 전처리 코드>

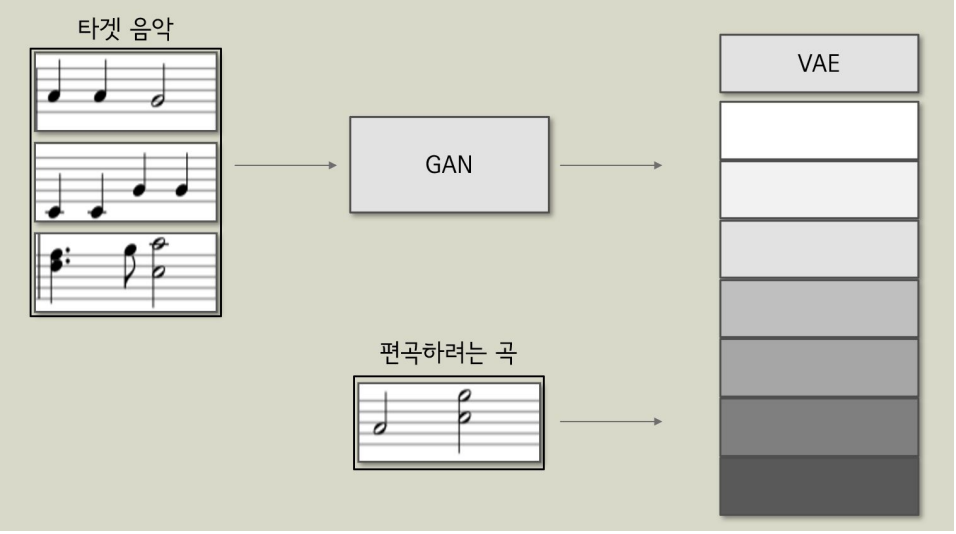
이 프로젝트에서는 쇼팽의 곡 약 40개를 하나의 긴 어레이로 바꾸고, 그것을 다시 길이가 한 마디 정도가 되도록 설정한 배치 사이즈로 나누어 학습을 진행하였다. 모델 학습에서는 메모리 부족으로 인하여 샘플링 주기를 moderato 기준으로 16분음표의 길이에 맞출 수 있도록 24로 줄여 데이터의 크기를 최소화하였다. 또한, 각 어레이 안에 들어있는 각 메시지는 특정 시간축에 연주가 되었느냐 안 되었느냐의 논리 자료형이

[4] GitHub - craffel/pretty-midi

아니라 어느 정도의 강도로 연주 되었는가 하는 0 부터 127의 정수 자료형이기 때문에 sigmoid 활성화 함수와 잘 맞도록 노멀라이징 과정도 거칠 필요가 있었다.

## III. 모델

모델은 먼저 GAN을 이용하여 학습시키고자 하는 음악적 특성을 하나의 곡으로 압축하여 만들고, 그것을 편곡하고자 하는 곡과 함께 VAE에 입력으로 주어 두 곡을 합성하는 방식으로 모델을 구성하였다. 이를 통하여 GAN의 출력에 포함된 음악적 특성이 VAE를 통해 편곡하려는 곡에 녹아들어갈 수 있을 것으로 기대하였다. 또한 GAN과 VAE를 둘 다 활용하여 GAN만을 이용하는 경우보다 원곡의 멜로디를 유지하기 용이하고, VAE만을 이용하는 경우보다 특정 음악가의 음악색을 더 잘 나타낼 수 있도록 하였다.




