```
  """Plot embeddings of single word predicates."""
  model = create_model(pca=True)
  #syms = "abcdefghijklmnopqrstuvwxyz"
  syms = WORD_INDEX.keys()
  ctx, splits = list(), list()
  # preds = list("pqrv")
  # preds.extend(['',join([e]*2) for e in preds])
  for i in WORD_INDEX.keys():
    ctx.append(i)
    splits.append(len(ctx))
  # for p in preds:
  #   for c in syms:
  #     ctx.append("{}({}).".format(p,c))
  #   splits.append(len(ctx))
  embds = get_pca(ctx, model, dims=64)
  from mpl_toolkits.mplot3d import Axes3D
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  prev_sp = 0
  for sp in splits:
    x, y, z = embds[prev_sp:sp, 0], embds[prev_sp:sp, 1], embds[prev_sp:sp, -1]
    ax.scatter(x, y, z, depthshade=False)
    for i in map(syms.index, ["blue","someone","are","bob"]):
      ax.text(x[i], y[i], z[i], ctx[prev_sp+i])
    prev_sp = sp
  showsave_plot()

def plot_pred_saturation():
  """Plot predicate embedding saturation."""
  model = create_model(pca=True)
  ctx, splits = list(), list()
  for i in range(2, 65):
    p = "".join(["p"]*i)
    ctx.append("{}(a).".format(p))
    splits.append(len(ctx))
  embds = get_pca(ctx, model)
  plt.scatter(embds[::2, 0], embds[::2, 1])
  plt.scatter(embds[1::2, 0], embds[1::2, 1])
  prev_sp = 0
  for i, sp in enumerate(splits):
    pred, x, y = ctx[prev_sp], embds[prev_sp, 0], embds[prev_sp, 1]
    count = pred.count('p')
    if count <= 6:
      xf, yf = offset(x), offset(y)
        plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    elif i % 3 == 0 and i < 50 or i == len(splits)-1 or i == len(splits)-2:
      pred = str(count)+"*p(a)"
      xf, yf = offset(x), offset(y)
        plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle':
'-'})
    prev_sp = sp
  # Plot contour
  plt.xlim(-2, 2)
```

```
  xmin, xmax = plt.xlim()
  X = np.linspace(xmin, xmax, 40)
  ymin, ymax = plt.ylim()
  Y = np.linspace(ymin, ymax, 40)
  X, Y = np.meshgrid(X, Y)
  Z = np.sqrt((X-embds[-1,0])**2 + (Y-embds[-1,1])**2)
  plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
  Z = np.sqrt((X-embds[-2,0])**2 + (Y-embds[-2,1])**2)
  plt.contour(X, Y, Z, colors='grey', alpha=0.2, linestyles='dashed')
  showsave_plot()

def plot_template(preds, temps):
  """Plot PCA of templates with given predicates."""
  model = create_model(pca=True)
  ctx, splits = list(), list()
  for t in temps:
    for p in preds:
      ctx.append(t.format(p))
    splits.append(len(ctx))
  embds = get_pca(ctx, model)
  prev_sp = 0
  for sp in splits:
    plt.scatter(embds[prev_sp:sp, 0], embds[prev_sp:sp, 1])
    prev_sp = sp
    pred, x, y = ctx[sp-1], embds[sp-1, 0], embds[sp-1, 1]
    xf, yf = offset(x), offset(y)
    plt.annotate(pred, xy=(x, y), xytext=(xf, yf), textcoords='offset points', arrowprops={'arrowstyle': '-'})
  showsave_plot()

def plot_struct_preds():
  """Plot embeddings of different structural predicates."""
  ps = ['w', 'q', 'r', 's', 't', 'v', 'u', 'p']
  temps = ["{}(X,Y).", "{}(X,X).", "{}(X).", "{}(Z).",
           "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
           "-{}(a,b).", "-{}(x,y).", "-{}(a).", "-{}(xy).",
           "-{}(X,Y).", "-{}(X,X).", "-{}(X).", "-{}(Z)."]
  plot_template(ps, temps)

def plot_rules():
  """Plot embeddings of rules."""
  ps = ['w', 'a', 'b', 'c', 'd', 'e', 's', 't', 'v', 'u', 'p']
  temps = ["{}(X):-q(X).", "{}(X):--q(X).", "{}(X):-q(X);r(X).", "{}(X).",
           "{}(X,Y).", "{}(X,Y):--q(Y,X).", "{}(X,Y):--q(X,Y).",
           "{}(X,Y):-q(X,Y).", "{}(X,Y):-q(Y,X).", "{}(X,Y):-q(X);r(Y).",
           "{}(a,b).", "{}(x,y).", "{}(a).", "{}(xy).",
           "{}(X):--q(X);r(X).", "{}(X):-q(X);-r(X)."]
  plot_template(ps, temps)

def plot_attention():
  """Plot attention vector over given context."""
  model = create_model(iterations=ARGS.iterations, training=False)
  ctxs = ["If someone is cold then they are red. if someone is red then they are smart. Fiona is cold."]
  ctxs = [s.lower().replace(",","") for s in ctxs]
  q = "Fiona is smart."
```

```
  q = q .lower()
  q = q.replace(".","")
  q_list = [q]
  fig, axes = plt.subplots(1,1)
  for i, ctx in enumerate(ctxs):
    for j, t in enumerate(q_list):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r for r in rs], t, 1)]], 1, False, False, pad=ARGS.pad)
      out = model.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      #sims = sims.reshape(36,1)
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes.get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1, ARGS.iterations+1),
                  linewidths=0.5, square=True, cbar=False, ax=axes)
      axes.set_xlabel(q+" "+str(out) if j % 2 == 0 else "p(b) "+str(out))
  #plt.tight_layout()
  showsave_plot()

def plot_dual_attention():
  """Plot dual attention vectors of 2 models over given context."""
  modelbw = build_model("imasm", ARGS.model_dir+"curr_imasm64.h5",
                        char_size=len(CHAR_IDX)+1,
                        dim=ARGS.dim, iterations=ARGS.iterations,
                        training=False)
  modelfw = build_model("fwimarsm", ARGS.model_dir+"curr_fwimarsm64.h5",
                        char_size=len(CHAR_IDX)+1,
                        dim=ARGS.dim, iterations=ARGS.iterations,
                        training=False)
  ctxs = ["p(X):-q(X).q(X):-r(X).r(X):-s(X).s(a).s(b).",
          "p(X):-q(X);r(X).r(a).q(a).r(b).q(b).",
          "p(X):-q(X).p(X):-r(X).p(b).r(a).q(b)."]
  fig, axes = plt.subplots(1, 6)
  # Plot the attention
  for i, ctx in enumerate(ctxs):
    for j, m in enumerate([modelbw, modelfw]):
      rs = ctx.split('.')[:-1]
      dgen = LogicSeq([[([r + '.' for r in rs], "p(a).", 0)]], 1, False, False, pad=ARGS.pad)
      out = m.predict_generator(dgen)
      sims = out[:-1]
      out = np.round(np.asscalar(out[-1]), 2)
      sims = np.stack(sims, axis=0).squeeze()
      sims = sims.T
      ticks = (["()"] if ARGS.pad else []) + ["$\phi$"]
      axes[i*2+j].get_xaxis().set_ticks_position('top')
      sns.heatmap(sims, vmin=0, vmax=1, cmap="Blues", yticklabels=rs + ticks if j % 2 == 0 else False,
                  xticklabels=range(1,5), linewidths=0.5, square=True, cbar=False, ax=axes[i*2+j])
      # axes[i*2+j].set_xlabel("p(Q|C)=" + str(out))
      axes[i*2+j].set_xlabel("backward" if j % 2 == 0 else "forward")
  plt.tight_layout()
```

```
