Coding Project N°7 - Music generation: Implement a music generation deep neural network integrating a variational auto-encoder approach with transformers.

Dipartimento di Informatica
Master Degree - Computer Science - AI Curriculum

Intelligent Systems for Pattern Recognition
9 CFU

PROF: Davide Bacciu
Niko Paterniti Barbino 638257

IN SUPREMÆ DIGNITATIS 1343
UNIVERSITÀ DI PISA

Index:

INTRODUCTION

The focus of this project is the development of a music generation Deep Neural Network integrating a Variational AutoEncoder approach with Transformers.

For this purpose we used:

- Lakh MIDI dataset to facilitate large-scale music information retrieval
- Pytorch implementation of Compressive Transformers

Lakh MIDI DATASET

The Lakh MIDI dataset is a collection of 176,581 unique MIDI files used to facilitate large-scale music information retrieval, both symbolic (using the MIDI files alone) and audio content-based (using information extracted from the MIDI files as annotations for the matched audio files).

Lakh MIDI Dataset

Data Partition

From the whole “Clean MIDI dataset” we extracted for each song its genre and then we randomly and equally sampled 120 .mid files among 3 genres:

- AlternativeRock
- ArtRock
- BluesRock

Making sure that the sampled songs all had the “acoustic guitar” as instrument (identified by the program ‘25’)

Music21

Music21 is a Python toolkit used for computer-aided musicology. It allows us to teach the fundamentals of music theory, generate music examples and study music. The toolkit provides a simple interface to acquire the musical notation of MIDI files. Additionally, it allows us to create Note and Chord objects so that we can make our own MIDI files easily.

```
def create_network(sequence_length, n_vocab):
    model = CompressiveTransformer(
    num_tokens = n_vocab,
    dim = sequence_length,
    depth = 6,
    seq_len = sequence_length,
    mem_len = sequence_length,
    cmem_len = 256,
    cmem_ratio = 4,
    memory_layers = [5,6],
    gru_gated_residual = False,
    #post-attention dropout
    attn_dropout = 0.1,
    #feedforward dropout
    ff_dropout = 0.1,
    #attention layer output dropout
    attn_layer_dropout = 0.1
    )

    model = AutoregressiveWrapper(model)
    model.cuda()
    return model
```

Neural Network Structure

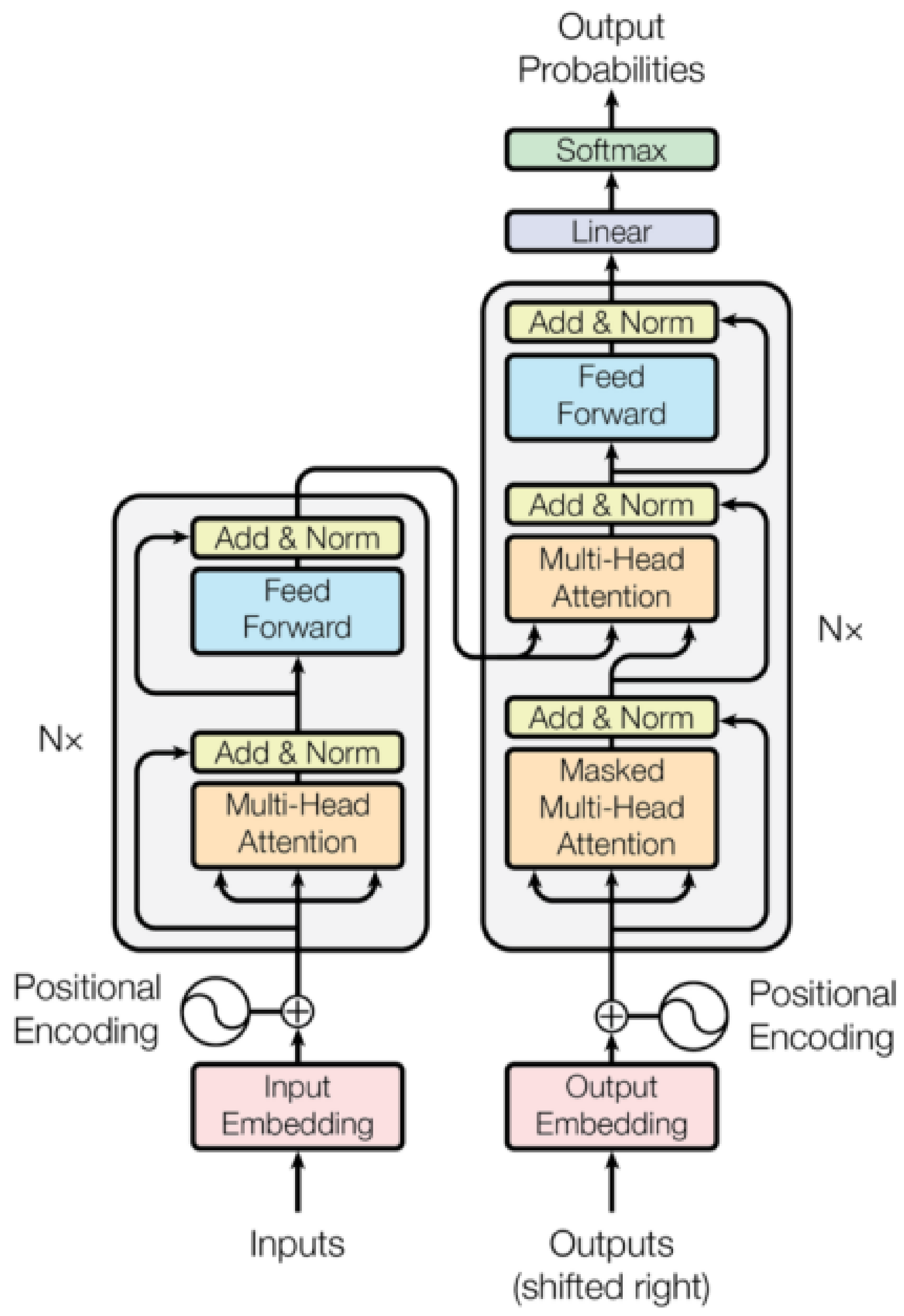
The architecture used in this project is the **“Pytorch Compressive Transformer”**:

