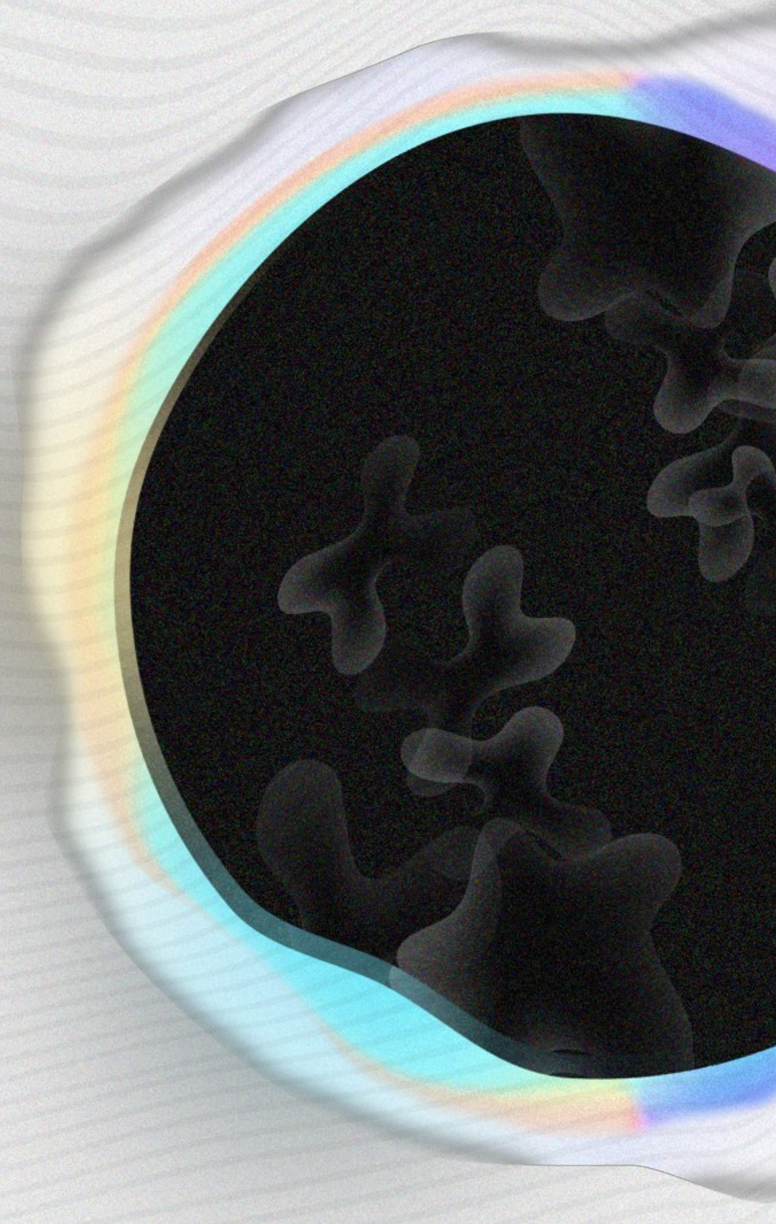
$ https://github.com/singnet/opencog-workshops/intro-singnet