  showsave_plot()

if __name__ == '__main__':
  globals()[ARGS.function]()

==== explainer_utils.py ====
import torch
from tqdm.auto import tqdm

from crm.core import Network


def get_explanations(
    n: Network, X_test, y_test, true_explanations, k=3, verbose=False, all_layers=True
):
    tp = torch.zeros(n.num_neurons)
    fp = torch.zeros(n.num_neurons)
    tn = torch.zeros(n.num_neurons)
    fn = torch.zeros(n.num_neurons)

    tp_scores, tp_count = 0, 0
    fp_scores, fp_count = 0, 0
    tn_scores, tn_count = 0, 0
    fn_scores, fn_count = 0, 0

    for i in tqdm(range(len(X_test)), desc="Explanations XTest"):
        n.reset()
        pred = torch.argmax(n.forward(X_test[i]))
        if pred == 1:
            n.lrp(torch.tensor(100.0), n.num_neurons - 1)
        else:
            n.lrp(torch.tensor(100.0), n.num_neurons - 2)
        rels = [[] * n.num_layers for _ in range(n.num_layers)]
        for j in range(n.num_neurons):
            if n.neurons[j] not in [n.num_neurons - 2, n.num_neurons - 1]:
                try:
                    rels[n.neurons[j].layer].append(
                        (torch.tensor(n.neurons[j].relevance).item(), j)
                    )
                except Exception as e:
                    print(j, n.neurons[j].layer, n.neurons[j].relevance)
                    print(e)
                    assert False

        rels = [sorted(x, key=lambda x: x, reverse=True) for x in rels]
        if verbose:
            if all_layers:
                print(f"{i}: pred: {pred}, true: {y_test[i]}")
                for l_id in range(n.num_layers):
                    print(f"top-{k}-L{l_id}: {rels[l_id][:k]}")
            else:
                print(
                    f"{i}: pred = {pred.item()}, true: {y_test[i].item()}, top-{k}: {rels[n.num_layers-1][:k]}"
                )
```

```
    print("-" * 20)
if pred == 1 and y_test[i] == 1:
    tp += torch.tensor([n.neurons[i].relevance for i in range(n.num_neurons)])
    tp_scores += (
        1
        if len(
            list(
                set(true_explanations)
                & set(
                    [
                        rels[n.num_layers - 1][j][1]
                        for j in range(min(k, len(rels[n.num_layers - 1])))
                    ]
                )
            )
        )
        > 0
        else 0
    )
    tp_count += 1
if pred == 1 and y_test[i] == 0:
    fp += torch.tensor([n.neurons[i].relevance for i in range(n.num_neurons)])
    fp_scores += (
        1
        if len(
            list(
                set(true_explanations)
                & set(
                    [
                        rels[n.num_layers - 1][j][1]
                        for j in range(min(k, len(rels[n.num_layers - 1])))
                    ]
                )
            )
        )
        > 0
        else 0
    )
    fp_count += 1
if pred == 0 and y_test[i] == 0:
    tn += torch.tensor([n.neurons[i].relevance for i in range(n.num_neurons)])
    tn_scores += (
        1
        if len(
            list(
                set(true_explanations)
                & set(
                    [
                        rels[n.num_layers - 1][j][1]
                        for j in range(min(k, len(rels[n.num_layers - 1])))
                    ]
                )
            )
        )
```

```
            > 0
            else 0
        )
        tn_count += 1
    if pred == 0 and y_test[i] == 1:
        fn += torch.tensor([n.neurons[i].relevance for i in range(n.num_neurons)])
        fn_scores += (
            1
            if len(
                list(
                    set(true_explanations)
                    & set(
                        [
                            rels[n.num_layers - 1][j][1]
                            for j in range(min(k, len(rels[n.num_layers - 1])))
                        ]
                    )
                )
            )
            > 0
            else 0
        )
        fn_count += 1

print("SCORES")
print(f"TP:{tp_scores}/{tp_count}")
print(f"FP:{fp_scores}/{fp_count}")
print(f"TN:{tn_scores}/{tn_count}")
print(f"FN:{fn_scores}/{fn_count}")
print("####################################")

print("SUMMED RELS")

tp_values, tp_indices = tp.sort(descending=True)
fp_values, fp_indices = fp.sort(descending=True)
tn_values, tn_indices = tn.sort(descending=True)
fn_values, fn_indices = fn.sort(descending=True)

tp_rels = []
for j in range(len(tp_indices)):
    if n.neurons[tp_indices[j]].successor_neurons == [
        n.num_neurons - 2,
        n.num_neurons - 1,
    ]:
        tp_rels.append((tp_values[j].item(), tp_indices[j].item()))
print(f"TP (top-{k}): {tp_rels[:k]}")

fp_rels = []
for j in range(len(fp_indices)):
    if n.neurons[fp_indices[j]].successor_neurons == [
        n.num_neurons - 2,
        n.num_neurons - 1,
    ]:
        fp_rels.append((fp_values[j].item(), fp_indices[j].item()))
```

```
    print(f"FP (top-{k}): {fp_rels[:k]}")

    tn_rels = []
    for j in range(len(tn_indices)):
        if n.neurons[tn_indices[j]].successor_neurons == [
            n.num_neurons - 2,
            n.num_neurons - 1,
        ]:
            tn_rels.append((tn_values[j].item(), tn_indices[j].item()))
    print(f"TN (top-{k}): {tn_rels[:k]}")

    fn_rels = []
    for j in range(len(fn_indices)):
        if n.neurons[fn_indices[j]].successor_neurons == [
            n.num_neurons - 2,
            n.num_neurons - 1,
        ]:
            fn_rels.append((fn_values[j].item(), fn_indices[j].item()))
    print(f"FN (top-{k}): {fn_rels[:k]}")


# added by T:BFS to get the ancestors of neurons
def get_ancestors_of_neurons(n: Network, current_neurons):
    visited = list(current_neurons)
    queue = list(current_neurons)
    while queue:
        visit = queue.pop(0)
        for predecessor_neuron in n.neurons[visit].predecessor_neurons:
            if predecessor_neuron not in visited:
                visited.append(predecessor_neuron)
                queue.append(predecessor_neuron)
    return visited


# added by T: Get maximal explanation of a CRM
def get_max_explanations(
    n: Network, X_test, y_test, true_explanations, k=1, verbose=False
):
    tp_count = 0
    fn_count = 0
    tn_count = 0
    fp_count = 0

    cep_count = 0  # correctly explained positives
