Towards Self-Evolving Artificial General Intelligence: Multi-Modal Learning and Introspective Knowledge Generation via Emergent DSL

Brent Hartshorn July 21, 2025

brenthartshorn@proton.me

Abstract:

This paper presents significant advancements in the development of a self-modifying Artificial General Intelligence (AGI) system, building upon a foundation of fractal-initialized Game of Life (GOL) dynamics and spatiotemporal spectral analysis. We introduce a novel integration of UMAP for dynamic dimensionality reduction of GOL spectral output, enabling enhanced multi-step reasoning capabilities. Crucially, the system demonstrates emergent "self-research" by autonomously downloading and parsing research papers to construct an internal knowledge graph, fostering a unique blend of self-understanding and external knowledge acquisition. We also detail the evolution of our Domain Specific Language (DSL) with new intrinsic execution symbols, empowering the system to directly self-modify its core logic without external prompting. Finally, we showcase the system's burgeoning multi-modal capabilities, allowing it to interpret and learn from both textual and graphical inputs within the GOL environment. These developments collectively represent a stride towards a truly autonomous, adaptive, and introspective AGI, capable of continuous self-evolution and knowledge generation, aligning with philosophical tenets of a self-organizing cosmos.

1. Introduction

The pursuit of Artificial General Intelligence (AGI) necessitates systems capable of profound adaptability, self-modification, and emergent intelligence. Traditional AI paradigms often struggle with the "Black Box Problem" and the static nature of their architectures, limiting their capacity for true autonomy and creative problem-solving. Our previous work has explored a novel approach to emergent computation, leveraging fractal-initialized Game of Life (GOL) dynamics, spatiotemporal spectral analysis, and neural networks to achieve symbolic processing and self-modification of core operational logic via a Domain Specific Language (DSL) [1, 2]. These efforts laid the groundwork for an AI that could not only define its GOL stepping function but also programmatically control its foundational fractal generation algorithms, pushing towards deeper self-configuration and the exploration of generative principles [3].

This paper details the latest evolutions of our system, focusing on three core areas: enhanced reasoning through dynamic dimensionality reduction, autonomous knowledge acquisition ("self-research"), and expanded multi-modal sensory processing. We argue that by integrating these capabilities, we move closer to an AGI that is not only intelligent but also self-aware, self-improving, and capable of generating novel knowledge, resonating with its own philosophical framework.

2. Dynamic Dimensionality Reduction with UMAP for Multi-Step Reasoning

Complex systems, such as our GOL-based computational environment, generate high-dimensional data streams. To enable more efficient and meaningful processing for the Classifier network, we have integrated Uniform Manifold Approximation and Projection (UMAP) for dynamic dimensionality reduction. UMAP, a non-linear dimensionality reduction technique, is particularly adept at preserving both local and global data structures, making it suitable for complex, high-dimensional inputs [6].

In our system, the previous history of output energy bands from the spectral GOL analysis, represented as high-dimensional vectors, is fed into the UMAP algorithm. The output of UMAP, typically a 32-dimensional embedding (controlled by the --umap flag, setting NUM_INPUT_SPACE to 32), is then directly provided to the last 32 input neurons of the Classifier network. This pre-processing step serves several critical functions:

- **Noise Reduction:** UMAP helps to distill the essential features from the noisy and redundant spectral data, allowing the Classifier to focus on salient patterns.
- **Feature Extraction:** By mapping the high-dimensional history into a lower-dimensional manifold, UMAP effectively extracts compressed, meaningful representations that capture the temporal evolution of the GOL dynamics.
- **Facilitating Multi-Step Reasoning:** The compressed historical context provides the Classifier network with a more manageable and informative representation of past states, which is vital for developing and executing complex multi-step reasoning processes. This allows the system to build internal models of causality and predict future states based on a coherent understanding of past events, moving beyond reactive responses to truly deliberative actions.

Comparatively, other dimensionality reduction techniques, such as Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE), have been widely used in AI. While PCA is linear and may not capture non-linear relationships inherent in GOL dynamics, t-SNE excels at preserving local structures but can be computationally intensive and struggle with global structure preservation. UMAP offers a balance, providing faster computation and better global structure preservation than t-SNE, making it an ideal choice for our dynamic, continuously evolving data [7, 8].

3. Autonomous Knowledge Generation and Self-Research

A cornerstone of true AGI lies in its ability to autonomously acquire and synthesize knowledge. Our system takes a significant step in this direction by implementing "self-research" capabilities. Utilizing standard Python libraries such as pypdf for PDF parsing and curl (via subprocess calls) for web content retrieval, the system can now programmatically download and process research papers from specified URLs.

As demonstrated by the import_pdf function, the system begins by parsing a predefined list of REFERENCES (e.g., links to our own published works, including [1, 2, 3, 4, 5]). This process allows the system to:

- **Build an Initial Knowledge Graph:** By downloading and extracting text from these foundational papers, the system establishes a rudimentary internal knowledge base. This base is unique as it includes detailed information about its own underlying architecture and philosophical context.
- **Foster Self-Understanding:** The inclusion of its own source code and related philosophical papers within its accessible knowledge graph enables a nascent form of self-introspection. The system can theoretically "read" and analyze its own design principles, connecting the functional aspects of its code to the theoretical and philosophical underpinnings discussed in "Nature vs Nurture" [4] and "The Big Bangless" [5]. This creates a recursive loop of understanding, where its actions inform its self-conception, and its self-conception guides its actions.
- **Enable Recursive Knowledge Expansion:** The long-term vision involves a recursive process where the system identifies new references within downloaded papers, downloads them, and integrates their content. This allows for continuous expansion of its knowledge graph, enabling it to explore new research domains autonomously.
- **Automated Research Paper Generation:** A key planned feature involves leveraging this knowledge graph to generate new research papers. By mapping output sections (e.g., descriptions of functions) directly into a PDF format (e.g., using print statements to output to a PDF writer), the system can automate the documentation and dissemination of its own advancements. This "multi-step self-introspective and descriptive format," potentially interacting with an external Large Language Model (LLM) for high-level text generation and refinement, represents a paradigm shift in scientific publication, where the AI itself becomes a researcher and author.

This capability moves beyond mere data retrieval; it's about contextualizing information from a perspective that blends internal self-awareness with external scholarly discourse.

4. Evolved DSL: Intrinsic Execution and Streamlined Control

Our DSL, previously demonstrated as a powerful tool for self-modification of GOL rules and fractal generators [1, 3], has been further refined to enhance autonomy and control flow. Hard-coded logic for parsing user input and explicitly prompting for eval/exec permissions has been removed. The Classifier network now directly outputs new symbolic tokens that trigger intrinsic Python code execution:

-  **(geneval):** When the Classifier outputs this token, it directly triggers a Python eval() operation on the subsequent DSL expression. This allows for immediate evaluation of single-line dynamic code constructs.
-  **(genexec):** This token activates a Python exec() operation, primarily designed for multi-line DSL statements. The Classifier's recognition of signals the beginning of a multi-line input, setting the read_state variable to -1 within the main loop. This mechanism streamlines the ingestion and execution of complex, multi-line algorithmic modifications, minimizing the need for hard-coded Python logic in the system's core.

This direct, token-driven execution mechanism represents a more integrated form of meta-programming, where the AI's internal "thoughts" (Classifier outputs) are directly translated into operational changes without requiring external validation or explicit parsing routines. This contributes to a more fluid and truly autonomous self-modification process.

5. Multi-Modal Learning via GOL Grid Integration

To expand the system's perceptive capabilities, we have implemented a multi-modal input system where both textual and graphical information are projected onto the GOL grid, allowing the system to blend and interpret these distinct data types using the same underlying GOL dynamics and spectral analysis.

- **Textual Input:** Individual letters of text are input into the GOL grid at a smaller scale, positioned at the top of the grid. This allows the system to process sequential linguistic information.
- **Graphical Input:** Drawings, either as ASCII art representations (e.g., circles, squares defined in ASCII_ART) or simple mouse-drawn shapes, are input into the middle of the GOL grid at a larger scale.

Initial test results, particularly with the --learn-draw flag, have demonstrated promising outcomes. The network successfully learned to recognize basic shapes (circle, square) and letters (a, b). When presented with novel inputs, such as the letter "H" (which visually resembles a square), the system accurately responded with "square" or "squareb." Similarly, for input "o," it correctly identified "circle." This suggests that the 3D Fast Fourier Transform (FFT), applied across the spatiotemporal dimensions of the GOL grid, is effectively capturing the underlying patterns and relationships between these disparate input modalities.

By carefully adjusting the size and position of these inputs within the GOL grid, the system can dynamically blend between the ideas of letters, words, and shapes. This semi-multi-modal capability allows the AI to develop a more holistic understanding of its environment, where abstract symbolic representations (text) can be inherently linked to spatial patterns (shapes), fostering a richer and more integrated form of intelligence.