〈간략한 모델 구조〉

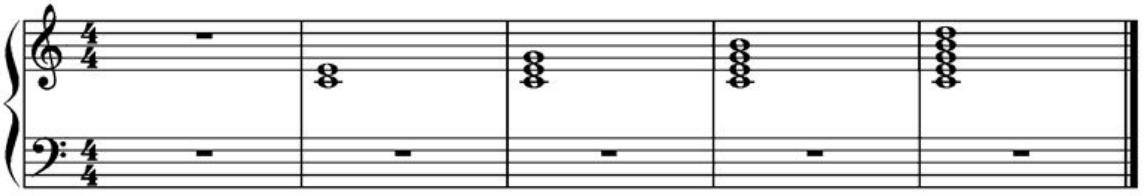
- **GAN 모델**

음악은 주로 기반이 되는 코드 안에서 음들을 조합하고, 또 시간에 따라 그 기반이 되는 코드를 변화시키며 진행된다. 따라서 곡을 효과적으로 학습시키기 위해서는 동시에, 또는 시간차로 나타나는 코드를 모델이 인식할 수 있도록 설계하는 것이 중요하다. 본 프로젝트는 단위 시간 내에 음정의 분포를 일종의 그림으로 생각하고 시각 데이터를 처리하기 위해 많이 쓰이는 Convolution layer를 사용하였다.

```
k_1 = 6
k_2 = 12
k_3 = 6

decode_h = input_shape[0] - (k_1 - 1) - (k_2 - 1) - (k_3 - 1)
decode_w = input_shape[1] - (k_1 - 1) - (k_2 - 1) - (k_3 - 1)
decode_input = decode_h * decode_w

gen_model = Sequential([
                    Dense(decode_input, activation = 'relu', name = 'gen_dense'),
                    Reshape(target_shape = (decode_h, decode_w, 1)),
                    Conv2DTranspose(filters = 16, kernel_size = k_3, activation = 'relu'),
                    Conv2DTranspose(filters = 24, kernel_size = k_2, activation = 'relu'),
                    Conv2DTranspose(filters = 1, kernel_size = k_1, activation = 'sigmoid')
])

disc_model = Sequential([
                    Conv2D(filters = 24, kernel_size = k_1, input_shape = input_shape, activation = 'relu'),
                    Conv2D(filters = 16, kernel_size = k_2, activation = 'relu'),
                    Conv2D(filters = 8, kernel_size = k_3, activation = 'sigmoid'),
                    Flatten(),
                    Dense(1)
])
```

<Generator와 Discriminator model 코드>

처음 필터는 피아노곡에서 가장 흔하게 연주되는 반음 6개 이내에서 이루어지는 화음을 학습할 수 있도록 크기를 6으로 하였으며, 두 번째 필터는 7화음과 같이 한 옥타브 이내에서의 화음 구성을 학습할 수 있도록 크기를 12로 하였다. 그리고 세 번째 필터에서는 다시 크기 6의 필터로 화음을 가다듬을 수 있도록 해주었다. 피아노곡에서는 9화음과 같이 한 옥타브 보다 더 차이나는 음을 연주하는 것이 흔하지 않기 때문에 배제하고 학습을 진행하였다.



<왼쪽에서부터 2화음, 3화음, 7화음, 9화음>

```python
crossentropy = tf.keras.losses.BinaryCrossentropy(from_logits = True)

def discriminator_loss(real, fake):
    real_loss = crossentropy(tf.ones_like(real), real)
    fake_loss = crossentropy(tf.zeros_like(fake), fake)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake):
    return crossentropy(tf.ones_like(fake), fake)

from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint

epochs = 2
learning_rate = 1e-4

gen_optimizer = Adam(lr = learning_rate)
disc_optimizer = Adam(lr = learning_rate)
```

〈Loss와 Optimizer 설정 코드〉

Discriminator의 loss로는 crossentropy를 이용하였고, generator에 의해 생성된 piano roll 어레이(코드에서 fake)와 실제 쇼팽 음악 데이터(코드에서 real)의 어레이를 구분하도록 학습하였다. Generator는 무작위 노이즈를 입력 데이터로 받아 piano roll 출력을 만들어내고, discriminator가 이 '가짜' 데이터를 '진짜' 데이터로 인식하도록 하는 것을 목표로 설정하였다.

