```
  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_densel(sim_vec) # (?, rules, dim//2)
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = mult([sim_vec, rule_mask])
    sim_vec = squeeze2(sim_vec) # (?, rules)
    # sim_vec = L.Softmax(axis=1)(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    new_state = dot11([sim_vec, new_states])
    # Apply gating
    gate = gating(new_state)
    outs.append(gate)
    new_state = gate2([new_state, state, gate])
    state = new_state

    # Apply gating
    # gate = gating(state)
    # outs.append(gate)
    # state = gate2([state, new_state, gate])

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== ima_gate_4.py ====
"""Iterative memory attention model."""
import numpy as np
import keras.backend as K
import keras.layers as L
from keras.models import Model
import tensorflow as tf
```

```
import random
import random as python_random
import os

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long
def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp

  # Contextual embeddeding of symbols
  onehot_weights = np.eye(char_size)
  onehot_weights[0, 0] = 0 # Clear zero index
  onehot = L.Embedding(char_size, char_size,
                       trainable=False,
                       weights=[onehot_weights],
                       name='onehot')
  embedded_ctx = onehot(context) # (?, rules, preds, chars, char_size)
  embedded_q = onehot(query) # (?, chars, char_size)
  K.print_tensor(embedded_q)
  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
  embedded_ctx_preds = NestedTimeDist(NestedTimeDist(embed_pred, name='nest1'), name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, go_backwards=True, name='embed_rule')
  # embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  rule_to_att = L.TimeDistributed(L.Dense(dim//2, name='rule_to_att'), name='d_rule_to_att')
  state_to_att = L.Dense(dim//2, name='state_to_att')
  repeat_toctx = L.RepeatVector(K.shape(context)[1], name='repeat_to_ctx')
  att_dense = L.TimeDistributed(L.Dense(1), name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')

  unifier = NestedTimeDist(ZeroGRU(dim, name='unifier'), name='dist_unifier')
  # dot11 = L.Dot((1, 1))
```

```
  gating = L.Dense(1, activation='sigmoid', name='gating')
  gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')

  # Reasoning iterations
  state = L.Dense(dim, activation='tanh', name='init_state')(embedded_predq)
  ctx_rules = rule_to_att(embedded_rules)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    att_state = state_to_att(state)  # (?, ATT_LATENT_DIM)
    att_state = repeat_toctx(att_state)  # (?, rules, ATT_LATENT_DIM)
    sim_vec = L.multiply([ctx_rules, att_state])
    sim_vec = att_dense(sim_vec)  # (?, rules, 1)
    sim_vec = squeeze2(sim_vec)  # (?, rules)
    sim_vec = L.Softmax(axis=1)(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    new_state = L.dot([sim_vec, new_states], (1, 1))
    s_m_ns = L.multiply([state, new_state])
    s_s_ns = L.subtract([state, new_state])
    gate = L.concatenate([state, new_state, s_m_ns, s_s_ns])
    gate = gating(gate)
    outs.append(gate)
    new_state = gate2([state, new_state, gate])
    state = new_state

    # Apply gating
    # gate = gating(state)
    # outs.append(gate)
    # state = gate2([state, new_state, gate])

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    # optimizer = tf.keras.optimizers.Adam(lr=0.01)
    model.compile(loss='binary_crossentropy',
                  optimizer="adam",
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== ima_glove.py ====
"""Iterative memory attention model with Glove as pre-training embedding."""
import numpy as np
import keras.backend as K
```

```
import keras.layers as L
from keras.models import Model
import os
import re
import json_lines
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.np_utils import *
from keras.layers import Embedding
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  # context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  # query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp

  # Contextual embeddeding of symbols
  # texts = []  # list of text samples
  # id_list = []
  # question_list = []
  # label_list = []
  # labels_index = {}  # dictionary mapping label name to numeric id
  # labels = []  # list of label ids
  # TEXT_DATA_DIR = os.path.abspath('.') + "/data/pararule"
  # # TEXT_DATA_DIR = "D:\\AllenAI\\20_newsgroup"
  # Str = '.jsonl'
  # CONTEXT_TEXTS = []
  # test_str = 'test'
  # meta_str = 'meta'

  # for name in sorted(os.listdir(TEXT_DATA_DIR)):
  #   path = os.path.join(TEXT_DATA_DIR, name)
  #   if os.path.isdir(path):
  #     label_id = len(labels_index)
  #     labels_index[name] = label_id
  #     for fname in sorted(os.listdir(path)):
  #       fpath = os.path.join(path, fname)
  #       if Str in fpath:
  #         if test_str not in fpath:
  #           if meta_str not in fpath:
  #             with open(fpath) as f:
  #               for l in json_lines.reader(f):
  #                 if l["id"] not in id_list:
  #                   id_list.append(l["id"])
  #                   questions = l["questions"]
```

```
#                   context = l["context"].replace("\n", " ")
#                   context = re.sub(r'\s+', ' ', context)
#                   CONTEXT_TEXTS.append(context)
#                   for i in range(len(questions)):
#                     text = questions[i]["text"]
#                     label = questions[i]["label"]
#                     if label == True:
#                       t = 1
#                     else:
#                       t = 0
#                     q = re.sub(r'\s+', ' ', text)
#                     texts.append(context)
#                     question_list.append(q)
#                     label_list.append(int(t))
#             f.close()
#       # labels.append(label_id)

print('Found %s texts.' % len(CONTEXT_TEXTS))

# MAX_NB_WORDS = 20000
# MAX_SEQUENCE_LENGTH = 1000
# tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
# tokenizer.fit_on_texts(texts)
# #sequences = tokenizer.texts_to_sequences(texts)

word_index = WORD_INDEX
print('Found %s unique tokens.' % len(word_index))

#data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

# labels = to_categorical(np.asarray(labels))
#print('Shape of data tensor:', data.shape)
# print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
# indices = np.arange(data.shape[0])
# np.random.shuffle(indices)
# data = data[indices]
# labels = labels[indices]

embeddings_index = {}
GLOVE_DIR = os.path.abspath('.') + "/data/glove"
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
for line in f:
  values = line.split()
  word = values[0]
  coefs = np.asarray(values[1:], dtype='float32')
  embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))

EMBEDDING_DIM = 100
```

```
  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
      # words not found in embedding index will be all-zeros.
      embedding_matrix[i] = embedding_vector

  embedding_layer = L.Embedding(len(word_index) + 1,
                                EMBEDDING_DIM,
                                weights=[embedding_matrix],
                                trainable=False)

  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  embedded_ctx = embedding_layer(context) # (?, rules, preds, chars, char_size)
  embedded_q = embedding_layer(query) # (?, chars, char_size)
  #onehot_weights = np.eye(char_size)
  #onehot_weights[0, 0] = 0 # Clear zero index
  # onehot = L.Embedding(char_size, char_size,
  #                      trainable=False,
  #                      weights=[onehot_weights],
  #                      name='onehot')
  # embedded_ctx = onehot(context) # (?, rules, preds, chars, char_size)
  # embedded_q = onehot(query) # (?, chars, char_size)

  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
            embedded_ctx_preds      =      L.TimeDistributed(L.TimeDistributed(embed_pred,      name='nest1'),
name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, go_backwards=True, name='embed_rule')
  # embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  repeat_toctx = L.RepeatVector(K.shape(embedded_ctx)[1], name='repeat_to_ctx')
  diff_sq = L.Lambda(lambda xy: K.square(xy[0]-xy[1]), output_shape=(None, dim), name='diff_sq')
  mult = L.Multiply()
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_densel = L.Dense(dim//2, activation='tanh', name='att_densel')
  att_dense = L.Dense(1, activation='sigmoid', name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
    rule_mask  =  L.Lambda(lambda  x:  K.cast(K.any(K.not_equal(x,  0),  axis=-1,  keepdims=True),  'float32'),
```

```
name='rule_mask')(embedded_rules)

  unifier = NestedTimeDist(ZeroGRU(dim, name='unifier'), name='dist_unifier')
  dot11 = L.Dot((1, 1))
  # gating = L.Dense(1, activation='sigmoid', name='gating')
  # gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')

  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_densel(sim_vec) # (?, rules, dim//2)
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = mult([sim_vec, rule_mask])
    sim_vec = squeeze2(sim_vec) # (?, rules)
    # sim_vec = L.Softmax(axis=1)(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    state = dot11([sim_vec, new_states])

    # Apply gating
    # gate = gating(state)
    # outs.append(gate)
    # state = gate2([state, new_state, gate])

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== ima_glove_gate3.py ====
"""Iterative memory attention model with Glove as pre-training embedding."""
import numpy as np
import keras.backend as K
import keras.layers as L
```

```
from keras.models import Model
from keras.optimizers import adam
import os
import re
import json_lines
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.np_utils import *
from keras.layers import Embedding
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  # context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  # query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp

  # Contextual embeddeding of symbols
  # texts = []  # list of text samples
  # id_list = []
  # question_list = []
  # label_list = []
  # labels_index = {}  # dictionary mapping label name to numeric id
  # labels = []  # list of label ids
  # TEXT_DATA_DIR = os.path.abspath('.') + "/data/pararule"
  # # TEXT_DATA_DIR = "D:\\AllenAI\\20_newsgroup"
  # Str = '.jsonl'
  # CONTEXT_TEXTS = []
  # test_str = 'test'
  # meta_str = 'meta'

  # for name in sorted(os.listdir(TEXT_DATA_DIR)):
  #   path = os.path.join(TEXT_DATA_DIR, name)
  #   if os.path.isdir(path):
  #     label_id = len(labels_index)
  #     labels_index[name] = label_id
  #     for fname in sorted(os.listdir(path)):
  #       fpath = os.path.join(path, fname)
  #       if Str in fpath:
  #         if test_str not in fpath:
  #           if meta_str not in fpath:
  #             with open(fpath) as f:
  #               for l in json_lines.reader(f):
  #                 if l["id"] not in id_list:
  #                   id_list.append(l["id"])
  #                   questions = l["questions"]
```

```
#                   context = l["context"].replace("\n", " ")
#                   context = re.sub(r'\s+', ' ', context)
#                   CONTEXT_TEXTS.append(context)
#                   for i in range(len(questions)):
#                     text = questions[i]["text"]
#                     label = questions[i]["label"]
#                     if label == True:
#                       t = 1
#                     else:
#                       t = 0
#                     q = re.sub(r'\s+', ' ', text)
#                     texts.append(context)
#                     question_list.append(q)
#                     label_list.append(int(t))
#             f.close()
#       # labels.append(label_id)

print('Found %s texts.' % len(CONTEXT_TEXTS))

# MAX_NB_WORDS = 20000
# MAX_SEQUENCE_LENGTH = 1000
# tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
# tokenizer.fit_on_texts(texts)
# #sequences = tokenizer.texts_to_sequences(texts)

word_index = WORD_INDEX
print('Found %s unique tokens.' % len(word_index))

#data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

# labels = to_categorical(np.asarray(labels))
#print('Shape of data tensor:', data.shape)
# print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
# indices = np.arange(data.shape[0])
# np.random.shuffle(indices)
# data = data[indices]
# labels = labels[indices]

embeddings_index = {}
GLOVE_DIR = os.path.abspath('.') + "/data/glove"
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
for line in f:
  values = line.split()
  word = values[0]
  coefs = np.asarray(values[1:], dtype='float32')
  embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))

EMBEDDING_DIM = 100
```

