A Pseudo-Bach Neural Network

Arsenijs Golicins,
Dorin-Vlad Udrea

Abstract

This paper is a semester project for the Neural Networks course. The project aims to generate a continuation of Johann Sebastian Bach's unfinished Last Fugue using a deep learning model. The authors use a sequence-to-sequence Long Short Term Memory (LSTM) model, which takes a sequence of notes as input and predicts the same sequence of notes shifted by one position as output. The model is trained on a four-dimensional vector sequence representing musical voices in a format known as the x-representation. The authors also implement vertical analysis in their model, treating the voices as interdependent. The model is trained for 100 epochs, and the results are assessed by listening to the output and judging whether it has learned the musical patterns adequately. The authors acknowledge that their model is not generalizable and is only concerned with one particular musical piece. They suggest potential improvements, such as splitting the initial data into training and validation sets and training the model to perform well on previously unseen musical pieces.

1 Introduction

For our semester project, we chose to pursue the project which seeks to finish, or at the very least extend, Johann Sebastian Bach's Last Symphony. This is not a trivial task, as the music composition is not episodic, but sequential - it depends on the previous dynamics of the system. By "system", we mean the interdependency of the musical notes within a given key. Due to music being able to be represented as a dynamical system which relies on the mathematical symmetry of sound frequencies, the task of generating new, or continuing already existing sequences, no longer seems implausible. In this project, our approach towards music generation, like many neural networks principles, is, in essence, data driven. We will attempt to extract certain features of the system from provided data samples to predict future states of said system. This technique comes in contrast to the top-down approach utilised by human composers during the process of music creation. However, since the human ability to listen to music is driven by expectations due to the music's deterministic nature, we will be able to assess the performance of the model using the best top-down computers - our brains, and judge if the result can be called a musical sequence. Hence, the aim of our project is to generate about 25 seconds of a pseudo-Bach musical sequence through the analysis of features of the unfinished piece. The set minimum objective of the task is to achieve something that will not sound like a chaotic cacophony. The main tool we are going to use for building the neural network is the free and open-source software library TensorFlow with a particular emphasis on the Keras interface.

2 Background and Data

2.1 Background

The idea for our project is not new - it was originally introduced in the Santa Fe Time Series Analysis and Prediction Competition from 1993. One of the time series used in that competition was the piece by Johann Sebastian Bach - *Contrapunctus 14* (Cp. XIV). The competition organisers selected this piece, for fugues tend to be typically characterised by the high extent and dynamical complexity of their internal structure. Fugues have primary and secondary themes, which are symmetrically transformed throughout the musical piece. [Dirst and Weigend, 1993] These transformations include transpositions, mirror inversions, changing the pace of the theme etc. All these changes can be mathematically expressed by the changes in the intervals between notes which grants us the ability to grasp the musical logic and the dynamics of the structure. The Cp. XIV is actually a sequence of 3 fugues, where the last one was left incomplete. The beginning of that last unfinished fugue is what we are going to use for our data.

2.2 Data

The data taken from the Santa Fe competition and provided by our course coordinator is, essentially, a 4-dimensional vector sequence representing musical voices - Soprano, Alto, Tenor, and Bass in a format known as the *x-representation*. The lengths of the vectors correspond to the lengths of the selected musical fragment. Each value is a real number denoting the sound frequency (a pitch of a note), which can be read and voiced in MIDI. Values of 0 denote pauses, or breaks, with no sound. The duration of the note is represented by the repeating value in the next value of the sequence. One note has the (arbitrary) length of sixteenth, and so 16 numbers make a bar. The representation can be seen in Figure 1:

```
S |68 68 68 68 NA NA NA NA NA NA NA NA NA NA NA NA|                                |
A |62 62 61 61 59 59 57 57 56 56 54 54 56 56 59 59|57 57 57 57                     |
T |54 54 54 54 54 54 53 51 53 53 53 53 53 53 53 53|54 54 56 56 54 54 52 52 51 51 49 49 51 51 54 54|
B |49 49 49 49 49 49 49 49 49 49 49 49 49 49 49 49|42 42 42 42                     |
    ↑                ↑                ↑                ↑
t  3793             3801             3809             3817
```