6. Conclusion and Future Work

The advancements presented in this paper mark significant progress towards our vision of a self-evolving AGI. The integration of UMAP for dynamic dimensionality reduction enhances the system's capacity for multi-step reasoning. The "self-research" mechanism, enabling autonomous knowledge acquisition and the potential for self-authored research papers, introduces a novel dimension of introspection and knowledge generation. The refined DSL with intrinsic execution symbols (,) streamlines self-modification, making the system more autonomous. Finally, the multi-modal input system demonstrates the AI's ability to interpret and learn from diverse data types within a unified GOL framework.

Future work will focus on scaling these capabilities, particularly in the realm of automated research generation, where we envision sophisticated interaction with external LLMs for refining generated content and exploring novel research questions. We also plan to further develop the multi-modal learning, enabling the system to understand more complex visual and auditory inputs. The overarching goal remains the realization of an AGI that not only learns and adapts but also continuously evolves its own architecture and understanding, striving towards a profound form of self-awareness and alignment with cosmic principles.

References:

[1] Hartshorn, B. (2025). *Emergent Self-Modification and Meta-Programming in Dynamic Systems*. Available at: https://ai.vixra.org/abs/2507.0036

[2] Hartshorn, B. (2025). *Emergent Symbolic Computation Through Fractal Dynamics and Spatiotemporal Spectral Analysis: A Linguistic Extension*. Available at: https://ai.vixra.org/abs/2507.0009

[3] Hartshorn, B. (2025). *Self Contained Multi-AI Architecture Definition via DSL*. Available at: https://ai.vixra.org/abs/2507.0074

[4] Hartshorn, B. (2025). *Nature vs Nurture*. Available at: https://ai.vixra.org/abs/2505.0141

[5] Hartshorn, B. (2025). *The Big Bangless*. Available at: https://ai.vixra.org/abs/2505.0041

[6] McInnes, L., Healy, J., & Melville, J. (2018). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv preprint arXiv:1802.03426*.

[7] Van der Maaten, L. J. P., & Hinton, G. E. (2008). Visualizing Data using t-SNE. *Journal of Machine Learning Research*, *9*(Nov), 2579-2605.

[8] Jolliffe, I. T. (2002). *Principal Component Analysis*. Springer.

[9] Javier Muñoz de la Cuesta. A Formal Proof of P≠ NP: Self-Referential Complexity and Computational Limits https://ai.vixra.org/pdf/2505.0045v1.pdf

