МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по Индивидуальному домашнему заданию
по дисциплине «Искусственные нейронные сети»
Тема: «Midi Music Generator»

Студент гр. 7381 _____ Минуллин М.А.
Студент гр. 7381 _____ Лукашев Р.С.
Студентка гр. 7381 _____ Машина Ю.Д.
Преподаватель _____ Жангиров Т.Р

Санкт-Петербург
2020

Цель работы.

В рамках выполнения ИДЗ необходимо разработать и реализовать модель ИНС решающую определенную задачу на заданном датасете.

Задачи.

Задачей данной работы является реализация генератора мелодий в формате Midi с использованием нейронной сети.

Требования.

- Модель должна быть разработана на языке Python с использованием Keras API
- Исходный код проекта должен быть в формате PEP8
- В исходном коде должны быть поясняющие комментарии
- Модель не должна быть избыточной (должен соблюдаться баланс между размером сети [кол-во слоев и нейронов] и качеством выдаваемого результата)
- Обучение модели должно быть стабильно (для предложенной архитектуры ИНС обучение должно приводить к примерно одним и тем же результатом, то есть не должно быть такого, что при обучении 10 сетей удовлетворительный результат дают только 5 из них)
- *Плюсом будет анализ с использованием callback'а TensorBoard*
- *Плюсом будет разработка собственных callback'ов*
- *Плюсом будет создание модели из ансамбля ИНС*

Распределение ролей.

Минуллин Михаил – настройка модели для обучения на GPU и тренировка сети. Подготовка данных для обучения.

Лукашев Роман – подбор датасета и архитектуры сети.

Машина Юлия – тестирование архитектур и написание генератора.

Описание датасета.

Датасет сонат Моцарта на фортепиано в midi формате. Особенностью классической музыки является в общем отсутствие четко выраженной структуры, т.е. у нее нет куплетов, припевов и т.д. Классическая музыка по сути является длинной мелодией. Было решено объединить все мелодии в один файл, и тренировать сеть на этом большом файле.

Ход работы.

Для решения задачи необходимо построить генеративную модель, учитывающую предыдущую историю (предыдущие ноты), поскольку иначе не получится мелодии. Поэтому было решено использовать рекуррентную сеть в качестве генеративной модели.

Подготовка данных для обучения.

Проанализируем датасет и определим необходимые для подготовки к обработке сетью действия\преобразования:

1. Для удобной работы с midi форматом воспользуемся свободной библиотекой Music21.
2. При считывании midi файла с помощью Music21, данные представляются в виде специальных объектов Music21 – нот и аккордов (а также отдыхов). У каждого объекта ноты и отдыха есть свой оффсет (когда нота играется по времени) и длительность. Аккорд является по сути набором нот, играющихся одновременно. Для того, чтобы правильно сгенерировать мелодию, достаточно предсказать следующую ноту, аккорд или отдых. Таким образом, нашу задачу можно свести к задаче генерации текста, в котором вместо букв используется алфавит, состоящий из всех возможных нот, аккордов и отдыхов.

Считывание данных и сохранение дампа в отдельный файл:

```
def get_notes():
    """ Get all the notes and chords from the midi files in the ./midi_songs directory
"""
    notes = []

    for file in glob.glob("midis/*.mid"):
        midi = converter.parse(file)

        print("Parsing %s" % file)

        notes_to_parse = None

        try:  # file has instrument parts
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except:  # file has notes in a flat structure
            notes_to_parse = midi.flat.notesAndRests

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch))
            elif isinstance(element, chord.Chord):
                notes.append('.'.join(str(n) for n in element.normalOrder))
            elif isinstance(element, note.Rest):
                notes.append(element.name)

    with open('parsed/notes', 'wb') as filepath:
        pickle.dump(notes, filepath)

    return notes
```

3. Для того, чтобы привести данные в формат, пригодный для обучения LSTM в Keras, разобьем наш большой файл, содержащий мелодии на под последовательности длины 100. Каждый обучающий шаблон сети состоит из 100 временных шагов одного символа (X), за которыми следует один символьный вывод (y). Создадим словарь всех возможных символов алфавита (ноты, аккорды) и вместо непосредственно символа будем подавать сети целое число – позиция символа в словаре. Затем нам нужно изменить масштаб целых чисел в

диапазоне от 0 до 1, чтобы облегчить изучение шаблонов сетью LSTM, которая по умолчанию использует функцию активации сигмовидной кишки.