SingularityNET

Introduction to OpenCog

Anatoly Belikov
abelikov@singularitynet.io

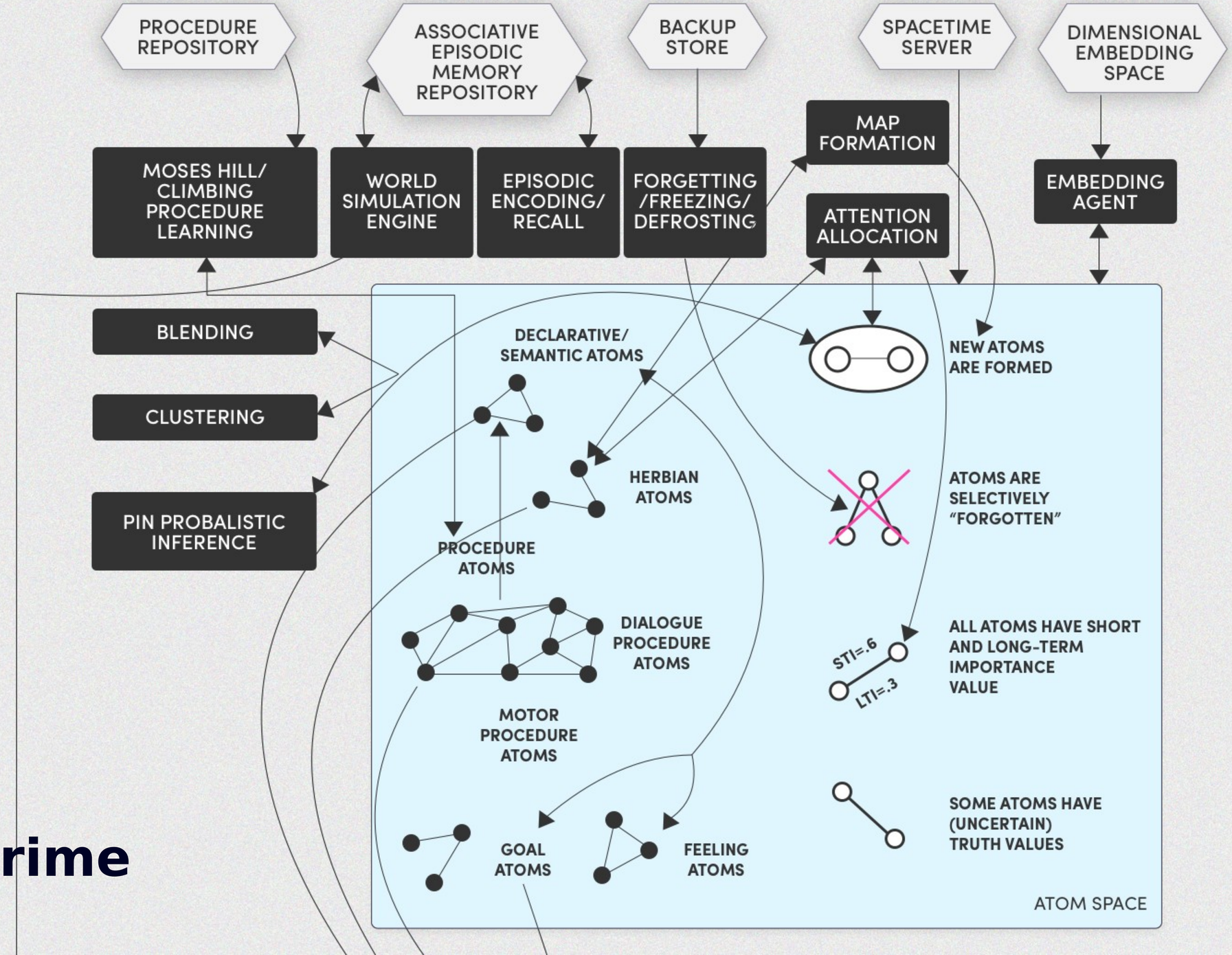


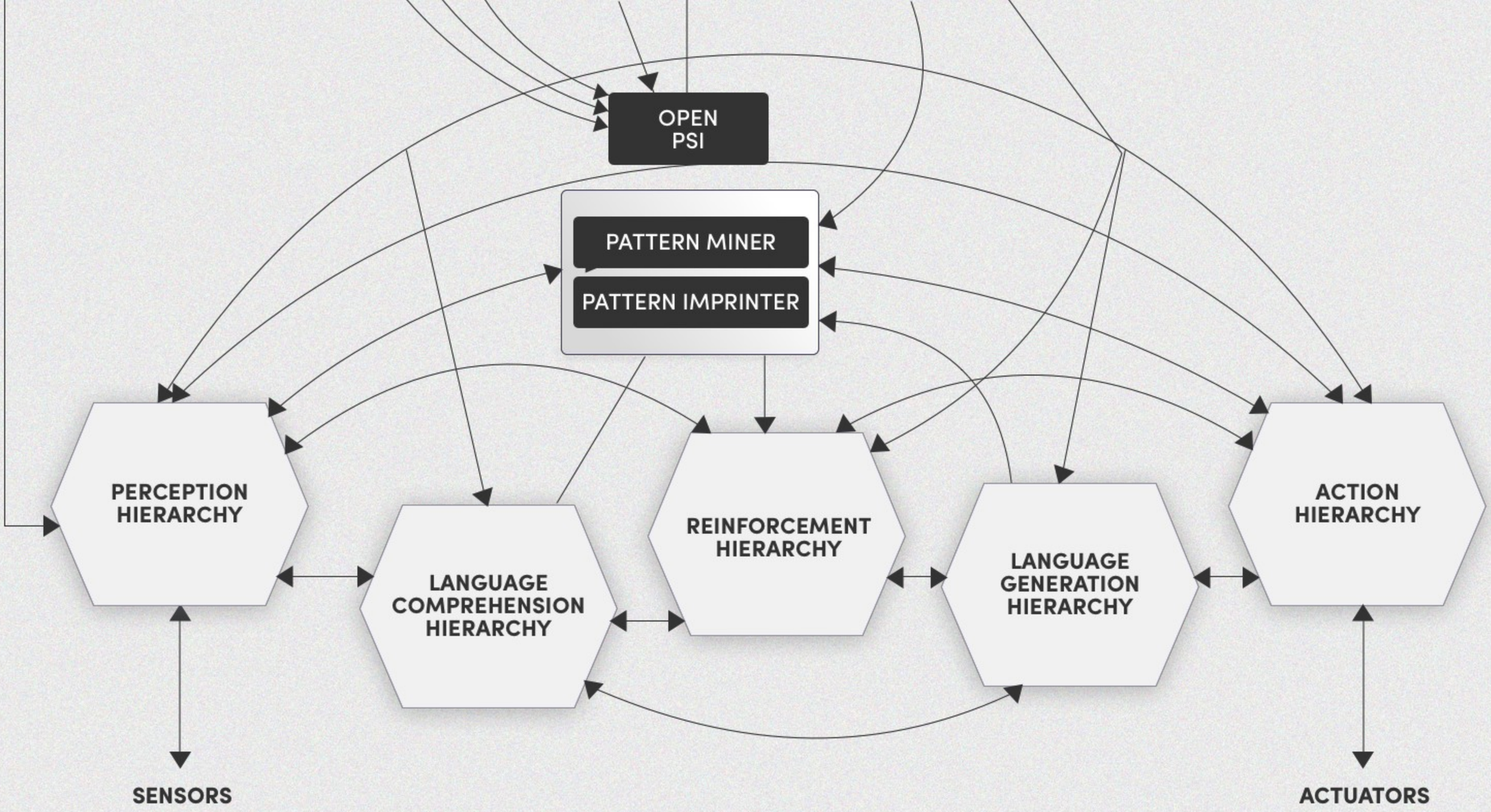
Content:

Cognitive architectures
Representing knowledge with atomspace
Inference example
Integration with pytorch