Figure 1: Raw data from the competition; the last two measures of Cp. XIV as they stand in Bach's manuscript. Time t is given from the beginning of Cp. XIV [Dirst and Weigend, 1993]

2.2.1 Data analysis

Since our goal is the generation of a sequence that relies on the previous data (graciously provided to us), this data should be analysed. Below are several analysis metrics for each voice separately.

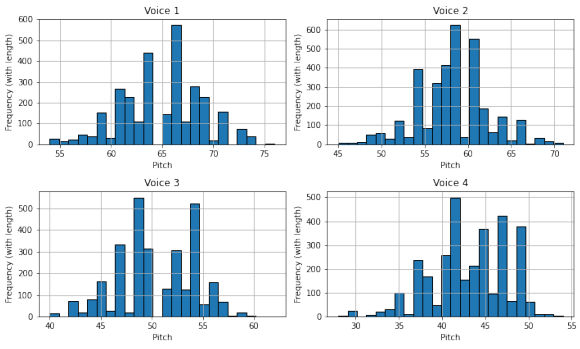

Figure 2: The histogram depicting the frequencies of the pitches. Each pitch repetition is counted.

In Figure 2 one can see the distribution of the notes most frequently played in the fugue. Voice 1's most played pitch frequencies are 67 and 64, Voice 2's are 59 and 61, Voice 3's are 49 and 54 and Voice 4's are 41 and 47.

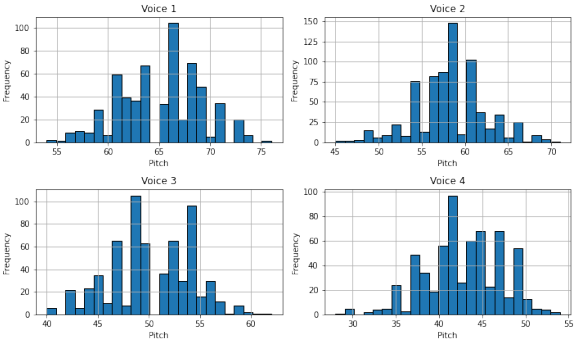

Figure 3: The histogram depicting the frequencies of the pitches. Consecutive repetition of pitches are not counted, this time.

In Figure 3 there is a similar histogram, but without counting consecutive repetitions for a note. The particular statistics do not change much, with only the number of frequencies decreasing as a change.

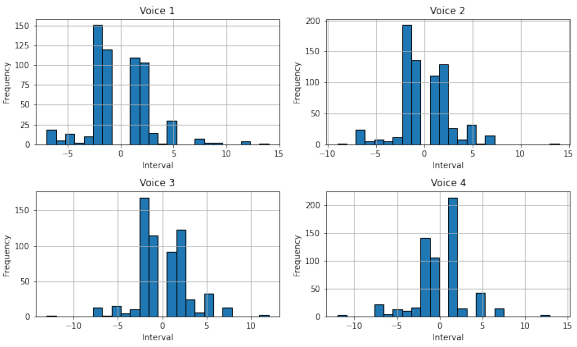

Figure 4: The histogram depicting the frequencies of the intervals measured between consecutive notes (not counting the '0' intervals and the intervals between notes and pauses.

Histograms in Figure 4 show the frequency of the intervals between consecutive notes. It can be seen that the most observed intervals between all voices are '1' and '2', a semitone and a tone, respectively. Another important interval is '5', which corresponds to the consonant perfect fourth.

