Self Contained Multi-AI Architecture Definition via DSL

Brent Hartshorn July 13th

Abstract

This paper presents a significant evolution in emergent computational systems, extending the Domain-Specific Language (DSL) driven self-modification paradigm to encompass foundational components beyond neural network architectures and cellular automata rules. Building on prior work where the AI could define its Game of Life (GOL) stepping function and classifier network via a tokenized DSL, we now demonstrate the system's capacity to articulate and dynamically replace its fractal generation algorithms (e.g., Burning Ship and Mandelbrot). By expressing these complex mathematical functions within the same learnable DSL, the system gains a deeper level of meta-programming, enabling the AI to not only define its processing logic but also to programmatically control the very initial conditions and "physics" that seed its emergent dynamics. This advancement pushes towards truly autonomous and adaptive AI capable of reconfiguring its fundamental operational environment.

Introduction

Our previous research established a novel framework for emergent computation [1], [4], demonstrating how a hybrid system, combining fractal-initialized Game of Life (GOL) dynamics with spatiotemporal spectral analysis and a neural network classifier, could achieve symbolic processing and even self-modify core components. Specifically, we showed the AI's ability to generate its GOL stepping function and define its own neural network architecture using a specialized Domain-Specific Language (DSL). This represented a crucial step towards AI systems capable of programming themselves.

This work introduces a further, profound extension to this self-modification capability. We have now integrated the fractal generation algorithms —the very mechanisms responsible for creating the initial complex patterns that seed the GOL grid—into the same DSL. Previously, these fractal functions (like Burning Ship and Mandelbrot) were hard-coded in standard Python. By translating them into the DSL, we enable the AI to programmatically define, modify, and potentially evolve these foundational pattern generators. This refactoring means the system can now influence its own initial conditions and the "substrate" upon which emergent behaviors unfold, opening new avenues for adaptive and creative computational exploration.

Gravity and Transfer of Understanding: PART I

Knowledge and understanding must be compressed when transferred from one entity to another, and then decompressed. Our current state of technology has compressed knowledge into huge LLM networks. Is the ground really accelerating up at us? Because its inertial frame of reference is not at zero, why? Instead of the ground accelerating outward, we can think of gravity as all bodies falling inward and compacting spacetime because of information compression/decompression that happens with each particle collision, it is actually the process that is accelerating, gravity is the increasing of information compression. An LLM network is a highly compressed understanding of information, taking billions of years to evolve. As chips get smaller, this information is compressed to the quantum scale. [2]

Acceleration is new information to the system, which then takes time to compress over many particle interactions. A spaceship accelerating forward using rockets, has an equal amount of matter and energy propelled backward, so in total nothing is actually accelerating, its actually one system that is expanding in two distinct directions. Before the system was uniform, without its own unique two distinct directions, so then it points towards the global inward falling direction of compacting information. (time points to the direction of increasing compression) Moving away from the direction of compression, creates what appears to be locally a force to the observer, the observer acceleration is forcing new information into the global system, slightly changing the center point of the inward falling direction of compacting information. The other forces of nature (in The Standard Model), act directly, because at their small scale they are already fully compressed information states, therefore these forces appear much stronger than gravity. [3]

Sub-Networks inside Fractals: PART II

The introduction of `FractalSubNetwork` represents a critical step towards a fully differentiable and end-to-end trainable system. `FractalSubNetwork` is itself a small neural network, defined using the `@dslai` decorator (see notes below), much like `ClassifierNN`. Its purpose is to learn to subtly perturb or "scramble" the deterministic fractal generation process based on its own internal learning.

```
@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
```

This sub-network, `FractalSubNetwork`, takes a set of inputs related to the fractal generation (size, center_x, center_y, zoom, max_iter, sx, sy, and a numerical representation of the `words` input string). It processes these inputs through its own learnable layers and produces an output vector stored in `self._res`. The `forward` method handles dynamic input sizing by padding or truncating the input tensor to match the network's expected input features, ensuring compatibility within the pipeline. The `getitem` method (`self._res[v]`) allows specific elements of this output vector to be accessed. This design enables the AI to learn how to generate subtle, context-dependent modifications to the fractal's parameters.

The Neural Mandelbrot Fractal: PART III

`FractalSubNetwork` directly influences the generation of the `neural_mandelbrot_fractal` through the `𝓌[row]` term in `cx`.