Anatoly Belikov
abelikov@singularitynet.io

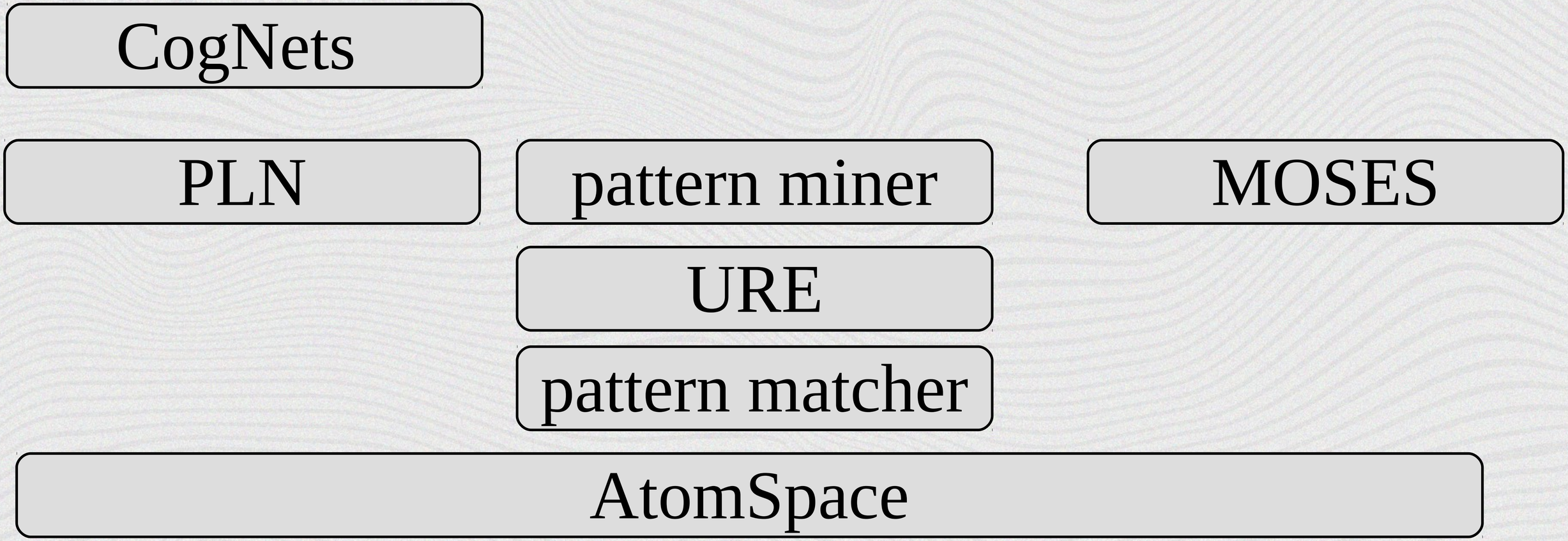
CogPrime


PROCEDURE REPOSITORY
ASSOCIATIVE EPISODIC MEMORY REPOSITORY
BACKUP STORE
SPACETIME SERVER
DIMENSIONAL EMBEDDING SPACE
MAP FORMATION
MOSES HILL/ CLIMBING PROCEDURE LEARNING
WORLD SIMULATION ENGINE
EPISODIC ENCODING/ RECALL
FORGETTING /FREEZING/ DEFROSTING
ATTENTION ALLOCATION
EMBEDDING AGENT
BLENDING
CLUSTERING
PIN PROBALISTIC INFERENCE
DECLARATIVE/ SEMANTIC ATOMS
NEW ATOMS ARE FORMED
HERBIAN ATOMS
ATOMS ARE SELECTIVELY "FORGOTTEN"
PROCEDURE ATOMS
DIALOGUE PROCEDURE ATOMS
STI=.6
LTI=.3
ALL ATOMS HAVE SHORT AND LONG-TERM IMPORTANCE VALUE
MOTOR PROCEDURE ATOMS
SOME ATOMS HAVE (UNCERTAIN) TRUTH VALUES
GOAL ATOMS
FEELING ATOMS
ATOM SPACE


OPEN PSI
PATTERN MINER
PATTERN IMPRINTER
PERCEPTION HIERARCHY
LANGUAGE COMPREHENSION HIERARCHY
REINFORCEMENT HIERARCHY
LANGUAGE GENERATION HIERARCHY
ACTION HIERARCHY
SENSORS
ACTUATORS

CogPrime

OpenCog


CogNets
PLN
pattern miner
MOSES
URE
pattern matcher
AtomSpace

Use cases

- Chatbots
- Bioinformatics
- Unsupervised language learning
- Visual Question Answering VQA
- ...

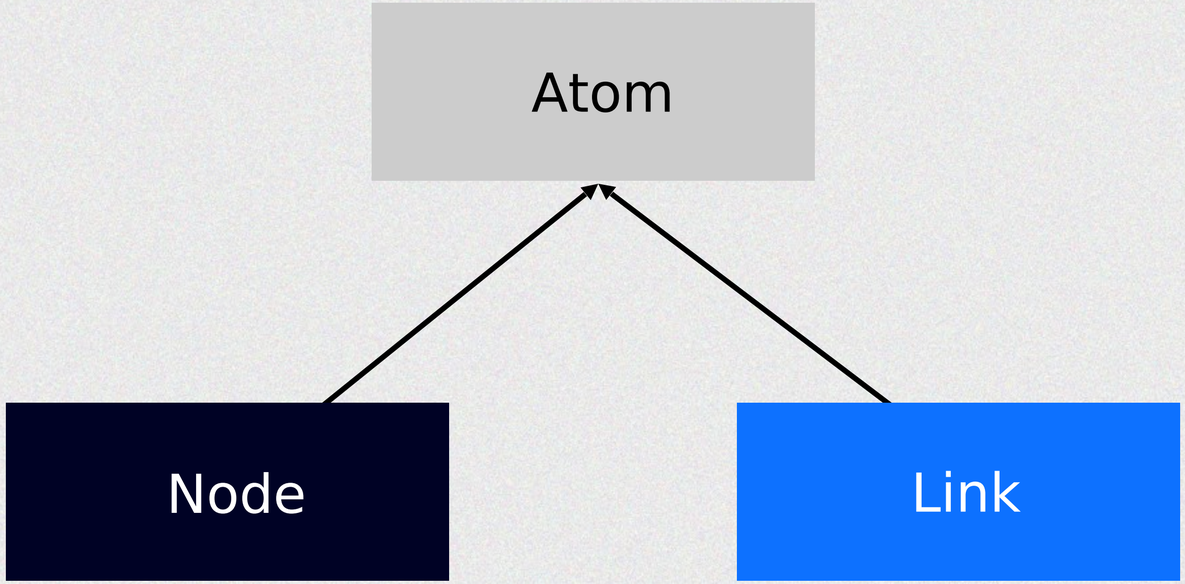
Atomspace - graph database

Hypergraph database designed to be primarily used as a storage of explicit, declarative knowledge