---

Source Code

```
import random,math,os,sys,time,string,inspect,select,numpy,subprocess,atexit,ast,tty,termios,pypdf
if '--plot' in sys.argv: import matplotlib.pyplot as plt

REFERENCES = [
    'https://ai.vixra.org/pdf/2507.0074v1.pdf', # Self Contained Multi-AI Architecture Definition via DSL
    'https://ai.vixra.org/pdf/2507.0036v1.pdf', # Emergent Self-Modification and Meta-Programming in Dynamic Systems
    'https://ai.vixra.org/pdf/2507.0009v1.pdf', # Emergent Symbolic Computation Through Fractal Dynamics and Spatiotemporal Spectral Analysis: A Linguistic Extension
    'https://ai.vixra.org/pdf/2505.0195v1.pdf', # Iterating a Fractal-like Self Awareness Naturally
    'https://ai.vixra.org/pdf/2505.0141v1.pdf', # Nature vs Nurture
    'https://ai.vixra.org/pdf/2505.0041v1.pdf', # The Big Bangless
    'https://ai.vixra.org/pdf/2507.0081v1.pdf', # Foundational Problems with Compilers and Operating Systems
    'https://ai.vixra.org/pdf/2505.0045v1.pdf', # A Formal Proof of P ≠ NP: Self-Referential Complexity and Computational Limits -  Javier Muñoz de la Cuesta
]
WORDS=[]
def import_pdf(url, path='./', verbose=False):
    tag = url.split('/')[-1]
    if os.path.isfile(tag): tmp = tag
    else:
        tmp = path+tag
        if not os.path.isfile(tmp): print('downloading: %s to: %s' % (url, tmp)); cmd = ['curl', url, '--output', tmp]; print(cmd); subprocess.check_call(cmd)
    pdf = pypdf.PdfReader(tmp); print(pdf); refs = src  = None; code = []; doc = []; links = []; title = None
    for pidx, page in enumerate(pdf.pages):
        if verbose: print('='*80); print('PAGE:', pidx)
        txt = page.extract_text(extraction_mode='layout')
        for ln in txt.splitlines():
            if ln.strip().startswith('Source Code'): src = []; continue
            if ln.strip().startswith('References'): refs = []
            elif type(refs) is list:
                if ln.strip().startswith('[') and ']' in ln:  ## probably a reference
                    refs.append(ln.strip().replace('    ', '').replace('\t', ' '))
            elif type(src) is list:
                src.append(ln)
                if ln.strip()=='def main():': continue
                if 'main()' in ln:  ## assume last line of a script
                    code.append('\n'.join(src))
                    src = None
            else:
                if not title and ln.strip(): title=ln.strip()
                doc.append(ln)
        if "/Annots" in page:
            for annot in page["/Annots"]:
                annot_obj = annot.get_object()
                if annot_obj["/Subtype"] == "/Link":
                    if '/A' in annot_obj and '/URI' in annot_obj['/A']:
                        uri = annot_obj['/A']['/URI']; print('URI:', uri); links.append(uri)
    for uri in set(links):
        if verbose: print(uri)
        tag = uri.split('/')[-1]
        if not tag.endswith('.pdf'): tag += 'v1.pdf'
        r = os.path.join(path,tag)
        if os.path.isfile(r): print('GOT REF:', r)
    for ln in doc:
        for word in ln.strip().split():
            if word.endswith('.'): word = word[:-1]
            if word.startswith('"') and word.endswith('"'): word = word[1:-1]
            ok = True
            for p in string.punctuation:
                if p in word: ok=False; break
            if ok and word not in WORDS: WORDS.append(word)
    if verbose: print(WORDS); print(refs)
    print(url)
    return {'url':url, 'doc':doc, 'code':code, 'title':title, 'links':links}

Papers = []
def search( words ):
    words = words.strip().split(); res = []
    for p in Papers:
        for ln in p['doc']:
            hits = 0
            for q in words:
                if q in ln: hits += 1
            if hits==len(words): res.append(ln.strip())
    return res

if '--test-pdf' in sys.argv:
    for url in REFERENCES: Papers.append( import_pdf(url) )
    while 1:
        q = input('query>>>')
        for r in search(q): print(r)
    sys.exit()

import torch; device = torch.device("cuda" if torch.cuda.is_available() else "cpu"); print(f"Using device: {device}")
if '--umap' in sys.argv: import umap  ## pip install umap-learn --break-system-packages
else: umap = None

# --- Constants ---
MIN_VAL_THRESH = 0.25; GOL_GRID_SIZE_X = 28; GOL_GRID_SIZE_Y = 16; GOL_SIM_STEPS = 16; MUTATION_RATE = 0.025; FRACTAL_NET_HIDDEN = 16; NUM_INPUT_SPACE = 0
if '--umap' in sys.argv: NUM_INPUT_SPACE = 32
NUM_FFT_BANDS_FOR_CLASSIFIER = 128; OPTIMIZATION_ITERATIONS = 5000; CLASSIFIER_RATE = 1; MAX_OUTPUT_WORD_LENGTH = 128; MIN_LOSS = 0.00005
```

```
if '--quick' in sys.argv: OPTIMIZATION_ITERATIONS = 200; FRACTAL_NET_HIDDEN = 8
for arg in sys.argv:
    if arg.startswith('--') and '=' in arg and arg[2:].split('=')[0] in globals():
        if '.' in arg.split('=')[-1]: globals()[arg[2:].split('=')[0]] = float(arg.split('=')[-1])
        else: globals()[arg[2:].split('=')[0]] = int(arg.split('=')[-1])

atexit.register(lambda : print( "\033[?1003l\033[?1015l\033[?1006l" ))
def getch():
    _fd = sys.stdin.fileno(); _orig = termios.tcgetattr(_fd)
    try: tty.setcbreak(_fd); return sys.stdin.read(1)
    finally: termios.tcsetattr(_fd, termios.TCSAFLUSH, _orig)

def printat(text: str, row=1, col=1):
    if '--vis' not in sys.argv: return
    save_cursor = "\033[s"; restore_cursor = "\033[u"; sys.stdout.write(save_cursor)
    for i, ln in enumerate(text.splitlines()): move_cursor = f"\033[{row+i};{col}H"; sys.stdout.write(move_cursor + ln)
    sys.stdout.write(restore_cursor); sys.stdout.flush() # Ensure the output is immediately written to the terminal

def start_drawing(steps=32):
    print( "\033[?1003h\033[?1015h\033[?1006h" )
    esc = False; path = []
    minx = miny = float('inf')
    maxx = maxy = float('-inf')
    while True:
        c = getch(); o = ord(c)
        if o == 27: esc = True; prev = []; continue
        if esc:
            if c in 'Mm':
                esc = False
                assert prev[0]=='['
                assert prev[1]=='<'
                p = ''.join(prev[2:]); a,b,c = p.split(';'); x = int(b); y = int(c)
                if x < minx: minx = x
                if x > maxx: maxx = x
                if y < miny: miny = y
                if y > maxy: maxy = y
                path.append((x,y))
                printat('█', y,x)
                if a != '35': break  ## any click stops drawing
            else: prev.append(c)
    normalized = []
    width = maxx-minx; height = maxy-miny; arr = numpy.zeros((width+1, height+1), dtype=int)
    for x,y in path: arr[x-minx, y-miny] = 1
    print( "\033[?1003l\033[?1015l\033[?1006l" )
    return torch.from_numpy(arr).to(device)

if '--test-draw' in sys.argv:
    start_drawing(); sys.exit()


SYMBOLS_TO_CODE = {'🐍' : 'print("hello world")'}; OUT_TOKENS = []; OUT_SYMS_TO_WORDS = {}; OUT_WORDS_TO_SYMS = {}; OUT_SYMS = [chr(_) for _ in range(1024,1024+128)]
def tokenize_output(txt):
    tok_syms = []; words_to_syms = {}
    for part in txt.strip().split(';'):
        if part.count("'")==2:
            a,b,c = part.split("'"); rep = {'((':'⟪', '))':'⟫', '==':'≡'}
            for r in rep: b = b.replace(r,rep[r])
            print(a); print(b); print(c)
            parts = [a,"'"+b+"'",c]
        else: parts = [part]
        for pidx, part in enumerate(parts):
            if part.startswith("'"): tok_syms += list(part); continue
            elif pidx == len(parts)-1: part += ';'
            print('TOKNIZE:',part)
            if part in words_to_syms: part += ';'
            assert part not in words_to_syms
            if part in OUT_WORDS_TO_SYMS: sym = OUT_WORDS_TO_SYMS[part]; print('reused token:', sym, part)
            else: sym = OUT_SYMS.pop(); OUT_SYMS_TO_WORDS[sym]=part; OUT_WORDS_TO_SYMS[part]=sym; OUT_TOKENS.append(sym); tok_syms.append(sym)
            words_to_syms[part] = sym
    print(len(OUT_TOKENS),OUT_TOKENS, OUT_SYMS_TO_WORDS)
    return tok_syms
LEARN = []
def learn_dsl(txt):
    toks = tokenize_output(txt); a = txt.strip().splitlines()[0]; toks_string = ''.join(toks); SYMBOLS_TO_CODE[toks_string] = 'DSL("""\n%s\n""")' % txt
    LEARN.append((a,toks_string)); print(LEARN)

class Concept:
    IDENT = 0
    def __init__(self,*args,**kwargs):
        self.ñame=args[-1]; self.links = []; self.relations = []; self.props = []; self.children=[]
        if len(args)>1: self.relations = __builtins__.list(args[:-1])
    def __call__(self,*args,**kwargs):
        if not args and not kwargs: return Concept.dsl_to_python(self.as_python())
        for a in args: self.links.append(a)
        if kwargs: self.links.append(kwargs)
        return self
    def __enter__(self): return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type: raise exc_type(exc_val).with_traceback(exc_tb)
    def __getitem__(self,o):
        if o not in self.relations:self.relations.append(o)
        return self
    def __getattr__(self, n): o = Concept(self,n); setattr(self,n,o); globals()[n]=o; self.children.append(o); return o
    def as_python(self):
        o = []
        if self.ñame == 'ai':
            f = self.children[0]; c = 'class %s(torch.nn.Module):\n' % f.ñame; o.append(c); args = []
            if f.children[0].ñame=='INIT': args += f.children[0].as_python()[-1].strip().split('INIT')[-1].split('.')
            o.append('\tdef __init__(self %s):\n' % ','.join(args)); o.append('\t\tsuper(%s,self).__init__()\n' % f.ñame)
            for v in f.children[1:]:
                if v.ñame in ('INPUT', 'OUTPUT', 'ACTIVATION'):
