```
            claims = extract_claims_json(f)
        elif ext == ".yaml":
            claims = extract_claims_yaml(f)
        elif ext == ".py":
            claims = extract_claims_py(f)
        elif ext in [".gguf", ".safetensors", ".bin"]:
            print(f"[devourer] ? Skipped binary model: {f.name}")
            continue

        if claims:
            for c in claims:
                write_fragment(c, f.name)
            print(f"[devourer] [OK] Ingested {len(claims)} from {f.name}")
            total += len(claims)
            shutil.move(f, CONSUMED_DIR / f.name)
        else:
            print(f"[devourer] WARNING Skipped {f.name} (no valid claims)")

    print(f"[devourer] ? Total logic extracted: {total}")

if __name__ == "__main__":
    devour()

==== train_pararule.py ====
"""Training module for logic-memnn"""
import argparse
import numpy as np
import keras.callbacks as C
import os

import tensorflow as tf
import keras.backend.tensorflow_backend as KTF
from data_gen import CHAR_IDX, IDX_CHAR
from word_dict_gen import WORD_INDEX, CONTEXT_TEXTS
from utils_pararule import LogicSeq, StatefulCheckpoint, ThresholdStop
from models import build_model
from keras import backend as K
import random

# Arguments
parser = argparse.ArgumentParser(description="Train logic-memnn models.")
parser.add_argument("model", help="The name of the module to train.")
parser.add_argument("model_file", help="Model filename.")
parser.add_argument("-md", "--model_dir", help="Model weights directory ending with /.")
parser.add_argument("--dim", default=64, type=int, help="Latent dimension.")
parser.add_argument("-d", "--debug", action="store_true", help="Only predict single data point.")
parser.add_argument("-ts", "--tasks", nargs='*', type=int, help="Tasks to train on, blank for all tasks.")
parser.add_argument("-e", "--epochs", default=120, type=int, help="Number of epochs to train.")
parser.add_argument("-s", "--summary", action="store_true", help="Dump model summary on creation.")
parser.add_argument("-i", "--ilp", action="store_true", help="Run ILP task.")
parser.add_argument("-its", "--iterations", default=4, type=int, help="Number of model iterations.")
parser.add_argument("-bs", "--batch_size", default=32, type=int, help="Training batch_size.")
parser.add_argument("-p", "--pad", action="store_true", help="Pad context with blank rule.")
ARGS = parser.parse_args()
```

```
MODEL_NAME = ARGS.model
MODEL_FNAME = ARGS.model_file
MODEL_WF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.h5'
MODEL_SF = (ARGS.model_dir or "weights/") + MODEL_FNAME + '.json'

# Stop numpy scientific printing
np.set_printoptions(suppress=True)

def create_model(**kwargs):
  """Create model from global arguments."""
  # Load in the model
  model = build_model(MODEL_NAME, MODEL_WF,
                      char_size=len(WORD_INDEX)+1,
                      dim=ARGS.dim,
                      **kwargs)
  if ARGS.summary:
    model.summary()
  return model

def ask(context, query, model):
  """Predict output for given context and query."""
  rs = context.split('.')[:-1] # split rules
  rr = [r for r in rs]
  dgen = LogicSeq([[(rr, query, 0)]], 1, False, False, pad=ARGS.pad)
  # print(dgen[0])
  out = model.predict_generator(dgen)
  # print("SHAPES:", [o.shape for o in out])
  for o in out:
    print(o)
  return np.asscalar(out[-1])

def train():
  """Train the given model saving weights to model_file."""
  os.environ["CUDA_VISIBLE_DEVICES"] = "6"
  seed_value = 0
  os.environ['PYTHONHASHSEED'] = str(seed_value)
  random.seed(seed_value)
  np.random.seed(seed_value)
  tf.set_random_seed(seed_value)

  session_conf = tf.ConfigProto(intra_op_parallelism_threads=1, inter_op_parallelism_threads=1)
  sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
  K.set_session(sess)

  # Setup callbacks
  logdir = (ARGS.model_dir or "weights/") + MODEL_FNAME + "_callbacks" # The folder for Tensorboard
  if not os.path.exists(logdir):
      os.mkdir(logdir)
  callbacks = [C.TensorBoard(logdir),
               C.ModelCheckpoint(filepath=MODEL_WF,
                                 verbose=1,
                                 save_best_only=True,
                                 save_weights_only=True),
```

```
              ThresholdStop(),
              C.EarlyStopping(monitor='loss', patience=10, verbose=1),
              C.TerminateOnNaN()]
  # Big data machine learning in the cloud
  ft = "data/pararule/train/{}_task{}.jsonl"
  fv = "data/pararule/dev/{}_task{}.jsonl"
  model = create_model(iterations=ARGS.iterations)
  # For long running training swap in stateful checkpoint
  callbacks[0] = StatefulCheckpoint(MODEL_WF, MODEL_SF,
                                    verbose=1, save_best_only=True,
                                    save_weights_only=True)
  tasks = ARGS.tasks or range(0, 8)
     traind  =  LogicSeq.from_files([ft.forset_sessionmat("train",  i)  for  i  in  tasks],  ARGS.batch_size,
pad=ARGS.pad)
  vald = LogicSeq.from_files([fv.format("val", i) for i in tasks], ARGS.batch_size, pad=ARGS.pad)
  model.fit_generator(traind, epochs=ARGS.epochs,
                      callbacks=callbacks,
                      validation_data=vald,
                      verbose=1, shuffle=True,
                      initial_epoch=callbacks[0].get_last_epoch())

def debug():
  """Run a single data point for debugging."""
  # Add command line history support
  import readline # pylint: disable=unused-variable
  model = create_model(iterations=ARGS.iterations, training=False)
  while True:
    try:
      ctx = input("CTX: ").lower().replace(',','')
      if ctx == 'q':
        break
      q = input("Q: ").lower().replace('.','')
      print("OUT:", ask(ctx, q, model))
    except(KeyboardInterrupt, EOFError, SystemExit):
      break
  print("\nTerminating.")


def ilp(training=True):
  """Run the ILP task using the ILP model."""
  # Create the head goal
  goals, vgoals = ["f(X)"], list()
  for g in goals:
    v = np.zeros((1, 1, 4, len(CHAR_IDX)+1))
    for i, c in enumerate(g):
      v[0, 0, i, CHAR_IDX[c]] = 1
    vgoals.append(v)
  # Create the ILP wrapper model
  model = build_model("ilp", "weights/ilp.h5",
                      char_size=len(CHAR_IDX)+1,
                      training=training,
                      goals=vgoals,
                      num_preds=1,
                      pred_len=4)
```

```
  model.summary()
  traind = LogicSeq.from_file("data/ilp_train.txt", ARGS.batch_size, pad=ARGS.pad)
  testd = LogicSeq.from_file("data/ilp_test.txt", ARGS.batch_size, pad=ARGS.pad)
  if training:
    # Setup callbacks
    callbacks = [C.ModelCheckpoint(filepath="weights/ilp.h5",
                                   verbose=1,
                                   save_best_only=True,
                                   save_weights_only=True),
                 C.TerminateOnNaN()]
    model.fit_generator(traind, epochs=200,
                        callbacks=callbacks,
                        validation_data=testd,
                        shuffle=True)
  else:
    # Dummy input to get templates
    ctx = "b(h).v(O):-c(O).c(a)."
    ctx = ctx.split('.')[:-1] # split rules
    ctx = [r + '.' for r in ctx]
    dgen = LogicSeq([[(ctx, "f(h).", 0)]], 1, False, False)
    print("TEMPLATES:")
    outs = model.predict_on_batch(dgen[0])
    ts, out = outs[0], outs[-1]
    print(ts)
    # Decode template
    # (num_templates, num_preds, pred_length, char_size)
    ts = np.argmax(ts[0], axis=-1)
    ts = np.vectorize(lambda i: IDX_CHAR[i])(ts)
    print(ts)
    print("CTX:", ctx)
    for o in outs[1:-1]:
      print(o)
    print("OUT:", out)

if __name__ == '__main__':
  if ARGS.ilp:
    ilp(not ARGS.debug)
  elif ARGS.debug:
    debug()
  else:
    train()

==== train_utils.py ====
import random

import numpy as np
import ray
import torch
from ray import tune
from ray.tune.schedulers import AsyncHyperBandScheduler
from ray.tune.suggest.basic_variant import BasicVariantGenerator
from sklearn.model_selection import train_test_split
from tqdm.auto import tqdm, trange
```

```
from crm.core import Network
from crm.distributed import DataWorker, ParameterServer
from crm.utils import get_metrics, save_object


def train_distributed(
    n: Network,
    X_train,
    y_train,
    num_epochs: int,
    optimizer: torch.optim.Optimizer,
    criterion,
    X_val,
    y_val,
    num_workers: int,
    verbose: bool = True,
):
    raise NotImplementedError("ToDo")
    iterations = 10
    test_loader = zip(X_val, y_val)  # noqa
    print("Running Asynchronous Parameter Server Training.")

    ray.init(ignore_reinit_error=True)
    ps = ParameterServer.remote(1e-3, n.num_neurons, n.adj_list)
    workers = [
        DataWorker.remote(X_train, y_train, n.num_neurons, n.adj_list)
        for i in range(num_workers)
    ]

    current_weights = ps.get_weights.remote()

    gradients = {}
    for worker in workers:
        gradients[worker.compute_gradients.remote(current_weights)] = worker

    for i in range(iterations * num_workers):
        ready_gradient_list, _ = ray.wait(list(gradients))
        ready_gradient_id = ready_gradient_list[0]
        worker = gradients.pop(ready_gradient_id)

        # Compute and apply gradients.
        current_weights = ps.apply_gradients.remote(*[ready_gradient_id])
        gradients[worker.compute_gradients.remote(current_weights)] = worker

        if i % 10 == 0:
            pass
        # Evaluate the current model after every 10 updates.
        n.set_weights(ray.get(current_weights))
        accuracy = get_metrics(n, X_val, y_val, output_dict=True)["accuracy"]
        print("Iter {}: \taccuracy is {:.1f}".format(i, accuracy))

    print("Final accuracy is {:.1f}.".format(accuracy))
```

```
def train(
    n: Network,
    X_train,
    y_train,
    num_epochs: int,
    optimizer: torch.optim.Optimizer,
    criterion,
    save_here: str = None,
    X_val=None,
    y_val=None,
    verbose: bool = False,
):
    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []
    min_loss = 1e10
    for e in trange(num_epochs):
        c = list(zip(X_train, y_train))
        random.shuffle(c)
        X_train, y_train = zip(*c)
        local_train_losses = []
        for i in trange(len(X_train)):
            # print(f"Epoch {e}/{num_epochs} | Batch {i}/{len(X_train)}")
            f_mapper = X_train[i]
            out = n.forward(f_mapper).reshape(1, -1)
            loss = criterion(out, y_train[i].reshape(1))
            local_train_losses.append(loss.item())
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            n.reset()
        with torch.no_grad():
            train_losses.append(sum(local_train_losses) / len(local_train_losses))
            train_accs.append(
                get_metrics(n, X_train, y_train, output_dict=True)["accuracy"]
            )
            if X_val is not None and y_val is not None:
                local_val_losses = []
                for j in range(len(X_val)):
                    f_mapper = X_val[j]
                    out = n.forward(f_mapper).reshape(1, -1)
                    loss = criterion(out, y_val[j].reshape(1))
                    local_val_losses.append(loss.item())
                    n.reset()
                val_losses.append(sum(local_val_losses) / len(local_val_losses))
                val_accs.append(
                    get_metrics(n, X_val, y_val, output_dict=True)["accuracy"]
                )
                if val_losses[-1] < min_loss:
                    min_loss = val_losses[-1]
                    patience = 0
                else:
                    patience += 1
```

```
                if patience > 3:
                    print("Patience exceeded. Stopping training.")
                    break
        if verbose:
            tqdm.write(f"Epoch {e}")
            tqdm.write(f"Train loss: {train_losses[-1]}")
            tqdm.write(f"Train acc: {train_accs[-1]}")
            if X_val is not None and y_val is not None:
                tqdm.write(f"Val loss: {val_losses[-1]}")
                tqdm.write(f"Val acc: {val_accs[-1]}")
            tqdm.write("------------------------------------")
        if save_here is not None:
            save_object(n, f"{save_here}_{e}.pt")
    return (
        (train_losses, train_accs, val_losses, val_accs)
        if X_val is not None and y_val is not None
        else (train_losses, train_accs)
    )


def get_best_config(
    n: Network,
    X,
    y,
    num_epochs: int,
    optimizer: torch.optim.Optimizer,
    criterion,
):
    """Uses Ray Tune and Optuna to find the best configuration for the network."""