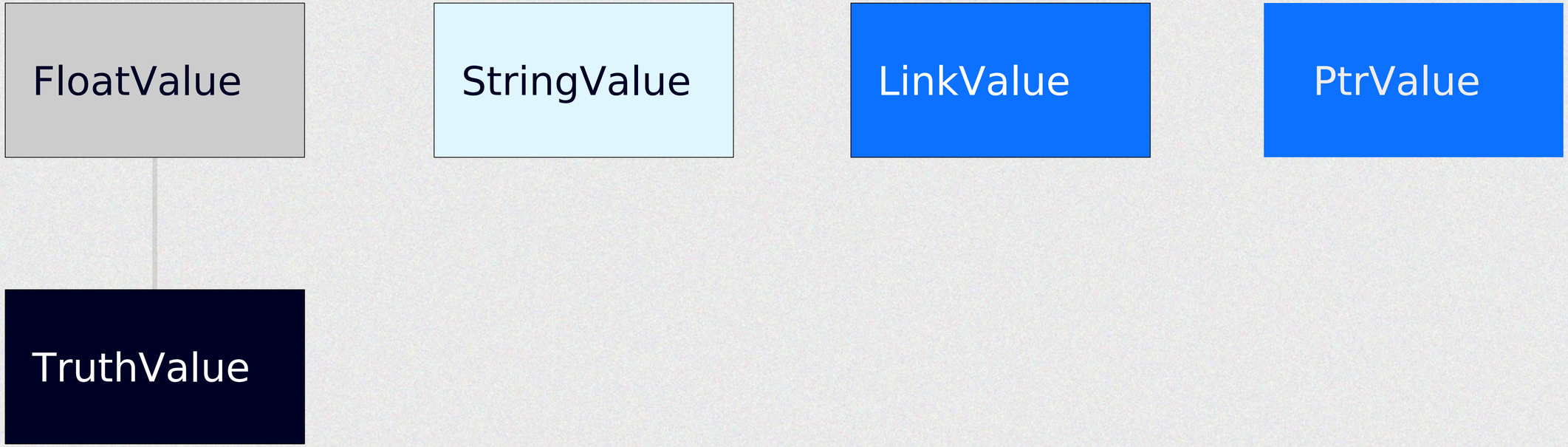
Hypergraph defines programming language - Atomese

Has a number of associated mechanisms:
pattern matching, unified rule engine, moses

Types of data in atomspace: Atoms



Types of data in atomspace: Values



Atomspace - API introduction

Python:

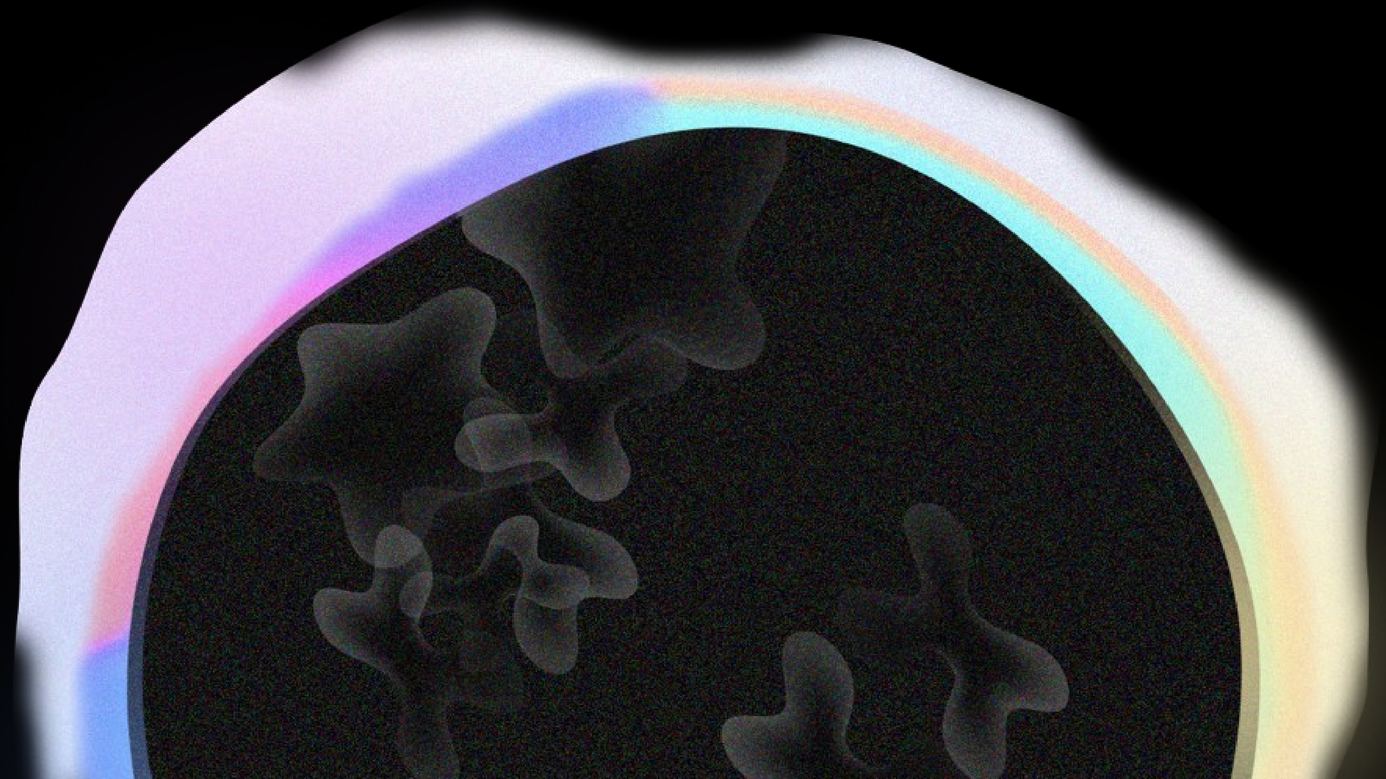
```
>> from opencog.atomspace import AtomSpace, types
>> atomspace = AtomSpace()
```