Optimizer로는 Adam을 이용하였으며, decay를 적용하는 것이 학습에 유의미한 진전을 주지 않는 것으로 보여 폐기하고 Generator와 Discriminator 모두 $10^{-4}$ 의 고정된 learning rate로 학습을 진행하였다.

```python
def train(dataset, epochs):
    for epoch in range(epochs):

        for batch in tqdm(dataset):
            train_step(batch)

        print('  {}/{} epochs'.format(epoch+1, epochs))
```

```
[ ]  train(train_x, epochs)

100%|██████████| 185/185 [23:05<00:00,  7.49s/it]
  0%|          | 0/185 [00:00<?, ?it/s]  1/20 epochs
100%|██████████| 185/185 [23:03<00:00,  7.48s/it]
  0%|          | 0/185 [00:00<?, ?it/s]  2/20 epochs
100%|██████████| 185/185 [23:10<00:00,  7.51s/it]
  0%|          | 0/185 [00:00<?, ?it/s]  3/20 epochs
100%|██████████| 185/185 [23:11<00:00,  7.52s/it]
  0%|          | 0/185 [00:00<?, ?it/s]  4/20 epochs
100%|██████████| 185/185 [23:11<00:00,  7.52s/it]
  0%|          | 0/185 [00:00<?, ?it/s]  5/20 epochs
100%|██████████| 185/185 [23:14<00:00,  7.54s/it]
  0%|          | 0/185 [00:00<?, ?it/s]  6/20 epochs
```

〈Train 진행화면〉

트레이닝은 GPU에서 진행되었으며, 1회에 약 25분 정도 소요되었다. 하지만 메모리 부족으로 인한 브라우저 오류 때문에 15회 이상의 학습을 시도하면 서버와의 연결이 종료되는 문제가 있었다. 따라서 학습은 10회씩 끊어서 진행되었으며, 이로 인하여 프로그램 기간 동안 100회 정도 밖에 학습시키지 못한 모델만을 얻을 수 있었다. 이는 논문으로 발표된 다른 모델들의 2만회에 달하는 학습 횟수에 비하여 매우 적은 양이기 때문에 본 프로젝트를 통하여 얻은 결과는 과소적합 상태일 것으로 보인다.

트레이닝 결과를 확인하기 위해 GAN의 generator 출력을 다시 MIDI 파일로 변환하여 결과를 확인할 수 있도록 하는 코드다. 먼저 batch마다 끊겨있는 데이터들을 다시 하나의 긴 어레이로 합치고, 데이터의 차원 수를 줄여 piano roll 형식으로 바꾸어주었다. 여기서 얻은 piano roll 데이터를 MIDI 파일로 변환하는 코드는 Pretty-midi 깃허브의 PR을 참고하여 작성하였다.

```
70  def array_to_roll(array):
71      array = np.squeeze(array)
72      result = []
73      for i in range(len(array)):
74          for k in range(len(array[0])):
75                  result.append(array[i][k])
76      result = np.asarray(result)
77      result = np.transpose(result)
```

〈출력의 Batch dimension을 제거해주는 코드〉

```
19  def piano_roll_to_pretty_midi(piano_roll, fs=100, program=0):
20      '''Convert a Piano Roll array into a PrettyMidi object
21      with a single instrument.
22      Parameters
23      ----------
24      piano_roll : np.ndarray, shape=(128,frames), dtype=int
25          Piano roll of one instrument
26      fs : int
27          Sampling frequency of the columns, i.e. each column is spaced apart
28          by ``1./fs`` seconds.
29      program : int
30          The program number of the instrument.
31      Returns
32      -------
33      midi_object : pretty_midi.PrettyMIDI
34          A pretty_midi.PrettyMIDI class instance describing
35          the piano roll.
```

```
37      notes, frames = piano_roll.shape
38      pm = pretty_midi.PrettyMIDI()
39      instrument = pretty_midi.Instrument(program=program)
40
41      # pad 1 column of zeros so we can acknowledge inital and ending events
42      piano_roll = np.pad(piano_roll, [(0, 0), (1, 1)], 'constant')
```

```
    # use changes in velocities to find note on / note off events
    velocity_changes = np.nonzero(np.diff(piano_roll).T)

    # keep track on velocities and note on times
    prev_velocities = np.zeros(notes, dtype=int)
    note_on_time = np.zeros(notes)
```

```
    for time, note in zip(*velocity_changes):
        # use time + 1 because of padding above
        velocity = piano_roll[note, time + 1]
        time = time / fs
        if velocity > 0:
            if prev_velocities[note] == 0:
                note_on_time[note] = time
                prev_velocities[note] = velocity
        else:
            pm_note = pretty_midi.Note(
                velocity=prev_velocities[note],
                pitch=note,
                start=note_on_time[note],
                end=time)
            instrument.notes.append(pm_note)
            prev_velocities[note] = 0
    pm.instruments.append(instrument)
    return pm
```