    cen_count = 0  # correctly explained negatives
    iep_count = 0  # incorrectly explained positives
    ien_count = 0  # incorrectly explained negatives

    print(f"Explaining {len(X_test)} test instances::")
    print(
                                                                                f"Instance:
y,y_pred,tp_count,fn_count,tn_count,fp_count,Top-{k}_neurons,cep_count,cen_count,iep_count,ien_count"
    )
    for i in tqdm(range(len(X_test)), desc="Explaining X_test"):
```

```
        n.reset()
        pred = torch.argmax(n.forward(X_test[i]))
        if pred == 1:
            n.lrp(torch.tensor(100.0), n.num_neurons - 1)
        else:
            n.lrp(torch.tensor(100.0), n.num_neurons - 2)

        rels = [[] * n.num_layers for _ in range(n.num_layers)]
        for j in range(n.num_neurons):
            if n.neurons[j] not in [n.num_neurons - 2, n.num_neurons - 1]:
                try:
                    rels[n.neurons[j].layer].append(
                        (round(torch.tensor(n.neurons[j].relevance).item(), 4), j)
                    )
                except Exception as e:
                    print(j, n.neurons[j].layer, n.neurons[j].relevance)
                    print(e)
                    assert False

        rels = [sorted(x, key=lambda x: x, reverse=True) for x in rels]
        topk_rels = rels[n.num_layers - 2][:k]
        topk_vertices = [x[1] for x in topk_rels]
        ancestors = get_ancestors_of_neurons(n, topk_vertices)

        # obtain eval metrics: accuracy and fidelity
        if y_test[i] == 1:
            if pred == 1:
                tp_count += 1
            else:
                fn_count += 1

            if set(true_explanations) & set(ancestors):
                cep_count += 1
            else:
                iep_count += 1

        if y_test[i] == 0:
            if pred == 0:
                tn_count += 1
            else:
                fp_count += 1

            if set(true_explanations) & set(ancestors):
                ien_count += 1
            else:
                cen_count += 1

        print(  # noqa
                                                                                f"Inst        {i}:
{y_test[i]},{pred},{tp_count},{fn_count},{tn_count},{fp_count},{topk_vertices},{cep_count},{cen_count},{iep_cou
nt},{ien_count}"  # noqa
        )  # noqa
        print(f"\tAncestors of {topk_vertices}: {ancestors}")
        print("\tTop-5 neurons in each CRM layer (ordered by relevance, descending):")
```

```
        for l_id in reversed(range(n.num_layers)):
            print(f"\t\tL{l_id}: {rels[l_id][:5]}")

    accuracy = (tp_count + tn_count) / (tp_count + fn_count + tn_count + fp_count)
    fidelity = (cep_count + cen_count) / (cep_count + cen_count + iep_count + ien_count)

    print("\n-------------------------------------")
    print("Explanation statistics:")
    print("-------------------------------------")
    print(f"TP: {tp_count}")
    print(f"FN: {fn_count}")
    print(f"TN: {tn_count}")
    print(f"FP: {fp_count}")
    print(f"CEP: {cep_count}")
    print(f"CEN: {cen_count}")
    print(f"IEP: {iep_count}")
    print(f"IEN: {ien_count}")
    print(f"Accuracy: {accuracy}, Fidelity: {fidelity}")

==== file_sage_agent.py ====
import os
import shutil
import argparse
from pathlib import Path
from datetime import datetime

# Define classification rules
FILE_MAP = {
    'data': [".csv", "train_", "test_", "repeat-config", "NCI-config"],
    'models': [".pt", ".pkl", ".gguf"],
    'fragments': [".yaml"],
    'logs': [".log", "mutation_log", "contradictions"],
    'runtime': ["vm_states", "snapshots"],
    'meta': ["audit_", "system_config", "optimized_paths"],
    'agents': [
        "validator.py", "dreamwalker.py", "quant_prompt_feeder.py",
        "run_logicshredder.py", "subcon_layer_mapper.py"
    ],
    'scripts': ["backup", "setup", "install", "crawler"],
    'media': [".jpg", ".jpeg", ".png", ".webp", ".bmp", ".gif"],
    'ui': [".html", ".css", ".js", ".mjs", "favicon"],
    'docs': ["readme", "README", ".md"],
    'ci': ["pytest", "tests.sh", "conftest.py"],
}

LOG_FILE = Path("logs/file_sage_log.txt")
LOG_FILE.parent.mkdir(parents=True, exist_ok=True)


def classify_file(file):
    name = file.name.lower()
    for folder, patterns in FILE_MAP.items():
        for pattern in patterns:
            if pattern in name:
```

```
                return folder
    if file.suffix == ".py":
        return "scripts"  # fallback for miscellaneous .py files
    return None


def move_file(file, target_folder, dry_run=False):
    target_path = Path(target_folder)
    target_path.mkdir(parents=True, exist_ok=True)
    new_loc = target_path / file.name

    if not dry_run:
        shutil.move(str(file), str(new_loc))

    with open(LOG_FILE, 'a', encoding='utf-8') as log:
        log.write(f"[{datetime.now()}] Moved '{file}' -> '{new_loc}'\n")
    print(f"[?] Moved '{file.name}' -> {target_folder}/")


def scan_and_sort(root, dry_run=False):
    all_files = [p for p in Path(root).rglob("*") if p.is_file() and not p.name.startswith(".")]
    for file in all_files:
        folder = classify_file(file)
        if folder:
            move_file(file, folder, dry_run=dry_run)
        else:
            print(f"[~] Unknown file: {file.name}")


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--preview", action="store_true", help="Run in dry mode (no file moves)")
    args = parser.parse_args()

    print("\n? FILE SAGE INITIATED\n============================\n")
    scan_and_sort(".", dry_run=args.preview)
    print("\n[?] File classification complete.\n")

==== fix_neurostore.py ====
import os
import zipfile
from pathlib import Path

def write_file(path_parts, content):
    path = Path(*path_parts)
    path.parent.mkdir(parents=True, exist_ok=True)
    path.write_text(content.strip() + "\n", encoding="utf-8")
    print(f"[OK] Fixed or created: {path}")

def fix_zip_project():
    zip_code = '''
import zipfile
from pathlib import Path
```

```
def zip_project():
    BASE = Path(".")
    zip_path = BASE.with_suffix(".zip")
    print(f"\\n[OK] Zipping project to: {zip_path}")
    with zipfile.ZipFile(zip_path, "w", zipfile.ZIP_DEFLATED) as zipf:
        for file in BASE.rglob("*"):
            if file.is_file():
                zipf.write(file, arcname=file.relative_to(BASE))