```
  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
      # words not found in embedding index will be all-zeros.
      embedding_matrix[i] = embedding_vector

  embedding_layer = L.Embedding(len(word_index) + 1,
                                EMBEDDING_DIM,
                                weights=[embedding_matrix],
                                trainable=False)

  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  embedded_ctx = embedding_layer(context) # (?, rules, preds, chars, char_size)
  embedded_q = embedding_layer(query) # (?, chars, char_size)
  #onehot_weights = np.eye(char_size)
  #onehot_weights[0, 0] = 0 # Clear zero index
  # onehot = L.Embedding(char_size, char_size,
  #                      trainable=False,
  #                      weights=[onehot_weights],
  #                      name='onehot')
  # embedded_ctx = onehot(context) # (?, rules, preds, chars, char_size)
  # embedded_q = onehot(query) # (?, chars, char_size)

  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
            embedded_ctx_preds      =      L.TimeDistributed(L.TimeDistributed(embed_pred,      name='nest1'),
name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, go_backwards=True, name='embed_rule')
  # embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  repeat_toctx = L.RepeatVector(K.shape(embedded_ctx)[1], name='repeat_to_ctx')
  diff_sq = L.Lambda(lambda xy: K.square(xy[0]-xy[1]), output_shape=(None, dim), name='diff_sq')
  mult = L.Multiply()
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_densel = L.Dense(dim//2, activation='tanh', name='att_densel', activity_regularizer="l1")
  att_dense = L.Dense(1, activation='sigmoid', name='att_dense', activity_regularizer="l1")
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
    rule_mask  =  L.Lambda(lambda  x:  K.cast(K.any(K.not_equal(x,  0),  axis=-1,  keepdims=True),  'float32'),
```

```
name='rule_mask')(embedded_rules)

  unifier = NestedTimeDist(ZeroGRU(dim, name='unifier'), name='dist_unifier')
  dot11 = L.Dot((1, 1))
  gating = L.Dense(1, activation='sigmoid', name='gating', activity_regularizer="l1")
  gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')

  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_densel(sim_vec) # (?, rules, dim//2)
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = mult([sim_vec, rule_mask])
    sim_vec = squeeze2(sim_vec) # (?, rules)
    # sim_vec = L.Softmax(axis=1)(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    new_state = dot11([sim_vec, new_states])
    # Apply gating
    gate = gating(new_state)
    outs.append(gate)
    new_state = gate2([new_state, state, gate])
    state = new_state

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out', activity_regularizer="l1")(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    # opt = adam(lr=0.00001)
    model.compile(loss='binary_crossentropy',
                  optimizer="adam",
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== ima_glove_gate4.py ====
"""Iterative memory attention model."""
import numpy as np
import keras.backend as K
```

```
import keras.layers as L
from keras.models import Model
import tensorflow as tf
import random
import random as python_random
import os
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long
def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp

  print('Found %s texts.' % len(CONTEXT_TEXTS))
  word_index = WORD_INDEX
  print('Found %s unique tokens.' % len(word_index))

  embeddings_index = {}
  GLOVE_DIR = os.path.abspath('.') + "/data/glove"
  f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
  for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
  f.close()

  print('Found %s word vectors.' % len(embeddings_index))

  EMBEDDING_DIM = 100

  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
      # words not found in embedding index will be all-zeros.
      embedding_matrix[i] = embedding_vector

  embedding_layer = L.Embedding(len(word_index) + 1,
                                EMBEDDING_DIM,
                                weights=[embedding_matrix],
                                trainable=False)

  # Contextual embeddeding of symbols
  # onehot_weights = np.eye(char_size)
  # onehot_weights[0, 0] = 0 # Clear zero index
```

```
  # onehot = L.Embedding(char_size, char_size,
  #                      trainable=False,
  #                      weights=[onehot_weights],
  #                      name='onehot')
  # embedded_ctx = onehot(context) # (?, rules, preds, chars, char_size)
  # embedded_q = onehot(query) # (?, chars, char_size)
  embedded_ctx = embedding_layer(context) # (?, rules, preds, chars, char_size)
  embedded_q = embedding_layer(query) # (?, chars, char_size)
  K.print_tensor(embedded_q)
  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
  embedded_ctx_preds = NestedTimeDist(NestedTimeDist(embed_pred, name='nest1'), name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, go_backwards=True, name='embed_rule')
  # embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  rule_to_att = L.TimeDistributed(L.Dense(dim//2, name='rule_to_att'), name='d_rule_to_att')
  state_to_att = L.Dense(dim//2, name='state_to_att')
  repeat_toctx = L.RepeatVector(K.shape(context)[1], name='repeat_to_ctx')
  att_dense = L.TimeDistributed(L.Dense(1), name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')

  unifier = NestedTimeDist(ZeroGRU(dim, name='unifier'), name='dist_unifier')
  # dot11 = L.Dot((1, 1))
  gating = L.Dense(1, activation='sigmoid', name='gating')
  gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')

  # Reasoning iterations
  state = L.Dense(dim, activation='tanh', name='init_state')(embedded_predq)
  ctx_rules = rule_to_att(embedded_rules)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    att_state = state_to_att(state)  # (?, ATT_LATENT_DIM)
    att_state = repeat_toctx(att_state)  # (?, rules, ATT_LATENT_DIM)
    sim_vec = L.multiply([ctx_rules, att_state])
    sim_vec = att_dense(sim_vec)  # (?, rules, 1)
    sim_vec = squeeze2(sim_vec)  # (?, rules)
    sim_vec = L.Softmax(axis=1)(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
```

```
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    new_state = L.dot([sim_vec, new_states], (1, 1))
    s_m_ns = L.multiply([state, new_state])
    s_s_ns = L.subtract([state, new_state])
    gate = L.concatenate([state, new_state, s_m_ns, s_s_ns])
    gate = gating(gate)
    outs.append(gate)
    new_state = gate2([state, new_state, gate])
    state = new_state

    # Apply gating
    # gate = gating(state)
    # outs.append(gate)
    # state = gate2([state, new_state, gate])

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    # optimizer = tf.keras.optimizers.Adam(lr=0.01)
    model.compile(loss='binary_crossentropy',
                  optimizer="adam",
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== ima_glove_gate_conceptrule.py ====
"""Iterative memory attention model with Glove as pre-training embedding."""
import numpy as np
import keras.backend as K
import keras.layers as L
from keras.models import Model
import os
import re
import json_lines
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.np_utils import *
from keras.layers import Embedding
from word_dict_gen_conceptrule import WORD_INDEX, CONTEXT_TEXTS

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
```

```
# Context: (rules, preds, chars,)
# context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
# query = L.Input(shape=(None,), name='query', dtype='int32')

if ilp:
  context, query, templates = ilp

# Contextual embeddeding of symbols
# texts = []  # list of text samples
# id_list = []
# question_list = []
# label_list = []
# labels_index = {}  # dictionary mapping label name to numeric id
# labels = []  # list of label ids
# TEXT_DATA_DIR = os.path.abspath('.') + "/data/pararule"
# # TEXT_DATA_DIR = "D:\\AllenAI\\20_newsgroup"
# Str = '.jsonl'
# CONTEXT_TEXTS = []
# test_str = 'test'
# meta_str = 'meta'

# for name in sorted(os.listdir(TEXT_DATA_DIR)):
#   path = os.path.join(TEXT_DATA_DIR, name)
#   if os.path.isdir(path):
#     label_id = len(labels_index)
#     labels_index[name] = label_id
#     for fname in sorted(os.listdir(path)):
#       fpath = os.path.join(path, fname)
#       if Str in fpath:
#         if test_str not in fpath:
#           if meta_str not in fpath:
#             with open(fpath) as f:
#               for l in json_lines.reader(f):
#                 if l["id"] not in id_list:
#                   id_list.append(l["id"])
#                   questions = l["questions"]
#                   context = l["context"].replace("\n", " ")
#                   context = re.sub(r'\s+', ' ', context)
#                   CONTEXT_TEXTS.append(context)
#                   for i in range(len(questions)):
#                     text = questions[i]["text"]
#                     label = questions[i]["label"]
#                     if label == True:
#                       t = 1
#                     else:
#                       t = 0
#                     q = re.sub(r'\s+', ' ', text)
#                     texts.append(context)
#                     question_list.append(q)
#                     label_list.append(int(t))
#             f.close()
#       # labels.append(label_id)

print('Found %s texts.' % len(CONTEXT_TEXTS))
```

```
# MAX_NB_WORDS = 20000
# MAX_SEQUENCE_LENGTH = 1000
# tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
# tokenizer.fit_on_texts(texts)
# #sequences = tokenizer.texts_to_sequences(texts)

word_index = WORD_INDEX
print('Found %s unique tokens.' % len(word_index))

#data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)

# labels = to_categorical(np.asarray(labels))
#print('Shape of data tensor:', data.shape)
# print('Shape of label tensor:', labels.shape)

# split the data into a training set and a validation set
# indices = np.arange(data.shape[0])
# np.random.shuffle(indices)
# data = data[indices]
# labels = labels[indices]

embeddings_index = {}
GLOVE_DIR = os.path.abspath('.') + "/data/glove"
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
for line in f:
  values = line.split()
  word = values[0]
  coefs = np.asarray(values[1:], dtype='float32')
  embeddings_index[word] = coefs
f.close()

print('Found %s word vectors.' % len(embeddings_index))

EMBEDDING_DIM = 100

embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
  embedding_vector = embeddings_index.get(word)
  if embedding_vector is not None:
