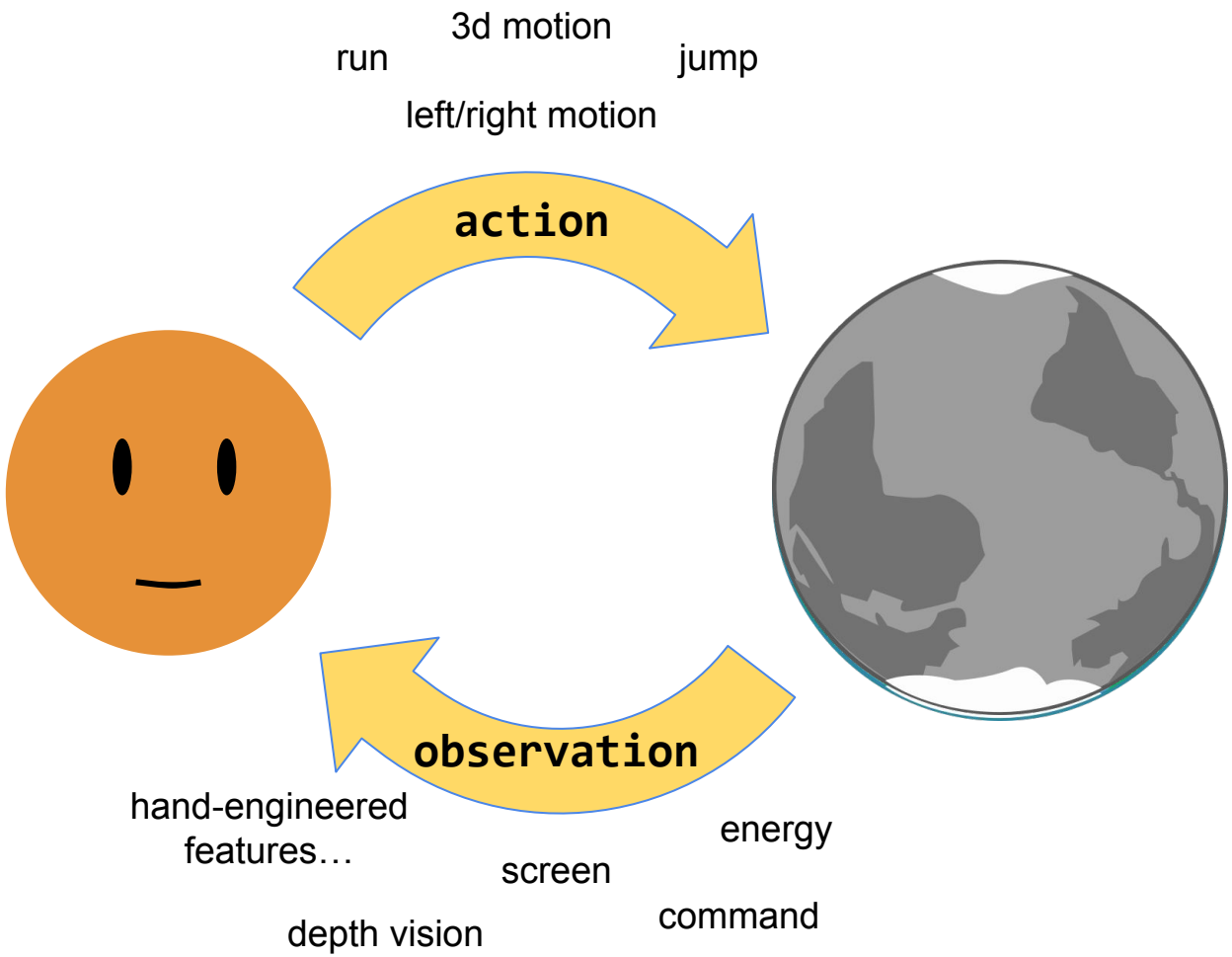
TeNΣ0RCode

 github.com/limboid/tensorcode

TeNΣORCode

- encode and decode arbitrary objects
- develop and evolve multi-modal models
- runtime code generation
- differentiable programming
- develop cognitive architectures