Scheme:

```
scheme> (use-modules (opencog))
scheme> (define atomspace (cog-new-atomspace))
```

AtomSpace examples

```
$ docker run -p8888:8888  -it singnet-demo
$ localhost:8888
```



jupyter

Qu

Files Running Clusters

Select items to perform actions on them.

Uploa

0 / Name Last Modified

docker		2 hours ag
and-link-example.ipynb	Running	2 hours ag
atomese-examples.ipynb		2 hours ag
socrates-explained.ipynb		2 hours ag
socrates.ipynb		2 hours ag
common.py		2 hours ag

knowledge base

Color and size are attributes.

InheritanceLink is similiar to verb "is" like it is used in natural languages.

```
InheritanceLink(ConceptNode("size"), ConceptNode("attribute"))
InheritanceLink(ConceptNode("color"), ConceptNode("attribute"))
```

Different types of color and size

PredicateNode is used to define relations between different objects.

```
InheritanceLink(PredicateNode("small"), ConceptNode("size"))
InheritanceLink(PredicateNode("big"), ConceptNode("size"))
InheritanceLink(PredicateNode("green"), ConceptNode("color"))
InheritanceLink(PredicateNode("red"), ConceptNode("color"))
InheritanceLink(PredicateNode("black"), ConceptNode("color"))
```

TruthValue

```
> from opencog.atomspace import TruthValue

> tv = TruthValue(0.5, 0.92)

P(Heads) = 0.5, .3

> ConceptNode("Heads").tv = tv
```

animals which have some attributes defined using EvaluationLink

EvaluationLink defines predicate for particular objects.

```
def assign_property(concept, relation, tv):
    EvaluationLink(relation,
                   concept).tv = TruthValue(*tv)

assign_property(ConceptNode("frog"), PredicateNode("small"), (0.8, 0.6))
assign_property(ConceptNode("frog"), PredicateNode("green"), (0.7, 0.91))
assign_property(ConceptNode("tiger"), PredicateNode("big"), (0.9, 0.92))
assign_property(ConceptNode("tiger"), PredicateNode("red"), (0.6, 0.83))
assign_property(ConceptNode("sparrow"), PredicateNode("red"), (0.1, 0.64))
assign_property(ConceptNode("sparrow"), PredicateNode("small"), (0.8, 0.77))
assign_property(ConceptNode("zebra"), PredicateNode("black"), (0.5, 0.9))
assign_property(ConceptNode("zebra"), PredicateNode("small"), (0.38, 0.89))
```

Define that all animals are actually animals

```
InheritanceLink(ConceptNode("frog"), ConceptNode("animal"))
InheritanceLink(ConceptNode("frog"), ConceptNode("animal"))
InheritanceLink(ConceptNode("tiger"), ConceptNode("animal"))
InheritanceLink(ConceptNode("tiger"), ConceptNode("animal"))
InheritanceLink(ConceptNode("sparrow"), ConceptNode("animal")
InheritanceLink(ConceptNode("sparrow"), ConceptNode("animal")
InheritanceLink(ConceptNode("zebra"), ConceptNode("animal"))
InheritanceLink(ConceptNode("zebra"), ConceptNode("animal"))
```

Query example

let's select all animals and attributes, filtering by some threshold

first define function and EvaluationLink

```
threshold = 0.6

def apply_threshold(atom):
    if atom.tv.mean < threshold:
        return TruthValue(0, 1)
    return TruthValue(1, 1)
```

BindLink and pattern matcher

BindLink

<variables> - optional
<pattern to match>
<rewrite term>

Pattern example:

```
ListLink(ConceptNode("red"),
        VariableNode("$X"),
        ConceptNode("blue"))
```

Constrain Variable("X") is animal

```
is_animal = InheritanceLink(VariableNode("X"), ConceptNode("animal"))
```

EvaluationLink will accept link X <- "animal" and return flag if it "matches"

```
predicate = EvaluationLink(VariableNode("$B"),
                           VariableNode("X"))

eval_threshold = EvaluationLink(
                    GroundedPredicateNode("py: apply_threshold"),
                    ListLink(predicate))
```

first define pattern that will be grounded during the search

```
and_link = AndLink(is_animal,
                   predicate,
                   eval_threshold
                  )
```

---

now build the BindLink

BindLink is constructed using pattern to ground and pattern to return.
pattern to return if it contains variables,
should be part of pattern to ground

```
bindlink_filter = BindLink(and_link,
                           predicate)
```

```python
print(execute_atom(atomspace, bindlink_filter))
```

```
(SetLink
  (EvaluationLink (stv 0.900000 0.920000)
    (PredicateNode "big")
    (ConceptNode "tiger")
  )
  (EvaluationLink (stv 0.800000 0.770000)
    (PredicateNode "small")
    (ConceptNode "sparrow")
  )
  (EvaluationLink (stv 0.600000 0.830000)
    (PredicateNode "red")
    (ConceptNode "tiger")
  )
  (EvaluationLink (stv 0.700000 0.910000)
    (PredicateNode "green")
    (ConceptNode "frog")
  )
```

Unified Rule Engine

Inference engine based on graph rewriting rules.