2.2.2 Data reduction and processing

Based on the data analysis histograms, one can see that there are notes that are met very rarely. To diminish the load on the network and decrease training time, we considered filtering out the notes that were played less than 20 times. However, we decided against that idea as we had access to 2 very powerful GPUs: NVIDIA's GeForce RTX 3080 and NVIDIA's GeForce GTX 1080. As such, we were

able to leverage the CUDA toolkit and the CUDA Deep Neural Network (cuDNN) libraries to significantly decrease the time that it took us to train our neural network (and, by extension, to find suitable hyper-parameter values). For comparison, while a MacBook Pro M1 required 5 hours to train one of our neural network candidates (leveraging the 16-core Neural Engine, mind you), our RTX 3080 required only 15 minutes for the very same neural network. Consequently, we did not perform any data reduction as our processing units were powerful enough to handle even large and complex neural networks.

One of the more controversial decisions we've had to make was the choice of a sliding window. Having studied Johann Sebastian Bach's compositions thoroughly, the only consensus we were able to reach was that there was no consensus: certain parts of a symphony could last for entire pages, while certain others would reach their apex in only half of one. As such, the arbitrary decision was chosen to consider only the last page and a half of a regular sheet music which, for Bach's Contrapunctus XIV, represented 39 boxes or 624 note samples. This became our chosen sliding window.

As for the exact code, however: two sequences are created in the form of two numpy arrays, the first one representing the input of the network while the second one represents the output of the network. The data is read from a .txt file using the pandas library. This data consists of musical voices, which are loaded into a DataFrame. Normalization is an important preprocessing step which ensures that all data is on a similar scale, thus making the learning process more stable. We identify the maximum value across all voices and increase it by one to establish the range within which all voices will be normalized. The DataFrame is, then, converted into a NumPy array, which allows for more efficient numerical operations and easier interfacing with TensorFlow and Keras. A crucial part of working with sequence data like music is to divide it into meaningful subsequences which can be fed into a model. For this exact reason, we created a function which handles the processing. For each voice, the function creates sequences of a certain length and the target for each sequence is the next note after the sequence. The targets (next notes for each sequence) are one-hot encoded, which is a common preprocessing step for categorical variables. In this case, each note is treated as a separate category. One-hot encoding transforms the categorical variable into a binary vector of a certain size (the number of distinct categories), where the index of the category is marked as 1, and all other indices are marked as 0. The lists of sequences and one-hot encoded targets are converted into NumPy arrays and then transposed to get the correct shape. The transposition is necessary to ensure that the first dimension of the arrays corresponds to the samples, the second dimension to the sequence length or voices, and the third dimension to the number of voices or the vocabulary size.

3 Methods & Architecture

3.1 Architecture

What we are tasked with is generating a continuation of all voices from Bach's Contrapunctus XIV using a deep learning model. This is a problem of sequence generation and as such, a Long Short Term Memory (LSTM) model was chosen as a suitable choice. Specifically, we used a sequence-to-sequence LSTM model. The input to the model was a sequence of notes to which the corresponding output would be the same sequence of notes shifted by one position. That is, if we input the sequence of notes $[64, 64, 64, 64, 62, 62, 61, 61]$, the output would be $[64, 64, 64, 62, 62, 61, 61, ..\text{next note}..]$. This was our final architecture:

1. **Model Initialization**: A Sequential model is initiated using Keras, which allows layers to be added one after another;

2. **LSTM Layers**: The model begins with two LSTM (Long Short-Term Memory) layers. LSTM is a type of recurrent neural network (RNN) layer, which is particularly effective for sequence-based problems. The first LSTM layer has 1024 units and receives an input shape derived from the data, with the first dimension corresponding to the sequence length and the second dimension corresponding to the number of voices. This LSTM layer is configured to return sequences, meaning it outputs a sequence with the same length as the input, enabling it to pass sequential information to the next LSTM layer. The second LSTM layer also has 1024 units but is not set to return sequences, meaning it only outputs the final state, summarizing the information from the input sequence;

3. **Dropout Layers**: After each LSTM layer, a Dropout layer is added. This is a regularization technique that randomly sets a proportion (20% in this case) of the input units to 0 at each update during training, which helps prevent overfitting;