- A variant of Transformer-XL with compressed memory for long-range language modelling.
- An attentive sequence model which compresses past memories for long-range sequence learning.

Neural Network Structure

Compressive Transformer

- Based on the Transformer XL architecture with the addition of compressed memory.
- Encoder-decoder design
- Self-Attention
- Data flow: given a sequence of tokens x they are mapped in a continuous representation z. The decoder takes as input one element of z at a time with the additional input of the previous output generated by itself.
- AutoRegressive Model



Neural Network Structure

Self Attention:

- Learn and describe the relationship between the sequence tokens.
- Have an attention function that given a vector query Q. the matrices of keys K and values V (dimension d^k) computes their relationship.

$$Attention(Q, K, V) = softmax(\frac{Q * K^T}{\sqrt{d^k}}) * V$$

Neural Network Structure

Recurrence Mechanism

Technique that enables long-term dependencies using information from previous segments conditioning the values of K and the V of the segment with the previously hidden state sequence produced by the previous segment concatenated with the earlier hidden state sequence produced by the current segment.

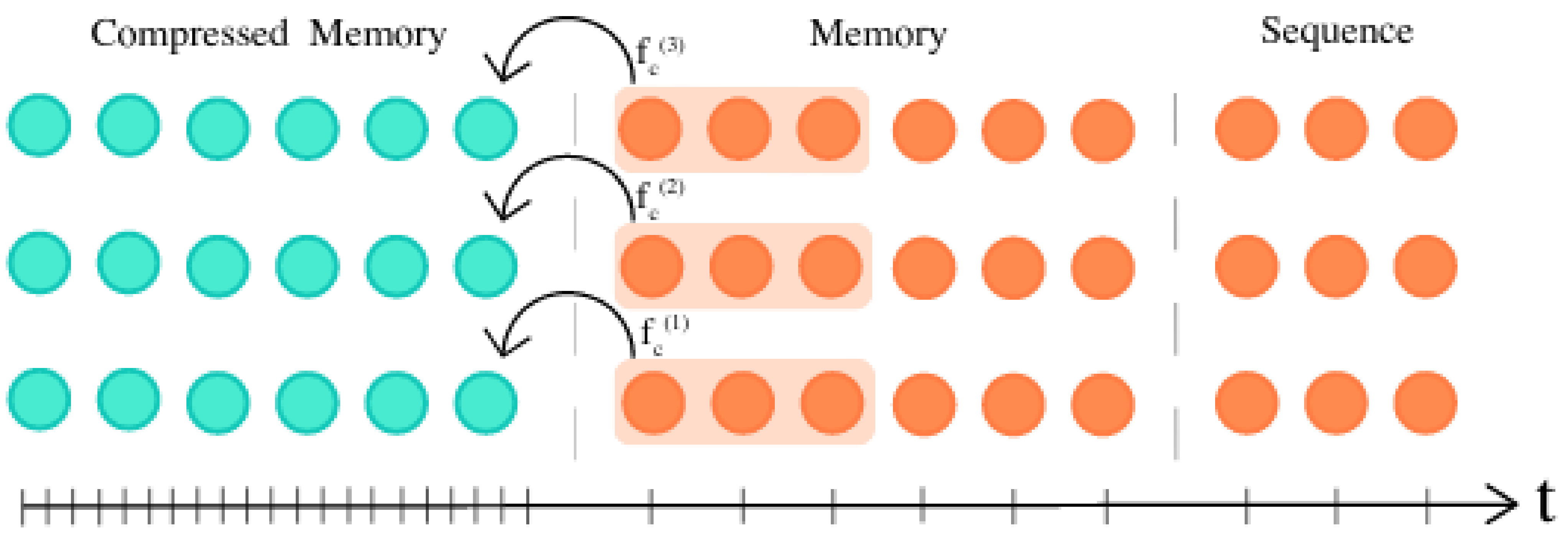
Relative Positional Encoding

Introduces a relative position encoding to correctly compute the self attention of the current segment because the current sequence has the same positional encoding as the previous

Neural Network Structure

Memory Compression

The Recurrence Mechanisms leads us to the requirement of additional memory to store the informations of past segments The memory compression helps us store as much information as we can through the compressed memory.



Example of compressed memory in a transformer:

- 3 layer,
- sequence length = 3,
- memory size = 6,
- compressed memory size = 6
- rate of compression = 3.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    25W / 300W |      0MiB / 16384MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

EXPERIMENTS

To run the code and perform the experiments i bought **Google Colab Pro** due to the long time required for the training and the limited resources available with the free version.

So the training was computed on Google Colab using the **Tesla V100 machine** (See the image on the left for more details)

Experiments

- **Architecture**: based on the implementation of the following notebook: *https://github.com/lucidrains/compressive-transformer-pytorch*