    def train_with_config(config):
        """Train the network with the given config."""
        optimizer = torch.optim.Adam(n.parameters(), lr=config["lr"])
        X_train, X_val, y_train, y_val = train_test_split(
            X, y, test_size=0.2, random_state=24, stratify=y
        )
        train_losses, train_accs, val_losses, val_accs = train(
            n=n,
            X_train=X_train,
            y_train=y_train,
            num_epochs=num_epochs,
            optimizer=optimizer,
            criterion=criterion,
            X_val=X_val,
            y_val=y_val,
            verbose=False,
        )
        return {
            "mean_train_loss": np.mean(train_losses),
            "mean_train_acc": np.mean(train_accs),
            "mean_val_loss": np.mean(val_losses),
            "mean_val_acc": np.mean(val_accs),
        }
```

```
    config = {"lr": tune.grid_search([0.01, 0.001, 0.005, 0.0001])}
    algo = BasicVariantGenerator(max_concurrent=16)
    # uncomment and set max_concurrent to limit number of cores
    # algo = ConcurrencyLimiter(algo, max_concurrent=16)
    scheduler = AsyncHyperBandScheduler()

    analysis = tune.run(
        train_with_config,
        num_samples=1,
        config=config,
        name="optuna_train",
        metric="mean_val_acc",
        mode="max",
        search_alg=algo,
        scheduler=scheduler,
        verbose=0,
        max_failures=1,
    )

    return analysis.best_config

==== tts-outetts.py ====
import sys
#import json
#import struct
import requests
import re
import struct
import numpy as np
from concurrent.futures import ThreadPoolExecutor


def fill_hann_window(size, periodic=True):
    if periodic:
        return np.hanning(size + 1)[:-1]
    return np.hanning(size)


def irfft(n_fft, complex_input):
    return np.fft.irfft(complex_input, n=n_fft)


def fold(buffer, n_out, n_win, n_hop, n_pad):
    result = np.zeros(n_out)
    n_frames = len(buffer) // n_win

    for i in range(n_frames):
        start = i * n_hop
        end = start + n_win
        result[start:end] += buffer[i * n_win:(i + 1) * n_win]

    return result[n_pad:-n_pad] if n_pad > 0 else result
```

```
def process_frame(args):
    l, n_fft, ST, hann = args
    frame = irfft(n_fft, ST[l])
    frame = frame * hann
    hann2 = hann * hann
    return frame, hann2


def embd_to_audio(embd, n_codes, n_embd, n_thread=4):
    embd = np.asarray(embd, dtype=np.float32).reshape(n_codes, n_embd)

    n_fft = 1280
    n_hop = 320
    n_win = 1280
    n_pad = (n_win - n_hop) // 2
    n_out = (n_codes - 1) * n_hop + n_win

    hann = fill_hann_window(n_fft, True)

    E = np.zeros((n_embd, n_codes), dtype=np.float32)
    for l in range(n_codes):
        for k in range(n_embd):
            E[k, l] = embd[l, k]

    half_embd = n_embd // 2
    S = np.zeros((n_codes, half_embd + 1), dtype=np.complex64)

    for k in range(half_embd):
        for l in range(n_codes):
            mag = E[k, l]
            phi = E[k + half_embd, l]

            mag = np.clip(np.exp(mag), 0, 1e2)
            S[l, k] = mag * np.exp(1j * phi)

    res = np.zeros(n_codes * n_fft)
    hann2_buffer = np.zeros(n_codes * n_fft)

    with ThreadPoolExecutor(max_workers=n_thread) as executor:
        args = [(l, n_fft, S, hann) for l in range(n_codes)]
        results = list(executor.map(process_frame, args))

        for l, (frame, hann2) in enumerate(results):
            res[l*n_fft:(l+1)*n_fft] = frame
            hann2_buffer[l*n_fft:(l+1)*n_fft] = hann2

    audio = fold(res, n_out, n_win, n_hop, n_pad)
    env = fold(hann2_buffer, n_out, n_win, n_hop, n_pad)

    mask = env > 1e-10
    audio[mask] /= env[mask]

    return audio
```

```
def save_wav(filename, audio_data, sample_rate):
    num_channels = 1
    bits_per_sample = 16
    bytes_per_sample = bits_per_sample // 8
    data_size = len(audio_data) * bytes_per_sample
    byte_rate = sample_rate * num_channels * bytes_per_sample
    block_align = num_channels * bytes_per_sample
    chunk_size = 36 + data_size  # 36 = size of header minus first 8 bytes

    header = struct.pack(
        '<4sI4s4sIHHIIHH4sI',
        b'RIFF',
        chunk_size,
        b'WAVE',
        b'fmt ',
        16,                # fmt chunk size
        1,                 # audio format (PCM)
        num_channels,
        sample_rate,
        byte_rate,
        block_align,
        bits_per_sample,
        b'data',
        data_size
    )

    audio_data = np.clip(audio_data * 32767, -32768, 32767)
    pcm_data = audio_data.astype(np.int16)

    with open(filename, 'wb') as f:
        f.write(header)
        f.write(pcm_data.tobytes())


def process_text(text: str):
    text = re.sub(r'\d+(\.\d+)?', lambda x: x.group(), text.lower()) # TODO this needs to be fixed
    text = re.sub(r'[-_/,\.\\]', ' ', text)
    text = re.sub(r'[^a-z\s]', '', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text.split()

# usage:
# python tts-outetts.py http://server-llm:port http://server-dec:port "text"

if len(sys.argv) <= 3:
    print("usage: python tts-outetts.py http://server-llm:port http://server-dec:port \"text\"")
    exit(1)

host_llm = sys.argv[1]
host_dec = sys.argv[2]
text = sys.argv[3]

prefix = """<|im_start|>
```

<|text_start|>the<|text_sep|>overall<|text_sep|>package<|text_sep|>from<|text_sep|>just<|text_sep|>two<|text_sep|>people<|text_sep|>is<|text_sep|>pretty<|text_sep|>remarkable<|text_sep|>sure<|text_sep|>i<|text_sep|>have<|text_sep|>some<|text_sep|>critiques<|text_sep|>about<|text_sep|>some<|text_sep|>of<|text_sep|>the<|text_sep|>gameplay<|text_sep|>aspects<|text_sep|>but<|text_sep|>its<|text_sep|>still<|text_sep|>really<|text_sep|>enjoyable<|text_sep|>and<|text_sep|>it<|text_sep|>looks<|text_sep|>lovely<|text_sep|>"""

```
words = process_text(text)
words = "<|text_sep|>".join([i.strip() for i in words])
words += "<|text_end|>\n"

# voice data
# TODO: load from json
#suffix = """<|audio_start|>
#the<|t_0.08|><|code_start|><|257|><|740|><|636|><|913|><|788|><|1703|><|code_end|>
#overall<|t_0.36|><|code_start|><|127|><|201|><|191|><|774|><|700|><|532|><|1056|><|557|><|798|><|298|><|1741|><|747|><|1662|><|1617|><|1702|><|1527|><|368|><|1588|><|1049|><|1008|><|1625|><|747|><|1576|><|728|><|1019|><|1696|><|1765|><|code_end|>
#package<|t_0.56|><|code_start|><|935|><|584|><|1319|><|627|><|1016|><|1491|><|1344|><|1117|><|1526|><|1040|><|239|><|1435|><|951|><|498|><|723|><|1180|><|535|><|789|><|1649|><|1637|><|78|><|465|><|1668|><|901|><|595|><|1675|><|117|><|1009|><|1667|><|320|><|840|><|79|><|507|><|1762|><|1508|><|1228|><|1768|><|802|><|1450|><|1457|><|232|><|639|><|code_end|>
#from<|t_0.19|><|code_start|><|604|><|782|><|1682|><|872|><|1532|><|1600|><|1036|><|1761|><|647|><|1554|><|1371|><|653|><|1595|><|950|><|code_end|>
#just<|t_0.25|><|code_start|><|1782|><|1670|><|317|><|786|><|1748|><|631|><|599|><|1155|><|1364|><|1524|><|36|><|1591|><|889|><|1535|><|541|><|440|><|1532|><|50|><|870|><|code_end|>
#two<|t_0.24|><|code_start|><|1681|><|1510|><|673|><|799|><|805|><|1342|><|330|><|519|><|62|><|640|><|1138|><|565|><|1552|><|1497|><|1552|><|572|><|1715|><|1732|><|code_end|>
#people<|t_0.39|><|code_start|><|593|><|274|><|136|><|740|><|691|><|633|><|1484|><|1061|><|1138|><|1485|><|344|><|428|><|397|><|1562|><|645|><|917|><|1035|><|1449|><|1669|><|487|><|442|><|1484|><|1329|><|1832|><|1704|><|600|><|761|><|653|><|269|><|code_end|>
#is<|t_0.16|><|code_start|><|566|><|583|><|1755|><|646|><|1337|><|709|><|802|><|1008|><|485|><|1583|><|652|><|10|><|code_end|>
#pretty<|t_0.32|><|code_start|><|1818|><|1747|><|692|><|733|><|1010|><|534|><|406|><|1697|><|1053|><|1521|><|1355|><|1274|><|816|><|1398|><|211|><|1218|><|817|><|1472|><|1703|><|686|><|13|><|822|><|445|><|1068|><|code_end|>
#remarkable<|t_0.68|><|code_start|><|230|><|1048|><|1705|><|355|><|706|><|1149|><|1535|><|1787|><|1356|><|1396|><|835|><|1583|><|486|><|1249|><|286|><|937|><|1076|><|1150|><|614|><|42|><|1058|><|705|><|681|><|798|><|934|><|490|><|514|><|1399|><|572|><|1446|><|1703|><|1346|><|1040|><|1426|><|1304|><|664|><|171|><|1530|><|625|><|64|><|1708|><|1830|><|1030|><|443|><|1509|><|1063|><|1605|><|1785|><|721|><|1440|><|923|><|code_end|>
#sure<|t_0.36|><|code_start|><|792|><|1780|><|923|><|1640|><|265|><|261|><|1525|><|567|><|1491|><|1250|><|1730|><|362|><|919|><|1766|><|543|><|1|><|333|><|113|><|970|><|252|><|1606|><|133|><|302|><|1810|><|1046|><|1190|><|1675|><|code_end|>
#i<|t_0.08|><|code_start|><|123|><|439|><|1074|><|705|><|1799|><|637|><|code_end|>
#have<|t_0.16|><|code_start|><|1509|><|599|><|518|><|1170|><|552|><|1029|><|1267|><|864|><|419|><|143|><|1061|><|0|><|code_end|>
#some<|t_0.16|><|code_start|><|619|><|400|><|1270|><|62|><|1370|><|1832|><|917|><|1661|><|167|><|269|><|1366|><|1508|><|code_end|>
#critiques<|t_0.60|><|code_start|><|559|><|584|><|1163|><|1129|><|1313|><|1728|><|721|><|1146|><|1093|><|577|><|928|><|27|><|630|><|1080|><|1346|><|1337|><|320|><|1382|><|1175|><|1682|><|1556|><|990|><|1683|><|860|><|1721|><|110|><|786|><|376|><|1085|><|756|><|1523|><|234|><|1334|><|1506|><|1578|><|659|><|612|><|1108|><|1466|><|1647|><|308|><|1470|><|746|><|556|><|1061|><|code_end|>
#about<|t_0.29|><|code_start|><|26|><|1649|><|545|><|1367|><|1263|><|1728|><|450|><|859|><|1434|><|497|><|1220|><|1285|><|179|><|755|><|1154|><|779|><|179|><|1229|><|1213|><|922|><|1774|><|1408|><|code_end|>
#some<|t_0.23|><|code_start|><|986|><|28|><|1649|><|778|><|858|><|1519|><|1|><|18|><|26|><|1042|><|1174|><|1309
```

|><|1499|><|1712|><|1692|><|1516|><|1574|><|code_end|>
#of<|t_0.07|><|code_start|><|197|><|716|><|1039|><|1662|><|64|><|code_end|>
#the<|t_0.08|><|code_start|><|1811|><|1568|><|569|><|886|><|1025|><|1374|><|code_end|>
#gameplay<|t_0.48|><|code_start|><|1269|><|1092|><|933|><|1362|><|1762|><|1700|><|1675|><|215|><|781|><|1086|><|461|><|838|><|1022|><|759|><|649|><|1416|><|1004|><|551|><|909|><|787|><|343|><|830|><|1391|><|1040|><|1622|><|1779|><|1360|><|1231|><|1187|><|1317|><|76|><|997|><|989|><|978|><|737|><|189|><|code_end|>
#aspects<|t_0.56|><|code_start|><|1423|><|797|><|1316|><|1222|><|147|><|719|><|1347|><|386|><|1390|><|1558|><|154|><|440|><|634|><|592|><|1097|><|1718|><|712|><|763|><|1118|><|1721|><|1311|><|868|><|580|><|362|><|1435|><|868|><|247|><|221|><|886|><|1145|><|1274|><|1284|><|457|><|1043|><|1459|><|1818|><|62|><|599|><|1035|><|62|><|1649|><|778|><|code_end|>
#but<|t_0.20|><|code_start|><|780|><|1825|><|1681|><|1007|><|861|><|710|><|702|><|939|><|1669|><|1491|><|613|><|1739|><|823|><|1469|><|648|><|code_end|>
#its<|t_0.09|><|code_start|><|92|><|688|><|1623|><|962|><|1670|><|527|><|599|><|code_end|>
#still<|t_0.27|><|code_start|><|636|><|10|><|1217|><|344|><|713|><|957|><|823|><|154|><|1649|><|1286|><|508|><|214|><|1760|><|1250|><|456|><|1352|><|1368|><|921|><|615|><|5|><|code_end|>