Подготовка данных для обучения:

```
def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 100

    # get all pitch names
    pitchnames = sorted(set(item for item in notes))

    # create a dictionary to map pitches to integers
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    network_input = []
    network_output = []

    # create input sequences and the corresponding outputs
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        network_output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    network_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    network_input = network_input / float(n_vocab)

    network_output = np_utils.to_categorical(network_output)

    return (network_input, network_output)
```

Создание модели.

На первой итерации было решено использовать следующую архитектуру сети:

```
def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(LSTM(
        400,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        dropout=0.25,
        recurrent_dropout=0.3,
        return_sequences=True))
    model.add(LSTM(400))
    model.add(BatchNorm())
    model.add(Dense(200))
    model.add(Activation('relu'))
    model.add(BatchNorm())
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    return model
```

У данной модели есть ряд проблем – слишком долгое обучение, отсутствие приемлемых результатов к 60й эпохе и переобучение.

На второй итерации слои LSTM были заменены на CuDNNLSTM для возможности эффективной тренировки сети на GPU. Если раньше на одну эпоху уходило 20 минут, то сейчас всего 7.

На третьей итерации добавим Несколько Dropout слоев для предотвращения переобучения.

На четвертой итерации первый LSTM слой был сделан двунаправленным – это значительно улучшает качество выдаваемых сетью мелодий.

Итоговый вид архитектуры модели:

```
def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(Bidirectional(CuDNNLSTM(
```

```
            400,
            input_shape=(network_input.shape[1], network_input.shape[2]),
            return_sequences=True
        ), input_shape=(network_input.shape[1], network_input.shape[2])))
        model.add(Dropout(0.3))
        model.add(CuDNNLSTM(400))
        model.add(BatchNorm())
        model.add(Dropout(0.3))
        model.add(Dense(200))
        model.add(Activation('relu'))
        model.add(BatchNorm())
        model.add(Dropout(0.2))
        model.add(Dense(n_vocab))
        model.add(Activation('softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

        return model
```

Генератор мелодий.

Следующий код генерирует двухминутное миди (500 нот):

```
def generate_notes(model, network_input, pitchnames, n_vocab):
    """ Generate notes from the neural network based on a sequence of notes """
    # pick a random sequence from the input as a starting point for the prediction
    start = numpy.random.randint(0, len(network_input)-1)

    int_to_note = dict((number, note) for number, note in enumerate(pitchnames))

    pattern = network_input[start]
    prediction_output = []

    # generate 500 notes
    for note_index in range(500):
        prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
        prediction_input = prediction_input / float(n_vocab)

        prediction = model.predict(prediction_input, verbose=0)

        index = numpy.argmax(prediction)
        result = int_to_note[index]
        prediction_output.append(result)
```

```
        pattern.append(index)
        pattern = pattern[1:len(pattern)]

    return prediction_output
```

Сохранение миди производится с помощью библиотеки Music21.

Выводы.

В ходе работы была написана генеративная рекуррентная сеть с долговременной памятью, генерирующая мелодии, похожие на сонаты моцарта.

ПРИЛОЖЕНИЕ А.