Allows to perform declarative inference.

Rules and data defined in the same language - atomese

Can do backward and forward chaining

Unified Rule Engine

Inference in first-order logic

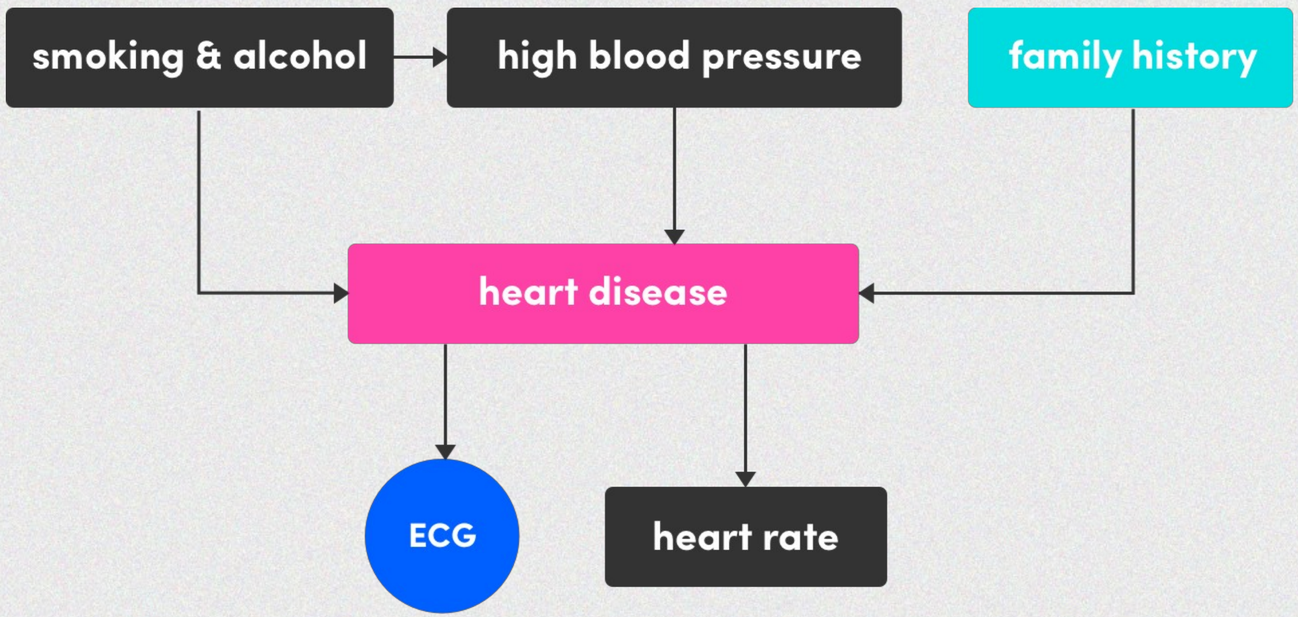
Automatic differentiation

Graphical models

Probabilistic logic inference

Fuzzy logic inference

...



	docker	
	and-link-example.ipynb	
	atomese-examples.ipynb	
	socrates-explained.ipynb	
	socrates.ipynb	Running
	common.py	
	LICENSE	
	notebook.sh	
	opencog.log	
	README.md	

Defining problem

```
> InheritanceLink(ConceptNode("Socrates"),
                  ConceptNode("man")).tv =
                TruthValue(0.97, 0.92)
> InheritanceLink(ConceptNode("man"),
                  ConceptNode("mortal")).tv =
                TruthValue(0.98, 0.94)
InheritanceLink(ConceptNode("Socrates"),
                  ConceptNode("mortal")).tv = ???
```

[5]:
```
pattern = InheritanceLink(ConceptNode("Socrates"), ConceptNode("mortal"))
```

Specify rule base.
All rule inheriting from ConceptNode("PLN") will be considired by the rule engine

[6]:
```
rule_base = ConceptNode("PLN")
```

Construct BackwardChainer object
It's arguments are atomspace, rule_base, and pattern to infer

[7]:
```
chainer = BackwardChainer(atomspace, rule_base, pattern)
chainer.do_chain()
print(chainer.get_results())
```

```
(SetLink
```

Running backward chainer

```
> request =
InheritanceLink(ConceptNode("Socrates"),
                ConceptNode("mortal"))
> tr = AtomSpace() # trace atomspace
> rule_base = ConceptNode("PLN") # rules to use
> chainer = BackwardChainer(atomspace, rule_base,
      request,
      trace_as=tr)
> chainer.do_chain()
> print(chainer.get_results())
```

Result

```
> (SetLink
    (InheritanceLink (stv 0.9508 0.828)
    (ConceptNode(“Socrates”),
    (ConceptNode("mortal"))
```

Deduction rule example

premise

A -> B

B -> C

condition = AndLink(BA, CB)

BA = InheritanceLink(var_b, var_a)

CB = InheritanceLink(var_c, var_b)

Ergo: A -> C

```
rewrite = ExecutionOutputLink(
        GroundedSchemaNode("scm: deduction-formula"),
        ListLink(CA, CB, BA))

deduction_link = BindLink(condition, rewrite)
```

Rule structure:

BindLink

```
<variables>
AndLink
    <premise-1>
    ...
    <premise-n>
<conclusion-pattern>
```

Calculate conclusion with ExecutionOutputLink

ExecutionOutputLink

GroundedSchemaNode <formula-name>

List

<conclusion-pattern>

<other arguments>

1 / cog_module

- ..
- mnist.ipynb
- example.py
- mnist.py
- module.py
- opencog.log
- pln.py
- test.py

CogModule

CogModule is python class that attaches itself to an Atom.