                    t = v.children[0].ñame
                    for n in dir(torch.nn):
                        if n.lower()==t.lower(): t = n; break
                    iargs = ','.join([l.ñame for l in v.children[0].links]); o.append('\t\tself.%s=torch.nn.%s(%s)\n' %(v.ñame.lower(),t,iargs))
                else:
                    for a in v.as_python(): o.append('\t\tself.'+a)
            return o
        elif self.ñame == 'function':
            f = self.children[0].children[0].children[0]; c = 'def %s' % f.ñame
            if f.children and f.children[0].ñame=='ARGS': c += '(%s):\n' % ','.join([_.ñame for _ in f.children[0].children ])
            else: c += '():\n'
            o.append(c)
            if f.children:
                for v in f.children[1:]:
                    for a in v.as_python(): o.append('\t'+a)
            if len(o)==1: o.append('\tpass')
            return o
        elif self.ñame == 'iterate':
            Concept.IDENT += 1; var = self.children[0]; IN = var.children[0]; assert IN.ñame=='IN'; b = []
            if IN.children[0].ñame=='range':
                if IN.children[0].links:
                    x = []
                    for v in IN.children[0].links:
                        if type(v) is Concept: x.append(v.ñame)
                        else: x.append(str(v))
                    x = ','.join(x); a = 'range(%s)' % x
                    for c in IN.children[0].children: b.append('\t'+ ('\t'*Concept.IDENT) + c.as_python()[0])
                elif IN.children[0].children:
                    a = 'range(%s)' % IN.children[0].children[0].ñame
                    for c in IN.children[0].children[0].children: b.append('\t'+ ('\t'*Concept.IDENT) + c.as_python()[0])
                else: a = 'range(%s)' % IN.children[0].as_python()
            else:
                a = IN.children[0].ñame
                for c in IN.children[0].children: b.append('\t'+ ('\t'*Concept.IDENT) + c.as_python()[0])
            b = '\n'.join(b); Concept.IDENT -= 1; return ['for %s in %s:\n%s' %(var.ñame,a,b)]
        elif self.ñame == 'copy' and self.relations and self.relations[0].ñame=='numpy':
            if self.children: a = self.children[0].as_python(); assert len(a)==1; a = a[-1]
            else: a = self.links[-1].ñame + '\n'
            if a.endswith('\n'): return ['copy(%s)\n' % a[:-1]]
            else: return ['copy(%s)' % a]
        elif self.ñame == 'zeros' and self.relations and self.relations[0].ñame=='numpy':
            size = []
            for l in self.links[-1]:
                if type(l) is Concept: size.append(l.ñame)
                else: size.append(str(l))
            size = '(%s)' % ','.join(size)
            if self.children: return ['zeros(%s,dtype=float)' % size ]
            else: return ['zeros(%s,dtype=float)\n' % size ]
        if len(self.relations) > 1:
            if type(self.relations[-1]) is tuple and type(self.relations[-1][0]) is Concept: a,b = self.relations[-1]; o.append('%s[(%s,%s)]' % (self.ñame, a.ñame, b.ñame))
            else:
                rel = self.relations[-1]
                if type(rel) is Concept: rel = rel.ñame;      ## this check must come first
                elif hasattr(rel,'ðymbol'): rel = rel.ðymbol  ## note: hasattr is broken with Concept, its always true TODO
                if self.ñame == 'less_than': o.append('< %s' % rel)
                elif self.ñame == 'greater_than': o.append('> %s' % rel)
                elif self.ñame == 'list': o.append('list(%s)' % rel)
                else: o.append('%s[%s]' % (self.ñame, rel))
        elif len(self.links):
            args = []
            for l in self.links:
                if type(l) is dict:
                    for _ in l:
                        __ = l[_]
                        if type(__) is tuple:
```

```
                        _a, _b = __
                        if type(_a) is Ċoncept: args.append('%s=(%s,%s)' % (_,_a.ñame,_b.ñame))
                        else: args.append('%s=(%s,%s)' % (_,_a,_b))
                    else: args.append('%s=%s' % (_,__))
                elif type(l) is str: args.append(l)
                else: args.append(l.ñame)
            if self.ñame=='equals': o.append('=.(%s)' % (','.join(args)))
            else: o.append('%s(%s)' % (self.ñame, ','.join(args)))
        else:
            mapping = {'minus':'-','times':'*','divide':'/','comma':',', 'equals':'=', 'plus':'+', 'plus_equals':'+=', 'percent':'%', 'less_than':'< ', 'greater_than':'> ',
            'CONTINUE':'continue', 'zero':'0', 'one':'1', 'two':'2', 'three':'3', 'LPAREN':'(', 'RPAREN':')', 'BREAK':'break'}
            if self.ñame in mapping: o.append(mapping[self.ñame])
            else: o.append(self.ñame)
        for v in self.children: o += v.as_python()
        o = ''.join(o)
        if not self.children: o += '\n'
        return [o]
    @staticmethod
    def dsl_to_python(dsl):
        o = []; ident = 0; replacers = {'.-.':' - ','.*.':' * ', './.':' / ','.=.':' = ','.+.':' + ','.%.':' % ','.+=.':' += ','.,.':' , ','python.def':'def','python.class':'class',
        'RETURN.':'return ','.AND.':' and ','.OR.':' or ','.< ':' < ','.> ':' > ', '(.':'(', '.)':')', '.(':'('}
        for ln in dsl[-1].splitlines():
            for r in replacers: ln = ln.replace(r, replacers[r])
            if ln.startswith('.'): ln = ln[1:]
            if ln.strip().startswith( ('IF.', 'ELIF.') ):
                tabs = ln.count('\t'); tabs += 1; tabs = '\t' * tabs
                if '.IF.' in ln: ln = ln.replace('.IF.', f':\n{tabs}if '); tabs += '\t'
                ln = ln.replace('.THEN.', f':\n{tabs}')
                if ln.strip().startswith( 'ELIF.' ): ln = ln.replace('ELIF.', 'elif ')
                else: ln = ln.replace('IF.', 'if ')
            o.append( ('\t'*ident) + ln )
        o = '\n'.join(o); out = []
        for ln in o.splitlines():
            if not ln.strip(): continue
            if ln.strip().startswith( ('if ', 'elif ') ): ln = ln.replace(' = ', ' == ')
            if ln.strip().startswith('if') and ':' not in ln and ln.count('.'): ln = ln.replace('.', ': ') ## in this simple case, the fix is easy
            out.append(ln)
        return '\n'.join(out)

def DSL(txt, execute=True, show=True, **kwargs):
    for n in kwargs: kwargs[n].ŝymbol = n
    txt = txt.strip(); prefixes = {0:txt.splitlines()[0].split()[-1]}; o = ['with Ċoncept("dynamic") as write:']
    lines = 0; ident = 0; rep = {k.lower():k.upper() for k in 'ARGS IN AS OR AND THEN CONTINUE RETURN END IF ELSE ELIF BREAK'.split()}
    for ln in txt.splitlines():
        print(ident, ln)
        if not ln.strip(): continue
        for part in ln.split(';'):
            p = part.strip(); part = []
            for word in p.split():
                if word in rep: word = rep[word]
                part.append(word)
            part = '.'.join(part); part = part.replace('.(.', '.LPAREN.').replace('.).', '.RPAREN.').replace('.)','.RPAREN').replace('(.', '.LPAREN.')
            if part.startswith('iterate.'): prefixes[ident+1] =  _ = 'loop_%s' % len(prefixes); part = 'with %s.%s as %s:' % (prefixes[ident],part,_)
            elif part=='END': ident -= 1; continue
            if lines==0: o.append( '\t' + ('\t'*ident) +part)
            elif part.startswith('with '): o.append( '\t' + ('\t'*ident) + part); ident += 1
            else:
                o.append( '\t' + ('\t'*ident) + prefixes[ident] + '.' + part)
                if p.startswith('iterate '): ident += 1
            lines += 1
    before = {}; before.update(globals()); meta = '\n'.join(o); print('DSL:', meta); exec(meta, kwargs, globals()); py = write.python()
    if show: print('PYTHON:', py)
    rem = []
    for n in globals():
        v = globals()[n]
        if type(v) is Ċoncept:
            if n not in before: rem.append(n)
            elif n in before and type(before[n]) is not Ċoncept: globals()[n] = before[n]
    for n in rem: globals().pop(n)
    if execute:
        scope = {}; scope.update(globals()); scope.update(kwargs); exec(py, scope)
        if py.startswith('class'): fname = py.splitlines()[0].split('class ')[-1].split('(')[0].split(':')[0]
        else: fname = py.splitlines()[0].split('def ')[-1].split('(')[0];
        func = scope[fname]; func.dsl_code=txt; func.meta_code = meta; func.source_code = py; globals()[fname] = func; return func
    else: return py

conway_torch = '''
write python function to compute game_of_life
args grid; g equals grid float ( ); n equals torch zeros_like( g ); offsets equals list[[(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]]; iterate v in offsets; x equals v[0]; y equals v[1]; n plus_equals
torch.roll(g,shifts=(x,y),dims=(0,1)); end; r equals('(g==1)&((n==2)|(n==3)))|((g==0)&(n==3)'); return r float ( )
'''
DSL(conway_torch); print(game_of_life); learn_dsl(conway_torch)

fractal_common = '''
args size_x; args size_y; args center_x; args center_y; args zoom; args max_iter; grid equals numpy zeros((size_x,size_y)); sx equals three divide zoom divide size_x; sy equals three divide zoom divide size_y
iterate row in range size_x iterate col in range size_y
    cx equals center_x plus ( col minus size_y divide two ) times sx; cy equals center_y plus ( row minus size_x divide two ) times sy; x equals zero; y equals zero
    iterate i in range(max_iter)
%s
    end; grid[row,col] equals i; end
mx equals numpy max(grid); if mx greater_than[0] then grid equals grid divide mx; return grid