- **Input**: Vector of 64 notes and chords.
- **6 layers** for the encoder and the decoder
- **Optimizer**: Adam PyTorch
- **Sequence length**: 32, 64,128. Due to time required for training, probability of overfitting/underfitting and other criteria i decided to use 64 as sequence length.
- **Regularization**: Dropout = 0.1

I studied the task and the strategy used in this project from the paper:

https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5

https://colinraffel.com/publications/thesis.pdf

And the relative code demonstration to better understand the task of music generation:

https://colab.research.google.com/drive/19TQqekOlnOSW36VCL8CPVEQKBBukmaEQ#scrollTo=DDOBVWULXfpz

Experiments

Initialliy i performed my experiments with the whole dataset of 120 MIDI files equally and randomly divided among 3 genres (AlternativeRock, ArtRock, BluesRock) setting a very high number of epochs. I found out that the time required for each epoch was in the range of 500 seconds so the time to get the results was definitely to high and i quickly run out of resources from the Colab runtime, so i decided to perform the training on a smaller portion of the dataset using a total of 40 MIDI files for the training and the validation with a time for each epoch in the range of 25 seconds

Results

Epoch: 0 |Training loss: 0.8095
Epoch: 1 |Training loss: 0.8072
Epoch: 2 |Training loss: 0.8051
Epoch: 3 |Training loss: 0.8027
Epoch: 4 |Training loss: 0.8011
validation loss: 6.0538

Epoch: 5 |Training loss: 0.7980
Epoch: 6 |Training loss: 0.7956
Epoch: 7 |Training loss: 0.7951
Epoch: 8 |Training loss: 0.7916
Epoch: 9 |Training loss: 0.7905
validation loss: 6.0906

Epoch: 10 |Training loss: 0.7874
Epoch: 11 |Training loss: 0.7859
Epoch: 12 |Training loss: 0.7829
Epoch: 13 |Training loss: 0.7819
Epoch: 14 |Training loss: 0.7792
validation loss: 6.1258

Epoch: 15 |Training loss: 0.7785
Epoch: 16 |Training loss: 0.7751
Epoch: 17 |Training loss: 0.7733
Epoch: 18 |Training loss: 0.7707
Epoch: 19 |Training loss: 0.7693
validation loss: 6.1418

Epoch: 20 |Training loss: 0.7668
Epoch: 21 |Training loss: 0.7654
Epoch: 22 |Training loss: 0.7622
Epoch: 23 |Training loss: 0.7618
Epoch: 24 |Training loss: 0.7585
validation loss: 6.1651

Epoch: 25 |Training loss: 0.7577
Epoch: 26 |Training loss: 0.7546
Epoch: 27 |Training loss: 0.7529
Epoch: 28 |Training loss: 0.7507
Epoch: 29 |Training loss: 0.7482
validation loss: 6.2097