It has number of helper methods to generate queries in Atomese

```
class MnistNet(CogModule):
    def __init__(self, atom):
        super().__init__(atom)
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
```

```python
class SumProb(CogModule):
    def forward(self, x, y):
        return x * y

class ProbOfDigit(CogModule):
    def forward(self, probs, i, p_in_range):
        return probs[0][i] * p_in_range


class TorchSum(CogModule):
    def forward(self, *args):
        result = sum(args)
        if result > 1:
            raise RuntimeError("probability exceeds 1")
        return result
```

CogModel

CogModel is class for holding all learnable parameters. Also it provides methods to execute queries. Such wrappers are necessary to deal with caching.

```
class MnistModel(CogModel):
    def __init__(self, atomspace):
        super().__init__()
        self.atomspace = atomspace
        self.mnist = MnistNet(ConceptNode("mnist"))
        self.sum_prob = SumProb(ConceptNode("SumProb"))
        self.digit_prob = ProbOfDigit(ConceptNode("ProbOfDigit"))
        self.torch_sum = TorchSum(ConceptNode("TorchSum"))
        self.inh_weights = torch.nn.Parameter(torch.Tensor([0.3] * 10))
        #  create NumberNodes
        #  attach tensor representing p(number in range)
        for i in range(10):
            NumberNode(str(i)).set_value(PredicateNode("cogNet"), PtrVal
```

```python
def process(self, data, label):
    """
    Accepts batch with features and labels,
    returns probability of labels
    """
    with tmp_atomspace() as atomspace:
        #  compute possible pairs of NumberNodes
        pairs = self.get_all_pairs(label, atomspace)

        # setup input images
        inp1 = InputModule(ConceptNode("img1"), data[0].reshape([1,
        inp2 = InputModule(ConceptNode("img2"), data[1].reshape([1,
        return self.p_correct_answer(pairs, inp1, inp2)


def p_correct_answer(self, pairs, inp1, inp2):
    """
    compute probability of earch pair
```

```
def get_all_pairs(self, label, atomspace):
    """
    Calculate all suitable pairs of digits for given
    """
    label = str(int(label.sum()))
    var_x = atomspace.add_node(types.VariableNode, "X
    var_y = atomspace.add_node(types.VariableNode, "Y
    vardecl = VariableList(TypedVariableLink(var_x, T
    eq = EqualLink(PlusLink(var_x, var_y), NumberNode
    inh1 = InheritanceLink(var_x, ConceptNode("range
    inh2 = InheritanceLink(var_y, ConceptNode("range
    bindlink = BindLink(vardecl, AndLink(inh1, inh2,
    return execute_atom(atomspace, bindlink)
```

```python
def p_correct_answer(self, pairs, inp1, inp2):
    """
    compute probability of earch pair
    compute total probability - sum of pairs
    """
    lst = []
    p_digit = lambda mnist, digit, inh: self.digit_prob.execute(mnist
    for pair in pairs.out:
        p_digit1 = p_digit(self.mnist.execute(inp1.execute()),
                pair.out[0],
                InheritanceLink(pair.out[0], ConceptNode("range")))
        p_digit2 = p_digit(self.mnist.execute(inp2.execute()),
                pair.out[1],
                InheritanceLink(pair.out[1], ConceptNode("range")))
        sum_expr = self.sum_prob.execute(p_digit1, p_digit2)
        lst.append(sum_expr)
    sum_query = self.torch_sum.execute(*lst)
    result = self.execute_atom(sum_query)
    return result
```

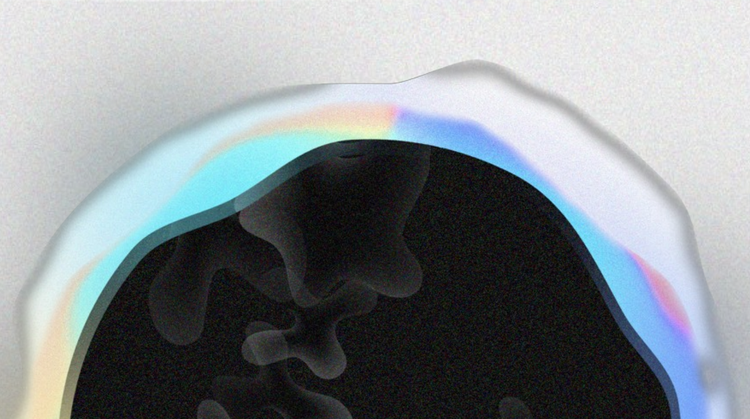
Integrating opencog with neural networks