    print("[OK] ZIP complete.")

if __name__ == "__main__":
    zip_project()
'''
    write_file(["zip_project.py"], zip_code)

def fix_inject_profiler():
    inject_code = '''
import time
import psutil

class InjectProfiler:
    def __init__(self, label="logic_injection"):
        self.label = label
        self.snapshots = []

    def snapshot(self):
        mem = psutil.virtual_memory().percent
        cpu = psutil.cpu_percent(interval=0.1)
        self.snapshots.append((time.time(), mem, cpu))

    def report(self):
        print(f"[Profiler:{self.label}] Total snapshots: {len(self.snapshots)}")
        for t, mem, cpu in self.snapshots:
            print(f" - {round(t, 2)}s :: MEM {mem}% | CPU {cpu}%")

if __name__ == "__main__":
    p = InjectProfiler()
    for _ in range(5):
        p.snapshot()
        time.sleep(1)
    p.report()
'''
    write_file(["inject_profiler.py"], inject_code)

def main():
    print("[?] Repairing NeuroStore core...")
    fix_zip_project()
    fix_inject_profiler()
    print("[OK] Repair complete. Ready to run.")

if __name__ == "__main__":
    main()

==== fragment_decay_engine.py ====
```

```
# fragment_decay_engine.py
# ? Symbolic fragment rot system
# Rewrites fragment metadata to simulate aging, decay, and drift

import os
import yaml
import random
from datetime import datetime, timedelta
from pathlib import Path

# Configurable decay rules
DECAY_RULES = {
    "certainty": lambda x: max(0.0, round(x - random.uniform(0.05, 0.2), 3)),
    "urgency": lambda x: max(0.0, round(x - random.uniform(0.01, 0.05), 3)),
    "doubt": lambda x: min(1.0, round(x + random.uniform(0.05, 0.2), 3)),
    "confidence": lambda x: max(0.0, round(x - random.uniform(0.1, 0.3), 3))
}

# Optional field shuffler
def shuffle_fields(fragment):
    if 'claim' in fragment and random.random() < 0.4:
        fragment['claim'] = f"[fragmented] {fragment['claim']}"
    if 'tags' in fragment and isinstance(fragment['tags'], list):
        random.shuffle(fragment['tags'])
    return fragment

# Age threshold (e.g. 10 days = eligible for decay)
DECAY_AGE_DAYS = 10

# Base path for fragments
FRAGMENTS_PATH = Path("C:/Users/PC/Desktop/Operation Future/Allinonepy/fragments")

# Rot target output
DECAYED_PATH = Path("C:/Users/PC/Desktop/Operation Future/Allinonepy/fragments/decayed")
DECAYED_PATH.mkdir(parents=True, exist_ok=True)


def should_decay(file_path):
    modified = datetime.fromtimestamp(file_path.stat().st_mtime)
    return datetime.now() - modified > timedelta(days=DECAY_AGE_DAYS)


def decay_fragment(frag):
    if not isinstance(frag, dict):
        return frag

    # Apply decay rules
    for field, fn in DECAY_RULES.items():
        if field in frag:
            frag[field] = fn(frag[field])

    # Simulate drift
    frag = shuffle_fields(frag)
    frag['decayed'] = True
```

```
    frag['decay_timestamp'] = datetime.now().isoformat()
    return frag


def process_fragments():
    for path in FRAGMENTS_PATH.rglob("*.yaml"):
        if should_decay(path):
            with open(path, 'r', encoding='utf-8') as f:
                try:
                    data = yaml.safe_load(f)
                except yaml.YAMLError:
                    continue
            decayed = decay_fragment(data)
            out_path = DECAYED_PATH / path.name
            with open(out_path, 'w', encoding='utf-8') as f:
                yaml.safe_dump(decayed, f, sort_keys=False)
            print(f"? Decayed: {path.name} -> {out_path.name}")


if __name__ == "__main__":
    print("INFO Starting fragment decay scan...")
    process_fragments()
    print("[OK] Decay cycle complete.")

==== fragment_migrator.py ====
"""
LOGICSHREDDER :: fragment_migrator.py
Purpose: Migrate legacy flat fragments to structured subject-predicate-object format
"""

import yaml
import re
import time
from pathlib import Path

TARGET_DIRS = [
    Path("fragments/core"),
    Path("fragments/incoming")
]

LOG_PATH = Path("logs/migration_log.txt")
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)

def parse_claim(claim_text):
    match = re.match(r"The (\w+) (is|are|was|were|has|have) (.+)", claim_text.strip(), re.IGNORECASE)
    if match:
        return {
            "subject": match.group(1),
            "predicate": match.group(2),
            "object": match.group(3).strip(". ")
        }
    return None

def migrate_fragment(path):
```

```
    try:
        frag = yaml.safe_load(path.read_text(encoding='utf-8'))
        if not frag or "claim" not in frag:
            return False  # Already migrated or malformed

        claim = frag.pop("claim")
        struct = parse_claim(claim)

        if not struct:
            print(f"[migrator] ERROR Unable to structure: {claim}")
            return False

        frag["structure"] = struct
        path.write_text(yaml.dump(frag), encoding='utf-8')

        with open(LOG_PATH, 'a', encoding='utf-8') as log:
            log.write(f"[{int(time.time())}] Migrated {path.name} -> structured format\n")

        print(f"[migrator] [OK] Migrated: {path.name}")
        return True

    except Exception as e:
        print(f"[migrator] ERROR Error on {path.name}: {e}")
        return False

def run_migration():
    print("CONFIG Starting fragment migration...")
    for dir_path in TARGET_DIRS:
        if not dir_path.exists():
            continue
        for frag_file in dir_path.glob("*.yaml"):
            migrate_fragment(frag_file)
    print("[OK] Migration complete.")

if __name__ == "__main__":
    run_migration()

==== fragment_teleporter.py ====
"""
LOGICSHREDDER :: fragment_teleporter.py
Purpose: Move fragments to their optimized locations as defined in system_config.yaml
"""

import os
import shutil
import yaml
from pathlib import Path
from core.config_loader import load_config

def get_paths():
    config = load_config()
    if "paths" not in config:
        print("[teleporter] ERROR Config missing 'paths'.")
        return None
```

```
    return {k: Path(v) for k, v in config["paths"].items()}

def move_all(source_dir, dest_dir):
    count = 0
    if not source_dir.exists():
        return 0

    dest_dir.mkdir(parents=True, exist_ok=True)
    for file in source_dir.glob("*.yaml"):
        try:
            shutil.move(str(file), str(dest_dir / file.name))