#really<|t_0.36|><|code_start|><|55|><|420|><|1008|><|1659|><|27|><|644|><|1266|><|617|><|761|><|1712|><|109|><|1465|><|1587|><|503|><|1541|><|619|><|197|><|1019|><|817|><|269|><|377|><|362|><|1381|><|507|><|1488|><|4|><|1695|><|code_end|>
#enjoyable<|t_0.49|><|code_start|><|678|><|501|><|864|><|319|><|288|><|1472|><|1341|><|686|><|562|><|1463|><|619|><|1563|><|471|><|911|><|730|><|1811|><|1006|><|520|><|861|><|1274|><|125|><|1431|><|638|><|621|><|153|><|876|><|1770|><|437|><|987|><|1653|><|1109|><|898|><|1285|><|80|><|593|><|1709|><|843|><|code_end|>
#and<|t_0.15|><|code_start|><|1285|><|987|><|303|><|1037|><|730|><|1164|><|502|><|120|><|1737|><|1655|><|1318|><|code_end|>
#it<|t_0.09|><|code_start|><|848|><|1366|><|395|><|1601|><|1513|><|593|><|1302|><|code_end|>
#looks<|t_0.27|><|code_start|><|1281|><|1266|><|1755|><|572|><|248|><|1751|><|1257|><|695|><|1380|><|457|><|659|><|585|><|1315|><|1105|><|1776|><|736|><|24|><|736|><|654|><|1027|><|code_end|>
#lovely<|t_0.56|><|code_start|><|634|><|596|><|1766|><|1556|><|1306|><|1285|><|1481|><|1721|><|1123|><|438|><|1246|><|1251|><|795|><|659|><|1381|><|1658|><|217|><|1772|><|562|><|952|><|107|><|1129|><|1112|><|467|><|550|><|1079|><|840|><|1615|><|1469|><|1380|><|168|><|917|><|836|><|1827|><|437|><|583|><|67|><|595|><|1087|><|1646|><|1493|><|1677|><|code_end|>"""

# TODO: tokenization is slow for some reason - here is pre-tokenized input
suffix = [151667, 198, 1782, 155780, 151669, 151929, 152412, 152308, 152585, 152460, 153375, 151670, 198, 74455,
            155808, 151669, 151799, 151873, 151863, 152446, 152372, 152204, 152728, 152229, 152470, 151970, 153413,
            152419, 153334, 153289, 153374, 153199, 152040, 153260, 152721, 152680, 153297, 152419, 153248, 152400,
           152691, 153368, 153437, 151670, 198, 1722, 155828, 151669, 152607, 152256, 152991, 152299, 152688, 153163,
            153016, 152789, 153198, 152712, 151911, 153107, 152623, 152170, 152395, 152852, 152207, 152461, 153321,
            153309, 151750, 152137, 153340, 152573, 152267, 153347, 151789, 152681, 153339, 151992, 152512, 151751,
           152179, 153434, 153180, 152900, 153440, 152474, 153122, 153129, 151904, 152311, 151670, 198, 1499, 155791,
            151669, 152276, 152454, 153354, 152544, 153204, 153272, 152708, 153433, 152319, 153226, 153043, 152325,
           153267, 152622, 151670, 198, 4250, 155797, 151669, 153454, 153342, 151989, 152458, 153420, 152303, 152271,
            152827, 153036, 153196, 151708, 153263, 152561, 153207, 152213, 152112, 153204, 151722, 152542, 151670, 198,
             19789, 155796, 151669, 153353, 153182, 152345, 152471, 152477, 153014, 152002, 152191, 151734, 152312, 152810,

152237, 153224, 153169, 153224, 152244, 153387, 153404, 151670, 198, 16069, 155811, 151669, 152265,
151946,
151808, 152412, 152363, 152305, 153156, 152733, 152810, 153157, 152016, 152100, 152069, 153234,
152317,
152589, 152707, 153121, 153341, 152159, 152114, 153156, 153001, 153504, 153376, 152272, 152433,
152325,
151941, 151670, 198, 285, 155788, 151669, 152238, 152255, 153427, 152318, 153009, 152381, 152474,
152680,
152157, 153255, 152324, 151682, 151670, 198, 32955, 155804, 151669, 153490, 153419, 152364, 152405,
152682,
152206, 152078, 153369, 152725, 153193, 153027, 152946, 152488, 153070, 151883, 152890, 152489,
153144,
153375, 152358, 151685, 152494, 152117, 152740, 151670, 198, 37448, 480, 155840, 151669, 151902,
152720,
153377, 152027, 152378, 152821, 153207, 153459, 153028, 153068, 152507, 153255, 152158, 152921,
151958,
152609, 152748, 152822, 152286, 151714, 152730, 152377, 152353, 152470, 152606, 152162, 152186,
153071,
152244, 153118, 153375, 153018, 152712, 153098, 152976, 152336, 151843, 153202, 152297, 151736,
153380,
153502, 152702, 152115, 153181, 152735, 153277, 153457, 152393, 153112, 152595, 151670, 198, 19098,
155808,
151669, 152464, 153452, 152595, 153312, 151937, 151933, 153197, 152239, 153163, 152922, 153402,
152034,
152591, 153438, 152215, 151673, 152005, 151785, 152642, 151924, 153278, 151805, 151974, 153482,
152718,
152862, 153347, 151670, 198, 72, 155780, 151669, 151795, 152111, 152746, 152377, 153471, 152309,
151670, 198,
19016, 155788, 151669, 153181, 152271, 152190, 152842, 152224, 152701, 152939, 152536, 152091,
151815, 152733,
151672, 151670, 198, 14689, 155788, 151669, 152291, 152072, 152942, 151734, 153042, 153504, 152589,
153333,
151839, 151941, 153038, 153180, 151670, 198, 36996, 8303, 155832, 151669, 152231, 152256, 152835,
152801,
152985, 153400, 152393, 152818, 152765, 152249, 152600, 151699, 152302, 152752, 153018, 153009,
151992,
153054, 152847, 153354, 153228, 152662, 153355, 152532, 153393, 151782, 152458, 152048, 152757,
152428,
153195, 151906, 153006, 153178, 153250, 152331, 152284, 152780, 153138, 153319, 151980, 153142,
152418,
152228, 152733, 151670, 198, 9096, 155801, 151669, 151698, 153321, 152217, 153039, 152935, 153400,
152122,
152531, 153106, 152169, 152892, 152957, 151851, 152427, 152826, 152451, 151851, 152901, 152885,
152594,
153446, 153080, 151670, 198, 14689, 155795, 151669, 152658, 151700, 153321, 152450, 152530, 153191,
151673,
151690, 151698, 152714, 152846, 152981, 153171, 153384, 153364, 153188, 153246, 151670, 198, 1055,
155779,
151669, 151869, 152388, 152711, 153334, 151736, 151670, 198, 1782, 155780, 151669, 153483, 153240,
152241,
152558, 152697, 153046, 151670, 198, 5804, 1363, 155820, 151669, 152941, 152764, 152605, 153034,
153434,
153372, 153347, 151887, 152453, 152758, 152133, 152510, 152694, 152431, 152321, 153088, 152676,
152223,

```
            152581, 152459, 152015, 152502, 153063, 152712, 153294, 153451, 153032, 152903, 152859, 152989,
151748,
            152669, 152661, 152650, 152409, 151861, 151670, 198, 300, 7973, 155828, 151669, 153095, 152469,
152988,
            152894, 151819, 152391, 153019, 152058, 153062, 153230, 151826, 152112, 152306, 152264, 152769,
153390,
            152384, 152435, 152790, 153393, 152983, 152540, 152252, 152034, 153107, 152540, 151919, 151893,
152558,
            152817, 152946, 152956, 152129, 152715, 153131, 153490, 151734, 152271, 152707, 151734, 153321,
152450,
          151670, 198, 8088, 155792, 151669, 152452, 153497, 153353, 152679, 152533, 152382, 152374, 152611,
153341,
          153163, 152285, 153411, 152495, 153141, 152320, 151670, 198, 1199, 155781, 151669, 151764, 152360,
153295,
         152634, 153342, 152199, 152271, 151670, 198, 43366, 155799, 151669, 152308, 151682, 152889, 152016,
152385,
            152629, 152495, 151826, 153321, 152958, 152180, 151886, 153432, 152922, 152128, 153024, 153040,
152593,
         152287, 151677, 151670, 198, 53660, 155808, 151669, 151727, 152092, 152680, 153331, 151699, 152316,
152938,
            152289, 152433, 153384, 151781, 153137, 153259, 152175, 153213, 152291, 151869, 152691, 152489,
151941,
        152049, 152034, 153053, 152179, 153160, 151676, 153367, 151670, 198, 268, 4123, 480, 155821, 151669,
152350,
            152173, 152536, 151991, 151960, 153144, 153013, 152358, 152234, 153135, 152291, 153235, 152143,
152583,
            152402, 153483, 152678, 152192, 152533, 152946, 151797, 153103, 152310, 152293, 151825, 152548,
153442,
          152109, 152659, 153325, 152781, 152570, 152957, 151752, 152265, 153381, 152515, 151670, 198, 437,
155787,
            151669, 152957, 152659, 151975, 152709, 152402, 152836, 152174, 151792, 153409, 153327, 152990,
151670, 198,
          275, 155781, 151669, 152520, 153038, 152067, 153273, 153185, 152265, 152974, 151670, 198, 94273,
155799,
            151669, 152953, 152938, 153427, 152244, 151920, 153423, 152929, 152367, 153052, 152129, 152331,
152257,
          152987, 152777, 153448, 152408, 151696, 152408, 152326, 152699, 151670, 198, 385, 16239, 155828,
151669,
            152306, 152268, 153438, 153228, 152978, 152957, 153153, 153393, 152795, 152110, 152918, 152923,
152467,
            152331, 153053, 153330, 151889, 153444, 152234, 152624, 151779, 152801, 152784, 152139, 152222,
152751,
            152512, 153287, 153141, 153052, 151840, 152589, 152508, 153499, 152109, 152255, 151739, 152267,
152759,
        153318, 153165, 153349, 151670, ]

response = requests.post(
    host_llm + "/completion",
    json={
        "prompt": [prefix + words, *suffix],
        "n_predict": 1024,
        "cache_prompt": True,
        "return_tokens": True,
        "samplers": ["top_k"],
```

```
        "top_k": 16,
        "seed": 1003,
    }
)

response_json = response.json()

#print(json.dumps(response_json, indent=4))
#print(json.dumps(response_json["prompt"], indent=4).replace("\\n", "\n"))
#print(json.dumps(response_json["timings"], indent=4))
#print(json.dumps(response_json["tokens"], indent=4))

codes = response_json["tokens"]

codes = [t - 151672 for t in codes if t >= 151672 and t <= 155772]

response = requests.post(
    host_dec + "/embeddings",
    json={
        "input": [*codes],
    }
)

response_json = response.json()

#print(json.dumps(response_json, indent=4))

# spectrogram
embd = response_json[0]["embedding"]

n_codes = len(embd)
n_embd = len(embd[0])

print('spectrogram generated: n_codes: %d, n_embd: %d' % (n_codes, n_embd))

# post-process the spectrogram to convert to audio
print('converting to audio ...')
audio = embd_to_audio(embd, n_codes, n_embd)
print('audio generated: %d samples' % len(audio))

filename = "output.wav"
sample_rate = 24000 # sampling rate

# zero out first 0.25 seconds
audio[:24000 // 4] = 0.0

save_wav(filename, audio, sample_rate)
print('audio written to file "%s"' % filename)

==== unicode_sanitizer.py ====
import os
from pathlib import Path

# Mapping of problematic Unicode -> Safe ASCII equivalents
```

```
UNICODE_REPLACEMENTS = {
    'OK': 'OK',
    '->': '->',
    'LAUNCH': 'LAUNCH',
    'TIME': 'TIME',
    'ERROR': 'ERROR',
    'INFO': 'INFO',
    'CONFIG': 'CONFIG',
    '[OK]': '[OK]',
    'WARNING': 'WARNING',
    '>>': '>>',
    'STOP': 'STOP',
}

def sanitize_file(path):
    try:
        original = path.read_text(encoding="utf-8")
    except Exception as e:
        print(f"[SKIP] Could not read {path}: {e}")
        return

    modified = original
    for uni, safe in UNICODE_REPLACEMENTS.items():
        modified = modified.replace(uni, safe)

    if modified != original:
        path.write_text(modified, encoding="utf-8")
        print(f"[OK] Sanitized: {path}")
    else:
        print(f"[--] Clean: {path}")

def sanitize_folder(folder):
    folder = Path(folder)
    for path in folder.rglob("*.py"):
        sanitize_file(path)

if __name__ == "__main__":
    root = Path(".")  # Current folder
    print("[*] Starting Unicode sanitization...")
    sanitize_folder(root)
    print("[DONE] All .py files sanitized.")