    # words not found in embedding index will be all-zeros.
    embedding_matrix[i] = embedding_vector

embedding_layer = L.Embedding(len(word_index) + 1,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            trainable=False)

context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
query = L.Input(shape=(None,), name='query', dtype='int32')

embedded_ctx = embedding_layer(context) # (?, rules, preds, chars, char_size)
embedded_q = embedding_layer(query) # (?, chars, char_size)
#onehot_weights = np.eye(char_size)
```

```
  #onehot_weights[0, 0] = 0 # Clear zero index
  # onehot = L.Embedding(char_size, char_size,
  #                      trainable=False,
  #                      weights=[onehot_weights],
  #                      name='onehot')
  # embedded_ctx = onehot(context) # (?, rules, preds, chars, char_size)
  # embedded_q = onehot(query) # (?, chars, char_size)

  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
            embedded_ctx_preds      =      L.TimeDistributed(L.TimeDistributed(embed_pred,      name='nest1'),
name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, go_backwards=True, name='embed_rule')
  # embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  repeat_toctx = L.RepeatVector(K.shape(embedded_ctx)[1], name='repeat_to_ctx')
  diff_sq = L.Lambda(lambda xy: K.square(xy[0]-xy[1]), output_shape=(None, dim), name='diff_sq')
  mult = L.Multiply()
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_densel = L.Dense(dim//2, activation='tanh', name='att_densel')
  att_dense = L.Dense(1, activation='sigmoid', name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
    rule_mask  =  L.Lambda(lambda  x:  K.cast(K.any(K.not_equal(x,  0),  axis=-1,  keepdims=True),  'float32'),
name='rule_mask')(embedded_rules)

  unifier = NestedTimeDist(ZeroGRU(dim, name='unifier'), name='dist_unifier')
  dot11 = L.Dot((1, 1))
  gating = L.Dense(1, activation='sigmoid', name='gating')
  gate2 = L.Lambda(lambda xyg: xyg[2]*xyg[0] + (1-xyg[2])*xyg[1], name='gate')

  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_densel(sim_vec) # (?, rules, dim//2)
```

```
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = mult([sim_vec, rule_mask])
    sim_vec = squeeze2(sim_vec) # (?, rules)
    # sim_vec = L.Softmax(axis=1)(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    new_state = dot11([sim_vec, new_states])
    # Apply gating
    gate = gating(new_state)
    outs.append(gate)
    new_state = gate2([new_state, state, gate])
    state = new_state

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== imarsm_glove.py ====
"""Iterative memory attention model."""
import numpy as np
import os
import keras.backend as K
import keras.layers as L
from keras.models import Model
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp
```

```
  print('Found %s texts.' % len(CONTEXT_TEXTS))
  word_index = WORD_INDEX
  print('Found %s unique tokens.' % len(word_index))

  embeddings_index = {}
  GLOVE_DIR = os.path.abspath('.') + "/data/glove"
  f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
  for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
  f.close()

  print('Found %s word vectors.' % len(embeddings_index))

  EMBEDDING_DIM = 100

  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
      # words not found in embedding index will be all-zeros.
      embedding_matrix[i] = embedding_vector

  # Contextual embeddeding of symbols
  # onehot_weights = np.eye(char_size)
  # onehot_weights[0, 0] = 0 # Clear zero index
  # onehot = L.Embedding(char_size, char_size,
  #                      trainable=False,
  #                      weights=[onehot_weights],
  #                      name='onehot')
  embedding_layer = L.Embedding(len(word_index) + 1,
                                EMBEDDING_DIM,
                                weights=[embedding_matrix],
                                trainable=False)
  embedded_ctx = embedding_layer(context) # (?, rules, preds, chars, char_size)
  embedded_q = embedding_layer(query) # (?, chars, char_size)

  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
  embedded_ctx_preds = NestedTimeDist(NestedTimeDist(embed_pred, name='nest1'), name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  embed_rule = ZeroGRU(dim, name='embed_rule')
  embedded_rules = NestedTimeDist(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  # (?, rules, dim)
```

```
  # Reused layers over iterations
  repeat_toctx = L.RepeatVector(K.shape(embedded_ctx)[1], name='repeat_to_ctx')
  diff_sq = L.Lambda(lambda xy: K.square(xy[0]-xy[1]), output_shape=(None, dim), name='diff_sq')
  mult = L.Multiply()
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_densel = L.Dense(dim//2, activation='tanh', name='att_densel')
  att_dense = L.Dense(1, name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
  softmax1 = L.Softmax(axis=1)
  unifier = NestedTimeDist(ZeroGRU(dim, go_backwards=False, name='unifier'), name='dist_unifier')
  dot11 = L.Dot((1, 1))

  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_densel(sim_vec) # (?, rules, dim//2)
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = squeeze2(sim_vec) # (?, rules)
    sim_vec = softmax1(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    state = dot11([sim_vec, new_states])

  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== imasm_glove.py ====
"""Iterative memory attention model."""
import numpy as np
import os
import keras.backend as K
```

```
import keras.layers as L
from keras.models import Model
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

def build_model(char_size=27, dim=64, iterations=4, training=True, ilp=False, pca=False):
  """Build the model."""
  # Inputs
  # Context: (rules, preds, chars,)
  context = L.Input(shape=(None, None, None,), name='context', dtype='int32')
  query = L.Input(shape=(None,), name='query', dtype='int32')

  if ilp:
    context, query, templates = ilp

  print('Found %s texts.' % len(CONTEXT_TEXTS))
  word_index = WORD_INDEX
  print('Found %s unique tokens.' % len(word_index))

  embeddings_index = {}
  GLOVE_DIR = os.path.abspath('.') + "/data/glove"
  f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'), 'r', encoding='utf-8')
  for line in f:
      values = line.split()
      word = values[0]
      coefs = np.asarray(values[1:], dtype='float32')
      embeddings_index[word] = coefs
  f.close()

  print('Found %s word vectors.' % len(embeddings_index))

  EMBEDDING_DIM = 100

  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
      embedding_vector = embeddings_index.get(word)
      if embedding_vector is not None:
          # words not found in embedding index will be all-zeros.
          embedding_matrix[i] = embedding_vector

  # Contextual embeddeding of symbols
  # onehot_weights = np.eye(char_size)
  # onehot_weights[0, 0] = 0 # Clear zero index
  # onehot = L.Embedding(char_size, char_size,
  #                      trainable=False,
  #                      weights=[onehot_weights],
  #                      name='onehot')
  embedding_layer = L.Embedding(len(word_index) + 1,
                                EMBEDDING_DIM,
                                weights=[embedding_matrix],
                                trainable=False)
```

```
  embedded_ctx = embedding_layer(context) # (?, rules, preds, chars, char_size)
  embedded_q = embedding_layer(query) # (?, chars, char_size)

  if ilp:
    # Combine the templates with the context, (?, rules+temps, preds, chars, char_size)
        embedded_ctx  =  L.Lambda(lambda  xs:  K.concatenate(xs,  axis=1),  name='template_concat')([templates,
embedded_ctx])
    # embedded_ctx = L.concatenate([templates, embedded_ctx], axis=1)

  embed_pred = ZeroGRU(dim, go_backwards=True, name='embed_pred')
  embedded_predq = embed_pred(embedded_q) # (?, dim)
  # For every rule, for every predicate, embed the predicate
            embedded_ctx_preds      =      L.TimeDistributed(L.TimeDistributed(embed_pred,      name='nest1'),
name='nest2')(embedded_ctx)
  # (?, rules, preds, dim)

  # embed_rule = ZeroGRU(dim, name='embed_rule')
  # embedded_rules = L.TimeDistributed(embed_rule, name='d_embed_rule')(embedded_ctx_preds)
  get_heads = L.Lambda(lambda x: x[:, :, 0, :], name='rule_heads')
  embedded_rules = get_heads(embedded_ctx_preds)
  # (?, rules, dim)

  # Reused layers over iterations
  repeat_toctx = L.RepeatVector(K.shape(embedded_ctx)[1], name='repeat_to_ctx')
  diff_sq = L.Lambda(lambda xy: K.square(xy[0]-xy[1]), output_shape=(None, dim), name='diff_sq')
  mult = L.Multiply()
  concat = L.Lambda(lambda xs: K.concatenate(xs, axis=2), output_shape=(None, dim*5), name='concat')
  att_densel = L.Dense(dim//2, activation='tanh', name='att_densel')
  att_dense = L.Dense(1, name='att_dense')
  squeeze2 = L.Lambda(lambda x: K.squeeze(x, 2), name='sequeeze2')
  softmax1 = L.Softmax(axis=1)
  unifier = NestedTimeDist(ZeroGRU(dim, go_backwards=False, name='unifier'), name='dist_unifier')
  dot11 = L.Dot((1, 1))

  # Reasoning iterations
  state = embedded_predq
  repeated_q = repeat_toctx(embedded_predq)
  outs = list()
  for _ in range(iterations):
    # Compute attention between rule and query state
    ctx_state = repeat_toctx(state) # (?, rules, dim)
    s_s_c = diff_sq([ctx_state, embedded_rules])
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_densel(sim_vec) # (?, rules, dim//2)
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = squeeze2(sim_vec) # (?, rules)
    sim_vec = softmax1(sim_vec)
    outs.append(sim_vec)

    # Unify every rule and weighted sum based on attention
    new_states = unifier(embedded_ctx_preds, initial_state=[state])
    # (?, rules, dim)
    state = dot11([sim_vec, new_states])
```

```
  # Predication
  out = L.Dense(1, activation='sigmoid', name='out')(state)
  if ilp:
    return outs, out
  elif pca:
    model = Model([context, query], [embedded_rules])
  elif training:
    model = Model([context, query], [out])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['acc'])
  else:
    model = Model([context, query], outs + [out])
  return model

==== inject_profiler.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: inject_profiler.py (Windows Staging Edition)
Purpose: Inject agent_profiler code into all real agents inside Allinonepy\agents\
"""

import threading
from pathlib import Path

# Your specific folder path (Windows safe)
AGENTS_DIR = Path("C:/Users/PC/Desktop/Operation Future/Allinonepy/agents")
PROFILER_IMPORT = "from utils import agent_profiler"
PROFILER_BOOT = "threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()"
INJECTION_TAG = "# [PROFILER_INJECTED]"

def already_injected(code):
    return INJECTION_TAG in code or "agent_profiler" in code

def inject_profiler_code(file_path):
    try:
        code = file_path.read_text(encoding='utf-8')

        if already_injected(code):
            print(f"[inject_profiler] Skipped (already injected): {file_path.name}")
            return

        lines = code.splitlines()
        new_lines = []
        inserted = False

        for i, line in enumerate(lines):
            new_lines.append(line)
            if not inserted and line.strip().startswith("import"):
                if i + 1 < len(lines) and not lines[i + 1].startswith("import"):
                    new_lines.append("import threading")
                    new_lines.append(PROFILER_IMPORT)
                    new_lines.append(INJECTION_TAG)
```

```
                    new_lines.append(PROFILER_BOOT)
                    inserted = True

        if inserted:
            file_path.write_text("\n".join(new_lines), encoding='utf-8')
            print(f"[inject_profiler] [OK] Injected into: {file_path.name}")
        else:
            print(f"[inject_profiler] WARNING Could not inject into: {file_path.name}")

    except Exception as e:
        print(f"[inject_profiler] ERROR Error with {file_path.name}: {e}")

def main():
    if not AGENTS_DIR.exists():
        print(f"[inject_profiler] ERROR: Cannot find directory: {AGENTS_DIR}")
        return

    for py_file in AGENTS_DIR.glob("*.py"):
        inject_profiler_code(py_file)

if __name__ == "__main__":
    main()

# [CONFIG_PATCHED]

==== json_schema_pydantic_example.py ====
# Usage:
#! ./llama-server -m some-model.gguf &
#! pip install pydantic
#! python json_schema_pydantic_example.py

from pydantic import BaseModel, Field, TypeAdapter
from annotated_types import MinLen
from typing import Annotated, List, Optional
import json, requests

if True:

         def  create_completion(*,  response_model=None,  endpoint="http://localhost:8080/v1/chat/completions",
messages, **kwargs):
        '''
        Creates a chat completion using an OpenAI-compatible endpoint w/ JSON schema support
        (llama.cpp server, llama-cpp-python, Anyscale / Together...)