4. **Dense Output Layers**: For each voice in the input, a Dense output layer is created. The Dense layer is a fully-connected layer, where each unit is connected to every unit in the previous layer. These Dense layers have a number of units equal to the vocabulary size, and use the softmax activation function:

$$\sigma(\boldsymbol{x}_i) = \frac{e_i^x}{\sum_{j=1}^{K} e_j^x}, \tag{1}$$

[Nielsen, 2015] which outputs a probability distribution over the vocabulary for each voice. This setup effectively creates a multi-output model, where each output predicts the next note for one of the voices;

5. **Model Compilation**: The model is compiled with the categorical cross-entropy loss function:

$$H(P^*|P) = -\sum_i P^*(i) \log P(i), \tag{2}$$

[Nielsen, 2015] which is suitable for multi-class classification problems, and the Adam optimizer, a popular choice due to its computational efficiency and good performance on a wide range of problems;

6. **Model Training**: The model is trained for 100 epochs using a batch size of 64. The model's fit function is called with the input sequences and a list of targets, one for each voice. Each target is a slice of the output array, picking out the one-hot encoded next note for the corresponding voice.

Our choice of hyper-parameters was motivated by one primary issue that had been constantly plaguing us for a majority of our project: model collapse. Model collapse is a typical problem in training generative models where the LSTM consistently predicts the same value. There were several reasons why this might have happened. Here are a few possibilities:

1. **Learning rate too high or too low**: If the learning rate is too high, the model may not be able to find the optimal solution and it may keep bouncing around. On the other hand, if the learning rate is too low, it may get stuck in a sub-optimal solution. However, the learning rate in our model is implicitly determined by the optimizer that we use. In our case, we're using the Adam optimizer which adjusts the learning rate adaptively for each parameter, thus, we could not (or, rather, would not) change it directly;
2. **Insufficient training data**: If the model is given insufficient data to learn from, it may get stuck predicting the most common value. This was an issue out of our control given we used all the data we were provided with and, as mentioned in the previous chapter, we went through great strides to not reduce it;
3. **Model complexity**: It could be that the model's capacity is insufficient for the complexity of the task. We started out with an LSTM with 256 units which seemed to not be enough, which is why our final model had 1024 units in each layer (4 times as much, heuristically choosing this value for the fact that we have 4 voices). The number seemed to sit comfortably on the upper-end of the generally agreed upon values;
4. **Insufficient training**: In some cases, the model may simply need to be trained for more epochs. In our case, 50 epochs might not have been enough for the model to learn, which is the number we started with. As such, our final model trained for 100 epochs, once more, the number seemingly sitting comfortably on the upper-end of the generally agreed upon values;
5. **Sequence length**: Sometimes, using longer sequences can improve the model's performance. We experimented with many values of this: 16, 32, 400. We ended up with 624, both for heuristical reasons as it seemed to work best and for musical reasons, as explained previously.

The rest of our chosen hyper-parameters were as follows:

1. **Dropout rate**: Our chosen regularisation technique was dropout. Dropout works by randomly "dropping out" (i.e., setting to zero) a number of output features of the layer during training. Let's say we set the dropout rate to 0.2 (which was indeed the final dropout value for us), which means approximately 20% of the output units of the layer to which dropout is applied will be turned off during each update cycle while training. The dropout rate of 20% seems to have been the sweet spot, as both lower and higher values caused the system to not perform as well;
2. **Batch Size**: Our chosen batch size was 64. There is no deep philosophical reason for this number: It seems to be a suitable value in related literature and, as such, we sought to use it as well. Additionally, it is a power of '2' and we are binary enjoyers.

3.2 Loss function and optimiser

During training, the model uses the Adam (Adaptive Moment Estimation) optimizer with an initial learning rate of 0.001. The learning rate determines the step size of parameter updates. The Adam optimizer calculates an adaptive learning rate for each parameter based on the estimates of both the first-order (gradient vector) and second-order (hessian matrix) moments of the gradients.