ОБУЧЕНИЕ СЕТИ (lstm.py)

```
""" This module prepares midi file data and feeds it to the neural
    network for training """
import glob
import pickle
from datetime import datetime

import numpy
from music21 import converter, instrument, note, chord
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Bidirectional, Dropout, LSTM, Activation
from tensorflow.compat.v1.keras.layers import CuDNNLSTM
from tensorflow.keras.layers import BatchNormalization as BatchNorm
from keras.utils import np_utils
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard, ReduceLROnPlateau
from keras_self_attention import SeqSelfAttention

def train_network():
    """ Train a Neural Network to generate music """
    notes = get_notes()

    # get amount of pitch names
    n_vocab = len(set(notes))

    network_input, network_output = prepare_sequences(notes, n_vocab)

    model = create_network(network_input, n_vocab)

    train(model, network_input, network_output)


def get_notes():
    """ Get all the notes and chords from the midi files in the ./midi_songs directory
"""
    notes = []
    durations = []

    for file in glob.glob("midis/*.mid"):
        midi = converter.parse(file)
```

```
        print("Parsing %s" % file)

        notes_to_parse = None

        try:  # file has instrument parts
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except:  # file has notes in a flat structure
            notes_to_parse = midi.flat.notesAndRests

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch))
            elif isinstance(element, chord.Chord):
                notes.append('.'.join(str(n) for n in element.normalOrder))
            elif isinstance(element, note.Rest):
                notes.append(element.name)

    with open('parsed/notes', 'wb') as filepath:
        pickle.dump(notes, filepath)

    return notes


def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 100

    # get all pitch names
    pitchnames = sorted(set(item for item in notes))

    # create a dictionary to map pitches to integers
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    network_input = []
    network_output = []

    # create input sequences and the corresponding outputs
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
```

```
            sequence_out = notes[i + sequence_length]
            network_input.append([note_to_int[char] for char in sequence_in])
            network_output.append(note_to_int[sequence_out])

        n_patterns = len(network_input)

        # reshape the input into a format compatible with LSTM layers
        network_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
        # normalize input
        network_input = network_input / float(n_vocab)

        network_output = np_utils.to_categorical(network_output)

        return (network_input, network_output)


def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = Sequential()
    model.add(Bidirectional(CuDNNLSTM(
        400,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        return_sequences=True
    ), input_shape=(network_input.shape[1], network_input.shape[2])))
    model.add(Dropout(0.3))
    model.add(CuDNNLSTM(400))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(200))
    model.add(Activation('relu'))
    model.add(BatchNorm())
    model.add(Dropout(0.2))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

    return model


def train(model, network_input, network_output):
    """ train the neural network """
```

```
    filepath = "weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"

    checkpoint = ModelCheckpoint(
        filepath,
        monitor='loss',
        verbose=0,
        save_best_only=True,
        mode='min'
    )

    log_dir = "logs/fit/" + datetime.now().strftime("%Y%m%d-%H%M%S/")
    tensorBoard = TensorBoard(log_dir=log_dir,
                              histogram_freq=0,
                              write_graph=True,
                              write_images=True, profile_batch=100000000)

    callbacks_list = [checkpoint, tensorBoard]

    model.fit(network_input, network_output, epochs=150, batch_size=192,
callbacks=callbacks_list)


if __name__ == '__main__':
    train_network()
```

ПРИЛОЖЕНИЕ Б.

ГЕНЕРАЦИЯ МЕЛОДИИ (predict.py)

```
""" This module generates notes for a midi file using the
    trained neural network """
import pickle
import numpy
from lstm import create_network
from music21 import instrument, note, stream, chord


def generate():
    """ Generate a piano midi file """
    #load the notes used to train the model
    with open('parsed/notes', 'rb') as filepath:
        notes = pickle.load(filepath)

    # Get all pitch names
    pitchnames = sorted(set(item for item in notes))
    # Get all pitch names
    n_vocab = len(set(notes))

    network_input, normalized_input = prepare_sequences(notes, pitchnames, n_vocab)
    model = load_network(normalized_input, n_vocab)
    prediction_output = generate_notes(model, network_input, pitchnames, n_vocab)
    create_midi(prediction_output)


def prepare_sequences(notes, pitchnames, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    # map between notes and integers and back
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))

    sequence_length = 100
    network_input = []
    output = []
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        output.append(note_to_int[sequence_out])
```

```
    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    normalized_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    normalized_input = normalized_input / float(n_vocab)

    return network_input, normalized_input


def load_network(network_input, n_vocab):
    """ create the structure of the neural network """
    model = create_network(network_input, n_vocab)

    # Load the weights to each node
    model.load_weights('weights.hdf5')

    return model


def generate_notes(model, network_input, pitchnames, n_vocab):
    """ Generate notes from the neural network based on a sequence of notes """
    # pick a random sequence from the input as a starting point for the prediction
    start = numpy.random.randint(0, len(network_input)-1)

    int_to_note = dict((number, note) for number, note in enumerate(pitchnames))

    pattern = network_input[start]
    prediction_output = []

    # generate 500 notes
    for note_index in range(500):
        prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
        prediction_input = prediction_input / float(n_vocab)

        prediction = model.predict(prediction_input, verbose=0)

        index = numpy.argmax(prediction)
        result = int_to_note[index]
        prediction_output.append(result)
```

```
        pattern.append(index)
        pattern = pattern[1:len(pattern)]

    return prediction_output


def create_midi(prediction_output):
    """ convert the output from the prediction to notes and create a midi file
        from the notes """
    offset = 0
    output_notes = []

    # create note and chord objects based on the values generated by the model
    for pattern in prediction_output:
        # pattern is a chord
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                new_note = note.Note(int(current_note))
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        elif 'rest' in pattern:
            new_rest = note.Rest(pattern)
            new_rest.offset = offset
            new_rest.storedInstrument = instrument.Piano()
            output_notes.append(new_rest)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)

        # increase offset each iteration so that notes do not stack
        offset += 0.5
```

```
    midi_stream = stream.Stream(output_notes)

    midi_stream.write('midi', fp='test_output.mid')


if __name__ == '__main__':
    generate()
```