          The response_model param takes a type (+ supports Pydantic) and behaves just as w/ Instructor (see
below)
        '''
        response_format = None
        type_adapter = None

        if response_model:
            type_adapter = TypeAdapter(response_model)
            schema = type_adapter.json_schema()
            messages = [{
```

```
                "role": "system",
                     "content": f"You respond in JSON format with the following schema: {json.dumps(schema,
indent=2)}"
            }] + messages
            response_format={"type": "json_object", "schema": schema}

        data = requests.post(endpoint, headers={"Content-Type": "application/json"},
                             json=dict(messages=messages, response_format=response_format, **kwargs)).json()
        if 'error' in data:
            raise Exception(data['error']['message'])

        content = data["choices"][0]["message"]["content"]
        return type_adapter.validate_json(content) if type_adapter else content

else:

    # This alternative branch uses Instructor + OpenAI client lib.
    # Instructor support streamed iterable responses, retry & more.
    # (see https://python.useinstructor.com/)
    #! pip install instructor openai
    import instructor, openai
    client = instructor.patch(
        openai.OpenAI(api_key="123", base_url="http://localhost:8080"),
        mode=instructor.Mode.JSON_SCHEMA)
    create_completion = client.chat.completions.create


if __name__ == '__main__':

    class QAPair(BaseModel):
        class Config:
            extra = 'forbid'  # triggers additionalProperties: false in the JSON schema
        question: str
        concise_answer: str
        justification: str
        stars: Annotated[int, Field(ge=1, le=5)]

    class PyramidalSummary(BaseModel):
        class Config:
            extra = 'forbid'  # triggers additionalProperties: false in the JSON schema
        title: str
        summary: str
        question_answers: Annotated[List[QAPair], MinLen(2)]
        sub_sections: Optional[Annotated[List['PyramidalSummary'], MinLen(2)]]

    print("# Summary\n", create_completion(
        model="...",
        response_model=PyramidalSummary,
        messages=[{
            "role": "user",
            "content": f"""
                You are a highly efficient corporate document summarizer.
                Create a pyramidal summary of an imaginary internal document about our company processes
                (starting high-level, going down to each sub sections).
```

```
                Keep questions short, and answers even shorter (trivia / quizz style).
            """
        }]))

==== json_schema_to_grammar.py ====
#!/usr/bin/env python3
from __future__ import annotations

import argparse
import itertools
import json
import re
import sys
from typing import Any, List, Optional, Set, Tuple, Union

def _build_repetition(item_rule, min_items, max_items, separator_rule=None):

    if min_items == 0 and max_items == 1:
        return f'{item_rule}?'

    if not separator_rule:
        if min_items == 1 and max_items is None:
            return f'{item_rule}+'
        elif min_items == 0 and max_items is None:
            return f'{item_rule}*'
        else:
            return f'{item_rule}{{{min_items},{max_items if max_items is not None else ""}}}'

    result = item_rule + ' ' + _build_repetition(f'({separator_rule} {item_rule})', min_items - 1 if min_items
> 0 else 0, max_items - 1 if max_items is not None else None)
    return f'({result})?' if min_items == 0 else result

def _generate_min_max_int(min_value: Optional[int], max_value: Optional[int], out: list, decimals_left: int =
16, top_level: bool = True):
    has_min = min_value != None
    has_max = max_value != None

    def digit_range(from_char: str, to_char: str):
        out.append("[")
        if from_char == to_char:
            out.append(from_char)
        else:
            out.append(from_char)
            out.append("-")
            out.append(to_char)
        out.append("]")

    def more_digits(min_digits: int, max_digits: int):
        out.append("[0-9]")
        if min_digits == max_digits and min_digits == 1:
            return
        out.append("{")
        out.append(str(min_digits))
        if max_digits != min_digits:
```

```
        out.append(",")
        if max_digits != sys.maxsize:
            out.append(str(max_digits))
    out.append("}")

def uniform_range(from_str: str, to_str: str):
    i = 0
    while i < len(from_str) and from_str[i] == to_str[i]:
        i += 1
    if i > 0:
        out.append("\"")
        out.append(from_str[:i])
        out.append("\"")
    if i < len(from_str):
        if i > 0:
            out.append(" ")
        sub_len = len(from_str) - i - 1
        if sub_len > 0:
            from_sub = from_str[i+1:]
            to_sub = to_str[i+1:]
            sub_zeros = "0" * sub_len
            sub_nines = "9" * sub_len

            to_reached = False
            out.append("(")
            if from_sub == sub_zeros:
                digit_range(from_str[i], chr(ord(to_str[i]) - 1))
                out.append(" ")
                more_digits(sub_len, sub_len)
            else:
                out.append("[")
                out.append(from_str[i])
                out.append("] ")
                out.append("(")
                uniform_range(from_sub, sub_nines)
                out.append(")")
                if ord(from_str[i]) < ord(to_str[i]) - 1:
                    out.append(" | ")
                    if to_sub == sub_nines:
                        digit_range(chr(ord(from_str[i]) + 1), to_str[i])
                        to_reached = True
                    else:
                        digit_range(chr(ord(from_str[i]) + 1), chr(ord(to_str[i]) - 1))
                    out.append(" ")
                    more_digits(sub_len, sub_len)
            if not to_reached:
                out.append(" | ")
                digit_range(to_str[i], to_str[i])
                out.append(" ")
                uniform_range(sub_zeros, to_sub)
            out.append(")")
        else:
            out.append("[")
            out.append(from_str[i])
```

```
            out.append("-")
            out.append(to_str[i])
            out.append("]")

    if has_min and has_max:
        if min_value < 0 and max_value < 0:
            out.append("\"-\" (")
            _generate_min_max_int(-max_value, -min_value, out, decimals_left, top_level=True)
            out.append(")")
            return

        if min_value < 0:
            out.append("\"-\" (")
            _generate_min_max_int(0, -min_value, out, decimals_left, top_level=True)
            out.append(") | ")
            min_value = 0

        min_s = str(min_value)
        max_s = str(max_value)
        min_digits = len(min_s)
        max_digits = len(max_s)

        for digits in range(min_digits, max_digits):
            uniform_range(min_s, "9" * digits)
            min_s = "1" + "0" * digits
            out.append(" | ")
        uniform_range(min_s, max_s)
        return

    less_decimals = max(decimals_left - 1, 1)

    if has_min:
        if min_value < 0:
            out.append("\"-\" (")
            _generate_min_max_int(None, -min_value, out, decimals_left, top_level=False)
            out.append(") | [0] | [1-9] ")
            more_digits(0, decimals_left - 1)
        elif min_value == 0:
            if top_level:
                out.append("[0] | [1-9] ")
                more_digits(0, less_decimals)
            else:
                more_digits(1, decimals_left)
        elif min_value <= 9:
            c = str(min_value)
            range_start = '1' if top_level else '0'
            if c > range_start:
                digit_range(range_start, chr(ord(c) - 1))
                out.append(" ")
                more_digits(1, less_decimals)
                out.append(" | ")
            digit_range(c, "9")
            out.append(" ")
            more_digits(0, less_decimals)
```

```
        else:
            min_s = str(min_value)
            length = len(min_s)
            c = min_s[0]

            if c > "1":
                digit_range("1" if top_level else "0", chr(ord(c) - 1))
                out.append(" ")
                more_digits(length, less_decimals)
                out.append(" | ")
            digit_range(c, c)
            out.append(" (")
            _generate_min_max_int(int(min_s[1:]), None, out, less_decimals, top_level=False)
            out.append(")")
            if c < "9":
                out.append(" | ")
                digit_range(chr(ord(c) + 1), "9")
                out.append(" ")
                more_digits(length - 1, less_decimals)
        return

    if has_max:
        if max_value >= 0:
            if top_level:
                out.append("\"-\" [1-9] ")
                more_digits(0, less_decimals)
                out.append(" | ")
            _generate_min_max_int(0, max_value, out, decimals_left, top_level=True)
        else:
            out.append("\"-\" (")
            _generate_min_max_int(-max_value, None, out, decimals_left, top_level=False)
            out.append(")")
        return

    raise RuntimeError("At least one of min_value or max_value must be set")

class BuiltinRule:
    def __init__(self, content: str, deps: list | None = None):
        self.content = content
        self.deps = deps or []

# Constraining spaces to prevent model "running away".
SPACE_RULE = '| " " | "\\n"{1,2} [ \\t]{0,20}'

PRIMITIVE_RULES = {
    'boolean'      : BuiltinRule('("true" | "false") space', []),
    'decimal-part' : BuiltinRule('[0-9]{1,16}', []),
    'integral-part': BuiltinRule('[0] | [1-9] [0-9]{0,15}', []),
    'number'       : BuiltinRule('("-"? integral-part) ("." decimal-part)? ([eE] [-+]? integral-part)? space',
['integral-part', 'decimal-part']),
    'integer'      : BuiltinRule('("-"? integral-part) space', ['integral-part']),
     'value'          : BuiltinRule('object | array | string | number | boolean | null', ['object', 'array',
'string', 'number', 'boolean', 'null']),
    'object'       : BuiltinRule('"{" space ( string ":" space value ("," space string ":" space value)* )? "}"
```

```
space', ['string', 'value']),
    'array'        : BuiltinRule('"[" space ( value ("," space value)* )? "]" space', ['value']),
    'uuid'         : BuiltinRule(r'"\"" [0-9a-fA-F]{8} "-" [0-9a-fA-F]{4} "-" [0-9a-fA-F]{4} "-" [0-9a-fA-F]{4}
"-" [0-9a-fA-F]{12} "\"" space', []),
    'char'         : BuiltinRule(r'[^"\\\x7F\x00-\x1F] | [\\] (["\\bfnrt] | "u" [0-9a-fA-F]{4})', []),
    'string'       : BuiltinRule(r'"\"" char* "\"" space', ['char']),
    'null'         : BuiltinRule('"null" space', []),
}

# TODO: support "uri", "email" string formats
STRING_FORMAT_RULES = {
    'date'            : BuiltinRule('[0-9]{4} "-" ( "0" [1-9] | "1" [0-2] ) "-" ( \"0\" [1-9] | [1-2] [0-9] |
"3" [0-1] )', []),
    'time'            : BuiltinRule('([01] [0-9] | "2" [0-3]) ":" [0-5] [0-9] ":" [0-5] [0-9] ( "." [0-9]{3} )?