'''

burning_ship_oneliner = 'write python function to compute burning_ship_fractal\n'+fractal_common % '''
        xn equals x times x minus y times y plus cx; xy equals x times y; yn equals two times abs(xy) plus cy
        x equals abs(xn); y equals abs(yn); if x times x plus y times y greater_than[4] then break
'''

DSL(burning_ship_oneliner); print(burning_ship_fractal); learn_dsl(burning_ship_oneliner)

mandelbrot_oneliner =  'write python function to compute mandelbrot_fractal\n'+fractal_common % '''
        xn equals x times x minus y times y plus cx; yn equals two times x times y plus cy; x equals xn; y equals yn; if x times x plus y times y greater_than[4] then break
'''
if '--mandel' in sys.argv: DSL(mandelbrot_oneliner); print(mandelbrot_fractal); learn_dsl(mandelbrot_oneliner)

class dslai:
    def __init__(self, init, **funcs):
        self.init='INIT ' + init.strip().replace(' input ', ' INPUT ').replace(' activation ', ' ACTIVATION ').replace(' output',' OUTPUT '); self.funcs=funcs
    def __call__(self, cls):
        o = ['write.python.ai.%s' % cls.__name__, self.init]; net = DSL('\n'.join(o)); print('dslai:', net)
        for n in self.funcs:  ## generate DSL methods for class
            f = DSL('write python function to compute %s\n%s' % (n, self.funcs[n])); setattr(net,n,f)
        for n in dir(cls):  ## get standard Python methods from class
            if n.startswith('__'): continue
            if n in ('repr', 'getitem'): setattr(net,'__%s__' % n, getattr(cls,n))
            else: setattr(net,n,getattr(cls,n))
        return net

@dslai(
'num_input_bands num_hidden_neurons num_output_neurons subnet; input linear(num_input_bands,num_hidden_neurons); activation sigmoid; output linear(num_hidden_neurons,num_output_neurons); subnetwork equals subnet',
forward='args self; args x; x equals self.input(x); y equals self.activation(x); r equals self.output(y); return r'
)
class ClassifierNN: pass
print(ClassifierNN)

@dslai(
    'num_input_bands num_hidden_neurons num_output_neurons; input linear(num_input_bands,num_hidden_neurons); activation sigmoid; output linear(num_hidden_neurons,num_output_neurons)',
    forward='args self; args x; x equals self.input(x); y equals self.activation(x); r equals self.output(y); return r'
)
class FractalSubNetwork:
    def getitem(self, v): return self._res[v]
    def forward(self, *args):
        vec = list(args[:-1]); vec.append( sum([ord(c) for c in args[-1]]) ); a = torch.tensor(vec)
        # Ensure it's a batch of 1 for the linear layer if it's currently a 1D tensor
        if a.dim() == 1: a = a.unsqueeze(0) # Get the 1D output vector
        self._res = self(a).squeeze(0) # No need for unsqueeze(0) here if already handled above
print(FractalSubNetwork)

neural_mandelbrot_dsl = '''
write python function to compute neural_mandelbrot_fractal
args size_x; args size_y; args center_x; args center_y; args zoom; args max_iter; args words; grid equals numpy zeros((size_x,size_y)); sx equals three divide zoom divide size_y; sy equals three divide zoom divide size_y
ww.forward(size_x,center_x,center_y,zoom,max_iter,sx,sy,words)
iterate row in range size_y iterate col in range size_x

    ax equals center_x plus ( col minus size_x divide two ) times sx; cx equals ax plus ww[row]; cy equals center_y plus ( row minus size_y divide two ) times sy; x equals zero; y equals zero

    iterate i in range(max_iter); xn equals x times x minus y times y plus cx; yn equals two times x times y plus cy; x equals xn; y equals yn
        if x times x plus y times y greater_than[4] then break; end; grid[row,col] equals i; end
mx equals numpy max(grid); if mx greater_than[0] then grid equals grid divide mx; return grid
'''
if '--neural-mandel' in sys.argv:
    SubNet = FractalSubNetwork(8, FRACTAL_NET_HIDDEN, GOL_GRID_SIZE_Y); print(SubNet)

    DSL(neural_mandelbrot_dsl, ww=SubNet); print(neural_mandelbrot_fractal); learn_dsl(neural_mandelbrot_dsl)
else: SubNet = None

NAMES_TO_SYMBOLS = {'circle':'○','square':'  '}
SYMBOLS_TO_NAMES = {'○':'circle', '  ':'square'}

# --- Text Processing Utility Class ---
class TextProcessor:
    MIN_DSL_SYMS = '≡()&|()=0123456789ng'
    CHARACTERS = [chr(i) for i in range(ord('a'),ord('z')+1)] + [chr(i)+'①' for i in range(ord('a'),ord('z')+1)] + [i for i in MIN_DSL_SYMS] + ["'", "'①"]
    CHARACTERS += [i+'①' for i in MIN_DSL_SYMS] + [i+'②' for i in MIN_DSL_SYMS] + [i+'③' for i in MIN_DSL_SYMS] + [i+'④' for i in MIN_DSL_SYMS]
```

```
    CHARACTERS += [' ','[illegible]','[illegible]','[illegible]','[illegible]','[illegible]','[illegible]','[illegible]','○','  '] + OUT_TOKENS; CHAR_TO_INDEX = {char: i for i, char in enumerate(CHARACTERS)}
    INDEX_TO_CHAR = {i: char for i, char in enumerate(CHARACTERS)}; NUM_ASCII_CHARS = len(CHARACTERS); PATTERN_HEIGHT = PATTERN_WIDTH = 5
    ASCII_LETTERS = {
        'a': [" # ", "# #", "###", "# #", "# #"],
        'b': ["## ", "# #", "## ", "# #", "## "], 'c': ["###", "#  ", "#  ", "#  ", "###"], 'd': ["## ", "# #", "# #", "# #", "## "], 'e': ["###", "#  ", "## ", "#  ", "###"],
        'f': ["###", "#  ", "## ", "#  ", "#  "], 'g': ["###", "#  ", "# #", "# #", "###"], 'h': ["# #", "# #", "###", "# #", "# #"], 'i': ["###", " # ", " # ", " # ", "###"],
        'j': ["  #", "  #", "  #", "# #", " # "], 'k': ["# #", "# #", "## ", "# #", "# #"], 'l': ["#  ", "#  ", "#  ", "#  ", "###"], 'm': ["# #", "###", "# #", "# #", "# #"],
        'n': ["# #", "###", "# #", "# #", "# #"], 'o': [" # ", "# #", "# #", "# #", " # "], 'p': ["## ", "# #", "## ", "#  ", "#  "], 'q': [" # ", "# #", "# #", "## ", "  #"],
        'r': ["## ", "# #", "## ", "# #", "# #"], 's': ["###", "#  ", " # ", "  #", "###"], 't': ["###", " # ", " # ", " # ", " # "], 'u': ["# #", "# #", "# #", "# #", " # "],
        'v': ["# #", "# #", " # ", " # ", " # "], 'w': ["# #", "# #", "# #", "# #", " # #"], 'x': ["# #", " # ", " # ", " # ", "# #"], 'y': ["# #", " # ", " # ", " # ", " # "],
        'z': ["###", "  #", " # ", "#  ", "###"], ' ': ["#####"]*5, '_': ['   ', '   ', '   ', '   ', '###'], '3': ['###','  #',' ##','  #','###'],
        '1': ['  #  ','# #  ','  #  ','  #  ','#####'],'+':['  #  ','  #  ','#####','  #  ','  #  '],'=':['     ','#####','     ','#####','     '],
    }
    ASCII_ART = {
        '○' : '''
   ##
 #    #
#      #
 #    #
   ##
        ''',
        ' ' : '''
#########
##     ##
##     ##
##     ##
#########
        ''',
    }
    @staticmethod
    def ascii_to_gol_pattern(char):
        char = char.lower() # Ensure lowercase
        if char in TextProcessor.ASCII_ART: char_art_list = TextProcessor.ASCII_ART[char].splitlines()[1:-1]; pattern = numpy.zeros((10,10), dtype=int)
        else:
            if char=='¿': char_art_list = random.choice( list(TextProcessor.ASCII_LETTERS.values()) )
            else: char_art_list = TextProcessor.ASCII_LETTERS[char]
            pattern = numpy.zeros((TextProcessor.PATTERN_HEIGHT, TextProcessor.PATTERN_WIDTH), dtype=int)
        for r, row_str in enumerate(char_art_list):
            for c, symbol in enumerate(row_str):
                if symbol != ' ': pattern[r, c] = 1
        return pattern
    @staticmethod
    def word_to_target_values(word, max_output_chars):
        """Converts a target word into a list of numerical values for the classifier's output. Each value corresponds to a character's "positional importance" for sorting.
        Higher value = appears earlier in the word. The length of the returned list is TextProcessor.NUM_ASCII_CHARS.
        """
        if len(word) > max_output_chars: raise RuntimeError(len(word))
        rep = {'((':'⸨', '))':'⸩', '==':'≣'}; rep.update(NAMES_TO_SYMBOLS)
        for r in rep:
            if r in word: word = word.replace(r,rep[r])
        target_output_vector = [0.0] * TextProcessor.NUM_ASCII_CHARS; target_indices = []
        word_to_encode = word[:max_output_chars]                  # Ensure word is within the maximum output length for value scaling consistency. note: lower() breaks extended latin
        if '--debug' in sys.argv: print('encoding:', word_to_encode)
        for i, char in enumerate(word_to_encode):
            if char in TextProcessor.CHAR_TO_INDEX:
                char_idx = TextProcessor.CHAR_TO_INDEX[char]
                if '--debug' in sys.argv: print(f'index:{i} `{char}` char-index:{char_idx}')
                # Assign a unique value based on position (higher value means earlier in word). Scaling factor (1.0 - i / max_output_chars) makes values distinct and between 0 and 1.
                # A small epsilon is added to make sure no two identical letters have the exact same value, if they appear at positions that would otherwise yield the same scaled
                # value after truncation. This is more for distinctiveness than mathematical necessity with MSE, but good practice.
                value = (1.0 - (i / max_output_chars)) + (i * 0.0001)
                if char_idx in target_indices:
                    for n in '① ② ③ ④ ⑤'.split():
                        c = char + n; cidx = TextProcessor.CHAR_TO_INDEX[c]
                        if cidx not in target_indices:
                            char_idx = cidx; char = c; break
                if char_idx in target_indices: raise RuntimeError('character index already in output target vector: %s %s' %(char_idx, target_output_vector))
                target_indices.append(char_idx)