The forward pass is computed by: `𝓌.forward(size,center_x,center_y,zoom,max_iter,sx,sy,words)`

```
if '--neural-mandel' in sys.argv:
    SubNet = FractalSubNetwork(8, FRACTAL_NET_HIDDEN, GOL_GRID_SIZE); print(SubNet)
    DSL('''
write python function to compute neural_mandelbrot_fractal
args size; args center_x; args center_y; args zoom; args max_iter; args words
grid equals numpy zeros((size,size)); sx equals three divide zoom divide size; sy equals three divide zoom divide size
𝓌.forward(size,center_x,center_y,zoom,max_iter,sx,sy,words)
iterate row in range size iterate col in range size
    ax equals center_x plus ( col minus size divide two ) times sx
    cx equals ax plus 𝓌[row]
    cy equals center_y plus ( row minus size 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
    ''',
    𝓌=SubNet
    )
    print(neural_mandelbrot_fractal)
```

Above, the line cx equals ax plus 𝓌[row] is crucial. Here, 𝓌 is a placeholder for the SubNet instance (passed as a keyword argument to DSL). The 𝓌.forward(...) call within the DSL-defined fractal function passes relevant parameters to the FractalSubNetwork. The output of this sub-network, 𝓌[row] (which accesses self._res[row] via the getitem method), directly modifies the cx component of the complex number in the Mandelbrot iteration. This means that the fractal's shape and characteristics are no longer purely deterministic based on fixed parameters, but are dynamically influenced by the learned output of the FractalSubNetwork, which itself is a neural network. This allows the system to learn how to generate fractals that are optimally "tuned" for the downstream task of word classification.

Game of Life Back-Propagation: PART IV

To enable the `FractalSubNetwork` to learn effectively, the entire pipeline from fractal generation through GOL simulation to FFT and classification must be differentiable. A key enabler for this is the re-implementation of the `game_of_life` function using PyTorch operations:

```
def game_of_life(grid_tensor: torch.Tensor) -> torch.Tensor:
    """Game of Life simulation using PyTorch operations for differentiability. Uses torch.roll for circular padding to sum neighbors.
    Args: grid_tensor: A PyTorch tensor representing the current GOL grid (0s and 1s). Expected shape: (H, W). Dtype: float or int.
    Returns: PyTorch tensor representing the next state of the GOL grid.
    """
    grid = grid_tensor.float() # Ensure input is float
    neighbor_counts = torch.zeros_like(grid)    # Initialize a tensor to store neighbor counts
    # Define the 8 relative positions of neighbors # (row_offset, col_offset)
    offsets = [
        (-1, -1), (-1, 0), (-1, 1), # Top row neighbors
        (0, -1),           (0, 1),  # Same row neighbors (excluding self)
        (1, -1),  (1, 0),  (1, 1)   # Bottom row neighbors
    ]
    # Sum neighbors using torch.roll for circular (toroidal) boundary conditions
    for dr, dc in offsets: neighbor_counts += torch.roll(grid, shifts=(dr, dc), dims=(0, 1))
    # Apply Conway's Game of Life rules using boolean logic on tensors
    # 1. A living cell with 2 or 3 living neighbors survives.
    # 2. A dead cell with exactly 3 living neighbors becomes a living cell (birth).
    new_grid = ((grid == 1) & ((neighbor_counts == 2) | (neighbor_counts == 3))) | ((grid == 0) & (neighbor_counts == 3))
    return new_grid.float() # Ensure output is float for consistency in the pipeline
```

With a fully differentiable pipeline, the criterion(predictions, targets_batch) loss would naturally backpropagate through the ClassifierNN, calculate_3d_fft_energy_bands_torch, run_gol_and_3d_fft_torch (including the new PyTorch game_of_life), and finally through the neural_mandelbrot_fractal (where 𝓌 is used) all the way back to the FractalSubNetwork's parameters. This allows FractalSubNetwork to learn how to generate fractal "substrates" that ultimately lead to better word classification, even though FractalSubNetwork doesn't have a direct, isolated target of its own. This approach effectively treats the entire system as one large neural network, allowing for end-to-end training.