( "Z" | ( "+" | "-" ) ( [01] [0-9] | "2" [0-3] ) ":" [0-5] [0-9] )', []),
    'date-time'       : BuiltinRule('date "T" time', ['date', 'time']),
    'date-string'     : BuiltinRule('"\\"" date "\\"" space', ['date']),
    'time-string'     : BuiltinRule('"\\"" time "\\"" space', ['time']),
    'date-time-string': BuiltinRule('"\\"" date-time "\\"" space', ['date-time']),
}

DOTALL = '[\\U00000000-\\U0010FFFF]'
DOT = '[^\\x0A\\x0D]'

RESERVED_NAMES = set(["root", "dot", *PRIMITIVE_RULES.keys(), *STRING_FORMAT_RULES.keys()])

INVALID_RULE_CHARS_RE = re.compile(r'[^a-zA-Z0-9-]+')
GRAMMAR_LITERAL_ESCAPE_RE = re.compile(r'[\r\n"]')
GRAMMAR_RANGE_LITERAL_ESCAPE_RE = re.compile(r'[\r\n"\]\-\\]')
GRAMMAR_LITERAL_ESCAPES = {'\r': '\\r', '\n': '\\n', '"': '\\"', '-': '\\-', ']': '\\]'}

NON_LITERAL_SET = set('|.()[]{}*+?')
ESCAPED_IN_REGEXPS_BUT_NOT_IN_LITERALS = set('^$.[]()|{}*+?')



class SchemaConverter:
    def __init__(self, *, prop_order, allow_fetch, dotall, raw_pattern):
        self._prop_order = prop_order
        self._allow_fetch = allow_fetch
        self._dotall = dotall
        self._raw_pattern = raw_pattern
        self._rules = {
            'space': SPACE_RULE,
        }
        self._refs = {}
        self._refs_being_resolved = set()

    def _format_literal(self, literal):
        escaped = GRAMMAR_LITERAL_ESCAPE_RE.sub(
            lambda m: GRAMMAR_LITERAL_ESCAPES.get(m.group(0)) or m.group(0), literal
        )
        return f'"{escaped}"'

    def not_literal(self, literal: str, dotall: bool = True, maybe_escaped_underscores = False) -> str:
```

```
    '''
        not_literal('a') -> '[^a]'
        not_literal('abc') -> '([^a] | "a" ([^b] | "b" ([^c])?)?)?'
    '''
    assert len(literal) > 0, 'Empty literal not supported'
    def recurse(i: int):
        c = literal[i]
        if maybe_escaped_underscores and c == '_':
            yield f'[^{c}\\\\]'
            yield ' | '
            yield f'"\\\\"? "{c}"'
        else:
            yield f'[^{c}]'
        if i < len(literal) - 1:
            yield ' | '
            yield self._format_literal(c)
            yield ' ('
            yield from recurse(i + 1)
            yield ')?'

    return ''.join(('(', *recurse(0), ')'))

def _not_strings(self, strings):
    class TrieNode:
        def __init__(self):
            self.children = {}
            self.is_end_of_string = False

        def insert(self, string):
            node = self
            for c in string:
                node = node.children.setdefault(c, TrieNode())
            node.is_end_of_string = True

    trie = TrieNode()
    for s in strings:
        trie.insert(s)

    char_rule = self._add_primitive('char', PRIMITIVE_RULES['char'])
    out = ['["] ( ']

    def visit(node):
        rejects = []
        first = True
        for c in sorted(node.children.keys()):
            child = node.children[c]
            rejects.append(c)
            if first:
                first = False
            else:
                out.append(' | ')
            out.append(f'[{c}]')
            if child.children:
                out.append(f' (')
```

```
                    visit(child)
                    out.append(')')
                elif child.is_end_of_string:
                    out.append(f' {char_rule}+')
            if node.children:
                if not first:
                    out.append(' | ')
                out.append(f'[^"{"".join(rejects)}] {char_rule}*')
        visit(trie)

        out.append(f' ){"" if trie.is_end_of_string else "?"} ["] space')
        return ''.join(out)

    def _add_rule(self, name, rule):
        esc_name = INVALID_RULE_CHARS_RE.sub('-', name)
        if esc_name not in self._rules or self._rules[esc_name] == rule:
            key = esc_name
        else:
            i = 0
            while f'{esc_name}{i}' in self._rules and self._rules[f'{esc_name}{i}'] != rule:
                i += 1
            key = f'{esc_name}{i}'
        self._rules[key] = rule
        return key

    def resolve_refs(self, schema: dict, url: str):
        '''
            Resolves all $ref fields in the given schema, fetching any remote schemas,
            replacing $ref with absolute reference URL and populating self._refs with the
            respective referenced (sub)schema dictionaries.
        '''
        def visit(n: dict):
            if isinstance(n, list):
                return [visit(x) for x in n]
            elif isinstance(n, dict):
                ref = n.get('$ref')
                if ref is not None and ref not in self._refs:
                    if ref.startswith('https://'):
                          assert self._allow_fetch, 'Fetching remote schemas is not allowed (use --allow-fetch
for force)'
                        import requests

                        frag_split = ref.split('#')
                        base_url = frag_split[0]

                        target = self._refs.get(base_url)
                        if target is None:
                            target = self.resolve_refs(requests.get(ref).json(), base_url)
                            self._refs[base_url] = target

                        if len(frag_split) == 1 or frag_split[-1] == '':
                            return target
                    elif ref.startswith('#/'):
                        target = schema
```

```
                        ref = f'{url}{ref}'
                        n['$ref'] = ref
                    else:
                        raise ValueError(f'Unsupported ref {ref}')

                    for sel in ref.split('#')[-1].split('/')[1:]:
                         assert target is not None and sel in target, f'Error resolving ref {ref}: {sel} not in
{target}'
                        target = target[sel]

                    self._refs[ref] = target
                else:
                    for v in n.values():
                        visit(v)

            return n
        return visit(schema)

    def _generate_union_rule(self, name, alt_schemas):
        return ' | '.join((
            self.visit(alt_schema, f'{name}{"-" if name else "alternative-"}{i}')
            for i, alt_schema in enumerate(alt_schemas)
        ))

    def _visit_pattern(self, pattern, name):
        '''
            Transforms a regular expression pattern into a GBNF rule.

            Input: https://json-schema.org/understanding-json-schema/reference/regular_expressions
            Output: https://github.com/ggerganov/llama.cpp/blob/master/grammars/README.md

            Unsupported features: negative/positive lookaheads, greedy/non-greedy modifiers.

            Mostly a 1:1 translation, except for {x} / {x,} / {x,y} quantifiers for which
            we define sub-rules to keep the output lean.
        '''

         assert pattern.startswith('^') and pattern.endswith('$'), 'Pattern must start with "^" and end with
"$"'
        pattern = pattern[1:-1]
        sub_rule_ids = {}

        i = 0
        length = len(pattern)

        def to_rule(s: tuple[str, bool]) -> str:
            (txt, is_literal) = s
            return "\"" + txt + "\"" if is_literal else txt

        def transform() -> tuple[str, bool]:
            '''
                 Parse a unit at index i (advancing it), and return its string representation + whether it's a
literal.
            '''
```

```
            nonlocal i
            nonlocal pattern
            nonlocal sub_rule_ids