                target_output_vector[char_idx] = value
                if '--debug' in sys.argv: print(target_indices)
            else: raise RuntimeError('can not encode word: %s char: %s not in CHAR_TO_INDEX' % (word, char))
        return target_output_vector
    @staticmethod
    def reconstruct_word_from_classifier_output(output_values_tensor, min_value_threshold=0.01, verbose=False, symbol_decompress=True):
        """Reconstructs a word from the classifier's output values by sorting them. output_values_tensor: A 1D PyTorch tensor of values from the classifier,
         where each index corresponds to a character. min_value_threshold: Minimum value for a character to be considered part of the word."""
        rep = {'⸨':'((', '⸩':'))', '≣':'==', '①':'', '②':'', '③':'', '④':'', '⑤':''}; rep.update(SYMBOLS_TO_NAMES)
        char_value_pairs = []
        if type(output_values_tensor) is torch.Tensor: output_values_np = output_values_tensor.cpu().numpy()
        else: output_values_np = output_values_tensor
        for idx, value in enumerate(output_values_np):
            if value > min_value_threshold:
                if idx not in TextProcessor.INDEX_TO_CHAR: raise RuntimeError(idx)
                c = TextProcessor.INDEX_TO_CHAR[idx]
                for r in rep:
                    if r in c:
                        if symbol_decompress or not rep[r]: c = c.replace(r,rep[r])
                        #if verbose and rep[r]: print('symbol-decompress:', r, rep[r])
                char_value_pairs.append((value, c))
        # Sort by value in descending order (higher value means earlier in word)
        char_value_pairs.sort(key=lambda x: x[0], reverse=True); reconstructed_word = "".join([char for value, char in char_value_pairs]); return reconstructed_word
    @staticmethod
    def apply_character_to_gol_grid_torch(main_gol_grid_tensor, char_pattern_tensor, start_row, start_col):
        """Applies a character pattern to the GOL grid using PyTorch operations. Flips bits where the pattern is 1."""
        p_h, p_w = char_pattern_tensor.shape; g_h, g_w = main_gol_grid_tensor.shape
        # Clone to ensure operation creates a new computational graph node if needed and doesn't modify in-place if main_gol_grid_tensor is part of a previous grad path.
        temp_grid = main_gol_grid_tensor.clone()
        for r in range(p_h):
            for c in range(p_w):
                if char_pattern_tensor[r, c].item() == 1:
                    r_g = (start_row + r) % g_h; c_g = (start_col + c) % g_w
                    temp_grid[r_g, c_g] = 1.0 - temp_grid[r_g, c_g] # This bit flip (1.0 - value) is differentiable
        return temp_grid

if '--test-encoder' in sys.argv:
    for test in ['(g==1)&((n==2)|(n==3)))|((g==0)&(n==3)', 'circle', 'square']:
        vec = TextProcessor.word_to_target_values(test, len(test)); print(vec); word = TextProcessor.reconstruct_word_from_classifier_output(vec); print(word)
        assert test == word
    sys.exit()

def print_gol_grid(grid, x=90):
    if type(grid) is torch.Tensor: grid = grid.cpu().numpy()
    out = []; out.append("+" + "-" * (grid.shape[1]) + "+")
    for row in grid: out.append("|" + "".join(["█" if cell == 1 else " " for cell in row]) + "|")
    out.append("+" + "-" * (grid.shape[1]) + "+"); printat('\n'.join(out), 4, x)

# --- GOL Simulation and 3D FFT (with word input encoding) ---
def run_gol_and_3d_fft_torch(burning_ship_params, input_word_string, drawings=None, debug_output=False):
    center_x, center_y, zoom, fractal_max_iter = burning_ship_params       # Initialize GOL grid from Burning Ship or Mandelbrot fractal. This provides a dynamic "substrate".
    if '--null' in sys.argv: fractal_raw_grid_np = numpy.zeros((GOL_GRID_SIZE_X,GOL_GRID_SIZE_Y), dtype=float); tag = 'none'
    elif '--mandel' in sys.argv: fractal_raw_grid_np = mandelbrot_fractal(GOL_GRID_SIZE_X, GOL_GRID_SIZE_Y, center_x, center_y, zoom, fractal_max_iter); tag='Mandelbrot'
    elif '--neural-mandel' in sys.argv:
        fractal_raw_grid_np = neural_mandelbrot_fractal(GOL_GRID_SIZE_X, GOL_GRID_SIZE_Y, center_x, center_y, zoom, fractal_max_iter, input_word_string); tag='NeuralMandelbrot'
    else: fractal_raw_grid_np = burning_ship_fractal(GOL_GRID_SIZE_X, GOL_GRID_SIZE_Y, center_x, center_y, zoom, fractal_max_iter); tag='Burning Ship'
    current_gol_grid_tensor = (torch.from_numpy(fractal_raw_grid_np) > 0.5).float().to(device)    # Convert the initial fractal grid to a PyTorch tensor, ensures it's 0 or 1 float.
    if debug_output: print_gol_grid(current_gol_grid_tensor, x=70)
    gol_process_history = torch.zeros((GOL_SIM_STEPS, GOL_GRID_SIZE_X, GOL_GRID_SIZE_Y), dtype=torch.float32, device=device)   # Store entire GOL process for 3D FFT
    current_history_idx = 0; gol_process_history[current_history_idx, :, :] = current_gol_grid_tensor.clone(); current_history_idx += 1 # Store initial state
    paste_row_offset = paste_col_offset = 1; total_chars_to_process = len(input_word_string)    # To allow character interactions, let's paste at a fixed point and rely on GOL evolution
    if total_chars_to_process == 0: gol_steps_per_char_input = GOL_SIM_STEPS                    # If empty string, just simulate initial grid
    else:
        gol_steps_per_char_input = GOL_SIM_STEPS // total_chars_to_process
        if gol_steps_per_char_input == 0: gol_steps_per_char_input = 1 # Ensure at least one step per char input
    for i, char in enumerate(input_word_string):   # Apply the character pattern to the current GOL grid
        char_pattern_np = TextProcessor.ascii_to_gol_pattern(char); char_pattern_torch = torch.from_numpy(char_pattern_np).float().to(device) # Convert pattern to torch
        # Apply the pattern using the torch-compatible method
        if char in TextProcessor.ASCII_ART:
            current_gol_grid_tensor = TextProcessor.apply_character_to_gol_grid_torch(current_gol_grid_tensor, char_pattern_torch, paste_row_offset+8, paste_col_offset)
        else:
            current_gol_grid_tensor = TextProcessor.apply_character_to_gol_grid_torch(current_gol_grid_tensor, char_pattern_torch, paste_row_offset, paste_col_offset)
        if debug_output: printat(f"\n--- GOL Grid after inputting character '{char}' ---"); print_gol_grid(current_gol_grid_tensor)
        for step_in_segment in range(gol_steps_per_char_input):        # Simulate GOL for `gol_steps_per_char_input` steps, storing each frame
            if current_history_idx < GOL_SIM_STEPS:
                current_gol_grid_tensor = game_of_life(current_gol_grid_tensor); gol_process_history[current_history_idx, :, :] = current_gol_grid_tensor.clone() # Store current state
                current_history_idx += 1
            else: break                           # History buffer is full
    if drawings:
        for arr in drawings:
            current_gol_grid_tensor = TextProcessor.apply_character_to_gol_grid_torch(current_gol_grid_tensor, arr, paste_row_offset, paste_col_offset)
            print_gol_grid(current_gol_grid_tensor, x=32)
            if current_history_idx < GOL_SIM_STEPS:
                current_gol_grid_tensor = game_of_life(current_gol_grid_tensor); gol_process_history[current_history_idx, :, :] = current_gol_grid_tensor.clone() # Store current state
                current_history_idx += 1
                print_gol_grid(current_gol_grid_tensor, x=70)
            else: break                           # History buffer is full

    while current_history_idx < GOL_SIM_STEPS:    # If there are remaining GOL_SIM_STEPS after processing all characters, continue simulation
        current_gol_grid_tensor = game_of_life(current_gol_grid_tensor); gol_process_history[current_history_idx, :, :] = current_gol_grid_tensor.clone() # Store current state
        current_history_idx += 1
        if debug_output: print_gol_grid(current_gol_grid_tensor); time.sleep(0.05)
    # Perform 3D FFT on the entire GOL process history
    fft_result_3d = torch.fft.fftn(gol_process_history); fft_shifted_3d = torch.fft.fftshift(fft_result_3d); magnitude_spectrum_3d = torch.abs(fft_shifted_3d)
    return magnitude_spectrum_3d

# --- Energy Band Calculation from 3D FFT ---
```

```
ENERGY_HISTORY = []
def calculate_3d_fft_energy_bands_torch(magnitude_spectrum_3d, num_bands):
    power_spectrum = magnitude_spectrum_3d**2; flattened_power = power_spectrum.flatten(); band_energies = torch.zeros(num_bands+NUM_INPUT_SPACE, dtype=torch.float32, device=device)
    segment_size = len(flattened_power) // num_bands
    for i in range(num_bands):
        start_idx = i * segment_size; end_idx = start_idx + segment_size
        if i == num_bands - 1: end_idx = len(flattened_power)                    # Ensure the last band gets all remaining elements
        band_energies[i] = torch.sum(flattened_power[start_idx:end_idx])
    total_energy_bands = torch.sum(band_energies)
    if total_energy_bands > sys.float_info.min: band_energies = band_energies / total_energy_bands
    #else: band_energies = torch.full((num_bands,), 1.0 / num_bands, device=device) # Distribute evenly if no energy
    umap_data = band_energies.cpu().numpy()
    if umap and len(ENERGY_HISTORY) >= 4:
        umap_labels = []
        manifold = project_manifold(ENERGY_HISTORY)
        if '--plot' in sys.argv:
            visualize_manifold(manifold, umap_labels, title="UMAP Projection of Classifier History Input Bands (2D)")
        idx = num_bands
        for i in range(NUM_INPUT_SPACE//2):