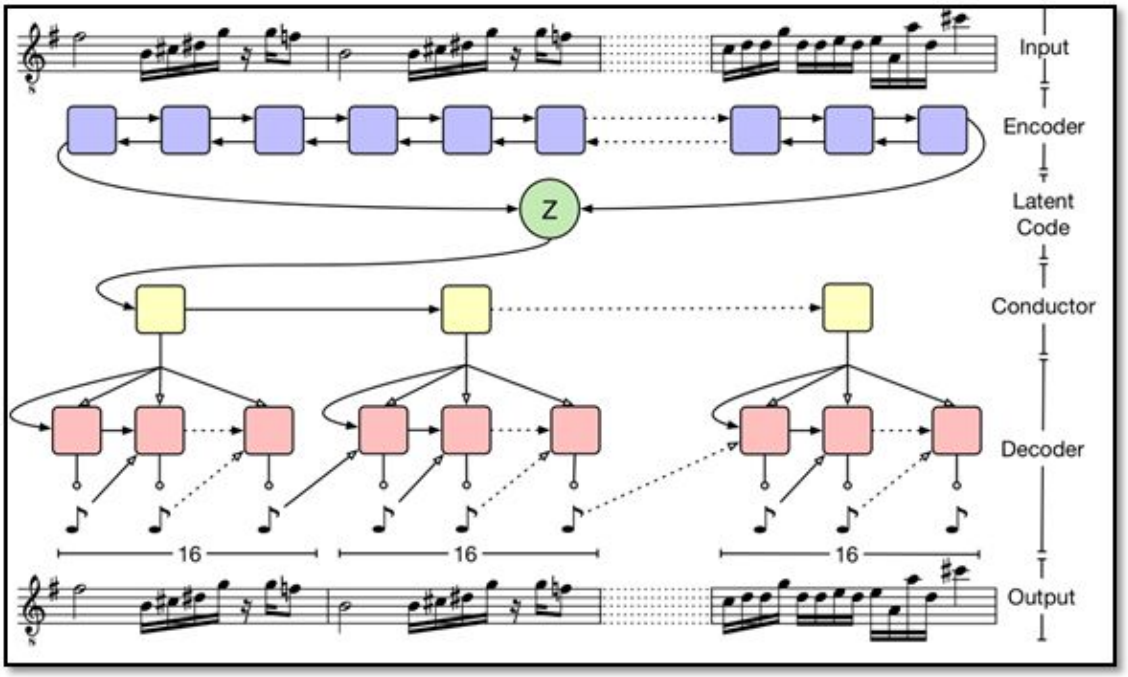
〈Piano roll을 다시 MIDI 파일로 변환하는 코드[5]〉

- **VAE 모델**

VAE 모델은 A. Roberts et. al.의 MusicVAE 모델[6]을 이용하였다. MusicVAE는 RNN을 사용하는 모델로, 인코더 모델은 양방향 LSTM 네트워크로 이루어져 있고, 디코딩 단계는 컨덕터와 디코더 두 개의 모델로 분리되어 각각 LSTM 네트워크로 이루어져 있다. 디코딩 단계가 이처럼 두 단계로 구분된 이유는 RNN을 이용하는 VAE 모델들에서 잠재벡터값을 무시하는 방향으로 학습이 진행되는 posterior collapse 문제가 도드라지게 나타나기 때문에 이를 보완하기 위해서다. 디코더가 잠재공간(latent space)에서 입력을 받기 전에 별개의 단방향 RNN 네트워크를 가진 컨덕터를 거치게 하여 잠재공간의 정보를 제대로 재현하지 못하는 RNN 기반 VAE의 한계를 극복하고자 하였다.

[5] piano roll to pretty_midi object util by jsleep · Pull Request #129 · craffel/pretty-midi · GitHub

[6] A. Roberts et. al. “A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music”. Proceedings of the 35th International Conference on Machine Learning, PMLR 80:4364-4373, 2018.