            start = i
            # For each component of this sequence, store its string representation and whether it's a literal.
            # We only need a flat structure here to apply repetition operators to the last item, and
            # to merge literals at the and (we're parsing grouped ( sequences ) recursively and don't treat '|'
specially
            # (GBNF's syntax is luckily very close to regular expressions!)
            seq: list[tuple[str, bool]] = []

            def get_dot():
                if self._dotall:
                    rule = DOTALL
                else:
                    # Accept any character... except \n and \r line break chars (\x0A and \xOD)
                    rule = DOT
                return self._add_rule(f'dot', rule)

            def join_seq():
                nonlocal seq
                ret = []
                for is_literal, g in itertools.groupby(seq, lambda x: x[1]):
                    if is_literal:
                        ret.append((''.join(x[0] for x in g), True))
                    else:
                        ret.extend(g)
                if len(ret) == 1:
                    return ret[0]
                return (' '.join(to_rule(x) for x in seq), False)

            while i < length:
                c = pattern[i]
                if c == '.':
                    seq.append((get_dot(), False))
                    i += 1
                elif c == '(':
                    i += 1
                    if i < length:
                         assert pattern[i] != '?', f'Unsupported pattern syntax "{pattern[i]}" at index {i} of
/{pattern}/'
                    seq.append((f'({to_rule(transform())})', False))
                elif c == ')':
                    i += 1
                     assert start > 0 and pattern[start-1] == '(', f'Unbalanced parentheses; start = {start}, i
= {i}, pattern = {pattern}'
                    return join_seq()
                elif c == '[':
                    square_brackets = c
                    i += 1
                    while i < length and pattern[i] != ']':
                        if pattern[i] == '\\':
                            square_brackets += pattern[i:i+2]
```

```
                        i += 2
                    else:
                        square_brackets += pattern[i]
                        i += 1
                    assert i < length, f'Unbalanced square brackets; start = {start}, i = {i}, pattern =
{pattern}'
                square_brackets += ']'
                i += 1
                seq.append((square_brackets, False))
            elif c == '|':
                seq.append(('|', False))
                i += 1
            elif c in ('*', '+', '?'):
                seq[-1] = (to_rule(seq[-1]) + c, False)
                i += 1
            elif c == '{':
                curly_brackets = c
                i += 1
                while i < length and pattern[i] != '}':
                    curly_brackets += pattern[i]
                    i += 1
                     assert i < length, f'Unbalanced curly brackets; start = {start}, i = {i}, pattern =
{pattern}'
                curly_brackets += '}'
                i += 1
                nums = [s.strip() for s in curly_brackets[1:-1].split(',')]
                min_times = 0
                max_times = None
                try:
                    if len(nums) == 1:
                        min_times = int(nums[0])
                        max_times = min_times
                    else:
                        assert len(nums) == 2
                        min_times = int(nums[0]) if nums[0] else 0
                        max_times = int(nums[1]) if nums[1] else None
                except ValueError:
                    raise ValueError(f'Invalid quantifier {curly_brackets} in /{pattern}/')

                (sub, sub_is_literal) = seq[-1]

                if not sub_is_literal:
                    id = sub_rule_ids.get(sub)
                    if id is None:
                        id = self._add_rule(f'{name}-{len(sub_rule_ids) + 1}', sub)
                        sub_rule_ids[sub] = id
                    sub = id

                 seq[-1] = (_build_repetition(f'"{sub}"' if sub_is_literal else sub, min_times, max_times),
False)
            else:
                literal = ''
                while i < length:
                    if pattern[i] == '\\' and i < length - 1:
```

```
                            next = pattern[i + 1]
                            if next in ESCAPED_IN_REGEXPS_BUT_NOT_IN_LITERALS:
                                i += 1
                                literal += pattern[i]
                                i += 1
                            else:
                                literal += pattern[i:i+2]
                                i += 2
                        elif pattern[i] == '"' and not self._raw_pattern:
                            literal += '\\"'
                            i += 1
                        elif pattern[i] not in NON_LITERAL_SET and \
                                (i == length - 1 or literal == '' or pattern[i+1] == '.' or pattern[i+1] not in
NON_LITERAL_SET):
                            literal += pattern[i]
                            i += 1
                        else:
                            break
                    if literal:
                        seq.append((literal, True))

            return join_seq()

        return self._add_rule(
            name,
            to_rule(transform()) if self._raw_pattern \
                else "\"\\\"\" (" + to_rule(transform()) + ") \"\\\"\" space")


    def _resolve_ref(self, ref):
        ref_name = ref.split('/')[-1]
        if ref_name not in self._rules and ref not in self._refs_being_resolved:
            self._refs_being_resolved.add(ref)
            resolved = self._refs[ref]
            ref_name = self.visit(resolved, ref_name)
            self._refs_being_resolved.remove(ref)
        return ref_name

    def _generate_constant_rule(self, value):
        return self._format_literal(json.dumps(value))

    def visit(self, schema, name):
        schema_type = schema.get('type')
        schema_format = schema.get('format')
        rule_name = name + '-' if name in RESERVED_NAMES else name or 'root'

        if (ref := schema.get('$ref')) is not None:
            return self._add_rule(rule_name, self._resolve_ref(ref))

        elif 'oneOf' in schema or 'anyOf' in schema:
                    return  self._add_rule(rule_name,  self._generate_union_rule(name,  schema.get('oneOf')  or
schema['anyOf']))

        elif isinstance(schema_type, list):
```

```
            return self._add_rule(rule_name, self._generate_union_rule(name, [{**schema, 'type': t} for t in schema_type]))

        elif 'const' in schema:
            return self._add_rule(rule_name, self._generate_constant_rule(schema['const']) + ' space')

        elif 'enum' in schema:
            rule = '(' + ' | '.join((self._generate_constant_rule(v) for v in schema['enum'])) + ') space'
            return self._add_rule(rule_name, rule)

        elif schema_type in (None, 'object') and \
             ('properties' in schema or \
              ('additionalProperties' in schema and schema['additionalProperties'] is not True)):
            required = set(schema.get('required', []))
            properties = list(schema.get('properties', {}).items())
                        return self._add_rule(rule_name, self._build_object_rule(properties, required, name, schema.get('additionalProperties')))

        elif schema_type in (None, 'object') and 'allOf' in schema:
            required = set()
            properties = []
            hybrid_name = name
            def add_component(comp_schema, is_required):
                if (ref := comp_schema.get('$ref')) is not None:
                    comp_schema = self._refs[ref]

                if 'properties' in comp_schema:
                    for prop_name, prop_schema in comp_schema['properties'].items():
                        properties.append((prop_name, prop_schema))
                        if is_required:
                            required.add(prop_name)

            for t in schema['allOf']:
                if 'anyOf' in t:
                    for tt in t['anyOf']:
                        add_component(tt, is_required=False)
                else:
                    add_component(t, is_required=True)

                    return self._add_rule(rule_name, self._build_object_rule(properties, required, hybrid_name, additional_properties=None))

        elif schema_type in (None, 'array') and ('items' in schema or 'prefixItems' in schema):
            items = schema.get('items') or schema['prefixItems']
            if isinstance(items, list):
                return self._add_rule(
                    rule_name,
                    '"[" space ' +
                    ' "," space '.join(
                        self.visit(item, f'{name}{"-" if name else ""}tuple-{i}')
                        for i, item in enumerate(items)) +
                    ' "]" space')
            else:
                item_rule_name = self.visit(items, f'{name}{"-" if name else ""}item')
```

```
                min_items = schema.get("minItems", 0)
                max_items = schema.get("maxItems")
                  return self._add_rule(rule_name, '"[" space ' + _build_repetition(item_rule_name, min_items,
max_items, separator_rule='"," space') + ' "]" space')

        elif schema_type in (None, 'string') and 'pattern' in schema:
            return self._visit_pattern(schema['pattern'], rule_name)

        elif schema_type in (None, 'string') and re.match(r'^uuid[1-5]?$', schema_format or ''):
            return self._add_primitive(
                'root' if rule_name == 'root' else schema_format,
                PRIMITIVE_RULES['uuid']
            )

        elif schema_type in (None, 'string') and f'{schema_format}-string' in STRING_FORMAT_RULES:
            prim_name = f'{schema_format}-string'
            return self._add_rule(rule_name, self._add_primitive(prim_name, STRING_FORMAT_RULES[prim_name]))

        elif schema_type == 'string' and ('minLength' in schema or 'maxLength' in schema):
            char_rule = self._add_primitive('char', PRIMITIVE_RULES['char'])
            min_len = schema.get('minLength', 0)
            max_len = schema.get('maxLength')

              return self._add_rule(rule_name, r'"\"" ' + _build_repetition(char_rule, min_len, max_len) + r'
"\"" space')

        elif schema_type in (None, 'integer') and \
                        ('minimum' in schema or 'exclusiveMinimum' in schema or 'maximum' in schema or
'exclusiveMaximum' in schema):
            min_value = None
            max_value = None
            if 'minimum' in schema:
                min_value = schema['minimum']
            elif 'exclusiveMinimum' in schema:
                min_value = schema['exclusiveMinimum'] + 1
            if 'maximum' in schema:
                max_value = schema['maximum']
            elif 'exclusiveMaximum' in schema:
                max_value = schema['exclusiveMaximum'] - 1

            out = ["("]
            _generate_min_max_int(min_value, max_value, out)
            out.append(") space")
            return self._add_rule(rule_name, ''.join(out))

        elif (schema_type == 'object') or (len(schema) == 0):
            return self._add_rule(rule_name, self._add_primitive('object', PRIMITIVE_RULES['object']))

        else:
            assert schema_type in PRIMITIVE_RULES, f'Unrecognized schema: {schema}'
            # TODO: support minimum, maximum, exclusiveMinimum, exclusiveMaximum at least for zero
                        return  self._add_primitive('root'  if  rule_name  ==  'root'  else  schema_type,
PRIMITIVE_RULES[schema_type])
```

```
    def _add_primitive(self, name: str, rule: BuiltinRule):
        n = self._add_rule(name, rule.content)

        for dep in rule.deps:
            dep_rule = PRIMITIVE_RULES.get(dep) or STRING_FORMAT_RULES.get(dep)
            assert dep_rule, f'Rule {dep} not known'
            if dep not in self._rules:
                self._add_primitive(dep, dep_rule)
        return n

        def _build_object_rule(self, properties: List[Tuple[str, Any]], required: Set[str], name: str,
additional_properties: Optional[Union[bool, Any]]):
        prop_order = self._prop_order
        # sort by position in prop_order (if specified) then by original order
                    sorted_props = [kv[0] for _, kv in sorted(enumerate(properties), key=lambda ikv:
(prop_order.get(ikv[1][0], len(prop_order)), ikv[0]))]

        prop_kv_rule_names = {}
        for prop_name, prop_schema in properties:
            prop_rule_name = self.visit(prop_schema, f'{name}{"-" if name else ""}{prop_name}')
            prop_kv_rule_names[prop_name] = self._add_rule(
                f'{name}{"-" if name else ""}{prop_name}-kv',
                fr'{self._format_literal(json.dumps(prop_name))} space ":" space {prop_rule_name}'
            )
        required_props = [k for k in sorted_props if k in required]
        optional_props = [k for k in sorted_props if k not in required]

        if additional_properties is not None and additional_properties != False:
            sub_name = f'{name}{"-" if name else ""}additional'
                                value_rule = self.visit(additional_properties, f'{sub_name}-value') if
isinstance(additional_properties, dict) else \
                self._add_primitive('value', PRIMITIVE_RULES['value'])
            key_rule = self._add_primitive('string', PRIMITIVE_RULES['string']) if not sorted_props \
                else self._add_rule(f'{sub_name}-k', self._not_strings(sorted_props))

            prop_kv_rule_names["*"] = self._add_rule(
                f'{sub_name}-kv',
                f'{key_rule} ":" space {value_rule}'
            )
            optional_props.append("*")

        rule = '"{" space '
        rule += ' "," space '.join(prop_kv_rule_names[k] for k in required_props)

        if optional_props:
            rule += ' ('
            if required_props:
                rule += ' "," space ( '

            def get_recursive_refs(ks, first_is_optional):
                [k, *rest] = ks
                kv_rule_name = prop_kv_rule_names[k]
                comma_ref = f'( "," space {kv_rule_name} )'
                if first_is_optional:
```

```
                    res = comma_ref + ('*' if k == '*' else '?')
                else:
                    res = kv_rule_name + (' ' + comma_ref + "*" if k == '*' else '')
                if len(rest) > 0:
                    res += ' ' + self._add_rule(
                        f'{name}{"-" if name else ""}{k}-rest',
                        get_recursive_refs(rest, first_is_optional=True)
                    )
                return res

            rule += ' | '.join(
                get_recursive_refs(optional_props[i:], first_is_optional=False)
                for i in range(len(optional_props))
            )
            if required_props:
                rule += ' )'
            rule += ' )?'