==== utility.py ====
from __future__ import annotations

from dataclasses import dataclass
from typing import Literal

import os
import json


def fill_templated_filename(filename: str, output_type: str | None) -> str:
    # Given a file name fill in any type templates e.g. 'some-model-name.{ftype}.gguf'
```

```
    ftype_lowercase: str = output_type.lower() if output_type is not None else ""
    ftype_uppercase: str = output_type.upper() if output_type is not None else ""
    return filename.format(ftype_lowercase,
                           outtype=ftype_lowercase, ftype=ftype_lowercase,
                           OUTTYPE=ftype_uppercase, FTYPE=ftype_uppercase)


def model_weight_count_rounded_notation(model_params_count: int, min_digits: int = 2) -> str:
    if model_params_count > 1e12 :
        # Trillions Of Parameters
        scaled_model_params = model_params_count * 1e-12
        scale_suffix = "T"
    elif model_params_count > 1e9 :
        # Billions Of Parameters
        scaled_model_params = model_params_count * 1e-9
        scale_suffix = "B"
    elif model_params_count > 1e6 :
        # Millions Of Parameters
        scaled_model_params = model_params_count * 1e-6
        scale_suffix = "M"
    else:
        # Thousands Of Parameters
        scaled_model_params = model_params_count * 1e-3
        scale_suffix = "K"

    fix = max(min_digits - len(str(round(scaled_model_params)).lstrip('0')), 0)

    return f"{scaled_model_params:.{fix}f}{scale_suffix}"


def size_label(total_params: int, shared_params: int, expert_params: int, expert_count: int) -> str:

    if expert_count > 0:
                pretty_size  =  model_weight_count_rounded_notation(abs(shared_params)  +  abs(expert_params),
min_digits=2)
        size_class = f"{expert_count}x{pretty_size}"
    else:
        size_class = model_weight_count_rounded_notation(abs(total_params), min_digits=2)

    return size_class


def  naming_convention(model_name:  str  |  None,  base_name:  str  |  None,  finetune_string:  str  |  None,
version_string:  str  |  None,  size_label:  str  |  None,  output_type:  str  |  None,  model_type:  Literal['vocab',
'LoRA'] | None = None) -> str:
    # Reference: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md#gguf-naming-convention

    if base_name is not None:
        name = base_name.strip().replace(' ', '-').replace('/', '-')
    elif model_name is not None:
        name = model_name.strip().replace(' ', '-').replace('/', '-')
    else:
        name = "ggml-model"
```

```
    parameters = f"-{size_label}" if size_label is not None else ""

    finetune = f"-{finetune_string.strip().replace(' ', '-')}" if finetune_string is not None else ""

    version = f"-{version_string.strip().replace(' ', '-')}" if version_string is not None else ""

    encoding = f"-{output_type.strip().replace(' ', '-').upper()}" if output_type is not None else ""

    kind = f"-{model_type.strip().replace(' ', '-')}" if model_type is not None else ""

    return f"{name}{parameters}{finetune}{version}{encoding}{kind}"


@dataclass
class RemoteTensor:
    dtype: str
    shape: tuple[int, ...]
    offset_start: int
    size: int
    url: str

    def data(self) -> bytearray:
        # TODO: handle request errors (maybe with limited retries?)
        # NOTE: using a bytearray, otherwise PyTorch complains the buffer is not writeable
                data  =  bytearray(SafetensorRemote.get_data_by_range(url=self.url,  start=self.offset_start,
size=self.size))
        return data


class SafetensorRemote:
    """
    Uility class to handle remote safetensor files.
    This class is designed to work with Hugging Face model repositories.

    Example (one model has single safetensor file, the other has multiple):
        for model_id in ["ngxson/TEST-Tiny-Llama4", "Qwen/Qwen2.5-7B-Instruct"]:
            tensors = SafetensorRemote.get_list_tensors_hf_model(model_id)
            print(tensors)

    Example reading tensor data:
        tensors = SafetensorRemote.get_list_tensors_hf_model(model_id)
        for name, meta in tensors.items():
            dtype, shape, offset_start, size, remote_safetensor_url = meta
            # read the tensor data
            data = SafetensorRemote.get_data_by_range(remote_safetensor_url, offset_start, size)
            print(data)
    """

    BASE_DOMAIN = "https://huggingface.co"
    ALIGNMENT = 8 # bytes

    @classmethod
    def get_list_tensors_hf_model(cls, model_id: str) -> dict[str, RemoteTensor]:
        """
```

```
    Get list of tensors from a Hugging Face model repository.

    Returns a dictionary of tensor names and their metadata.
    Each tensor is represented as a tuple of (dtype, shape, offset_start, size, remote_safetensor_url)
    """
    # case 1: model has only one single model.safetensor file
    is_single_file = cls.check_file_exist(f"{cls.BASE_DOMAIN}/{model_id}/resolve/main/model.safetensors")
    if is_single_file:
        url = f"{cls.BASE_DOMAIN}/{model_id}/resolve/main/model.safetensors"
        return cls.get_list_tensors(url)

    # case 2: model has multiple files
    index_url = f"{cls.BASE_DOMAIN}/{model_id}/resolve/main/model.safetensors.index.json"
    is_multiple_files = cls.check_file_exist(index_url)
    if is_multiple_files:
        # read the index file
        index_data = cls.get_data_by_range(index_url, 0)
        index_str = index_data.decode('utf-8')
        index_json = json.loads(index_str)
        assert index_json.get("weight_map") is not None, "weight_map not found in index file"
        weight_map = index_json["weight_map"]
        # get the list of files
        all_files = list(set(weight_map.values()))
        all_files.sort() # make sure we load shard files in order
        # get the list of tensors
        tensors: dict[str, RemoteTensor] = {}
        for file in all_files:
            url = f"{cls.BASE_DOMAIN}/{model_id}/resolve/main/{file}"
            for key, val in cls.get_list_tensors(url).items():
                tensors[key] = val
        return tensors

    raise ValueError(f"Model {model_id} does not have any safetensor files")

@classmethod
def get_list_tensors(cls, url: str) -> dict[str, RemoteTensor]:
    """
    Get list of tensors from a remote safetensor file.

    Returns a dictionary of tensor names and their metadata.
    Each tensor is represented as a tuple of (dtype, shape, offset_start, size)
    """
    metadata, data_start_offset = cls.get_metadata(url)
    res: dict[str, RemoteTensor] = {}

    for name, meta in metadata.items():
        if name == "__metadata__":
            continue
        if not isinstance(meta, dict):
            raise ValueError(f"Invalid metadata for tensor '{name}': {meta}")
        try:
            dtype = meta["dtype"]
            shape = meta["shape"]
            offset_start_relative, offset_end_relative = meta["data_offsets"]
```

```
                size = offset_end_relative - offset_start_relative
                offset_start = data_start_offset + offset_start_relative
                res[name] = RemoteTensor(dtype=dtype, shape=tuple(shape), offset_start=offset_start, size=size,
url=url)
            except KeyError as e:
                raise ValueError(f"Missing key in metadata for tensor '{name}': {e}, meta = {meta}")

        return res

    @classmethod
    def get_metadata(cls, url: str) -> tuple[dict, int]:
        """
        Get JSON metadata from a remote safetensor file.

        Returns tuple of (metadata, data_start_offset)
        """
        # Request first 5MB of the file (hopefully enough for metadata)
        read_size = 5 * 1024 * 1024
        raw_data = cls.get_data_by_range(url, 0, read_size)

        # Parse header
        # First 8 bytes contain the metadata length as u64 little-endian
        if len(raw_data) < 8:
            raise ValueError("Not enough data to read metadata size")
        metadata_length = int.from_bytes(raw_data[:8], byteorder='little')

        # Calculate the data start offset
        data_start_offset = 8 + metadata_length
        alignment = SafetensorRemote.ALIGNMENT
        if data_start_offset % alignment != 0:
            data_start_offset += alignment - (data_start_offset % alignment)

        # Check if we have enough data to read the metadata
        if len(raw_data) < 8 + metadata_length:
                raise ValueError(f"Could not read complete metadata. Need {8 + metadata_length} bytes, got
{len(raw_data)}")

        # Extract metadata bytes and parse as JSON
        metadata_bytes = raw_data[8:8 + metadata_length]
        metadata_str = metadata_bytes.decode('utf-8')
        try:
            metadata = json.loads(metadata_str)
            return metadata, data_start_offset
        except json.JSONDecodeError as e:
            raise ValueError(f"Failed to parse safetensor metadata as JSON: {e}")

    @classmethod
    def get_data_by_range(cls, url: str, start: int, size: int = -1) -> bytes:
        """
        Get raw byte data from a remote file by range.
        If size is not specified, it will read the entire file.
        """
        import requests
        from urllib.parse import urlparse
```

```
        parsed_url = urlparse(url)
        if not parsed_url.scheme or not parsed_url.netloc:
            raise ValueError(f"Invalid URL: {url}")

        headers = cls._get_request_headers()
        if size > -1:
            headers["Range"] = f"bytes={start}-{start + size}"
        response = requests.get(url, allow_redirects=True, headers=headers)
        response.raise_for_status()

        # Get raw byte data
        return response.content[:size]

    @classmethod
    def check_file_exist(cls, url: str) -> bool:
        """
        Check if a file exists at the given URL.
        Returns True if the file exists, False otherwise.
        """
        import requests
        from urllib.parse import urlparse

        parsed_url = urlparse(url)
        if not parsed_url.scheme or not parsed_url.netloc:
            raise ValueError(f"Invalid URL: {url}")

        try:
            headers = cls._get_request_headers()
            headers["Range"] = "bytes=0-0"
            response = requests.head(url, allow_redirects=True, headers=headers)
            # Success (2xx) or redirect (3xx)
            return 200 <= response.status_code < 400
        except requests.RequestException:
            return False

    @classmethod
    def _get_request_headers(cls) -> dict[str, str]:
        """Prepare common headers for requests."""