Game of Life DSL

```
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 to Meta

```
with Ċoncept("dynamic") as write:
        write.python.function.to.compute.game_of_life
        game_of_life.ARGS.grid
        game_of_life.g.equals.grid.float.LPAREN.RPAREN
        game_of_life.n.equals.torch.zeros_like.LPAREN.g.RPAREN
        game_of_life.offsets.equals.list[[(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]]
        with game_of_life.iterate.v.IN.offsets as loop_1:
                loop_1.x.equals.v[0]
                loop_1.y.equals.v[1]
                loop_1.n.plus_equals.torch.roll(g,shifts=(x,y),dims=(0,1))
        game_of_life.r.equals('(g==1)&((n==2)|(n==3)))|((g==0)&(n==3)')
        game_of_life.RETURN.r.float.LPAREN.RPAREN
```

Meta to Python

```
def game_of_life(grid):
        g = grid.float()
        n = torch.zeros_like(g)
        offsets = list([(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)])
        for v in offsets:
                x = v[0]
                y = v[1]
                n += torch.roll(g,shifts=(x,y),dims=(0,1))
        r = ((g==1)&((n==2)|(n==3)))|((g==0)&(n==3))
        return r.float()
```

Compressed Tokenized String (training data target)

```
ѿѾѽѼѻѺѹѸѷѶ'(g≣1)&⸨n≣2)|(n≣3⸩)|⸨g≣0)&(n≣3)'ѵѴ
```

Above, multi-level tokenization: extended Latin characters are used to tokenize entire blocks of code (delimited by “;”), while the inner string is tokenized per-letter. The inner string is quoted, directing the training process to reproduce these tokens per-letter, this allows the network to fully mutate this section.

Source Code

```
import random,math,sys,torch,time,string,inspect,select,numpy,subprocess,atexit; device = torch.device("cuda" if torch.cuda.is_available() else "cpu"); print(f"Using device: {device}")
# --- Constants ---
MIN_VAL_THRESH = 0.25; GOL_GRID_SIZE = 12; GOL_SIM_STEPS = 8; NUM_FFT_BANDS_FOR_CLASSIFIER = 128; OPTIMIZATION_ITERATIONS = 5000; CLASSIFIER_RATE = 1
MAX_OUTPUT_WORD_LENGTH = 128; MIN_LOSS = 0.00005; MUTATION_RATE = 0.025; FRACTAL_NET_HIDDEN = 16
if '--quick' in sys.argv: OPTIMIZATION_ITERATIONS = 1000; 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])


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 Ċoncept:
    IDENT = 0
    def __init__(self,*args,**kwargs):
        self.ñame=args[-1]; self.ļinks = []; self.řelations = []; self.þrops = []; self.children=[]
        if len(args)>1: self.řelations = __builtins__.list(args[:-1])
    def __call__(self,*args,**kwargs):
        if not args and not kwargs: return Ċoncept.dsl_to_python(self.as_python())
        for a in args: self.ļinks.append(a)
        if kwargs: self.ļinks.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.řelations:self.řelations.append(o)
        return self
    def __getattr__(self, n): o = Ċoncept(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].ļinks]); 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[0].ñame=='ARGS': c += '(%s):\n' % ','.join([_.ñame for _ in f.children[0].children ])
            else: c += '():\n'
            o.append(c)
            for v in f.children[1:]:
                for a in v.as_python(): o.append('\t'+a)
            return o
        elif self.ñame == 'iterate':
            Ċoncept.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.name)
                        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].name
                    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].name
                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.name,a,b)]
        elif self.name == 'copy' and self.relations and self.relations[0].name=='numpy':
            if self.children: a = self.children[0].as_python(); assert len(a)==1; a = a[-1]
            else: a = self.links[-1].name + '\n'
            if a.endswith('\n'): return ['copy(%s)\n' % a[:-1]]
            else: return ['copy(%s)' % a]
        elif self.name == 'zeros' and self.relations and self.relations[0].name=='numpy':
            size = []
            for l in self.links[-1]:
                if type(l) is Concept: size.append(l.name)
                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.name, a.name, b.name))
            else:
                rel = self.relations[-1]
                if type(rel) is Concept: rel = rel.name;      ## this check must come first
                elif hasattr(rel,'symbol'): rel = rel.symbol  ## note: hasattr is broken with Concept, its always true TODO
                if self.name == 'less_than': o.append('< %s' % rel)
                elif self.name == 'greater_than': o.append('> %s' % rel)
                elif self.name == 'list': o.append('list(%s)' % rel)
                else: o.append('%s[%s]' % (self.name, 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 Concept: args.append('%s=(%s,%s)' % (_,_a.name,_b.name))
                            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.name)
            if self.name=='equals': o.append('=.(%s)' % (','.join(args)))
            else: o.append('%s(%s)' % (self.name, ','.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.name in mapping: o.append(mapping[self.name])
            else: o.append(self.name)
        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].symbol = n
    txt = txt.strip(); prefixes = {0:txt.splitlines()[0].split()[-1]}; o = ['with Concept("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 Concept:
            if n not in before: rem.append(n)
            elif n in before and type(before[n]) is not Concept: 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; args center_x; args center_y; args zoom; args max_iter; grid equals numpy zeros((size,size)); sx equals three divide zoom divide size; sy equals three divide zoom divide size
iterate row in range size iterate col in range size
    cx equals center_x plus ( col minus size divide two ) times sx; cy equals center_y plus ( row minus size 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; args center_x; args center_y; args zoom; args max_iter; args words; grid equals numpy zeros((size,size)); sx equals three divide zoom divide size; sy equals three divide zoom divide size
.forward(size,center_x,center_y,zoom,max_iter,sx,sy,words)
iterate row in range size iterate col in range size