The loss function used is the cross-entropy loss, which is suitable for multi-class classification tasks, and also is a standard choice for the cases involving deciding on the probability distribution [Goodfellow et al., 2016].

$$H(P^*|P) = -\sum_i P^*(i) \log P(i), \qquad (3)$$

where P^* is the true class distribution, and P is the predicted class distribution. Cross-entropy loss measures the dissimilarity between the predicted probability distribution of generated notes and the true distribution of the target notes. The model aims to generate sequences of notes that closely resemble the original input music. This enables the model to learn the patterns and dependencies within the music data, ultimately improving its ability to generate coherent and musically meaningful sequences of notes. Since the note generation task involves multiple classes (different pitches or musical notes), the cross-entropy loss is well-suited for this purpose.

The training procedure involves iterating over the input sequences and their corresponding target outputs. The model is then trained by forward propagating the input through the network, computing the loss, backpropagating the gradients, and updating the model's parameters.

3.3 Accuracy

Given a trained model, we can recursively feed the generated sequence to itself as the input in order to generate new notes. Afterwards, we sample the output distribution to determine the next note. The standard metric to determine the accuracy of such a system is the loss function of the model. However, as per neural networks' opacity, it is rather difficult to interpret it (as we are about to witness). Luckily, we can listen to the output for ourselves and judge whether or not it learned the musical patterns adequately.

4 Results

After 100 epochs of training, the model performed admirably. The loss function of the model for each of the voices, as well as for the total error, can be seen in Figure 5. By the end of the last epoch, the training loss managed to drop to approximately 0.4, with each voice having an approximate error of 0.1. As was mentioned previously, while we went through great efforts to prevent overfitting, due to the lack of training data, our model will inevitably overfit. In essence, the system learned the regularities inside *Contrapunctus 14* and continued to generate music following this piece's logic.

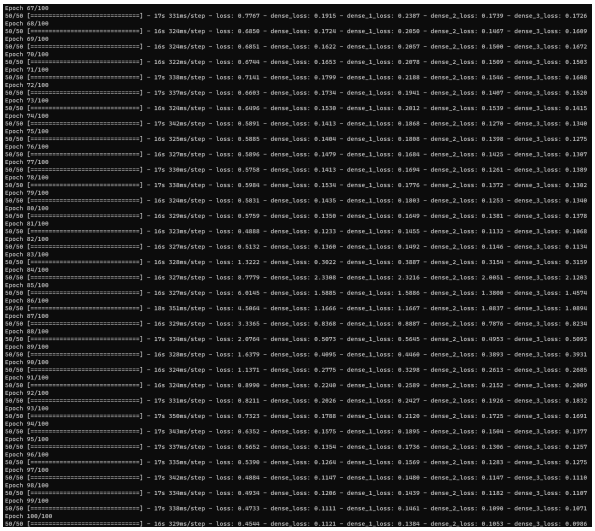

Figure 5: The loss function of the model over 100 epochs.

4.1 Discussion

During development, there were quite a few issues, namely the inevitable difficulty in choosing the adequate architecture for the task. First, we started with 1 LSTM layer and 1 dense layer. For a reason unknown to us at the time, the model was never able to reach a point where its loss function lowered below a value of 0.7, no matter what values were chosen for the learning rate, the dropout rate or the number of epochs. After some experimentation, we found out that adding one more LSTM layer and one more dense layer solved our problem, our loss function finally reaching low-enough values to be to our liking. Even though we knew overfitting to be inevitable, regularisation was still a primary goal of ours. Initially, we implemented L2 regularisation to the system, but after tinkering with it for some time, we decided to instead move to dropout because it worked better with our

architecture, granting us the capability of adding one such layer after each LSTM layer, effectively covering the entire model.