        rule += ' "}" space'

        return rule

    def format_grammar(self):
        return '\n'.join(
            f'{name} ::= {rule}'
            for name, rule in sorted(self._rules.items(), key=lambda kv: kv[0])
        )


def main(args_in = None):
    parser = argparse.ArgumentParser(
        description='''
            Generates a grammar (suitable for use in ./llama-cli) that produces JSON conforming to a
            given JSON schema. Only a subset of JSON schema features are supported; more may be
            added in the future.
        ''',
    )
    parser.add_argument(
        '--prop-order',
        default=[],
        type=lambda s: s.split(','),
        help='''
            comma-separated property names defining the order of precedence for object properties;
            properties not specified here are given lower precedence than those that are, and
            are kept in their original order from the schema. Required properties are always
            given precedence over optional properties.
        '''
    )
    parser.add_argument(
        '--allow-fetch',
        action='store_true',
        default=False,
        help='Whether to allow fetching referenced schemas over HTTPS')
    parser.add_argument(
```

```
        '--dotall',
        action='store_true',
        default=False,
          help='Whether to treat dot (".") as matching all chars including line breaks in regular expression
patterns')
    parser.add_argument(
        '--raw-pattern',
        action='store_true',
        default=False,
        help='Treats string patterns as raw patterns w/o quotes (or quote escapes)')

    parser.add_argument('schema', help='file containing JSON schema ("-" for stdin)')
    args = parser.parse_args(args_in)

    if args.schema.startswith('https://'):
        url = args.schema
        import requests
        schema = requests.get(url).json()
    elif args.schema == '-':
        url = 'stdin'
        schema = json.load(sys.stdin)
    else:
        url = f'file://{args.schema}'
        with open(args.schema) as f:
            schema = json.load(f)
    converter = SchemaConverter(
        prop_order={name: idx for idx, name in enumerate(args.prop_order)},
        allow_fetch=args.allow_fetch,
        dotall=args.dotall,
        raw_pattern=args.raw_pattern)
    schema = converter.resolve_refs(schema, url)
    converter.visit(schema, '')
    print(converter.format_grammar())


if __name__ == '__main__':
    main()

==== lazy.py ====
from __future__ import annotations
from abc import ABC, ABCMeta, abstractmethod

import logging
from typing import Any, Callable

import numpy as np
from numpy.typing import DTypeLike


logger = logging.getLogger(__name__)


class LazyMeta(ABCMeta):
```

```
def __new__(cls, name: str, bases: tuple[type, ...], namespace: dict[str, Any], **kwargs):
    def __getattr__(self, name: str) -> Any:
        meta_attr = getattr(self._meta, name)
        if callable(meta_attr):
            return type(self)._wrap_fn(
                (lambda s, *args, **kwargs: getattr(s, name)(*args, **kwargs)),
                use_self=self,
            )
        elif isinstance(meta_attr, self._tensor_type):
            # e.g. self.T with torch.Tensor should still be wrapped
            return type(self)._wrap_fn(lambda s: getattr(s, name))(self)
        else:
            # no need to wrap non-tensor properties,
            # and they likely don't depend on the actual contents of the tensor
            return meta_attr

    namespace["__getattr__"] = __getattr__

    # need to make a builder for the wrapped wrapper to copy the name,
    # or else it fails with very cryptic error messages,
    # because somehow the same string would end up in every closures
    def mk_wrap(op_name: str, *, meta_noop: bool = False):
        # need to wrap the wrapper to get self
        def wrapped_special_op(self, *args, **kwargs):
            return type(self)._wrap_fn(
                getattr(type(self)._tensor_type, op_name),
                meta_noop=meta_noop,
            )(self, *args, **kwargs)
        return wrapped_special_op

    # special methods bypass __getattr__, so they need to be added manually
    # ref: https://docs.python.org/3/reference/datamodel.html#special-lookup
    # NOTE: doing this from a metaclass is very convenient
    # TODO: make this even more comprehensive
    for binary_op in (
        "lt", "le", "eq", "ne", "ge", "gt", "not"
        "abs", "add", "and", "floordiv", "invert", "lshift", "mod", "mul", "matmul",
        "neg", "or", "pos", "pow", "rshift", "sub", "truediv", "xor",
        "iadd", "iand", "ifloordiv", "ilshift", "imod", "imul", "ior", "irshift", "isub", "ixor",
        "radd", "rand", "rfloordiv", "rmul", "ror", "rpow", "rsub", "rtruediv", "rxor",
    ):
        attr_name = f"__{binary_op}__"
        # the result of these operators usually has the same shape and dtype as the input,
        # so evaluation on the meta tensor can be skipped.
        namespace[attr_name] = mk_wrap(attr_name, meta_noop=True)

    for special_op in (
        "getitem", "setitem", "len",
    ):
        attr_name = f"__{special_op}__"
        namespace[attr_name] = mk_wrap(attr_name, meta_noop=False)

    return super().__new__(cls, name, bases, namespace, **kwargs)
```

```
# Tree of lazy tensors
class LazyBase(ABC, metaclass=LazyMeta):
    _tensor_type: type
    _meta: Any
    _data: Any | None
    _args: tuple
    _kwargs: dict[str, Any]
    _func: Callable[[Any], Any] | None

    def __init__(self, *, meta: Any, data: Any | None = None, args: tuple = (), kwargs: dict[str, Any] | None =
None, func: Callable[[Any], Any] | None = None):
        super().__init__()
        self._meta = meta
        self._data = data
        self._args = args
        self._kwargs = kwargs if kwargs is not None else {}
        self._func = func
        assert self._func is not None or self._data is not None

    def __init_subclass__(cls) -> None:
        if "_tensor_type" not in cls.__dict__:
            raise TypeError(f"property '_tensor_type' must be defined for {cls!r}")
        return super().__init_subclass__()

    @staticmethod
    def _recurse_apply(o: Any, fn: Callable[[Any], Any]) -> Any:
        # TODO: dict and set
        if isinstance(o, (list, tuple)):
            L = []
            for item in o:
                L.append(LazyBase._recurse_apply(item, fn))
            if isinstance(o, tuple):
                L = tuple(L)
            return L
        elif isinstance(o, LazyBase):
            return fn(o)
        else:
            return o

    @classmethod
      def _wrap_fn(cls, fn: Callable, *, use_self: LazyBase | None = None, meta_noop: bool | DTypeLike |
tuple[DTypeLike, Callable[[tuple[int, ...]], tuple[int, ...]]] = False) -> Callable[[Any], Any]:
        def wrapped_fn(*args, **kwargs):
            if kwargs is None:
                kwargs = {}
            args = ((use_self,) if use_self is not None else ()) + args

            meta_args = LazyBase._recurse_apply(args, lambda t: t._meta)
            # TODO: maybe handle tensors in kwargs too

            if isinstance(meta_noop, bool) and not meta_noop:
                try:
                    res = fn(*meta_args, **kwargs)
```

```
                except NotImplementedError:
                    # running some operations on PyTorch's Meta tensors can cause this exception
                    res = None
            else:
                # some operators don't need to actually run on the meta tensors
                assert len(args) > 0
                res = args[0]
                assert isinstance(res, cls)
                res = res._meta
                # allow operations to override the dtype and shape
                if meta_noop is not True:
                    if isinstance(meta_noop, tuple):
                        dtype, shape = meta_noop
                        assert callable(shape)
                        res = cls.meta_with_dtype_and_shape(dtype, shape(res.shape))
                    else:
                        res = cls.meta_with_dtype_and_shape(meta_noop, res.shape)

            if isinstance(res, cls._tensor_type):
                return cls(meta=cls.eager_to_meta(res), args=args, kwargs=kwargs, func=fn)
            elif isinstance(res, tuple) and all(isinstance(t, cls._tensor_type) for t in res):
                # share the evaluation between lazy tuple elements
                shared_args: list = [args, None]

                def eager_tuple_element(a: list[Any], i: int = 0, /, **kw) -> LazyBase:
                    assert len(a) == 2
                    if a[1] is None:
                        a[1] = fn(*a[0], **kw)
                    return a[1][i]
                return tuple(cls(meta=cls.eager_to_meta(res[i]), args=(shared_args, i), kwargs=kwargs,
func=eager_tuple_element) for i in range(len(res)))
            else:
                del res  # not needed
                # non-tensor return likely relies on the contents of the args
                # (e.g. the result of torch.equal)
                eager_args = cls.to_eager(args)
                return fn(*eager_args, **kwargs)
        return wrapped_fn

    @classmethod
    def to_eager(cls, t: Any) -> Any:
        def simple_to_eager(_t: LazyBase) -> Any:
            if _t._data is not None:
                return _t._data

            # NOTE: there's a recursion limit in Python (usually 1000)

            assert _t._func is not None
            _t._args = cls._recurse_apply(_t._args, simple_to_eager)
            _t._data = _t._func(*_t._args, **_t._kwargs)
            # sanity check
            assert _t._data is not None
            assert _t._data.dtype == _t._meta.dtype
            assert _t._data.shape == _t._meta.shape
```

```
            return _t._data

        # recurse into lists and/or tuples, keeping their structure
        return cls._recurse_apply(t, simple_to_eager)

    @classmethod
    def eager_to_meta(cls, t: Any) -> Any:
        return cls.meta_with_dtype_and_shape(t.dtype, t.shape)

    # must be overridden, meta tensor init is backend-specific
    @classmethod
    @abstractmethod
    def meta_with_dtype_and_shape(cls, dtype: Any, shape: Any) -> Any: pass

    @classmethod
    def from_eager(cls, t: Any) -> Any:
        if type(t) is cls:
            # already lazy
            return t
        elif isinstance(t, cls._tensor_type):
            return cls(meta=cls.eager_to_meta(t), data=t)
        else:
            return TypeError(f"{type(t)!r} is not compatible with {cls._tensor_type!r}")


class LazyNumpyTensor(LazyBase):
    _tensor_type = np.ndarray

    shape: tuple[int, ...]  # Makes the type checker happy in quants.py

    @classmethod
    def meta_with_dtype_and_shape(cls, dtype: DTypeLike, shape: tuple[int, ...]) -> np.ndarray[Any, Any]:
        # The initial idea was to use np.nan as the fill value,
        # but non-float types like np.int16 can't use that.
        # So zero it is.
        cheat = np.zeros(1, dtype)
        return np.lib.stride_tricks.as_strided(cheat, shape, (0 for _ in shape))

    def astype(self, dtype, *args, **kwargs):
        meta = type(self).meta_with_dtype_and_shape(dtype, self._meta.shape)
        full_args = (self, dtype,) + args
              return type(self)(meta=meta, args=full_args, kwargs=kwargs, func=(lambda a, *args, **kwargs:
a.astype(*args, **kwargs)))

    def tofile(self, *args, **kwargs):
        eager = LazyNumpyTensor.to_eager(self)
        return eager.tofile(*args, **kwargs)

    # TODO: __array_function__

==== llava_surgery.py ====
import argparse
import glob
```

```
import os
import torch


ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", help="Path to LLaVA v1.5 model")
args = ap.parse_args()

# find the model part that includes the the multimodal projector weights
path = sorted(glob.glob(f"{args.model}/pytorch_model*.bin"))[-1]
checkpoint = torch.load(path)

# get a list of mm tensor names
mm_tensors = [k for k, v in checkpoint.items() if k.startswith("model.mm_projector")]

# store these tensors in a new dictionary and torch.save them
projector = {name: checkpoint[name].float() for name in mm_tensors}
torch.save(projector, f"{args.model}/llava.projector")

# BakLLaVA models contain CLIP tensors in it
clip_tensors = [k for k, v in checkpoint.items() if k.startswith("model.vision_tower")]
if len(clip_tensors) > 0:
    clip = {name.replace("vision_tower.vision_tower.", ""): checkpoint[name].float() for name in clip_tensors}
    torch.save(clip, f"{args.model}/llava.clip")



    # added tokens should be removed to be able to convert Mistral models
    if os.path.exists(f"{args.model}/added_tokens.json"):
        with open(f"{args.model}/added_tokens.json", "w") as f:
            f.write("{}\n")



print("Done!")
print(f"Now you can convert {args.model} to a regular LLaMA GGUF file.")
print(f"Also, use {args.model}/llava.projector to prepare a llava-encoder.gguf file.")

==== llava_surgery_v2.py ====
import argparse
import glob
import os
import torch
from safetensors import safe_open
from safetensors.torch import save_file
from typing import Any, ContextManager, cast

# Function to determine if file is a SafeTensor file
def is_safetensor_file(file_path):
    return file_path.endswith('.safetensors')


# Unified loading function
def load_model(file_path):
    if is_safetensor_file(file_path):
```

```
        tensors = {}
        with cast(ContextManager[Any], safe_open(file_path, framework="pt", device="cpu")) as f:
            for key in f.keys():
                tensors[key] = f.get_tensor(key).clone()
                # output shape
                print(f"{key} : {tensors[key].shape}")
        return tensors, 'safetensor'
    else:
        return torch.load(file_path, map_location=torch.device('cpu')), 'pytorch'


# Unified saving function
def save_model(model, file_path, file_type):
    if file_type == 'safetensor':
        # safe_save(model, file_path)
        save_file(model, file_path)
    else:
        torch.save(model, file_path)

# Helpers to match weight names from specific components or
# determine if a saved shard contains that component
def is_vision_tower(weight_name):
    return (
        weight_name.startswith("model.vision_tower") or
        weight_name.startswith("vit.") or
        weight_name.startswith("vision_tower")
    )

def is_newline(weight_name):
    return (
        weight_name.startswith("model.image_newline") or
        weight_name.startswith("image_newline")
    )

def is_mm_projector(weight_name):
    return (
        weight_name.startswith("model.mm_projector") or
        weight_name.startswith("vision_proj.") or
        weight_name.startswith("multi_modal_projector")
    )

def newline_criteria(checkpoint):
    return any(is_newline(k) for k in checkpoint.keys())

def proj_criteria(checkpoint):
    return any(is_mm_projector(k) for k in checkpoint.keys())

# Adapted function to clean vision tower from checkpoint
def clean_vision_tower_from_checkpoint(checkpoint_path):
    checkpoint, file_type = load_model(checkpoint_path)
    # file_type = 'pytorch'
    model_path = os.path.dirname(checkpoint_path)
    print(f"Searching for vision tower tensors in {checkpoint_path}")
    clip_tensors = [k for k, v in checkpoint.items() if is_vision_tower(k)]
```

```
    if len(clip_tensors) > 0:
        print(f"Found {len(clip_tensors)} tensors to extract from {checkpoint_path}")
        # Adapted for file type
        clip_path = os.path.join(model_path, "llava.clip")

        if os.path.exists(clip_path):
            print(f"Loading existing llava.clip from {clip_path}")
            existing_clip, _ = load_model(clip_path)
        else:
            print(f"Creating new llava.clip at {clip_path}")
            existing_clip = {}
        # Update existing_clip with new tensors, avoid duplicates
        for name in clip_tensors:
            simple_name = name[name.index('vision_model.'):] if 'vision_model.' in name else name
            print(f"Adding {simple_name} to llava.clip")
            if simple_name not in existing_clip:
                existing_clip[simple_name] = checkpoint[name]

        # Save the updated clip tensors back to llava.clip
        save_model(existing_clip, clip_path, 'pytorch')

        # Remove the tensors from the original checkpoint
        for name in clip_tensors:
            del checkpoint[name]

        checkpoint_path = checkpoint_path
        return True
    return False

def find_relevant_checkpoints(checkpoint_paths, newline_criteria, projector):
    newline_checkpoint_path = None
    projector_checkpoint_path = None

    for path in checkpoint_paths:
        checkpoint, _ = load_model(path)
        if newline_criteria(checkpoint) and newline_checkpoint_path is None:
            newline_checkpoint_path = path
        if projector(checkpoint):
            projector_checkpoint_path = path

    return newline_checkpoint_path, projector_checkpoint_path


# Command-line interface setup
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True, help="Path to LLaVA v1.5+ model")
ap.add_argument("-C", "--clean-vision-tower", action="store_true", help="Remove any vision tower from the model
files")
args = ap.parse_args()

if args.clean_vision_tower:
    # Generalized to handle both PyTorch and SafeTensors models
    model_files = sorted(glob.glob(f"{args.model}/*"), key=os.path.getmtime, reverse=True)
```

```
            # checkpoint_paths = [path for path in model_files if (path.endswith('.bin') and
path.startswith('pytorch')) or (path.endswith('.safetensors') and path.startswith('model'))]
        checkpoint_paths = [path for path in model_files if (path.endswith('.bin') and 'pytorch' in
path.split('/')[-1].split('\\')[-1]) or (path.endswith('.safetensors') and 'model' in
path.split('/')[-1].split('\\')[-1])]
    for projector_checkpoint_path in checkpoint_paths:
        print(f"Cleaning {projector_checkpoint_path}")
        if not clean_vision_tower_from_checkpoint(projector_checkpoint_path):
            print(f"No vision tower found in {projector_checkpoint_path}")
            # we break once none is found, so far all models append them at the end
            # break
    print("Done! All vision tower tensors are removed from the model files and stored in llava.clip file.")

# Now we look for the projector in the last checkpoint
model_files = sorted(glob.glob(f"{args.model}/*"), key=os.path.getmtime, reverse=True)
checkpoint_paths = [path for path in model_files if (path.endswith('.bin') and 'pytorch' in
path.split('/')[-1].split('\\')[-1]) or (path.endswith('.safetensors') and 'model' in
path.split('/')[-1].split('\\')[-1])]
# last_checkpoint_path = checkpoint_paths[0]
# first_checkpoint_path = checkpoint_paths[-1]
newline_checkpoint_path, projector_checkpoint_path = find_relevant_checkpoints(checkpoint_paths,
newline_criteria, proj_criteria)

print(f"Taking projector from {projector_checkpoint_path}")
first_mm_tensors = []
first_checkpoint = None
if newline_checkpoint_path is not None:
    print(f"Taking newline from {newline_checkpoint_path}")
    first_checkpoint, file_type = load_model(newline_checkpoint_path)
    first_mm_tensors = [k for k, v in first_checkpoint.items() if is_newline(k)]

# Load the checkpoint
mm_tensors = []
last_checkpoint = None
if projector_checkpoint_path is not None:
    last_checkpoint, file_type = load_model(projector_checkpoint_path)
    mm_tensors = [k for k, v in last_checkpoint.items() if is_mm_projector(k)]

if len(mm_tensors) == 0:
    if last_checkpoint is not None:
        for k, v in last_checkpoint.items():
            print(k)
    print(f"Found {len(mm_tensors)} tensors to extract out of {len(last_checkpoint) if last_checkpoint is not
None else 0} tensors.")
    print("No tensors found. Is this a LLaVA model?")
    exit()

print(f"Found {len(mm_tensors)} tensors to extract.")
print(f"Found additional {len(first_mm_tensors)} tensors to extract.")
# projector = {name: checkpoint.[name].float() for name in mm_tensors}
projector = {}
for name in mm_tensors:
    assert last_checkpoint is not None
    projector[name] = last_checkpoint[name].float()
```

```
for name in first_mm_tensors:
    assert first_checkpoint is not None
    projector[name] = first_checkpoint[name].float()

if len(projector) > 0:
    save_model(projector, f"{args.model}/llava.projector", 'pytorch')

print("Done!")
print(f"Now you can convert {args.model} to a regular LLaMA GGUF file.")
print(f"Also, use {args.model}/llava.projector to prepare a llava-encoder.gguf file.")

==== logic_dash.py ====
"""
LOGICSHREDDER :: logic_dash.py (Part 1/2)
Purpose: Live terminal dashboard showing system status and cognitive state
"""

import curses
import time
import os
import yaml
from pathlib import Path
import psutil
from core.config_loader import get

HEATMAP = Path("logs/logic_heatmap.yaml")
LOCK_FILE = Path("core/neuro.lock")
SNAPSHOT_DIR = Path("snapshots/")
AGENT_STATS = Path("logs/agent_stats/")

REFRESH_INTERVAL = 5

def get_hot_beliefs(n=5):
    if not HEATMAP.exists():
        return []
    try:
        data = yaml.safe_load(open(HEATMAP, 'r', encoding='utf-8'))
        sorted_data = sorted(data.items(), key=lambda x: x[1].get("heat_score", 0), reverse=True)
        return sorted_data[:n]
    except Exception as e:
        return [("error_loading", {"heat_score": 0.0})]

def get_lock_status():
    return LOCK_FILE.exists()

def get_last_snapshot_time():
    if not SNAPSHOT_DIR.exists():
        return "Never"
    files = list(SNAPSHOT_DIR.glob("*.tar.gz"))
    if not files:
        return "Never"
    latest = max(files, key=os.path.getctime)
    age = int(time.time() - os.path.getctime(latest))
    minutes = age // 60
```