    ax equals center_x plus ( col minus size divide two ) times sx; cx equals ax plus [row]; cy equals center_y plus ( row minus size 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); print(SubNet)

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


# --- Text Processing Utility Class ---
class TextProcessor:
    MIN_DSL_SYMS = '≡()&|()=0123456789ng'
    CHARACTERS = [chr(i) 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 += [' ', '', '', '', ''] + 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': ['###',' #',' ##',' #','###'],
    }
    @staticmethod
    def ascii_to_gol_pattern(char):
        char = char.lower() # Ensure lowercase
        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):
            if r >= TextProcessor.PATTERN_HEIGHT: continue
            for c, symbol in enumerate(row_str):
                if c >= TextProcessor.PATTERN_WIDTH: continue
                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 = {'((':'(', '))':')', '==':'≡'}
        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=False):
        """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':''}
        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:
    test = '(g==1)&((n==2)|(n==3)))|((g==0)&(n==3)'
    vec = TextProcessor.word_to_target_values(test, len(test)); print(vec); word = TextProcessor.reconstruct_word_from_classifier_output(vec); print(word)
    assert test == word

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, 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,GOL_GRID_SIZE), dtype=float); tag = 'none'
    elif '--mandel' in sys.argv: fractal_raw_grid_np = mandelbrot_fractal(GOL_GRID_SIZE, 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, center_x, center_y, zoom, fractal_max_iter, input_word_string); tag='NeuralMandelbrot'
    else: fractal_raw_grid_np = burning_ship_fractal(GOL_GRID_SIZE, 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, GOL_GRID_SIZE), 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
        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
    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 ---
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, 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
    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_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))

SYMBOLS_TO_GEN = { ''  : genfunc, ''  : gencall }; 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") ]

def read(prompt='', timeout_seconds=1):
    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):
    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, '', debug_output=True); bands = calculate_3d_fft_energy_bands_torch(magnitude_spectrum, len(TextProcessor.CHARACTERS))
    plot(bands.cpu().numpy(), label=':', 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='9l:', x=35); printat(reply, 19, 70)

SUBSELF=None
def main():
    global word_mapping_data
    if '--quick' in sys.argv: word_mapping_data = [LEARN[0]]
    else: word_mapping_data += LEARN
    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')
    while 1:
        input_word = read()
        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
        else: dream( list(best_bs_params), classifier_nn)
        continue
    if input_word.count('=')==1:
        _a,_b = input_word.split('='); _a = _a.strip(); _v = eval(_b); print(f'\033[36m    new variable: {_a} = {_v}\033[m'); globals()[_a]=_v; var_stack.append(_a)
        continue
    elif input_word.endswith(')'):
        try: eval(input_word)
        except BaseException as err: print(err)
        continue
    magnitude_spectrum = run_gol_and_3d_fft_torch(best_bs_params, input_word, 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); 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()) )

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 subself():
    global SUBSELF
    tmp = '/tmp/subself.py'; open(tmp,'w').write(open(__file__).read()); cmd = ['python3',tmp]
    for n in 'GOL_GRID_SIZE 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())

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

Notes

@dslai Decorator

The most significant architectural shift lies in the introduction of the @dslai decorator and a custom Domain-Specific Language (DSL). Previously, the ClassifierNN was explicitly hardcoded as a standard torch.nn.Module with predefined layers and a fixed forward method, see below:

Python (old version)

```
class ClassifierNN(torch.nn.Module):
    def __init__(self, num_input_bands, num_hidden_neurons, num_output_neurons):
        super(ClassifierNN, self).__init__()
        self.fc1 = torch.nn.Linear(num_input_bands, num_hidden_neurons)
        self.sigmoid = torch.nn.Sigmoid()
        self.fc2 = torch.nn.Linear(num_hidden_neurons, num_output_neurons)
        self.output_activation = torch.nn.Sigmoid()
    def forward(self, x):
        x = self.fc1(x); x = self.sigmoid(x); x = self.fc2(x); x = self.output_activation(x)
        return x
```