            count += 1
        except Exception as e:
            print(f"[teleporter] WARNING Failed to move {file.name}: {e}")
    return count

def run_teleport():
    print("[teleporter] ? Starting logic fragment migration...")
    paths = get_paths()
    if not paths:
        return

    old_dirs = [
        Path("fragments/core"),
        Path("fragments/incoming"),
        Path("fragments/cold"),
        Path("fragments/archive"),
        Path("fragments/overflow")
    ]

    teleport_map = {
        "fragments/core": paths["fragments"],
        "fragments/incoming": paths["fragments"],
        "fragments/cold": paths["cold"],
        "fragments/archive": paths["archive"],
        "fragments/overflow": paths["overflow"]
    }

    total = 0
    for old_dir in old_dirs:
        target = teleport_map.get(str(old_dir).replace("\\", "/"), None)
        if target:
            moved = move_all(old_dir, target)
            total += moved
            print(f"[teleporter] [OK] Moved {moved} from {old_dir.name} -> {target}")
        else:
            print(f"[teleporter] WARNING No target mapped for: {old_dir.name}")

    print(f"[teleporter] ? Total fragments relocated: {total}")

if __name__ == "__main__":
    run_teleport()
```

```
==== fragment_tools.py ====
from core.config_loader import get
"""
LOGICSHREDDER :: fragment_tools.py
Purpose: Compare symbolic YAML fragments, log diffs, and track mutation drift
"""

import yaml
import difflib
import threading
from utils import agent_profiler
# [PROFILER_INJECTED]
threading.Thread(target=agent_profiler.run_profile_loop, daemon=True).start()
from pathlib import Path
import time

FRAG_DIR = Path("fragments/core")
ARCHIVE_DIR = Path("fragments/archive")
LOG_PATH = Path("logs/fragment_diffs.txt")
LOG_PATH.parent.mkdir(parents=True, exist_ok=True)

def load_fragment(path):
    try:
        with open(path, 'r', encoding='utf-8') as file:
            return yaml.safe_load(file)
    except Exception as e:
        print(f"[fragment_tools] Failed to load {path}: {e}")
        return None

def compare_fragments(frag1, frag2):
    diffs = {}

    if frag1.get('claim') != frag2.get('claim'):
        diffs['claim'] = (frag1.get('claim'), frag2.get('claim'))

    if frag1.get('confidence') != frag2.get('confidence'):
        diffs['confidence'] = (frag1.get('confidence'), frag2.get('confidence'))

    if frag1.get('emotion') != frag2.get('emotion'):
        diffs['emotion'] = (frag1.get('emotion'), frag2.get('emotion'))

    return diffs

def log_diff(old_id, new_id, diffs):
    timestamp = int(time.time())
    with open(LOG_PATH, 'a', encoding='utf-8') as log:
        log.write(f"\n[{timestamp}] Mutation: {old_id} -> {new_id}\n")
        for field, change in diffs.items():
            before, after = change
            log.write(f"  {field}:\n")
            if isinstance(before, str) and isinstance(after, str):
                for line in difflib.unified_diff(
                    before.splitlines(), after.splitlines(),
                    fromfile='before', tofile='after', lineterm=''
```

```
                ):
                    log.write(f"    {line}\n")
            else:
                log.write(f"    before: {before}\n")
                log.write(f"    after : {after}\n")

def diff_pair(file1, file2):
    frag1 = load_fragment(file1)
    frag2 = load_fragment(file2)
    if not frag1 or not frag2:
        print("[fragment_tools] Could not load both fragments.")
        return

    diffs = compare_fragments(frag1, frag2)
    if diffs:
        log_diff(frag1.get('id', 'unknown'), frag2.get('id', 'unknown'), diffs)
        print(f"[fragment_tools] Diff recorded for: {frag1.get('id')} -> {frag2.get('id')}")
    else:
        print("[fragment_tools] No significant changes detected.")

if __name__ == "__main__":
    # Manual test mode
    files = list(FRAG_DIR.glob("*.yaml"))
    if len(files) >= 2:
        diff_pair(files[0], files[1])
    else:
        print("[fragment_tools] Not enough fragments to compare.")
# [CONFIG_PATCHED]

==== full_mesh_builder.py ====

import os
import yaml
import random

def create_mesh(base_dir="F:/logic_core", count=50):
    os.makedirs(base_dir, exist_ok=True)

    for i in range(1, count + 1):
        part_name = f"p_{i:03}"
        part_path = os.path.join(base_dir, part_name)
        os.makedirs(part_path, exist_ok=True)

        # Drop a config file with symbolic parameters
        config = {
            "id": part_name,
            "emotion_bias": random.choice(["neutral", "anger", "joy", "curiosity", "melancholy"]),
            "decay_rate": round(random.uniform(0.93, 0.99), 3),
            "walk_style": random.choice(["conservative", "aggressive", "balanced"]),
            "linked_neighbors": [],
        }

        # Mesh connection simulation ? link to 2 random other nodes
        available = [f"p_{j:03}" for j in range(1, count + 1) if j != i]
```

```
        config["linked_neighbors"] = random.sample(available, k=2)

        config_path = os.path.join(part_path, "config.yaml")
        with open(config_path, "w") as f:
            yaml.dump(config, f)

        print(f"INFO {part_name} initialized with links -> {config['linked_neighbors']}")

if __name__ == "__main__":
    create_mesh()

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

from .zerogru import ZeroGRU, NestedTimeDist

# pylint: disable=line-too-long

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

  if ilp:
    context, query, templates = ilp

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

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

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

  EMBEDDING_DIM = 100

  embedding_matrix = np.zeros((len(word_index) + 1, EMBEDDING_DIM))
  for word, i in word_index.items():
```

```
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
      # words not found in embedding index will be all-zeros.
      embedding_matrix[i] = embedding_vector

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

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

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

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

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

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