CogNets

Passing arbitrary python objects between ExecutionOutputLinks

Allows to express computation graph as pytorch expression

Allows to integrate and update ontologies



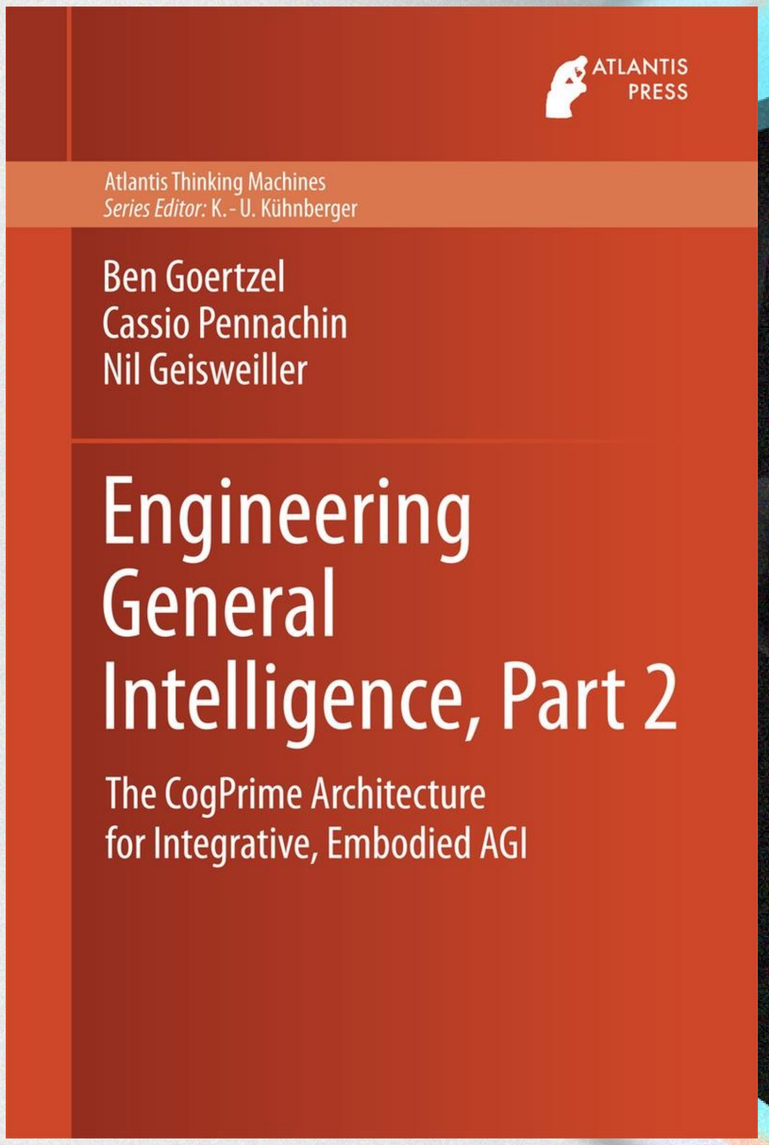
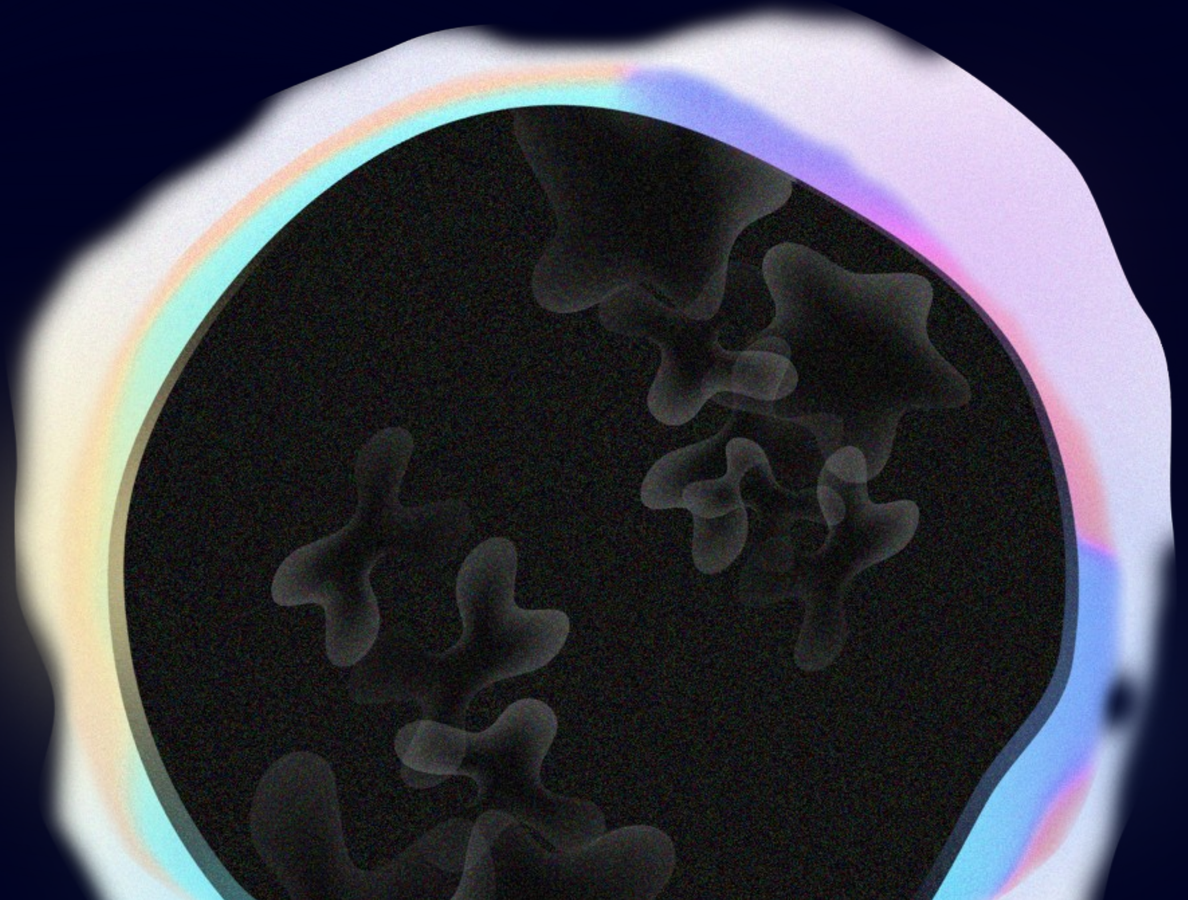
Resources

wiki.opencog.org/

github.com/opencog/

github.com/singnet/semantic-vision

blog.singularitynet.io



SingularityNET