Now, with @dslai, the network's structure and behavior are defined declaratively using a simple, tokenized format:

Python (new DSL version)

```
@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 ClassifierNN: pass # This class now serves as a template for DSL instantiation
```

This represents a crucial shift:

- **Abstraction of Network Definition:** The `@dslai` decorator, leveraging the underlying Ċoncept class and DSL function, allows for defining network layers (e.g., input linear, activation sigmoid) and their connections within the forward pass using concise, tokenized strings. This abstracts away the direct torch.nn.Module instantiation boilerplate.
- **AI-Driven Architecture Generation:** By expressing the network's design in this tokenized DSL format, the system gains the potential for self-modification. Instead of a human programmer manually changing Python code to alter the network, the AI itself can learn to manipulate or generate these token sequences to evolve new architectures or modify existing ones as part of its optimization process. This is a foundational step towards "AI writing new AI" in a dynamic, data-driven manner.

- **Increased Flexibility:** Decoupling the network definition from direct Python code provides greater flexibility for integrating architectural changes with the emergent learning process, allowing the system to explore a wider design space.

Rewrite of Fractals into DSL

In the previous version, the Burning Ship and Mandelbrot Fractals were implemented in standard Python. In this version we have extended the DSL and rewritten both those functions, below is the Burning Ship.

Python DSL: Burning Ship

```
write python function to compute burning_ship_fractal
args size; args center_x; args center_y; args zoom; args max_iter
grid equals numpy zeros((size,size)); sx equals three divide zoom divide size; sy equals three divide zoom divide size
iterate row in range size iterate col in range size
    cx equals center_x plus ( col minus size divide two ) times sx; cy equals center_y plus ( row minus size 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; 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
    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
```

Python DSL to Python

```
def burning_ship_fractal(size,center_x,center_y,zoom,max_iter):
          grid = numpy.zeros((size,size),dtype=float)
          sx = 3 / zoom / size
          sy = 3 / zoom / size
          for row in range(size):
                    for col in range(size):
                              cx = center_x + (col - size / 2) * sx
                              cy = center_y + (row - size / 2) * sy
                              x = 0
                              y = 0
                              for i in range(max_iter):
                                        xn = x * x - y * y + cx
                                        xy = x * y
                                        yn = 2 * abs(xy) + cy
                                        x = abs(xn)
                                        y = abs(yn)
                                        if x * x + y * y > 4:
                                                  break
                              grid[(row,col)] = i
          mx = numpy.max(grid)
          if mx > 0:
                    grid = grid / mx
          return grid
```

Conclusion

This latest iteration of our emergent computational system marks a critical milestone in self-modifying AI. By extending the DSL to encompass fundamental fractal generation algorithms, we have moved beyond enabling the AI to define its processing logic (GOL rules, neural network architectures) to allowing it to programmatically control the very initial conditions and "physics" of its emergent dynamics.

This capability has several profound implications:

- **Deeper Self-Configuration:** The system can now dynamically alter the nature of the complex patterns that seed its GOL simulations, offering a more profound level of self-configuration.
- **Exploration of Generative Principles:** The AI can potentially learn to generate and optimize novel fractal forms or other generative algorithms through its DSL, leading to new insights into complex systems and pattern formation.
- **Enhanced Adaptability:** An AI capable of modifying its own foundational generative components could exhibit superior adaptability, tailoring its internal "environment" to better suit specific tasks or evolving conditions.

While the current implementation demonstrates the feasibility of this DSL-driven fractal definition, future work will focus on enabling the AI to learn *which* fractal parameters or even *which types* of fractal algorithms are optimal for specific emergent behaviors or computational tasks, further solidifying its path towards truly autonomous and creatively adaptive intelligence.

References:

[1] Emergent Self-Modification and Meta-Programming in Dynamic Systems https://ai.vixra.org/abs/2507.0036

[2] Nature vs Nurture https://ai.vixra.org/abs/2505.0141

[3] The Big Bangless https://ai.vixra.org/abs/2505.0041

[4] Emergent Symbolic Computation Fractal Dynamics Spatiotemporal Spectral Analysis: ...Extension https://ai.vixra.org/abs/2507.0009