```
    s_m_c = mult([embedded_rules, state]) # (?, rules, dim)
    sim_vec = concat([s_s_c, s_m_c, ctx_state, embedded_rules, repeated_q])
    sim_vec = att_dense(sim_vec) # (?, rules, 1)
    sim_vec = squeeze2(sim_vec) # (?, rules)
    sim_vec = softmax1(sim_vec)
    outs.append(sim_vec)

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

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

==== gemma3_convert_encoder_to_gguf.py ====
import gguf
import argparse
import logging
import sys
import torch
import json
import os
import numpy as np
from typing import cast, ContextManager, Any, Iterator
from pathlib import Path
from torch import Tensor

logger = logging.getLogger("gemma3-mmproj")



# (copied from convert_hf_to_gguf.py)
# tree of lazy tensors
class LazyTorchTensor(gguf.LazyBase):
    _tensor_type = torch.Tensor
    # to keep the type-checker happy
    dtype: torch.dtype
    shape: torch.Size

    # only used when converting a torch.Tensor to a np.ndarray
    _dtype_map: dict[torch.dtype, type] = {
        torch.float16: np.float16,
```

```
        torch.float32: np.float32,
    }

    # used for safetensors slices
                                                                                # ref: https://github.com/huggingface/safetensors/blob/079781fd0dc455ba0fe851e2b4507c33d0c0d407/bindings/python/src/lib.rs#L1046
    # TODO: uncomment U64, U32, and U16, ref: https://github.com/pytorch/pytorch/issues/58734
    _dtype_str_map: dict[str, torch.dtype] = {
        "F64": torch.float64,
        "F32": torch.float32,
        "BF16": torch.bfloat16,
        "F16": torch.float16,
        # "U64": torch.uint64,
        "I64": torch.int64,
        # "U32": torch.uint32,
        "I32": torch.int32,
        # "U16": torch.uint16,
        "I16": torch.int16,
        "U8": torch.uint8,
        "I8": torch.int8,
        "BOOL": torch.bool,
        "F8_E4M3": torch.float8_e4m3fn,
        "F8_E5M2": torch.float8_e5m2,
    }

    def numpy(self) -> gguf.LazyNumpyTensor:
        dtype = self._dtype_map[self.dtype]
        return gguf.LazyNumpyTensor(
            meta=gguf.LazyNumpyTensor.meta_with_dtype_and_shape(dtype, self.shape),
            args=(self,),
            func=(lambda s: s.numpy())
        )

    @classmethod
    def meta_with_dtype_and_shape(cls, dtype: torch.dtype, shape: tuple[int, ...]) -> Tensor:
        return torch.empty(size=shape, dtype=dtype, device="meta")

    @classmethod
    def from_safetensors_slice(cls, st_slice: Any) -> Tensor:
        dtype = cls._dtype_str_map[st_slice.get_dtype()]
        shape: tuple[int, ...] = tuple(st_slice.get_shape())
        lazy = cls(meta=cls.meta_with_dtype_and_shape(dtype, shape), args=(st_slice,), func=lambda s: s[:])
        return cast(torch.Tensor, lazy)

    @classmethod
    def __torch_function__(cls, func, types, args=(), kwargs=None):
        del types  # unused

        if kwargs is None:
            kwargs = {}

        if func is torch.Tensor.numpy:
            return args[0].numpy()
```

```
        return cls._wrap_fn(func)(*args, **kwargs)


class Gemma3VisionTower:
    hparams: dict
    gguf_writer: gguf.GGUFWriter
    fname_out: Path
    ftype: gguf.LlamaFileType

    @staticmethod
    def load_hparams(dir_model: Path):
        with open(dir_model / "config.json", "r", encoding="utf-8") as f:
            return json.load(f)

    @staticmethod
    def get_model_part_names(dir_model: Path, prefix: str, suffix: str) -> list[str]:
        part_names: list[str] = []
        for filename in os.listdir(dir_model):
            if filename.startswith(prefix) and filename.endswith(suffix):
                part_names.append(filename)
        part_names.sort()
        return part_names

    def __init__(self,
                 dir_model: Path,
                 fname_out: Path,
                 ftype: gguf.LlamaFileType,
                 is_big_endian: bool,):
        hparams = Gemma3VisionTower.load_hparams(dir_model)
        self.hparams = hparams
        self.fname_out = fname_out
        self.ftype = ftype
        endianess = gguf.GGUFEndian.BIG if is_big_endian else gguf.GGUFEndian.LITTLE
        self.gguf_writer = gguf.GGUFWriter(path=None, arch="clip", endianess=endianess)

        text_config = hparams["text_config"]
        vision_config = hparams["vision_config"]

        assert hparams["architectures"][0] == "Gemma3ForConditionalGeneration"
        assert text_config is not None
        assert vision_config is not None

        self.gguf_writer.add_string ("clip.projector_type",              "gemma3")
        self.gguf_writer.add_bool   ("clip.has_text_encoder",            False)
        self.gguf_writer.add_bool   ("clip.has_vision_encoder",          True)
        self.gguf_writer.add_bool   ("clip.has_llava_projector",         False) # legacy
        self.gguf_writer.add_uint32 ("clip.vision.image_size",           vision_config["image_size"])
        self.gguf_writer.add_uint32 ("clip.vision.patch_size",           vision_config["patch_size"])
        self.gguf_writer.add_uint32 ("clip.vision.embedding_length",     vision_config["hidden_size"])
        self.gguf_writer.add_uint32 ("clip.vision.feed_forward_length",  vision_config["intermediate_size"])
        self.gguf_writer.add_uint32 ("clip.vision.projection_dim",       text_config["hidden_size"])
        self.gguf_writer.add_uint32 ("clip.vision.block_count",          vision_config["num_hidden_layers"])
        self.gguf_writer.add_uint32 ("clip.vision.attention.head_count", vision_config["num_attention_heads"])
```