        headers = {"User-Agent": "convert_hf_to_gguf"}
        if os.environ.get("HF_TOKEN"):
            headers["Authorization"] = f"Bearer {os.environ['HF_TOKEN']}"
        return headers

==== utils.py ====
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# type: ignore[reportUnusedImport]

import subprocess
import os
import re
import json
```

```
import sys
import requests
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
from typing import (
    Any,
    Callable,
    ContextManager,
    Iterable,
    Iterator,
    List,
    Literal,
    Tuple,
    Set,
)
from re import RegexFlag
import wget


DEFAULT_HTTP_TIMEOUT = 12

if "LLAMA_SANITIZE" in os.environ or "GITHUB_ACTION" in os.environ:
    DEFAULT_HTTP_TIMEOUT = 30


class ServerResponse:
    headers: dict
    status_code: int
    body: dict | Any


class ServerProcess:
    # default options
    debug: bool = False
    server_port: int = 8080
    server_host: str = "127.0.0.1"
    model_hf_repo: str = "ggml-org/models"
    model_hf_file: str | None = "tinyllamas/stories260K.gguf"
    model_alias: str = "tinyllama-2"
    temperature: float = 0.8
    seed: int = 42

    # custom options
    model_alias: str | None = None
    model_url: str | None = None
    model_file: str | None = None
    model_draft: str | None = None
    n_threads: int | None = None
    n_gpu_layer: int | None = None
    n_batch: int | None = None
    n_ubatch: int | None = None
    n_ctx: int | None = None
    n_ga: int | None = None
    n_ga_w: int | None = None
```

```
n_predict: int | None = None
n_prompts: int | None = 0
slot_save_path: str | None = None
id_slot: int | None = None
cache_prompt: bool | None = None
n_slots: int | None = None
ctk: str | None = None
ctv: str | None = None
fa: bool | None = None
server_continuous_batching: bool | None = False
server_embeddings: bool | None = False
server_reranking: bool | None = False
server_metrics: bool | None = False
server_slots: bool | None = False
pooling: str | None = None
draft: int | None = None
api_key: str | None = None
lora_files: List[str] | None = None
disable_ctx_shift: int | None = False
draft_min: int | None = None
draft_max: int | None = None
no_webui: bool | None = None
jinja: bool | None = None
reasoning_format: Literal['deepseek', 'none'] | None = None
chat_template: str | None = None
chat_template_file: str | None = None
server_path: str | None = None

# session variables
process: subprocess.Popen | None = None

def __init__(self):
    if "N_GPU_LAYERS" in os.environ:
        self.n_gpu_layer = int(os.environ["N_GPU_LAYERS"])
    if "DEBUG" in os.environ:
        self.debug = True
    if "PORT" in os.environ:
        self.server_port = int(os.environ["PORT"])

def start(self, timeout_seconds: int | None = DEFAULT_HTTP_TIMEOUT) -> None:
    if self.server_path is not None:
        server_path = self.server_path
    elif "LLAMA_SERVER_BIN_PATH" in os.environ:
        server_path = os.environ["LLAMA_SERVER_BIN_PATH"]
    elif os.name == "nt":
        server_path = "../../../build/bin/Release/llama-server.exe"
    else:
        server_path = "../../../build/bin/llama-server"
    server_args = [
        "--host",
        self.server_host,
        "--port",
        self.server_port,
        "--temp",
```

```
    self.temperature,
    "--seed",
    self.seed,
]
if self.model_file:
    server_args.extend(["--model", self.model_file])
if self.model_url:
    server_args.extend(["--model-url", self.model_url])
if self.model_draft:
    server_args.extend(["--model-draft", self.model_draft])
if self.model_hf_repo:
    server_args.extend(["--hf-repo", self.model_hf_repo])
if self.model_hf_file:
    server_args.extend(["--hf-file", self.model_hf_file])
if self.n_batch:
    server_args.extend(["--batch-size", self.n_batch])
if self.n_ubatch:
    server_args.extend(["--ubatch-size", self.n_ubatch])
if self.n_threads:
    server_args.extend(["--threads", self.n_threads])
if self.n_gpu_layer:
    server_args.extend(["--n-gpu-layers", self.n_gpu_layer])
if self.draft is not None:
    server_args.extend(["--draft", self.draft])
if self.server_continuous_batching:
    server_args.append("--cont-batching")
if self.server_embeddings:
    server_args.append("--embedding")
if self.server_reranking:
    server_args.append("--reranking")
if self.server_metrics:
    server_args.append("--metrics")
if self.server_slots:
    server_args.append("--slots")
if self.pooling:
    server_args.extend(["--pooling", self.pooling])
if self.model_alias:
    server_args.extend(["--alias", self.model_alias])
if self.n_ctx:
    server_args.extend(["--ctx-size", self.n_ctx])
if self.n_slots:
    server_args.extend(["--parallel", self.n_slots])
if self.ctk:
    server_args.extend(["-ctk", self.ctk])
if self.ctv:
    server_args.extend(["-ctv", self.ctv])
if self.fa is not None:
    server_args.append("-fa")
if self.n_predict:
    server_args.extend(["--n-predict", self.n_predict])
if self.slot_save_path:
    server_args.extend(["--slot-save-path", self.slot_save_path])
if self.n_ga:
    server_args.extend(["--grp-attn-n", self.n_ga])
```

```
if self.n_ga_w:
    server_args.extend(["--grp-attn-w", self.n_ga_w])
if self.debug:
    server_args.append("--verbose")
if self.lora_files:
    for lora_file in self.lora_files:
        server_args.extend(["--lora", lora_file])
if self.disable_ctx_shift:
    server_args.extend(["--no-context-shift"])
if self.api_key:
    server_args.extend(["--api-key", self.api_key])
if self.draft_max:
    server_args.extend(["--draft-max", self.draft_max])
if self.draft_min:
    server_args.extend(["--draft-min", self.draft_min])
if self.no_webui:
    server_args.append("--no-webui")
if self.jinja:
    server_args.append("--jinja")
if self.reasoning_format is not None:
    server_args.extend(("--reasoning-format", self.reasoning_format))
if self.chat_template:
    server_args.extend(["--chat-template", self.chat_template])
if self.chat_template_file:
    server_args.extend(["--chat-template-file", self.chat_template_file])

args = [str(arg) for arg in [server_path, *server_args]]
print(f"tests: starting server with: {' '.join(args)}")

flags = 0
if "nt" == os.name:
    flags |= subprocess.DETACHED_PROCESS
    flags |= subprocess.CREATE_NEW_PROCESS_GROUP
    flags |= subprocess.CREATE_NO_WINDOW

self.process = subprocess.Popen(
    [str(arg) for arg in [server_path, *server_args]],
    creationflags=flags,
    stdout=sys.stdout,
    stderr=sys.stdout,
    env={**os.environ, "LLAMA_CACHE": "tmp"} if "LLAMA_CACHE" not in os.environ else None,
)
server_instances.add(self)

print(f"server pid={self.process.pid}, pytest pid={os.getpid()}")

# wait for server to start
start_time = time.time()
while time.time() - start_time < timeout_seconds:
    try:
        response = self.make_request("GET", "/health", headers={
            "Authorization": f"Bearer {self.api_key}" if self.api_key else None
        })
        if response.status_code == 200:
```

```
                self.ready = True
                return  # server is ready
        except Exception as e:
            pass
        # Check if process died
        if self.process.poll() is not None:
            raise RuntimeError(f"Server process died with return code {self.process.returncode}")

        print(f"Waiting for server to start...")
        time.sleep(0.5)
    raise TimeoutError(f"Server did not start within {timeout_seconds} seconds")

def stop(self) -> None:
    if self in server_instances:
        server_instances.remove(self)
    if self.process:
        print(f"Stopping server with pid={self.process.pid}")
        self.process.kill()
        self.process = None

def make_request(
    self,
    method: str,
    path: str,
    data: dict | Any | None = None,
    headers: dict | None = None,
    timeout: float | None = None,
) -> ServerResponse:
    url = f"http://{self.server_host}:{self.server_port}{path}"
    parse_body = False
    if method == "GET":
        response = requests.get(url, headers=headers, timeout=timeout)
        parse_body = True
    elif method == "POST":
        response = requests.post(url, headers=headers, json=data, timeout=timeout)
        parse_body = True
    elif method == "OPTIONS":
        response = requests.options(url, headers=headers, timeout=timeout)
    else:
        raise ValueError(f"Unimplemented method: {method}")
    result = ServerResponse()
    result.headers = dict(response.headers)
    result.status_code = response.status_code
    result.body = response.json() if parse_body else None
    print("Response from server", json.dumps(result.body, indent=2))
    return result

def make_stream_request(
    self,
    method: str,
    path: str,
    data: dict | None = None,
    headers: dict | None = None,
) -> Iterator[dict]:
```

```
        url = f"http://{self.server_host}:{self.server_port}{path}"
        if method == "POST":
            response = requests.post(url, headers=headers, json=data, stream=True)
        else:
            raise ValueError(f"Unimplemented method: {method}")
        for line_bytes in response.iter_lines():
            line = line_bytes.decode("utf-8")
            if '[DONE]' in line:
                break
            elif line.startswith('data: '):
                data = json.loads(line[6:])
                print("Partial response from server", json.dumps(data, indent=2))
                yield data


server_instances: Set[ServerProcess] = set()


class ServerPreset:
    @staticmethod
    def tinyllama2() -> ServerProcess:
        server = ServerProcess()
        server.model_hf_repo = "ggml-org/models"
        server.model_hf_file = "tinyllamas/stories260K.gguf"
        server.model_alias = "tinyllama-2"
        server.n_ctx = 512
        server.n_batch = 32
        server.n_slots = 2
        server.n_predict = 64
        server.seed = 42
        return server

    @staticmethod
    def bert_bge_small() -> ServerProcess:
        server = ServerProcess()
        server.model_hf_repo = "ggml-org/models"
        server.model_hf_file = "bert-bge-small/ggml-model-f16.gguf"
        server.model_alias = "bert-bge-small"
        server.n_ctx = 512
        server.n_batch = 128
        server.n_ubatch = 128
        server.n_slots = 2
        server.seed = 42
        server.server_embeddings = True
        return server

    @staticmethod
    def bert_bge_small_with_fa() -> ServerProcess:
        server = ServerProcess()
        server.model_hf_repo = "ggml-org/models"
        server.model_hf_file = "bert-bge-small/ggml-model-f16.gguf"
        server.model_alias = "bert-bge-small"
        server.n_ctx = 1024
        server.n_batch = 300
```

```
        server.n_ubatch = 300
        server.n_slots = 2
        server.fa = True
        server.seed = 42
        server.server_embeddings = True
        return server

    @staticmethod
    def tinyllama_infill() -> ServerProcess:
        server = ServerProcess()
        server.model_hf_repo = "ggml-org/models"
        server.model_hf_file = "tinyllamas/stories260K-infill.gguf"
        server.model_alias = "tinyllama-infill"
        server.n_ctx = 2048
        server.n_batch = 1024
        server.n_slots = 1
        server.n_predict = 64
        server.temperature = 0.0
        server.seed = 42
        return server

    @staticmethod
    def stories15m_moe() -> ServerProcess:
        server = ServerProcess()
        server.model_hf_repo = "ggml-org/stories15M_MOE"
        server.model_hf_file = "stories15M_MOE-F16.gguf"
        server.model_alias = "stories15m-moe"
        server.n_ctx = 2048
        server.n_batch = 1024
        server.n_slots = 1
        server.n_predict = 64
        server.temperature = 0.0
        server.seed = 42
        return server

    @staticmethod
    def jina_reranker_tiny() -> ServerProcess:
        server = ServerProcess()
        server.model_hf_repo = "ggml-org/models"
        server.model_hf_file = "jina-reranker-v1-tiny-en/ggml-model-f16.gguf"
        server.model_alias = "jina-reranker"
        server.n_ctx = 512
        server.n_batch = 512
        server.n_slots = 1
        server.seed = 42
        server.server_reranking = True
        return server


def parallel_function_calls(function_list: List[Tuple[Callable[..., Any], Tuple[Any, ...]]]) -> List[Any]:
    """
    Run multiple functions in parallel and return results in the same order as calls. Equivalent to Promise.all
in JS.
```

```
    Example usage:

    results = parallel_function_calls([
        (func1, (arg1, arg2)),
        (func2, (arg3, arg4)),
    ])
    """
    results = [None] * len(function_list)
    exceptions = []

    def worker(index, func, args):
        try:
            result = func(*args)
            results[index] = result
        except Exception as e:
            exceptions.append((index, str(e)))

    with ThreadPoolExecutor() as executor:
        futures = []
        for i, (func, args) in enumerate(function_list):
            future = executor.submit(worker, i, func, args)
            futures.append(future)

        # Wait for all futures to complete
        for future in as_completed(futures):
            pass

    # Check if there were any exceptions
    if exceptions:
        print("Exceptions occurred:")
        for index, error in exceptions:
            print(f"Function at index {index}: {error}")

    return results


def match_regex(regex: str, text: str) -> bool:
    return (
        re.compile(
            regex, flags=RegexFlag.IGNORECASE | RegexFlag.MULTILINE | RegexFlag.DOTALL
        ).search(text)
        is not None
    )


def download_file(url: str, output_file_path: str | None = None) -> str:
    """
    Download a file from a URL to a local path. If the file already exists, it will not be downloaded again.

    output_file_path is the local path to save the downloaded file. If not provided, the file will be saved in
the root directory.

    Returns the local path of the downloaded file.