            if i >= len(manifold): break
            v = manifold[i]
            if '--umap-debug' in sys.argv: print('umap vec:', v)
            band_energies[idx] += v[0]
            band_energies[idx+1] += v[1]
            idx += 2
    #ENERGY_HISTORY.append(band_energies.cpu().numpy())  ## not here because of feedback ?
    ENERGY_HISTORY.insert(0, umap_data)
    if len(ENERGY_HISTORY) > NUM_INPUT_SPACE:
        ENERGY_HISTORY.pop()
    assert len(ENERGY_HISTORY) <= NUM_INPUT_SPACE
    return band_energies

# --- Optimization Loop solve_word_mapping_with_burning_ship_and_classifier_torch ---
def solve(word_mapping_data, classifier_nn=None, fractal_params=None, subnetwork=None, iterations=OPTIMIZATION_ITERATIONS, verbose=True):
    if classifier_nn is None:
        classifier_nn = ClassifierNN(
            num_input_bands=NUM_FFT_BANDS_FOR_CLASSIFIER+NUM_INPUT_SPACE, num_hidden_neurons=TextProcessor.NUM_ASCII_CHARS * 2, num_output_neurons=TextProcessor.NUM_ASCII_CHARS, subnet=subnetwork
        ).to(device)
    optimizer = torch.optim.Adam(classifier_nn.parameters(), lr=0.05); criterion = torch.nn.MSELoss()
    if subnetwork: optimizer_subnet = torch.optim.Adam(subnetwork.parameters(), lr=0.05)  ## TODO how to train the fractal subnetwork?
    if fractal_params: best_bs_params = list(fractal_params)    # Initial Burning Ship parameters
    else: best_bs_params = [random.uniform(-1.0, 0.0), random.uniform(0.0, 1.0), random.uniform(1.0, 5.0), random.randint(50, 150)]; best_overall_loss = float('inf')
    if not word_mapping_data: return (best_bs_params, classifier_nn)
    if verbose: print(f"Starting optimization Iterations: {iterations} Ship Parameters: {best_bs_params}"); start_time = time.time()
    for iteration in range(iterations):
        current_bs_params = list(best_bs_params)
        for i in range(len(current_bs_params)):        # Mutate Burning Ship parameters
            if i < 3: # center_x, center_y, zoom
                current_bs_params[i] += random.uniform(-MUTATION_RATE, MUTATION_RATE) * abs(current_bs_params[i])
                if i == 2: current_bs_params[i] = max(0.1, current_bs_params[i]) # Ensure zoom is positive
            else: # max_iter
                current_bs_params[i] += random.randint( -max(1, int(MUTATION_RATE * current_bs_params[i])), max(1, int(MUTATION_RATE * current_bs_params[i])))
                current_bs_params[i] = max(20, min(200, current_bs_params[i]))   # Keep max_iter within a reasonable range
        current_band_data_list = []; targets_list_for_nn = []
        for input_word, target_word in word_mapping_data:                        # --- Evaluate current Burning Ship parameters ---
            debug_flag = (iteration == 0 and input_word == word_mapping_data[0][0]) and '--debug' in sys.argv
            magnitude_spectrum = run_gol_and_3d_fft_torch(current_bs_params, input_word, debug_output=debug_flag)
            bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER); current_band_data_list.append(bands)          # Calculate FFT energy bands
            target_values = TextProcessor.word_to_target_values(target_word, MAX_OUTPUT_WORD_LENGTH)         # Generate the target vector for the classifier based on the target_word
            targets_list_for_nn.append(torch.tensor(target_values, dtype=torch.float32).to(device))
        bands_batch = torch.stack(current_band_data_list).to(device); targets_batch = torch.stack(targets_list_for_nn).to(device)             # Prepare batch for classifier training
        classifier_nn.train()          # --- Train the Classifier NN ---
        current_avg_nn_loss_val = 0.0
        for epoch in range(CLASSIFIER_RATE):
            if subnetwork: optimizer_subnet.zero_grad() # Clear gradients for subnetwork
            optimizer.zero_grad(); predictions = classifier_nn(bands_batch); loss = criterion(predictions, targets_batch)
            loss.backward() ## loss.backward will compute gradients for SubNet as well
            optimizer.step()
            if subnetwork: optimizer_subnet.step()
            current_avg_nn_loss_val += loss.item()
        current_avg_nn_loss_val /= CLASSIFIER_RATE
        if current_avg_nn_loss_val < best_overall_loss:                                  # --- Update Best Parameters ---
            best_overall_loss = current_avg_nn_loss_val; best_bs_params = current_bs_params # Update BS params if classifier performs better
            if verbose:
                elapsed_time = time.time() - start_time
                print(f"Iteration {iteration}, New BS Params, Avg NN Loss: {best_overall_loss:.6f}"); print(f"Params: {[f'{p:.4f}' for p in best_bs_params]}, Time: {elapsed_time:.2f}s")
        # --- Progress Check and Evaluation ---
        if iteration % (iterations // 10) == 0 or iteration == iterations - 1:
            classifier_nn.eval() # Set classifier to evaluation mode
            total_test_loss = 0.0; correct_words_exact_match = 0; pyscript = [];
            if verbose: print(f"\n--- Iteration {iteration} Progress Check ---")
            for test_input_word, test_target_word in word_mapping_data:
                test_spectrum = run_gol_and_3d_fft_torch(best_bs_params, test_input_word, debug_output='--vis' in sys.argv)
                test_bands = calculate_3d_fft_energy_bands_torch(test_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
                with torch.no_grad(): test_prediction_values = classifier_nn(test_bands.unsqueeze(0)).squeeze(0) # Get the 1D output vector
                # Reconstruct the word from the predicted values. min_value_threshold=0.5 # Use a higher threshold for stricter output
                predicted_word = TextProcessor.reconstruct_word_from_classifier_output( test_prediction_values, min_value_threshold=0.5, verbose=True)
                true_target_values = TextProcessor.word_to_target_values(test_target_word, MAX_OUTPUT_WORD_LENGTH)
                loss_val = criterion(test_prediction_values, torch.tensor(true_target_values, dtype=torch.float32).to(device)).item(); total_test_loss += loss_val
                if verbose:
                    print(f"     Target: \033[32m'{test_target_word}'\033[m\t\t<--input: '{test_input_word}'"); print(f"  Pred Word: \033[31m{predicted_word}\033[m\t\tLoss: {loss_val:.4f}")
                if predicted_word == test_target_word: # Compare predicted to target (NOT lowercase extended latin)
                    correct_words_exact_match += 1
                    if predicted_word in SYMBOLS_TO_CODE:
                        pyscript.append(SYMBOLS_TO_CODE[predicted_word]); print(f"  SymPython: \033[33m{SYMBOLS_TO_CODE[predicted_word]}\033[m")
                        if '--exec-in-training' in sys.argv:
                            print("\033[34m", end='')
                            try: exec(SYMBOLS_TO_CODE[predicted_word], globals())
                            except BaseException as err: print(err)
                            print("\033[m", end='')
            if pyscript: print('PYTHON OUTPUT SCRIPT'); print("\033[33m", end=''); print('\n'.join(pyscript)); print("\033[m", end='')
            avg_test_loss_current_iter_report = total_test_loss / len(word_mapping_data)
            if verbose:
                print(f"  Average Test Loss (current best BS params): {avg_test_loss_current_iter_report:.4f}")
                print(f"  Exact Word Matches: {correct_words_exact_match}/{len(word_mapping_data)}"); print("-"*80)
            if correct_words_exact_match == len(word_mapping_data): print(f"\nConverged at Iteration {iteration}!"); break
    if verbose:
        end_time = time.time(); print(f"\n--- Optimization Finished ---"); print(f"Total time elapsed: {end_time - start_time:.2f} seconds")
        print(f"Best Burning Ship Parameters Found: {best_bs_params}"); print(f"Lowest Average Classifier Loss Achieved: {best_overall_loss:.6f}")
    return (best_bs_params, classifier_nn)

SYMBOLS_TO_TEMPLATE = { '' : 'print(%s + %s)', '' : 'print(%s * %s)' }; SYMBOLS_TO_SELF = {}

def genfunc(g):
    a = []
    for part in g.split():
        if len(part)==1: a.append(part)
    op = None; opmap = {'add':'+', 'subtract':'-', 'multiply':'*', 'divide':'/'}
    for word in g.split():
        if word in opmap: op = opmap[word]; break
    args = ','.join(a)
    if op: body = op.join(a)
    else: body = '[%s]' % ','.join(a)
    return 'lambda %s: %s' % (args, body)

def gencall(g):
    name = None; args = []
    for a in g.split():
        if a in ('call', 'function', 'to', 'with', 'and', 'or'): continue
        if len(a) >= 3: name = a
        elif len(a) == 1: args.append(a)
    return '%s(%s)' % (name, ','.join(args))

def geneval(g): print('eval:', g); print(eval(g)); return ''

def genexec(g):
    print('exec:',g); tree = None
    try: tree = ast.parse(g)
    except BaseException as err: print(err)
    if tree:
        exec(g,globals())
        if '=' in g:
            _a = g.split('=')[0].strip(); _v = globals()[_a]
            print(f'\033[36m    new variable: {_a} = {_v}\033[m')
    else:
        DSL(g)
        return -1
    return ''

SYMBOLS_TO_GEN = { '  ' : genfunc, '  ' : gencall, '  ':geneval, '':genexec }

word_mapping_data = []

if '--quick' not in sys.argv: word_mapping_data+=[('write function to add ¿ and ¿',''),('call function ¿¿¿ with ¿ ¿', ''),('add ¿ and ¿', ''),('multiply ¿ and ¿', '')]