Epoch: 30 |Training loss: 0.7462
Epoch: 31 |Training loss: 0.7451
Epoch: 32 |Training loss: 0.7418
Epoch: 33 |Training loss: 0.7404
Epoch: 34 |Training loss: 0.7374
validation loss: 6.2456

Epoch: 35 |Training loss: 0.7360
Epoch: 36 |Training loss: 0.7338
Epoch: 37 |Training loss: 0.7326
Epoch: 38 |Training loss: 0.7298
Epoch: 39 |Training loss: 0.7286
validation loss: 6.2593

Epoch: 40 |Training loss: 0.7258
Epoch: 41 |Training loss: 0.7243
Epoch: 42 |Training loss: 0.7219
Epoch: 43 |Training loss: 0.7199
Epoch: 44 |Training loss: 0.7173
validation loss: 6.2856

Epoch: 45 |Training loss: 0.7164
Epoch: 46 |Training loss: 0.7135
Epoch: 47 |Training loss: 0.7116
Epoch: 48 |Training loss: 0.7091
Epoch: 49 |Training loss: 0.7081
validation loss: 6.3383

Epoch: 50 |Training loss: 0.7050
Epoch: 51 |Training loss: 0.7041
Epoch: 52 |Training loss: 0.7011
Epoch: 53 |Training loss: 0.6995
Epoch: 54 |Training loss: 0.6967
validation loss: 6.3687

Epoch: 55 |Training loss: 0.6952
Epoch: 56 |Training loss: 0.6933
Epoch: 57 |Training loss: 0.6908
Epoch: 58 |Training loss: 0.6886
Epoch: 59 |Training loss: 0.6873
validation loss: 6.3766

Epoch: 40 |Training loss: 0.7258
Epoch: 41 |Training loss: 0.7243
Epoch: 42 |Training loss: 0.7219
Epoch: 43 |Training loss: 0.7199
Epoch: 44 |Training loss: 0.7173
validation loss: 6.2856

Epoch: 45 |Training loss: 0.7164
Epoch: 46 |Training loss: 0.7135
Epoch: 47 |Training loss: 0.7116
Epoch: 48 |Training loss: 0.7091
Epoch: 49 |Training loss: 0.7081
validation loss: 6.3383

Epoch: 50 |Training loss: 0.7050
Epoch: 51 |Training loss: 0.7041
Epoch: 52 |Training loss: 0.7011
Epoch: 53 |Training loss: 0.6995
Epoch: 54 |Training loss: 0.6967
validation loss: 6.3687

Epoch: 55 |Training loss: 0.6952
Epoch: 56 |Training loss: 0.6933
Epoch: 57 |Training loss: 0.6908
Epoch: 58 |Training loss: 0.6886
Epoch: 59 |Training loss: 0.6873
validation loss: 6.3766

Epoch: 60 |Training loss: 0.6856
Epoch: 61 |Training loss: 0.6828
Epoch: 62 |Training loss: 0.6808
Epoch: 63 |Training loss: 0.6787
Epoch: 64 |Training loss: 0.6771
validation loss: 6.4080

Epoch: 65 |Training loss: 0.6751
Epoch: 66 |Training loss: 0.6731
Epoch: 67 |Training loss: 0.6716
Epoch: 68 |Training loss: 0.6684
Epoch: 69 |Training loss: 0.6671
validation loss: 6.4736

Epoch: 70 |Training loss: 0.6648
Epoch: 71 |Training loss: 0.6634
Epoch: 72 |Training loss: 0.6608
Epoch: 73 |Training loss: 0.6593
Epoch: 74 |Training loss: 0.6565
validation loss: 6.4970

Epoch: 75 |Training loss: 0.6547
Epoch: 76 |Training loss: 0.6532
Epoch: 77 |Training loss: 0.6517
Epoch: 78 |Training loss: 0.6501
Epoch: 79 |Training loss: 0.6468
validation loss: 6.4955

CONCLUSION

In this study we implemented a deep neural network for music generation integrating a variational auto-encoder approach with Transformers, i took inspiration from different projects and papers where i found many informations and different ways of approaching the problem such as LSTM, BI-LSTM ecc...
Due to the self attention mechanism the best way to approach this problem is transformer based approach, still it would be interesting making comparisons and trying to obtain better quality songs with a bigger dataset. Also we could study the behaviour of the model considering the encoding of more than one instrument for each note with also the temporal parameter set, so that more than one instrument could be associated with each note or chord generated making music more harmonious

Thanks

Niko Paterniti Barbino
638257

IN SUPREMÆ DIGNITATIS
1343
UNIVERSITÀ DI PISA