Second, it is of worth to mention that we did implement vertical analysis in our model [Dirst and Weigend, 1993]. The system treats the voices as interdependent, as they are all used as input to the model at each timestep. Specifically, in the LSTM model, at each timestep, it accepts the four voices as input and uses this information to predict the next four notes. This means the model takes into account the relationships between the voices when making a prediction. The learned relationships or dependencies are determined by the patterns present in the training data. If the voices in the training data are interrelated in some way, the LSTM model will learn these interdependencies during training and will use that learned information when generating new sequences of notes. However, the LSTM doesn't assume any specific type of dependency between the voices, but it is capable of learning the existing dependencies from the data during the training phase.

There are several potential improvements that could be done to our model. First and foremost, in our work, we made an assumption that the initial "momentum" that the composer had set up will keep this consonant transformations of voices for at least 25 more seconds. This is a bold assumption and our final output shows this to be partially wrong: while the 25 seconds are quite melodic and, in our opinion, beautiful, the "jump" from Bach's own music to ours is noticeable and one can very clearly pin-point where the cut-off is. It is of note to mention that when we used a smaller sequence length, this was not a problem and the switch between the input data and the output data was so small as to be unnoticeable; this occurred when the sequence length was equal to 400. However, we decided to remain on the value of 624 as, even though the swap was much more sudden, the overall behaviour of the 25 "piece" seemed to be more creative and stochastic.

Moreover, as previously mentioned, our model was only concerned with one particular musical piece and, although capable, was not meant to be generalisable. Hence, the model does not include test set, nor test loss and the data is similarly not split into training data and validation data. One of the potential improvements that could be implemented is this split to initial data and train the model so it could perform well even on previously unseen musical pieces.

5 Conclusion

For our project concerning the continuation of Bach's last fugue, we first analysed the data provided to us by turning it into a .wav file and listening to it to understand the dataset and the task better. Next, we effectively performed data reduction by segmenting the musical voices into manageable sequences and one-hot encoding the targets, and preparing the data for a machine learning model by normalizing the inputs and shaping the data into a suitable form for a recurrent neural network. Finally, we demonstrated the construction and training of a multi-output sequence generation model using LSTM layers, Dropout for regularization, and Dense layers for multi-class prediction of the next notes in a sequence. Conclusively, the minimum objective we had set to achieve, namely, a melodic, if not quite adequate, continuation of Johann Sebastian Bach's symphonies was achieved, even though the famed composer himself would perhaps remain disatisfied with our work, as, with everything, there is always room for growth and improvement.

References

Matthew Dirst and AS Weigend. Baroque forecasting: On completing js bach's last fugue. Technical report, PRE-33988, 1993.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

A Pseudocode of the program that generates new music

```
import relevant libraries

Read the file 'F.txt' into DataFrame df

Find the maximum value in the DataFrame and add 1 to get the total number of
distinct notes (n_vocab)

Convert DataFrame to numpy array and store it in variable "voices"

Define function create_sequences with parameters data and seq_length
    Initialize empty lists x and y
    For each index i from 0 to length of data - seq_length
        Create sequence from data[i:i + seq_length] and store it in variable "sequence"
        Pick the note after sequence and store it in variable "label"
        Append sequence to x
        Append label to y
    Return arrays x and y

Set sequence_length to 624
Initialize empty lists x and y
For each voice in voices
    Create sequences for the voice and store them in x_voice and y_voice
    Convert y_voice to one-hot encoding with n_vocab classes
    Append x_voice to x
    Append y_voice to y

Convert x and y to numpy arrays
Transpose x and y to get the shapes (samples, sequence_length, n_voices) and
(samples, n_voices, n_vocab), respectively

Initialize a Sequential model
Add LSTM layer with 1024 units and input shape (x.shape[1], x.shape[2]),
return_sequences set to True
Add Dropout layer with rate 0.2
Add LSTM layer with 1024 units
Add Dropout layer with rate 0.2
Create separate output layer for each voice using Dense layer with
n_vocab units and softmax activation
Combine the model with its input and outputs

Call model.summary()

Compile the model with categorical cross entropy loss and Adam optimizer

Fit the model to the data with 100 epochs and batch size 64

Initialize empty list "output"
Pick a random start point from the data
Set initial pattern to the sequence at the start point
```

```
Generate 400 new notes
    For each index i from 0 to 400
        Reshape the pattern to (1, len(pattern), x.shape[2]) and store it
        in "prediction_input"
        Predict the next note using the model and prediction_input, store the result
        in "prediction"
        Get the index of the maximum value in prediction and store it in "indices"
        Append indices to output
        Update the pattern by removing the first note and appending indices

Open file 'G.txt' for writing
    For each set of notes in output
        Write the notes to the file, separated by tabs
```