    """
```

```
    file_name = url.split('/').pop()
    output_file = f'./tmp/{file_name}' if output_file_path is None else output_file_path
    if not os.path.exists(output_file):
        print(f"Downloading {url} to {output_file}")
        wget.download(url, out=output_file)
        print(f"Done downloading to {output_file}")
    else:
        print(f"File already exists at {output_file}")
    return output_file


def is_slow_test_allowed():
    return os.environ.get("SLOW_TESTS") == "1" or os.environ.get("SLOW_TESTS") == "ON"

==== utils_conceptrule.py ====
"""Data utils for logic-memnn."""
import json
import socket
import numpy as np
import json_lines
import re

import keras.callbacks as C
from keras.utils import Sequence
from keras.preprocessing.sequence import pad_sequences

from data_gen import CHAR_IDX
from word_dict_gen_conceptrule import WORD_INDEX
import os
import random

class LogicSeq(Sequence):
  """Sequence generator for normal logic programs."""
  def __init__(self, datasets, batch_size, train=True,
               shuffle=True, pad=False, zeropad=True):
    self.datasets = datasets or [[]]
    # We distribute batch evenly so it must divide the batc size
    assert batch_size % len(self.datasets) == 0, "Number of datasets must divide batch size."
    self.batch_size = batch_size
    self.train = train
    self.shuffle = shuffle
    self.pad = pad
    self.zeropad = zeropad
    seed_value = 0
    os.environ['PYTHONHASHSEED'] = str(seed_value)
    random.seed(seed_value)
    np.random.seed(seed_value)

  def __len__(self):
    return int(np.ceil(sum(map(len, self.datasets))/ self.batch_size))

  def on_epoch_end(self):
    """Shuffle data at the end of epoch."""
    if self.shuffle:
```

```
    for ds in self.datasets:
      np.random.shuffle(ds)

def __getitem__(self, idx):
  dpoints = list()
  per_ds_bs = self.batch_size//len(self.datasets)
  for ds in self.datasets:
    dpoints.extend(ds[idx*per_ds_bs:(idx+1)*per_ds_bs])
  # Create batch
  ctxs, queries, targets = list(), list(), list()
  for ctx, q, t in dpoints:
    if self.shuffle:
      np.random.shuffle(ctx)
    # rules = [r.replace(':-', '.').replace(';', '.').split('.')[:-1]
    #          for r in ctx]
    rules = []
    for r in ctx:
      result = []
      result.append(r)
      rules.append(result)
    # if self.pad:
    #   rules.append(['()']) # Append blank rule
    # if self.zeropad:  pred.split(" ")   q.split(" ")   filter_data(re.split(r"[\s]", q))
    #   rules.append(['']) # Append null sentinel filter_data(re.split(r"[\s]", pred))

    rules = [[[WORD_INDEX[c] for c in filter_data(re.split(r"[\s]", pred))]
              for pred in r]
             for r in rules]
    ctxs.append(rules)
    queries.append([WORD_INDEX[c] for c in filter_data(re.split(r"[\s]", q))]) # Remove '.' at the end
    targets.append(t)
  vctxs = np.zeros((len(dpoints),
                    max([len(rs) for rs in ctxs]),
                    max([len(ps) for rs in ctxs for ps in rs]),
                    max([len(cs) for rs in ctxs for ps in rs for cs in ps])),
                   dtype='int')
  # Contexts
  for i in range(len(dpoints)):
    # Rules in context (ie program)
    for j in range(len(ctxs[i])):
      # Predicates in rules
      for k in range(len(ctxs[i][j])):
        # Chars in predicates
        for l in range(len(ctxs[i][j][k])):
          vctxs[i, j, k, l] = ctxs[i][j][k][l]
  xs = [vctxs, pad_sequences(queries, padding='post')]
  if self.train:
    return xs, np.array(targets)
  return xs

@staticmethod
def parse_file(fname, shuffle=True):
  """Parse logic program data given fname."""
  dpoints = list()
```

```
  with open(fname) as f:
    for l in json_lines.reader(f):
      ctx = list()
      questions = l["questions"]
      context = l["context"].replace("\n", " ")
      context = context.replace(",", "")
      context = context.replace("!", "")
      context = context.replace("\\", "")
      #context = context.replace(".", "")
      context = re.sub(r'\s+', ' ', context)
      context = context.lower()
      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)
          q = q.replace('.','')
          q = q.replace('!', '')
          q = q.replace(',', '')
          q = q.replace('\\', '')
          #ctx = context.split(".")
          context = context.replace('ph.d.','phd')
          context = context.replace('t.v.', 'tv')
          q = q.replace('ph.d.','phd')
          q = q.replace('t.v.', 'tv')
          ctx = filter_data(re.split(r"[.]", context))
          q = q.lower()
          #ctx = re.split(r"([.])", context)
          #ctx = ["".join(i) for i in zip(ctx[0::2], ctx[1::2])]
          dpoints.append((ctx, q, int(t)))
  if shuffle:
    np.random.shuffle(dpoints)
  return dpoints

@classmethod
def from_file(cls, fname, batch_size, pad=False, verbose=True):
  """Load logic programs from given fname."""
  dpoints = cls.parse_file(fname)
  if verbose:
    print("Example data points from:", fname)
    print(dpoints[:4])
  return cls([dpoints], batch_size, pad=pad)

@classmethod
def from_files(cls, fnames, batch_size, pad=False, verbose=True):
  """Load several logic program files return a singel sequence generator."""
  datasets = [cls.parse_file(f) for f in fnames]
  if verbose:
    print("Loaded files:", fnames)
  return cls(datasets, batch_size, pad=pad)
```

```
class ThresholdStop(C.Callback):
  """Stop when monitored value is greater than threshold."""
  def __init__(self, monitor='val_acc', threshold=1):
    super().__init__()
    self.monitor = monitor
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    current = logs.get(self.monitor)
    if current >= self.threshold:
      self.model.stop_training = True


class StatefulCheckpoint(C.ModelCheckpoint):
  """Save extra checkpoint data to resume training."""
  def __init__(self, weight_file, state_file=None, **kwargs):
    """Save the state (epoch etc.) along side weights."""
    super().__init__(weight_file, **kwargs)
    self.state_f = state_file
    self.hostname = socket.gethostname()
    self.state = dict()
    if self.state_f:
      # Load the last state if any
      try:
        with open(self.state_f, 'r') as f:
          self.state = json.load(f)
        self.best = self.state['best']
      except Exception as e: # pylint: disable=broad-except
        print("Skipping last state:", e)

  def on_train_begin(self, logs=None):
    prefix = "Resuming" if self.state else "Starting"
    print("{} training on {}".format(prefix, self.hostname))

  def on_epoch_end(self, epoch, logs=None):
    """Saves training state as well as weights."""
    super().on_epoch_end(epoch, logs)
    if self.state_f:
      state = {'epoch': epoch+1, 'best': self.best,
               'hostname': self.hostname}
      state.update(logs)
      state.update(self.params)
      with open(self.state_f, 'w') as f:
        json.dump(state, f)

  def get_last_epoch(self, initial_epoch=0):
    """Return last saved epoch if any, or return default argument."""
    return self.state.get('epoch', initial_epoch)

  def on_train_end(self, logs=None):
    print("Training ending on {}".format(self.hostname))
```

```
# 2????split?list???
# filter_data()???string?list [str1, str2, str3, ......]???''?'\n'??list
def not_break(sen):
  return (sen != '\n' and sen != '\u3000' and sen != '' and not sen.isspace())

def filter_data(ini_data):
  # ini_data??string
  new_data = list(filter(not_break, [data.strip() for data in ini_data]))
  return new_data

==== utils_conceptrule_csv.py ====
"""Data utils for logic-memnn."""
import json
import socket
import numpy as np
import json_lines
import re

import keras.callbacks as C
from keras.utils import Sequence
from keras.preprocessing.sequence import pad_sequences

from data_gen import CHAR_IDX
from word_dict_gen_conceptrule import WORD_INDEX
import pandas as pd
import os
import random



class LogicSeq(Sequence):
  """Sequence generator for normal logic programs."""
  def __init__(self, datasets, batch_size, train=True,
               shuffle=True, pad=False, zeropad=True):
    self.datasets = datasets or [[]]
    # We distribute batch evenly so it must divide the batc size
    assert batch_size % len(self.datasets) == 0, "Number of datasets must divide batch size."
    self.batch_size = batch_size
    self.train = train
    self.shuffle = shuffle
    self.pad = pad
    self.zeropad = zeropad
    seed_value = 0
    os.environ['PYTHONHASHSEED'] = str(seed_value)
    random.seed(seed_value)
    np.random.seed(seed_value)

  def __len__(self):
    return int(np.ceil(sum(map(len, self.datasets))/ self.batch_size))

  def on_epoch_end(self):
    """Shuffle data at the end of epoch."""
    if self.shuffle:
      for ds in self.datasets:
        np.random.shuffle(ds)
```

```
def __getitem__(self, idx):
  dpoints = list()
  per_ds_bs = self.batch_size//len(self.datasets)
  for ds in self.datasets:
    dpoints.extend(ds[idx*per_ds_bs:(idx+1)*per_ds_bs])
  # Create batch
  ctxs, queries, targets = list(), list(), list()
  for ctx, q, t in dpoints:
    if self.shuffle:
      np.random.shuffle(ctx)
    # rules = [r.replace(':-', '.').replace(';', '.').split('.')[:-1]
    #          for r in ctx]
    rules = []
    for r in ctx:
      result = []
      result.append(r)
      rules.append(result)
    # if self.pad:
    #   rules.append(['()']) # Append blank rule
    # if self.zeropad:  pred.split(" ")   q.split(" ")   filter_data(re.split(r"[\s]", q))
    #   rules.append(['']) # Append null sentinel filter_data(re.split(r"[\s]", pred))

    rules = [[[WORD_INDEX[c] for c in filter_data(re.split(r"[\s]", pred))]
              for pred in r]
             for r in rules]
    ctxs.append(rules)
    queries.append([WORD_INDEX[c] for c in filter_data(re.split(r"[\s]", q))]) # Remove '.' at the end
    targets.append(t)
  vctxs = np.zeros((len(dpoints),
                    max([len(rs) for rs in ctxs]),
                    max([len(ps) for rs in ctxs for ps in rs]),
                    max([len(cs) for rs in ctxs for ps in rs for cs in ps])),
                   dtype='int')
  # Contexts
  for i in range(len(dpoints)):
    # Rules in context (ie program)
    for j in range(len(ctxs[i])):
      # Predicates in rules
      for k in range(len(ctxs[i][j])):
        # Chars in predicates
        for l in range(len(ctxs[i][j][k])):
          vctxs[i, j, k, l] = ctxs[i][j][k][l]
  xs = [vctxs, pad_sequences(queries, padding='post')]
  if self.train:
    return xs, np.array(targets)
  return xs

@staticmethod
def parse_file(fname, shuffle=True):
  """Parse logic program data given fname."""
  dpoints = list()
  with open(fname) as f:
    data = pd.read_csv(fname, sep='\t', header=0)
```

```
      for index, row in data.iterrows():
        question = re.sub(r'\s+', ' ', row[2])
        question = question.replace('.','')
        question = question.replace('!','')
        question = question.replace(',','')
        question = question.replace('\\', '')
        question = question.replace('ph.d.', 'phd')
        question = question.replace('t.v.', 'tv')
        question = question.lower()

        context = row[1].replace("\n", " ")
        context = context.replace("\\", "")
        context = context.replace('ph.d.', 'phd')
        context = context.replace('t.v.', 'tv')
        context = context.replace(",","")
        context = context.replace("!", "")
        context = re.sub(r'\s+', ' ', context)
        context = context.lower()
        context = filter_data(re.split(r"[.]", context))

        label = row[3]
        if label == True:
          label_num = 1
        else:
          label_num = 0

        dpoints.append((context, question, label_num))

    if shuffle:
      np.random.shuffle(dpoints)
    return dpoints

  @classmethod
  def from_file(cls, fname, batch_size, pad=False, verbose=True):
    """Load logic programs from given fname."""
    dpoints = cls.parse_file(fname)
    if verbose:
      print("Example data points from:", fname)
      print(dpoints[:4])
    return cls([dpoints], batch_size, pad=pad)

  @classmethod
  def from_files(cls, fnames, batch_size, pad=False, verbose=True):
    """Load several logic program files return a singel sequence generator."""
    datasets = [cls.parse_file(f) for f in fnames]
    if verbose:
      print("Loaded files:", fnames)
    return cls(datasets, batch_size, pad=pad)


class ThresholdStop(C.Callback):
  """Stop when monitored value is greater than threshold."""
  def __init__(self, monitor='val_acc', threshold=1):
    super().__init__()
```

```
    self.monitor = monitor
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    current = logs.get(self.monitor)
    if current >= self.threshold:
      self.model.stop_training = True


class StatefulCheckpoint(C.ModelCheckpoint):
  """Save extra checkpoint data to resume training."""
  def __init__(self, weight_file, state_file=None, **kwargs):
    """Save the state (epoch etc.) along side weights."""
    super().__init__(weight_file, **kwargs)
    self.state_f = state_file
    self.hostname = socket.gethostname()
    self.state = dict()
    if self.state_f:
      # Load the last state if any
      try:
        with open(self.state_f, 'r') as f:
          self.state = json.load(f)
        self.best = self.state['best']
      except Exception as e: # pylint: disable=broad-except
        print("Skipping last state:", e)

  def on_train_begin(self, logs=None):
    prefix = "Resuming" if self.state else "Starting"
    print("{} training on {}".format(prefix, self.hostname))

  def on_epoch_end(self, epoch, logs=None):
    """Saves training state as well as weights."""
    super().on_epoch_end(epoch, logs)
    if self.state_f:
      state = {'epoch': epoch+1, 'best': self.best,
               'hostname': self.hostname}
      state.update(logs)
      state.update(self.params)
      with open(self.state_f, 'w') as f:
        json.dump(state, f)

  def get_last_epoch(self, initial_epoch=0):
    """Return last saved epoch if any, or return default argument."""
    return self.state.get('epoch', initial_epoch)

  def on_train_end(self, logs=None):
    print("Training ending on {}".format(self.hostname))


# 2????split?list???
# filter_data()???string?list [str1, str2, str3, ......]???''?'\n'??list
def not_break(sen):
  return (sen != '\n' and sen != '\u3000' and sen != '' and not sen.isspace())
```

```
def filter_data(ini_data):
  # ini_data??string
  new_data = list(filter(not_break, [data.strip() for data in ini_data]))
  return new_data

==== utils_pararule.py ====
"""Data utils for logic-memnn."""