```
                                        self.gguf_writer.add_float32("clip.vision.attention.layer_norm_epsilon",
vision_config.get("layer_norm_eps", 1e-6))
        # default values taken from HF tranformers code
        self.gguf_writer.add_array  ("clip.vision.image_mean", [0.5, 0.5, 0.5])
        self.gguf_writer.add_array  ("clip.vision.image_std",  [0.5, 0.5, 0.5])
        self.gguf_writer.add_bool   ("clip.use_gelu", True)

        # load tensors
        for name, data_torch in self.get_tensors(dir_model):
            # convert any unsupported data types to float32
            if data_torch.dtype not in (torch.float16, torch.float32):
                data_torch = data_torch.to(torch.float32)
            self.add_tensor(name, data_torch)

    def get_tensors(self, dir_model: Path) -> Iterator[tuple[str, Tensor]]:
        part_names = Gemma3VisionTower.get_model_part_names(dir_model, "model", ".safetensors")
        tensor_names_from_parts: set[str] = set()
        for part_name in part_names:
            logger.info(f"gguf: loading model part '{part_name}'")
            from safetensors import safe_open
            ctx = cast(ContextManager[Any], safe_open(dir_model / part_name, framework="pt", device="cpu"))
            with ctx as model_part:
                tensor_names_from_parts.update(model_part.keys())

                for name in model_part.keys():
                    data = model_part.get_slice(name)
                    data = LazyTorchTensor.from_safetensors_slice(data)
                    yield name, data

    def add_tensor(self, name: str, data_torch: Tensor):
        is_1d = len(data_torch.shape) == 1
        is_embd = ".embeddings." in name
        old_dtype = data_torch.dtype
        can_quantize = not is_1d and not is_embd
        data_qtype = gguf.GGMLQuantizationType.F32

        # this is to support old checkpoint
        # TODO: remove this when we have the final model
        name = name.replace("vision_model.vision_model.", "vision_tower.vision_model.")
        name = name.replace("multimodal_projector.", "multi_modal_projector.")

        # filter only vision tensors
        if not name.startswith("vision_tower.vision_model.") and not name.startswith("multi_modal_projector."):
            return
        # prefix
        name = name.replace("vision_tower.vision_model.encoder.layers.", "v.blk.")
        name = name.replace("vision_tower.vision_model.", "v.")
        # projector and input embd
        name = name.replace(".embeddings.patch_embedding.", ".patch_embd.")
        name = name.replace(".embeddings.position_embedding.", ".position_embd.")
        name = name.replace(
            "multi_modal_projector.mm_input_projection_weight",
            "mm.input_projection.weight"
        )
```

```
        name = name.replace(
            "multi_modal_projector.mm_soft_emb_norm.weight",
            "mm.soft_emb_norm.weight"
        )
        name = name.replace("post_layernorm.", "post_ln.")
        # each block
        name = name.replace(".self_attn.k_proj.", ".attn_k.")
        name = name.replace(".self_attn.v_proj.", ".attn_v.")
        name = name.replace(".self_attn.q_proj.", ".attn_q.")
        name = name.replace(".self_attn.out_proj.", ".attn_out.")
        name = name.replace(".layer_norm1.", ".ln1.")
        name = name.replace(".layer_norm2.", ".ln2.")
        name = name.replace(".mlp.fc1.", ".ffn_down.")
        name = name.replace(".mlp.fc2.", ".ffn_up.")

        if can_quantize:
            if self.ftype == gguf.LlamaFileType.ALL_F32:
                data_qtype = gguf.GGMLQuantizationType.F32
            elif self.ftype == gguf.LlamaFileType.MOSTLY_F16:
                data_qtype = gguf.GGMLQuantizationType.F16
            elif self.ftype == gguf.LlamaFileType.MOSTLY_BF16:
                data_qtype = gguf.GGMLQuantizationType.BF16
            elif self.ftype == gguf.LlamaFileType.MOSTLY_Q8_0:
                data_qtype = gguf.GGMLQuantizationType.Q8_0
            else:
                raise ValueError(f"Unsupported file type: {self.ftype}")

        # corrent norm value ; only this "soft_emb_norm" need to be corrected as it's part of Gemma projector
        # the other norm values are part of SigLIP model, and they are already correct
        # ref code: Gemma3RMSNorm
        if "soft_emb_norm.weight" in name:
            logger.info(f"Correcting norm value for '{name}'")
            data_torch = data_torch + 1

        data = data_torch.numpy()

        try:
            data = gguf.quants.quantize(data, data_qtype)
        except Exception as e:
            logger.error(f"Error quantizing tensor '{name}': {e}, fallback to F16")
            data_qtype = gguf.GGMLQuantizationType.F16
            data = gguf.quants.quantize(data, data_qtype)

        # reverse shape to make it similar to the internal ggml dimension order
        shape_str = f"{{{', '.join(str(n) for n in reversed(data_torch.shape))}}}"
        logger.info(f"{f'%-32s' % f'{name},'} {old_dtype} --> {data_qtype.name}, shape = {shape_str}")

        self.gguf_writer.add_tensor(name, data, raw_dtype=data_qtype)

    def write(self):
        self.gguf_writer.write_header_to_file(path=self.fname_out)
        self.gguf_writer.write_kv_data_to_file()
        self.gguf_writer.write_tensors_to_file(progress=True)
        self.gguf_writer.close()
```