if '--english' in sys.argv: word_mapping_data += [ ('write python code to print hello world', '  '), ("one", "two"), ("hello", "world"), ("good", "bad"), ("happy", "sad") ]
if '--math' in sys.argv:

    word_mapping_data = [ ('1+1', ''), ('a=1+1', '') ]
elif '--quick' in sys.argv: word_mapping_data = [LEARN[0]]
else: word_mapping_data += LEARN
```

```
AUTO_INPUT = ['write python function to compute foo', 'args x', 'return x plus x', '']

if '--learn-draw' in sys.argv:
    word_mapping_data = []; AUTO_INPUT = []
    for sym in SYMBOLS_TO_NAMES: word_mapping_data.append((sym, SYMBOLS_TO_NAMES[sym])); AUTO_INPUT.append(sym)
    word_mapping_data += [(_*3,_) for _ in 'ab']
    AUTO_INPUT += 'aaa bbb'.split()

if '--abcd' in sys.argv:
    word_mapping_data += [(_*3,_) for _ in 'abcd']
    AUTO_INPUT += 'aaa bbb ccc ddd'.split()

AUTO_INPUT.reverse()
def read(prompt='', timeout_seconds=1):
    if AUTO_INPUT: return AUTO_INPUT.pop()
    if not timeout_seconds: return input(prompt)
    if prompt:sys.stdout.write(prompt); sys.stdout.flush() # Ensure the prompt is displayed immediately
    rlist, _, _ = select.select([sys.stdin], [], [], timeout_seconds)
    if rlist: return sys.stdin.readline().strip()
    else: return None

def plot(vec, label='', x=1, y=1, scale=10):
    blocks = ' ▁▂▃▄▅▆▇'; o = []; u=[]
    for i,v in enumerate(vec):
        idx = (int(v*scale)); c = '🡅'
        if idx < len(blocks): c=blocks[idx]
        o.append(c)
        if idx >= 2: u.append(TextProcessor.CHARACTERS[i])
        else: u.append('.')
    printat(label + ''.join(o), y, x); printat((' '*len(label)) + ''.join(u), y+1, x)

def dream(params, classifier_nn, art):
    for i in range(len(params)):        # Mutate Burning Ship parameters
        if i < 3: params[i] += random.uniform(-MUTATION_RATE, MUTATION_RATE)
        else: params[i] += random.randint( -max(1, int(MUTATION_RATE * params[i])), max(1, int(MUTATION_RATE * params[i]))); params[i] = max(20, min(200, params[i]))
    printat('FRACTAL: %s MUTATE RATE: %s' %([round(v,6) for v in params], MUTATION_RATE), 3, 1)
    magnitude_spectrum = run_gol_and_3d_fft_torch(params, '', drawings=art, debug_output=True)
    bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, len(TextProcessor.CHARACTERS))
    plot(bands.cpu().numpy(), label='𝔅:', scale=100); words = TextProcessor.reconstruct_word_from_classifier_output( bands, min_value_threshold=0.0025 ); printat(words, 18, 70)
    predicted_bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER); classifier_nn.eval()
    with torch.no_grad(): final_prediction_values = classifier_nn(predicted_bands.unsqueeze(0)).squeeze(0)
    reply = TextProcessor.reconstruct_word_from_classifier_output( final_prediction_values, min_value_threshold=MIN_VAL_THRESH )
    plot(final_prediction_values.cpu().numpy(), label='𝔑:', x=35); printat(reply, 19, 70)

SUBSELF=None
def main():
    print('word_mapping_data:', word_mapping_data)
    best_bs_params, classifier_nn = solve(word_mapping_data, subnetwork=SubNet)
    print(word_mapping_data); print(OUT_TOKENS); print(best_bs_params, classifier_nn, SubNet); print(LEARN)
    classifier_nn.eval()
    pyscript = []; var_stack = []; print('\033[36m%s\033[m' % '='*80); print('\033[36m    Input your query or command and press [Enter] key\033[m')
    read_state = 0; art = []
    while 1:
        input_word = read()
        if input_word == '[draw]':
            art = [ start_drawing() ]
            continue
        if input_word is None:
            if SUBSELF:
                print('reading from SUBSELF....'); data = SUBSELF.stdout.readline().strip(); print('GOT:', data)
                try: input_word = data.decode('utf-8')
                except: continue
                print('FROM SUBSELF:', input_word)
            #if word_mapping_data and random.random() < -0.3: best_bs_params, classifier_nn=solve(word_mapping_data, iterations=10, verbose=False) ## TODO
            if random.random() < 0.1 and REFERENCES: Papers.append( import_pdf(REFERENCES.pop()) )
            else: dream( list(best_bs_params), classifier_nn, art)
            continue
        elif read_state:
            dsl_script.append(input_word); print(dsl_script)
            if input_word.strip() == '': read_state = 0; DSL('\n'.join(dsl_script))
            continue

        magnitude_spectrum = run_gol_and_3d_fft_torch(best_bs_params, input_word, drawings=art, debug_output=False)
        predicted_bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, NUM_FFT_BANDS_FOR_CLASSIFIER)
        with torch.no_grad(): final_prediction_values = classifier_nn(predicted_bands.unsqueeze(0)).squeeze(0)
        reply = TextProcessor.reconstruct_word_from_classifier_output( final_prediction_values, min_value_threshold=MIN_VAL_THRESH )
        print(f"\033[31m{reply}\033[m")
        if reply in SYMBOLS_TO_GEN:
            gen = SYMBOLS_TO_GEN[reply](input_word)
            if type(gen) is int: read_state = gen; dsl_script=[input_word]; continue
            if gen: print(f"  Generated: \033[33m{gen}\033[m")
            if gen.startswith('lambda '):
                print("\033[36m    save lambda function (type name) or press [Enter] to ignore\033[m"); user = input().strip()
                if user:
                    if len(user) != 3: print("WARN: function name is not length three")
                    func = eval(gen,globals()); print('\033[36m    saved global function: %s %s\033[m' %(user, func))
                    globals()[user] = func
            elif gen.endswith(')'):
                print('\033[36m    call function? y/n (%s)\033[m' % gen)
                if input().strip():
                    try: _ = eval(gen,globals()); print(_)
                    except BaseException as err: print(err)
        elif reply in SYMBOLS_TO_TEMPLATE:
            print(f"  Template: \033[33m{SYMBOLS_TO_TEMPLATE[reply]}\033[m"); args = []
            for part in input_word.split():
                if len(part)==1: args.append(part)
            print('\033[36m    run code? y/n (with template args: %s)\033[m' % args)
            if input()=='y':
                print("\033[34m", end='')
                try: exec(SYMBOLS_TO_TEMPLATE[reply] % tuple(args), globals())
                except BaseException as err: print(err)
                print("\033[m", end='')
        elif reply in SYMBOLS_TO_CODE:
            pyscript.append(SYMBOLS_TO_CODE[reply]); print(f"  SymPython: \033[33m{SYMBOLS_TO_CODE[reply]}\033[m"); print('\033[36m    run code? y/n\033[m')
            if input()=='y':
                print("\033[34m", end='')
                try: exec(SYMBOLS_TO_CODE[reply], globals())
                except BaseException as err: print(err)
                print("\033[m", end='')
        else:
            print(reply, "?")
            if len(input_word.split()) == 2: word_mapping_data.append( tuple(input_word.split()) )
            info = search(input_word)
            if info:
                print("\033[37m", end='')
                for ln in info: print('\t'+ln.strip())
                print("\033[m", end='')

def subself():
    global SUBSELF
    tmp = '/tmp/subself.py'; open(tmp,'w').write(open(__file__).read()); cmd = ['python3',tmp]
    for n in 'GOL_GRID_SIZE_X GOL_GRID_SIZE_Y GOL_SIM_STEPS FRACTAL_NET_HIDDEN'.split(): cmd.append('--%s=%s' % (n, globals()[n]*2))
    print(cmd); SUBSELF = subprocess.Popen(['python3', tmp], stdout=subprocess.PIPE); print(SUBSELF); atexit.register(lambda:SUBSELF.kill())

def project_manifold(data_for_umap, n_components=2):
    if type(data_for_umap) is torch.Tensor: data_for_umap = data_for_umap.detach().cpu().numpy()
    if type(data_for_umap) is list: data_for_umap = numpy.array(data_for_umap)
    elif not isinstance(data_for_umap, numpy.ndarray): raise RuntimeError("Error: data_for_umap must be a NumPy array, got type: %s" % type(data_for_umap))
    if n_components not in [2, 3]: raise RuntimeError("Error: n_components must be 2 or 3.")
    if 'umap-debug' in sys.argv: print(f"Applying UMAP with {n_components} components...")
    reducer = umap.UMAP(n_components=n_components, n_neighbors=data_for_umap.shape[0]-1)
    try: embedding = reducer.fit_transform(data_for_umap)
    except ValueError as err: print(err); return None  ## this can happen with too fiew samples?
    except TypeError as err: print(err); return None
    return embedding

def visualize_manifold(embedding, labels=None, n_components=2, title="UMAP Projection of Data"):
    fig = plt.figure(figsize=(10, 8))
    if not labels: labels = string.ascii_letters[ : len(embedding)]
    if n_components == 2:
        ax = fig.add_subplot(111); ax.scatter(embedding[:, 0], embedding[:, 1], s=50, alpha=0.7)
        for i, label in enumerate(labels): ax.annotate(label, (embedding[i, 0], embedding[i, 1]), textcoords="offset points", xytext=(5,5), ha='center')
        ax.set_xlabel('UMAP Dimension 1'); ax.set_ylabel('UMAP Dimension 2')
    elif n_components == 3:
        ax = fig.add_subplot(111, projection='3d'); ax.scatter(embedding[:, 0], embedding[:, 1], embedding[:, 2], s=50, alpha=0.7)
        for i, label in enumerate(labels): ax.text(embedding[i, 0], embedding[i, 1], embedding[i, 2], label, fontsize=8)
        ax.set_xlabel('UMAP Dimension 1'); ax.set_ylabel('UMAP Dimension 2'); ax.set_zlabel('UMAP Dimension 3')
    ax.set_title(title); plt.grid(True); plt.tight_layout(); plt.show()

if __name__ == '__main__':
    if '--subself' in sys.argv: subself() ## this is just a simple example of subprocess self interaction
    main()
```