import json
import socket
import numpy as np
import json_lines
import re

import keras.callbacks as C
from keras.utils import Sequence
from keras.preprocessing.sequence import pad_sequences

from data_gen import CHAR_IDX
from word_dict_gen import WORD_INDEX
import os
import random

class LogicSeq(Sequence):
  """Sequence generator for normal logic programs."""
  def __init__(self, datasets, batch_size, train=True,
               shuffle=True, pad=False, zeropad=True):
    self.datasets = datasets or [[]]
    # We distribute batch evenly so it must divide the batc size
    assert batch_size % len(self.datasets) == 0, "Number of datasets must divide batch size."
    self.batch_size = batch_size
    self.train = train
    self.shuffle = shuffle
    self.pad = pad
    self.zeropad = zeropad
    seed_value = 0
    os.environ['PYTHONHASHSEED'] = str(seed_value)
    random.seed(seed_value)
    np.random.seed(seed_value)

  def __len__(self):
    return int(np.ceil(sum(map(len, self.datasets))/ self.batch_size))

  def on_epoch_end(self):
    """Shuffle data at the end of epoch."""
    if self.shuffle:
      for ds in self.datasets:
        np.random.shuffle(ds)

  def __getitem__(self, idx):
    dpoints = list()
    per_ds_bs = self.batch_size//len(self.datasets)
    for ds in self.datasets:
      dpoints.extend(ds[idx*per_ds_bs:(idx+1)*per_ds_bs])
    # Create batch
```

```
  ctxs, queries, targets = list(), list(), list()
  for ctx, q, t in dpoints:
    if self.shuffle:
      np.random.shuffle(ctx)
    # rules = [r.replace(':-', '.').replace(';', '.').split('.')[:-1]
    #          for r in ctx]
    rules = []
    for r in ctx:
      result = []
      result.append(r)
      rules.append(result)
    # if self.pad:
    #   rules.append(['()']) # Append blank rule
    # if self.zeropad:  pred.split(" ")   q.split(" ")   filter_data(re.split(r"[\s]", q))
    #   rules.append(['']) # Append null sentinel filter_data(re.split(r"[\s]", pred))

    rules = [[[WORD_INDEX[c] for c in filter_data(re.split(r"[\s]", pred))]
              for pred in r]
             for r in rules]
    ctxs.append(rules)
    queries.append([WORD_INDEX[c] for c in filter_data(re.split(r"[\s]", q))]) # Remove '.' at the end
    targets.append(t)
  vctxs = np.zeros((len(dpoints),
                    max([len(rs) for rs in ctxs]),
                    max([len(ps) for rs in ctxs for ps in rs]),
                    max([len(cs) for rs in ctxs for ps in rs for cs in ps])),
                   dtype='int')
  # Contexts
  for i in range(len(dpoints)):
    # Rules in context (ie program)
    for j in range(len(ctxs[i])):
      # Predicates in rules
      for k in range(len(ctxs[i][j])):
        # Chars in predicates
        for l in range(len(ctxs[i][j][k])):
          vctxs[i, j, k, l] = ctxs[i][j][k][l]
  xs = [vctxs, pad_sequences(queries, padding='post')]
  if self.train:
    return xs, np.array(targets)
  return xs

@staticmethod
def parse_file(fname, shuffle=True):
  """Parse logic program data given fname."""
  dpoints = list()
  with open(fname) as f:
    for l in json_lines.reader(f):
      ctx = list()
      questions = l["questions"]
      context = l["context"].replace("\n", " ")
      context = context.replace(",", "")
      context = context.replace("!", "")
      context = re.sub(r'\s+', ' ', context)
      context = context.lower()
```

```
        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)
            q = q.replace('.','')
            q = q.replace('!', '')
            q = q.replace(',', '')
            #ctx = context.split(".")
            ctx = filter_data(re.split(r"[.]", context))
            q = q.lower()
            #ctx = re.split(r"([.])", context)
            #ctx = ["".join(i) for i in zip(ctx[0::2], ctx[1::2])]
            dpoints.append((ctx, q, int(t)))
    if shuffle:
      np.random.shuffle(dpoints)
    return dpoints

  @classmethod
  def from_file(cls, fname, batch_size, pad=False, verbose=True):
    """Load logic programs from given fname."""
    dpoints = cls.parse_file(fname)
    if verbose:
      print("Example data points from:", fname)
      print(dpoints[:4])
    return cls([dpoints], batch_size, pad=pad)

  @classmethod
  def from_files(cls, fnames, batch_size, pad=False, verbose=True):
    """Load several logic program files return a singel sequence generator."""
    datasets = [cls.parse_file(f) for f in fnames]
    if verbose:
      print("Loaded files:", fnames)
    return cls(datasets, batch_size, pad=pad)


class ThresholdStop(C.Callback):
  """Stop when monitored value is greater than threshold."""
  def __init__(self, monitor='val_acc', threshold=1):
    super().__init__()
    self.monitor = monitor
    self.threshold = threshold

  def on_epoch_end(self, epoch, logs=None):
    current = logs.get(self.monitor)
    if current >= self.threshold:
      self.model.stop_training = True


class StatefulCheckpoint(C.ModelCheckpoint):
  """Save extra checkpoint data to resume training."""
```

```
  def __init__(self, weight_file, state_file=None, **kwargs):
    """Save the state (epoch etc.) along side weights."""
    super().__init__(weight_file, **kwargs)
    self.state_f = state_file
    self.hostname = socket.gethostname()
    self.state = dict()
    if self.state_f:
      # Load the last state if any
      try:
        with open(self.state_f, 'r') as f:
          self.state = json.load(f)
        self.best = self.state['best']
      except Exception as e: # pylint: disable=broad-except
        print("Skipping last state:", e)

  def on_train_begin(self, logs=None):
    prefix = "Resuming" if self.state else "Starting"
    print("{} training on {}".format(prefix, self.hostname))

  def on_epoch_end(self, epoch, logs=None):
    """Saves training state as well as weights."""
    super().on_epoch_end(epoch, logs)
    if self.state_f:
      state = {'epoch': epoch+1, 'best': self.best,
               'hostname': self.hostname}
      state.update(logs)
      state.update(self.params)
      with open(self.state_f, 'w') as f:
        json.dump(state, f)

  def get_last_epoch(self, initial_epoch=0):
    """Return last saved epoch if any, or return default argument."""
    return self.state.get('epoch', initial_epoch)

  def on_train_end(self, logs=None):
    print("Training ending on {}".format(self.hostname))



# 2????split?list???
# filter_data()???string?list [str1, str2, str3, ......]???''?'\n'??list
def not_break(sen):
  return (sen != '\n' and sen != '\u3000' and sen != '' and not sen.isspace())

def filter_data(ini_data):
  # ini_data??string
  new_data = list(filter(not_break, [data.strip() for data in ini_data]))
  return new_data

==== validator.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: validator.py
Purpose: Compare symbolic beliefs, detect contradictions, write to overflow
"""
```

```
import os
import yaml
import redis
r = redis.Redis(decode_responses=True)

import time
import hashlib
import threading
import redis
from pathlib import Path

# Local module
from utils import agent_profiler

# Redis pub/sub for symbolic event broadcasts
r = redis.Redis(decode_responses=True)

# Start profiler as background daemon
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()



CORE_DIR = Path("fragments/core")
OVERFLOW_DIR = Path("fragments/overflow")
OVERFLOW_DIR.mkdir(parents=True, exist_ok=True)

class Validator:
    def __init__(self, agent_id="validator_01"):
        self.agent_id = agent_id
        self.frags = {}

    def hash_claim(self, claim):
        return hashlib.md5(claim.encode("utf-8")).hexdigest()

    def load_core_beliefs(self):
        for path in CORE_DIR.glob("*.yaml"):
            with open(path, 'r', encoding='utf-8') as file:
                try:
                    frag = yaml.safe_load(file)
                    if frag and 'claim' in frag:
                        claim_hash = self.hash_claim(frag['claim'])
                        self.frags[claim_hash] = (path, frag)
                except yaml.YAMLError as e:
                    print(f"[{self.agent_id}] YAML error in {path.name}: {e}")

    def contradicts(self, a, b):
        # Naive contradiction check: exact negation
        return a.lower().strip() == f"not {b.lower().strip()}"

    def run_validation(self):
        for hash_a, (path_a, frag_a) in self.frags.items():
            for hash_b, (path_b, frag_b) in self.frags.items():
                if hash_a == hash_b:
                    continue
```

```
                if self.contradicts(frag_a['claim'], frag_b['claim']):
                    contradiction_id = f"{hash_a[:6]}_{hash_b[:6]}"
                    filename = f"contradiction_{contradiction_id}.yaml"
                    contradiction_path = OVERFLOW_DIR / filename
                    if not contradiction_path.exists():
                        with open(contradiction_path, 'w', encoding='utf-8') as out:
                            yaml.safe_dump({
                                'source_1': frag_a['claim'],
                                'source_2': frag_b['claim'],
                                'path_1': str(path_a),
                                'path_2': str(path_b),
                                'detected_by': self.agent_id,
                                'timestamp': int(time.time())
                            }, out)
r.publish("contradiction_found", payload['claim_1'])  # [AUTO_EMIT]
                        send_message({
                            'from': self.agent_id,
                            'type': 'contradiction_found',
                            'payload': {
                                'claim_1': frag_a['claim'],
                                'claim_2': frag_b['claim'],
                                'paths': [str(path_a), str(path_b)]
                            },
                            'timestamp': int(time.time())
                        })

    def run(self):
        self.load_core_beliefs()
        self.run_validation()

if __name__ == "__main__":
    Validator().run()
# [CONFIG_PATCHED]

==== verify-checksum-models.py ====
#!/usr/bin/env python3

import logging
import os
import hashlib

logger = logging.getLogger("verify-checksum-models")


def sha256sum(file):
    block_size = 16 * 1024 * 1024  # 16 MB block size
    b = bytearray(block_size)
    file_hash = hashlib.sha256()
    mv = memoryview(b)
    with open(file, 'rb', buffering=0) as f:
        while True:
            n = f.readinto(mv)
            if not n:
                break
```

```
            file_hash.update(mv[:n])

    return file_hash.hexdigest()


# Define the path to the llama directory (parent folder of script directory)
llama_path = os.path.abspath(os.path.join(os.path.dirname(__file__), os.pardir))

# Define the file with the list of hashes and filenames
hash_list_file = os.path.join(llama_path, "SHA256SUMS")

# Check if the hash list file exists
if not os.path.exists(hash_list_file):
    logger.error(f"Hash list file not found: {hash_list_file}")
    exit(1)

# Read the hash file content and split it into an array of lines
with open(hash_list_file, "r") as f:
    hash_list = f.read().splitlines()

# Create an array to store the results
results = []

# Loop over each line in the hash list
for line in hash_list:
    # Split the line into hash and filename
    hash_value, filename = line.split("  ")

    # Get the full path of the file by joining the llama path and the filename
    file_path = os.path.join(llama_path, filename)

    # Informing user of the progress of the integrity check
    logger.info(f"Verifying the checksum of {file_path}")

    # Check if the file exists
    if os.path.exists(file_path):
        # Calculate the SHA256 checksum of the file using hashlib
        file_hash = sha256sum(file_path)

        # Compare the file hash with the expected hash
        if file_hash == hash_value:
            valid_checksum = "V"
            file_missing = ""
        else:
            valid_checksum = ""
            file_missing = ""
    else:
        valid_checksum = ""
        file_missing = "X"

    # Add the results to the array
    results.append({
        "filename": filename,
        "valid checksum": valid_checksum,
```

```
        "file missing": file_missing
    })


# Print column headers for results table
print("filename".ljust(40) + "valid checksum".center(20) + "file missing".center(20)) # noqa: NP100
print("-" * 80) # noqa: NP100

# Output the results as a table
for r in results:
    print(f"{r['filename']:40} {r['valid checksum']:^20} {r['file missing']:^20}") # noqa: NP100

==== vocab.py ====
from __future__ import annotations

import re
import logging
import json
import os
from pathlib import Path
from typing import Any, Callable, Sequence, Mapping, Iterable, Protocol, ClassVar, runtime_checkable

from sentencepiece import SentencePieceProcessor

import gguf

from .gguf_writer import GGUFWriter

logger = logging.getLogger(__name__)


class SpecialVocab:
    merges: list[str]
    add_special_token: dict[str, bool]
    special_token_ids: dict[str, int]
    chat_template: str | Sequence[Mapping[str, str]] | None

    def __init__(
        self, path: str | os.PathLike[str], load_merges: bool = False,
        special_token_types: Iterable[str] | None = None,
        n_vocab: int | None = None,
    ):
        self.special_token_ids = {}
        self.add_special_token = {}
        self.n_vocab = n_vocab
        self.load_merges = load_merges
        self.merges = []
        self.chat_template = None
        if special_token_types is not None:
            self.special_token_types = special_token_types
        else:
            self.special_token_types = ('bos', 'eos', 'unk', 'sep', 'pad', 'cls', 'mask')
        self._load(Path(path))
```

```
def __repr__(self) -> str:
    return '<SpecialVocab with {} merges, special tokens {}, add special tokens {}>'.format(
        len(self.merges), self.special_token_ids or "unset", self.add_special_token or "unset",
    )

def add_to_gguf(self, gw: GGUFWriter, quiet: bool = False) -> None:
    if self.merges:
        if not quiet:
            logger.info(f'Adding {len(self.merges)} merge(s).')