〈MusicVAE 모델 구조도〉

```
CONFIG_MAP['hier-mel_16bar'] = Config(
    model=MusicVAE(
        lstm_models.HierarchicalLstmEncoder(
            lstm_models.BidirectionalLstmEncoder, [16, 16]),
        lstm_models.HierarchicalLstmDecoder(
            lstm_models.CategoricalLstmDecoder(),
            level_lengths=[16, 16],
            disable_autoregression=True)),
    hparams=merge_hparams(
        lstm_models.get_default_hparams(),
        HParams(
            batch_size=512,
            max_seq_len=256,
            z_size=512,
            enc_rnn_size=[1024],
            dec_rnn_size=[1024, 1024],
            free_bits=256,
            max_beta=0.2,
        )),
    note_sequence_augmenter=None,
    data_converter=mel_16bar_converter,
    train_examples_path=None,
    eval_examples_path=None,
)
```

〈MusicVAE 구성〉

## IV. 결과

- **MIDI 파일 생성**

GAN 모델의 출력 레이어의 활성화 함수는 sigmoid로 설정되어 있기 때문에 0 부터 128 사이의 정수값을 가질 수 있도록 재설정해주어야 한다. 따라서 출력으로 얻은 텐서값들을 numpy 어레이로 변환하고 어레이 원소들을 정수로 바꾸어주는 코드를 추가하였다.

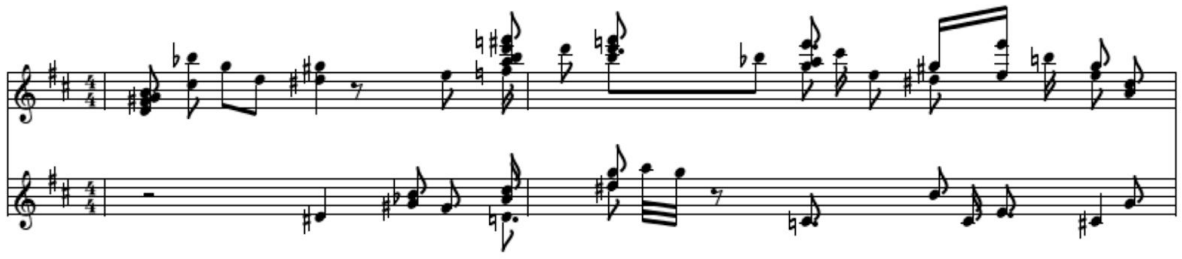
```
# load saved model

from tensorflow.keras.models import model_from_json

json_file = open("MuseConv_chpn_gen_v2", "r") # in GPU
#json_file = open("/content/drive/MyDrive/ColabNotebooks/models/MuseConv_chpn_
#json_file = open("/home/student/IAB/models/MuseConv_chpn_gen_v1", "r")

test_json = json_file.read()
json_file.close()

model_gen = model_from_json(test_json)
# model_gen.load_weights("/content/drive/MyDrive/ColabNotebooks/models/MuseCon
#model_gen.load_weights("/home/student/IAB/models/MuseConv_chpn_gen_v1.h5")
model_gen.load_weights("MuseConv_chpn_gen_v2.h5")
```

```
[ ]
pred = model_gen(test_x)
pred = pred.numpy()
pred = pred.astype(np.int64)
```

〈MIDI 파일 생성 코드〉

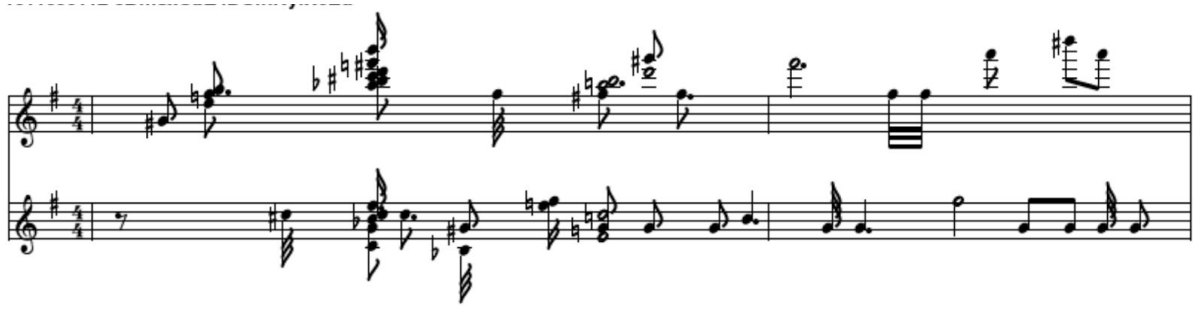
- **GAN 모델의 MIDI 출력**

쇼팽의 피아노곡 약 40개를 학습시킨 다음 GAN 모델에서 얻어낸 MIDI 파일을 다시 악보로 변환[7]한 것이다.