github.com/limboid/tensorcode


3d motion
run
jump
left/right motion
action
observation
hand-engineered
features…
energy
screen
command
depth vision

```python
def step(self, obs: Observation) -> Action:

    # encode inputs
    image_enc = self.ViT(obs.vision)
    text_enc = self.gpt(obs.text)
    latent = concat([image_enc, text_enc, obs.energy])

    # actually make decision
    action = self.mlp(latent)

    # convert back to python objects
    left_right_direction = clip(action[0], -1, +1)
    jump = action[1] > 0

    return Action(left_right_direction, jump)
```

Arbitrary object encoding and decoding

```
import tensorcode as tc


my_object_enc = tc.encode(my_object)
my_object_dec = tc.decode(action, MyObject)
```

```python
def step(self, obs: Observation) -> Action:

    # encode inputs
    image_enc = self.ViT(obs.vision)
    text_enc = self.gpt(obs.text)
    latent = concat([image_enc, text_enc, obs.energy])

    # actually make decision
    action = self.mlp(latent)

    # convert back to python objects
    left_right_direction = clip(action[0], -1, +1)
    jump = action[1] > 0

    return Action(left_right_direction, jump)
```

```python
import tensorcode as tc

def step(self, obs: Observation) -> Action:
    latent = tc.encode(obs)
    action = self.mlp(latent)
    return tc.decode(action, Action)
```

```python
import tensorcode as tc

def step(self, obs: Observation) -> Action:
    latent = tc.encode(obs)
    action = self.mlp(latent)
    return tc.decode(action, Action)
```

```python
class Observation:
    vision: Image
    text: str
    energy: float
```

```python
class Action:
    # -1 for left; 0 for still; +1 for right
    left_right_direction: int
    jump: bool
```

```python
import tensorcode as tc

def step(self, obs: Observation) -> Action:
    latent = tc.encode(obs)
    action = self.mlp(latent)
    return tc.decode(action, Action)
```

```python
class Observation:
    vision: Image
    extra_signal_A: ClassA
    extra_signal_B: ClassB
    extra_signal_C: ClassC
    text: str
    energy: float
```

```python
class Action:
    # -1 for left; 0 for still; +1 for right
    left_right_direction: int
    jump: bool
```

```python
import tensorcode as tc

def step(self, obs: Observation) -> Action:
    latent = tc.encode(obs)
    action = self.mlp(latent)
    return tc.decode(action, Action)
```

```python
class Observation:
    vision: Image
    extra_signal_A: ClassA
    extra_signal_B: ClassB
    extra_signal_C: ClassC
    text: str
    energy: float
```

```python
LEFT = -1; STILL = 0; RIGHT = +1
DIRECTION = Union[Literal[LEFT],
                  Literal[STILL],
                  Literal[RIGHT]]

class Action:
    left_right_direction: DIRECTION
    jump: bool
```

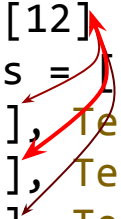
```python
import tensorcode as tc

def step(self, obs: Observation) -> Action:
    latent = tc.encode(obs)
    action = self.mlp(latent)
    return tc.decode(action, Action)
```