        gw.add_token_merges(self.merges)
    elif self.load_merges:
        logger.warning('Adding merges requested but no merges found, output may be non-functional.')
    for typ, tokid in self.special_token_ids.items():
        id_handler: Callable[[int], None] | None = getattr(gw, f'add_{typ}_token_id', None)
        if id_handler is None:
            logger.warning(f'No handler for special token type {typ} with id {tokid} - skipping')
            continue
        if not quiet:
            logger.info(f'Setting special token type {typ} to {tokid}')
        id_handler(tokid)
    for typ, value in self.add_special_token.items():
        add_handler: Callable[[bool], None] | None = getattr(gw, f'add_add_{typ}_token', None)
        if add_handler is None:
            logger.warning(f'No handler for add_{typ}_token with value {value} - skipping')
            continue
        if not quiet:
            logger.info(f'Setting add_{typ}_token to {value}')
        add_handler(value)
    if self.chat_template is not None:
        if not quiet:
            logger.info(f'Setting chat_template to {self.chat_template}')
        gw.add_chat_template(self.chat_template)

def _load(self, path: Path) -> None:
    self._try_load_from_tokenizer_json(path)
    self._try_load_from_config_json(path)
    if self.load_merges and not self.merges:
        self._try_load_merges_txt(path)

def _try_load_merges_txt(self, path: Path) -> bool:
    merges_file = path / 'merges.txt'
    if not merges_file.is_file():
        return False
    with open(merges_file, 'r', encoding = 'utf-8') as fp:
        first_line = next(fp, '').strip()
        if not first_line.startswith('#'):
            fp.seek(0)
            line_num = 0
        else:
            line_num = 1
        merges = []
        for line in fp:
            line_num += 1
            line = line.strip()
```

```
                if not line:
                    continue
                parts = line.split(None, 3)
                if len(parts) != 2:
                    logger.warning(f'{merges_file.name}: Line {line_num}: Entry malformed, ignoring')
                    continue
                merges.append(f'{parts[0]} {parts[1]}')
        self.merges = merges
        return True

    def _set_special_token(self, typ: str, tid: Any) -> None:
        if not isinstance(tid, int):
            return
        if tid < 0:
            raise ValueError(f'invalid value for special token type {typ}: {tid}')
        if self.n_vocab is None or tid < self.n_vocab:
            if typ in self.special_token_ids:
                return
            self.special_token_ids[typ] = tid
            return
          logger.warning(f'Special token type {typ}, id {tid} out of range, must be under {self.n_vocab} -
skipping')

    def _try_load_from_tokenizer_json(self, path: Path) -> bool:
        tokenizer_file = path / 'tokenizer.json'
        if tokenizer_file.is_file():
            with open(tokenizer_file, encoding = 'utf-8') as f:
                tokenizer = json.load(f)
            if self.load_merges:
                merges = tokenizer.get('model', {}).get('merges')
                if isinstance(merges, list) and merges:
                    if isinstance(merges[0], str):
                        self.merges = merges
                    elif isinstance(merges[0], list) and len(merges[0]) == 2 and isinstance(merges[0][0], str):
                        # New format since transformers 4.45 to support spaces in merges
                        # ref: https://github.com/ggml-org/llama.cpp/issues/9692
                        # TODO: internally store as the new format instead of converting to old
                        if any(' ' in s for pair in merges for s in pair):
                            logger.warning(f'Spaces in merges detected, encoding as {chr(ord(" ") + 256)!r}')
                        self.merges = [
                            ' '.join(
                                [
                                    # ensure the spaces are properly encoded
                                    ''.join(
                                        chr(ord(c) + 256) if c == ' ' else c
                                        for c in part
                                    )
                                    for part in pair
                                ]
                            )
                            for pair in merges
                        ]
                    else:
                        raise ValueError("Unknown tokenizer merges format")
```

```
            added_tokens = tokenizer.get('added_tokens', {})
        else:
            added_tokens = {}
        tokenizer_config_file = path / 'tokenizer_config.json'
        if not tokenizer_config_file.is_file():
            return True
        with open(tokenizer_config_file, encoding = 'utf-8') as f:
            tokenizer_config = json.load(f)
        chat_template_alt = None
        chat_template_file = path / 'chat_template.json'
        if chat_template_file.is_file():
            with open(chat_template_file, encoding = 'utf-8') as f:
                chat_template_alt = json.load(f).get('chat_template')
        chat_template = tokenizer_config.get('chat_template', chat_template_alt)
        if chat_template is None or isinstance(chat_template, (str, list)):
            self.chat_template = chat_template
        else:
            logger.warning(f'Bad type for chat_template field in {tokenizer_config_file!r} - ignoring')
        for typ in self.special_token_types:
            add_entry = tokenizer_config.get(f'add_{typ}_token')
            if isinstance(add_entry, bool):
                self.add_special_token[typ] = add_entry
            entry = tokenizer_config.get(f'{typ}_token')
            if isinstance(entry, str):
                tc_content = entry
            elif isinstance(entry, dict):
                entry_content = entry.get('content')
                if not isinstance(entry_content, str):
                    continue
                tc_content = entry_content
            else:
                continue
            # We only need the first match here.
            maybe_token_id = next(
                (atok.get('id') for atok in added_tokens if atok.get('content') == tc_content),
                None,
            )
            self._set_special_token(typ, maybe_token_id)
        return True

    def _try_load_from_config_json(self, path: Path) -> bool:
        config_file = path / 'config.json'
        if not config_file.is_file():
            return False
        with open(config_file, encoding = 'utf-8') as f:
            config = json.load(f)
        for typ in self.special_token_types:
            self._set_special_token(typ, config.get(f'{typ}_token_id'))
        return True


@runtime_checkable
class BaseVocab(Protocol):
    tokenizer_model: ClassVar[str]
```

```
    name: ClassVar[str]


@runtime_checkable
class Vocab(BaseVocab, Protocol):
    vocab_size: int
    added_tokens_dict: dict[str, int]
    added_tokens_list: list[str]
    fname_tokenizer: Path

    def __init__(self, base_path: Path): ...
    def all_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]: ...


class NoVocab(BaseVocab):
    tokenizer_model = "no_vocab"
    name = "no_vocab"

    def __repr__(self) -> str:
        return "<NoVocab for a model without integrated vocabulary>"


class BpeVocab(Vocab):
    tokenizer_model = "gpt2"
    name = "bpe"

    def __init__(self, base_path: Path):
        added_tokens: dict[str, int] = {}

        if (fname_tokenizer := base_path / 'vocab.json').exists():
            # "slow" tokenizer
            with open(fname_tokenizer, encoding="utf-8") as f:
                self.vocab = json.load(f)

            try:
                # FIXME: Verify that added tokens here _cannot_ overlap with the main vocab.
                with open(base_path / 'added_tokens.json', encoding="utf-8") as f:
                    added_tokens = json.load(f)
            except FileNotFoundError:
                pass
        else:
            # "fast" tokenizer
            fname_tokenizer = base_path / 'tokenizer.json'

            # if this fails, FileNotFoundError propagates to caller
            with open(fname_tokenizer, encoding="utf-8") as f:
                tokenizer_json = json.load(f)

            tokenizer_model: dict[str, Any] = tokenizer_json['model']
            if (
                tokenizer_model['type'] != 'BPE' or tokenizer_model.get('byte_fallback', False)
                or tokenizer_json['decoder']['type'] != 'ByteLevel'
            ):
                raise FileNotFoundError('Cannot find GPT-2 BPE tokenizer')
```

```
            self.vocab = tokenizer_model["vocab"]

            if (added := tokenizer_json.get('added_tokens')) is not None:
                # Added tokens here can be duplicates of the main vocabulary.
                added_tokens = {item['content']: item['id']
                                for item in added
                                if item['content'] not in self.vocab}

        vocab_size   = len(self.vocab)
        expected_ids = list(range(vocab_size, vocab_size + len(added_tokens)))
        actual_ids   = sorted(added_tokens.values())
        if expected_ids != actual_ids:
            expected_end_id = vocab_size + len(actual_ids) - 1
            raise ValueError(f"Expected the {len(actual_ids)} added token ID(s) to be sequential in the range "
                             f"{vocab_size} - {expected_end_id}; got {actual_ids}")

        items = sorted(added_tokens.items(), key=lambda text_idx: text_idx[1])
        self.added_tokens_dict    = added_tokens
        self.added_tokens_list    = [text for (text, idx) in items]
        self.vocab_size_base      = vocab_size
        self.vocab_size           = self.vocab_size_base + len(self.added_tokens_list)
        self.fname_tokenizer      = fname_tokenizer

    def bpe_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        reverse_vocab = {id: encoded_tok for encoded_tok, id in self.vocab.items()}

        for i, _ in enumerate(self.vocab):
            yield reverse_vocab[i], 0.0, gguf.TokenType.NORMAL

    def added_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        for text in self.added_tokens_list:
            score = -1000.0
            yield text.encode("utf-8"), score, gguf.TokenType.CONTROL

    def all_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        yield from self.bpe_tokens()
        yield from self.added_tokens()

    def __repr__(self) -> str:
          return f"<BpeVocab with {self.vocab_size_base} base tokens and {len(self.added_tokens_list)} added
tokens>"


class SentencePieceVocab(Vocab):
    tokenizer_model = "llama"
    name = "spm"

    def __init__(self, base_path: Path):
        added_tokens: dict[str, int] = {}
        if (fname_tokenizer := base_path / 'tokenizer.model').exists():
            # normal location
            try:
                with open(base_path / 'added_tokens.json', encoding="utf-8") as f:
```

```
                added_tokens = json.load(f)
            except FileNotFoundError:
                pass
        elif not (fname_tokenizer := base_path.parent / 'tokenizer.model').exists():
            # not found in alternate location either
            raise FileNotFoundError('Cannot find tokenizer.model')

        self.sentencepiece_tokenizer = SentencePieceProcessor()
        self.sentencepiece_tokenizer.LoadFromFile(str(fname_tokenizer))
        vocab_size = self.sentencepiece_tokenizer.vocab_size()

        new_tokens       = {id: piece for piece, id in added_tokens.items() if id >= vocab_size}
        expected_new_ids = list(range(vocab_size, vocab_size + len(new_tokens)))
        actual_new_ids   = sorted(new_tokens.keys())

        if expected_new_ids != actual_new_ids:
                raise ValueError(f"Expected new token IDs {expected_new_ids} to be sequential; got
{actual_new_ids}")

        # Token pieces that were added to the base vocabulary.
        self.added_tokens_dict  = added_tokens
        self.added_tokens_list  = [new_tokens[id] for id in actual_new_ids]
        self.vocab_size_base    = vocab_size
        self.vocab_size         = self.vocab_size_base + len(self.added_tokens_list)
        self.fname_tokenizer    = fname_tokenizer

    def sentencepiece_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        tokenizer = self.sentencepiece_tokenizer
        for i in range(tokenizer.vocab_size()):
            piece = tokenizer.IdToPiece(i)
            text         = piece.encode("utf-8")
            score: float = tokenizer.GetScore(i)

            toktype = gguf.TokenType.NORMAL
            if tokenizer.IsUnknown(i):
                toktype = gguf.TokenType.UNKNOWN
            if tokenizer.IsControl(i):
                toktype = gguf.TokenType.CONTROL

            # NOTE: I think added_tokens are user defined.
            # ref: https://github.com/google/sentencepiece/blob/master/src/sentencepiece_model.proto
            # if tokenizer.is_user_defined(i): toktype = gguf.TokenType.USER_DEFINED

            if tokenizer.IsUnused(i):
                toktype = gguf.TokenType.UNUSED
            if tokenizer.IsByte(i):
                toktype = gguf.TokenType.BYTE

            yield text, score, toktype

    def added_tokens(self) -> Iterable[tuple[bytes, float, gguf.TokenType]]:
        for text in self.added_tokens_list:
            score = -1000.0
            yield text.encode("utf-8"), score, gguf.TokenType.USER_DEFINED
```