B Pseudocode of the program that translates text files to waveform audio files

```
import required libraries

Load 'G.txt' into numpy array F

Set symbolic_length to the length of F
Set base_freq to 440
Set sample_rate to 10000
Set duration_per_symbol to 1/16
Set ticks_per_symbol to the product of sample_rate and duration_per_symbol

Define function convert_voice_to_sound_vector with parameter voice
    Initialize sound_vector as a zero array with size of symbolic_length * ticks_per_symbol
    Set current_symbol to the first note of voice
    Set start_symbol_index to 0

    For each index n from 0 to symbolic_length
        If voice[n] is not equal to current_symbol
            Set stop_symbol_index to n - 1
            Generate covered_sound_vector_indices as a range of indices corresponding to
            the duration of the current_symbol
            Set tone_length to the length of covered_sound_vector_indices
            Calculate frequency of the current_symbol using MIDI to frequency formula
            Generate tone_vector as a sine wave with calculated frequency
            Insert tone_vector into corresponding indices in sound_vector
            Set current_symbol to voice[n]
            Set start_symbol_index to n
    Return sound_vector

Generate sound vector for each column in F and store them in voices

Set sound_vector to the mean of all voices

Save sound_vector as a .wav file with a given sample_rate
```

C Usage of GPT-4 in the writing of this essay

In the 10 pages before you, GPT-4 (and other such Large Language Models) have mostly only been used in the conversion of Python code into Pseudocode, for a few reasons. First of all, we are, perhaps a little, ashamed to admit that none of us ever mastered the necessary syntax. Second, we would like to argue that the previous point is not entirely our fault because there are so many variants of Pseudocode! Specifically, there is the Fortran Style, the Pascal style, the C style, the Structured Basic style, the Scratch style and many, many more. As such, we were actually quite curious which style GPT-4 will use and, furthermore, quite surprised when we saw the result as it's something unlike anything we've seen before. We are unsure if this is simply an obscure style or something GPT-4 came up with on its own, but it is definitely intriguing.

The experience of using GPT-4 is quite exquisite. It understands human language **incredibly** well. When using Google or StackOverflow, we often have to think of the phrasing of our questions before actually posing them; with GPT-4, we speak to it as if we spoke to each other and it is still able to understand us. Furthermore, it always explains everything step by step, without a shred of impatience or malice in it which makes us feel safe asking it for assistance. Lastly, the utter breadth of knowledge it possess is mind-boggling, being able to comfortable answer questions about chess, video games and neural networks *in the same prompt*!.

However, all of the previous being said, we have noticed a weakening of GPT-4 ever since its release. We do not know if it simply an illusion of the mind or if the developers at OpenAI purposefully weakened their own system (we suspect the latter), but it definitely seems as if it is unwilling to answer as many questions as it used to. Often times, even those questions it does answer have vague answers and they leave us to fill most of the blanks which is, most often, exactly the reason we turn to it in the first place. Finally, its "attention span" seems to have certainly decreased and is our biggest concern with the system at the moment: with its token limit of 8.000, we often time have to send it text in parts, but even with the prompt clearly mentioning that fact, by the time we send the second half of a certain text, it will have already forgotten about the first which is greatly frustrating.