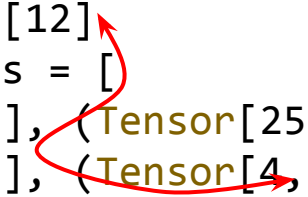
```python
class Observation:
    vision: Image
    extra_signal_A: ClassA
    extra_signal_B: ClassB
    extra_signal_C: ClassC
    text: str
    energy: float
```

```python
LEFT = -1; STILL = 0; RIGHT = +1
DIRECTION = Union[Literal[LEFT],
                  Literal[STILL],
                  Literal[RIGHT]]

class Action:
    direction: tuple[DIRECTION, DIRECTION]
    jump: bool
```

```
class Observation:
    vision = Image(...)
    text = 'go to the door'
    energy = 0.98


query = 'Observation'
keys_and_values = [
    ('vision', Image(...)),
    ('text', 'go to the door'),
    ('energy', 0.98)
]


query = Tensor[1, 768]
keys_and_values = [
    (Tensor[1, 768], Tensor[256, 256, 3]),  # vision
    (Tensor[1, 768], Tensor[4, 768]),  # text
    (Tensor[1, 768], Tensor[1])  # energy
]
```

```
query = Tensor[1, 768]
keys_and_values = [
    (Tensor[1, 768], Tensor[256, 256, 3]),  # vision
    (Tensor[1, 768], Tensor[4, 768]),  # text
    (Tensor[1, 768], Tensor[1])  # energy
]

query = Tensor[12]
keys_and_values = [
    (Tensor[12], Tensor[256, 256, 3]),
    (Tensor[12], Tensor[4, 768]),
    (Tensor[12], Tensor[1])
]

query = Tensor[12]
keys_and_values = [
    (Tensor[12], (Tensor[256, 12], Tensor[256, 16, 16, 3])),
    (Tensor[12], (Tensor[4, 12], Tensor[4, 768])),
    (Tensor[12], Tensor[1])
]
```