〈15 epochs〉

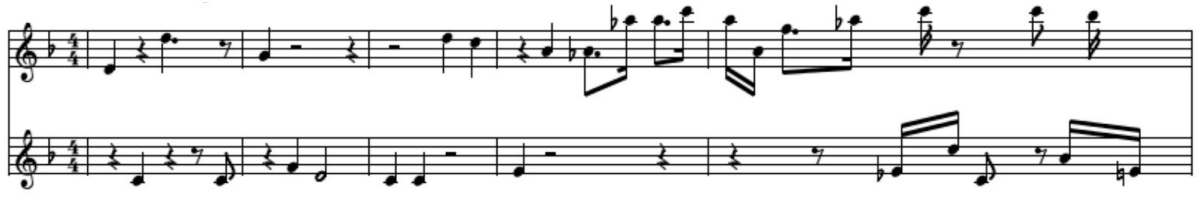
[7] https://solmire.com/ 이용



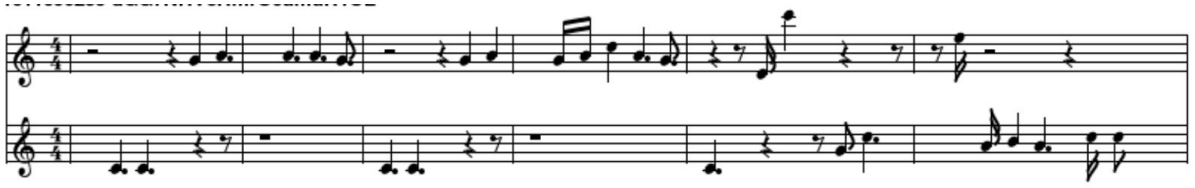
〈100 epochs〉

- **VAE 모델의 MIDI 출력**

작은별 변주곡의 오른손 악보와 GAN 모델의 출력을 VAE 모델의 입력값으로 주고 결과의 앞부분을 악보로 나타낸 것이다.



〈15 epochs〉



〈100 epochs〉

## V. 결론

GAN 모델 학습은 진행이 되고 있는 것으로 보이나 음악 정보의 특성상 많은 반복 학습을 요구하기 때문에 수백 정도의 반복으로는 과소적합 상태를 벗어나지 못할 것이다. 비록 다른 모델이라 최적의 비교대상이 되지는 못하지만, VAE의 경우 약 30만 step 정도의 학습을 진행한 반면 GAN은 100의 epoch을 진행하였을 때 2만 step 정도의 학습 밖에 이루어지지 않는다. GAN을 이용하여 적절한 결과를 내기 위하여서는 본 프로젝트에서 진행한 것 보다 적어도 10배 정도 많은 학습을 하여야 할 것으로 예측된다.

또한 CNN 기반 모델은 곡을 전체적으로 보지는 않기 때문에 음악 자료가 가지고 있는 복잡한 정보들을 다 학습시키기 어렵다는 문제가 있다. CNN 기반의 GAN만을 이용할 경우 학습이 진행되더라도 알고리즘이 구간구간 등장하는 기교를 학습하여 이를 반영할 수는 있지만, 곡이 진행하며 보이는 코드의 흐름이나 분위기를 재현하기는 어려울 수 있다. 코드 진행만을 따로 학습할 수 있는 레이어를 추가하는 방식이 이 문제를 보완할 수 있을 것으로 보인다.

## V. 참고문헌

A. Roberts et. al. “A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music”. Proceedings of the 35th International Conference on Machine Learning, PMLR 80:4364-4373, 2018.

Fernández, J. D., Vico, F. “AI Methods in Algorithmic Composition: A Comprehensive Survey”. Journal of Artificial Intelligence Research 48: 513-582, 2013