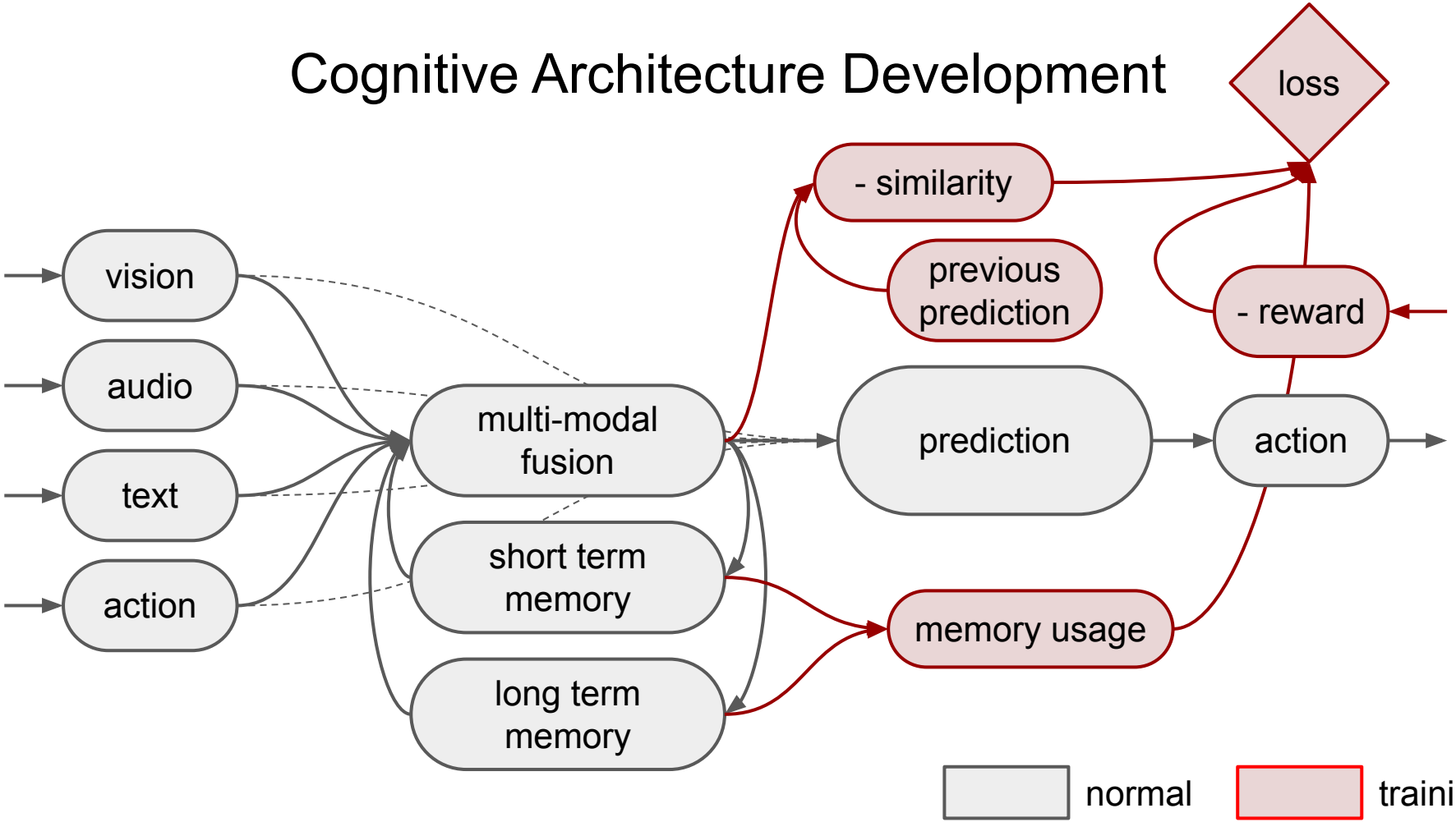


Runtime Code Generation

```
tc.exec("put worker in the READY state", worker)

winner = tc.select("the highest performing, most diverse, and
simplest model", models, n=1)

graph = tc.create(Graph, "John's memories",
                  photos, videos, messages, gps_data)

thought_factory = tc.patterns.Factory(Text, Image, Audio, Video)
```

Differentiable Programming

```
x = ... # some scalar tensor

@tc.If(x == 1)
def MyIf():
  return x
@MyIf.Elif(x == 2)
def ElifClause():
  return x ** 2
@MyIf.Else(x == 3)
def ElseClause():
  return -x

y = MyIf()
dx = y.grad(x)
```


Cognitive Architecture Development
loss
- similarity
previous prediction
- reward
vision
audio
text
action
multi-modal fusion
prediction
action
short term memory
memory usage
long term memory
normal
training

```
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

```
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

```
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

```
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

```
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

```python
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

```
def step(self, obs: Observation, training=False) -> Action:

    # input and recurrent information
    obs_enc = tc.encode(obs)
    stm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)
    ltm = tc.select(query=obs_enc @ self.W_stm, values=self.short_term_memory)

    # global workspace state
    perception_enc = self.perception_mlp(concat(obs_enc, stm, ltm))
    perception = tc.decode(perception_enc, Perception,
        context={'vision': obs.vision, 'prompt': obs.prompt})
        # extras kwarg useful when perception_enc bottleneck is too tight to squeeze information through

    # prediction
    prediction_enc = self.prediction_mlp(perception_enc)
    prediction = tc.decode(prediction_enc, Perception, context=perception)

    # get outputs
    self.monologue += perception.text
    self.short_term_memory.remember(tc.select('store short-term memories', prediction.thoughts))
    self.long_term_memory.remember(tc.select('store long-term memories', prediction.thoughts))
    action = prediction.action

    if training:  # feedback
       self.add_loss('reward', stopgrad(prediction.pain - prediction.pleasure)
       self.add_loss('pred_error', -tc.similarity(perception, self._prev_prediction)
       self._prev_prediction = prediction
       self.add_loss('pred_enc_error', -tc.similarity(perception_enc, self._prev_prediction_enc)
       self._prev_prediction_enc = prediction_enc
       self.add_loss('stm_cognitive_load', self.l_stm_len*len(stm))
       self.add_loss('ltm_cognitive_load', self.l_ltm_len*len(ltm))

    return action
```

TeNΣORCode

- encode and decode arbitrary objects
- develop and evolve multi-modal models
- runtime code generation
- differentiable programming
- develop cognitive architectures

github.com/limboid/tensorcode