```
def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="Convert Gemma 3 vision tower safetensors to GGUF format",)
    parser.add_argument(
        "--outfile", type=Path, default="mmproj.gguf",
        help="path to write to",
    )
    parser.add_argument(
        "--outtype", type=str, choices=["f32", "f16", "bf16", "q8_0"], default="f16",
        help="output format",
    )
    parser.add_argument(
        "--bigendian", action="store_true",
        help="model is executed on big endian machine",
    )
    parser.add_argument(
        "model", type=Path,
        help="directory containing model file",
        nargs="?",
    )
    parser.add_argument(
        "--verbose", action="store_true",
        help="increase output verbosity",
    )

    args = parser.parse_args()
    if args.model is None:
        parser.error("the following arguments are required: model")
    return args


def main() -> None:
    args = parse_args()

    if args.verbose:
        logging.basicConfig(level=logging.DEBUG)
    else:
        logging.basicConfig(level=logging.INFO)

    dir_model = args.model

    if not dir_model.is_dir():
        logger.error(f'Error: {args.model} is not a directory')
        sys.exit(1)

    ftype_map: dict[str, gguf.LlamaFileType] = {
        "f32": gguf.LlamaFileType.ALL_F32,
        "f16": gguf.LlamaFileType.MOSTLY_F16,
        "bf16": gguf.LlamaFileType.MOSTLY_BF16,
        "q8_0": gguf.LlamaFileType.MOSTLY_Q8_0,
    }

    logger.info(f"Loading model: {dir_model.name}")
```

```
    with torch.inference_mode():
        gemma3_vision_tower = Gemma3VisionTower(
            dir_model=dir_model,
            fname_out=args.outfile,
            ftype=ftype_map[args.outtype],
            is_big_endian=args.bigendian,
        )
        gemma3_vision_tower.write()


if __name__ == '__main__':
    main()


==== gen-unicode-data.py ====
from __future__ import annotations

import array
import unicodedata
import requests


MAX_CODEPOINTS = 0x110000

UNICODE_DATA_URL = "https://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt"


# see https://www.unicode.org/L2/L1999/UnicodeData.html
def unicode_data_iter():
    res = requests.get(UNICODE_DATA_URL)
    res.raise_for_status()
    data = res.content.decode()

    prev = []

    for line in data.splitlines():
        # ej: 0000;<control>;Cc;0;BN;;;;;N;NULL;;;;
        line = line.split(";")

        cpt = int(line[0], base=16)
        assert cpt < MAX_CODEPOINTS

        cpt_lower = int(line[-2] or "0", base=16)
        assert cpt_lower < MAX_CODEPOINTS

        cpt_upper = int(line[-3] or "0", base=16)
        assert cpt_upper < MAX_CODEPOINTS

        categ = line[2].strip()
        assert len(categ) == 2

        bidir = line[4].strip()
        assert len(categ) == 2
```

```
        name = line[1]
        if name.endswith(", First>"):
            prev = (cpt, cpt_lower, cpt_upper, categ, bidir)
            continue
        if name.endswith(", Last>"):
            assert prev[1:] == (0, 0, categ, bidir)
            for c in range(prev[0], cpt):
                yield (c, cpt_lower, cpt_upper, categ, bidir)

        yield (cpt, cpt_lower, cpt_upper, categ, bidir)


# see definition in unicode.h
CODEPOINT_FLAG_UNDEFINED   = 0x0001  #
CODEPOINT_FLAG_NUMBER      = 0x0002  # \p{N}
CODEPOINT_FLAG_LETTER      = 0x0004  # \p{L}
CODEPOINT_FLAG_SEPARATOR   = 0x0008  # \p{Z}
CODEPOINT_FLAG_MARK        = 0x0010  # \p{M}
CODEPOINT_FLAG_PUNCTUATION = 0x0020  # \p{P}
CODEPOINT_FLAG_SYMBOL      = 0x0040  # \p{S}
CODEPOINT_FLAG_CONTROL     = 0x0080  # \p{C}

UNICODE_CATEGORY_TO_FLAG = {
    "Cn": CODEPOINT_FLAG_UNDEFINED,    # Undefined
    "Cc": CODEPOINT_FLAG_CONTROL,      # Control
    "Cf": CODEPOINT_FLAG_CONTROL,      # Format
    "Co": CODEPOINT_FLAG_CONTROL,      # Private Use
    "Cs": CODEPOINT_FLAG_CONTROL,      # Surrrogate
    "Ll": CODEPOINT_FLAG_LETTER,       # Lowercase Letter
    "Lm": CODEPOINT_FLAG_LETTER,       # Modifier Letter
    "Lo": CODEPOINT_FLAG_LETTER,       # Other Letter
    "Lt": CODEPOINT_FLAG_LETTER,       # Titlecase Letter
    "Lu": CODEPOINT_FLAG_LETTER,       # Uppercase Letter
    "L&": CODEPOINT_FLAG_LETTER,       # Cased Letter
    "Mc": CODEPOINT_FLAG_MARK,         # Spacing Mark
    "Me": CODEPOINT_FLAG_MARK,         # Enclosing Mark
    "Mn": CODEPOINT_FLAG_MARK,         # Nonspacing Mark
    "Nd": CODEPOINT_FLAG_NUMBER,       # Decimal Number
    "Nl": CODEPOINT_FLAG_NUMBER,       # Letter Number
    "No": CODEPOINT_FLAG_NUMBER,       # Other Number
    "Pc": CODEPOINT_FLAG_PUNCTUATION,  # Connector Punctuation
    "Pd": CODEPOINT_FLAG_PUNCTUATION,  # Dash Punctuation
    "Pe": CODEPOINT_FLAG_PUNCTUATION,  # Close Punctuation
    "Pf": CODEPOINT_FLAG_PUNCTUATION,  # Final Punctuation
    "Pi": CODEPOINT_FLAG_PUNCTUATION,  # Initial Punctuation
    "Po": CODEPOINT_FLAG_PUNCTUATION,  # Other Punctuation
    "Ps": CODEPOINT_FLAG_PUNCTUATION,  # Open Punctuation
    "Sc": CODEPOINT_FLAG_SYMBOL,       # Currency Symbol
    "Sk": CODEPOINT_FLAG_SYMBOL,       # Modifier Symbol
    "Sm": CODEPOINT_FLAG_SYMBOL,       # Math Symbol
    "So": CODEPOINT_FLAG_SYMBOL,       # Other Symbol
    "Zl": CODEPOINT_FLAG_SEPARATOR,    # Line Separator
    "Zp": CODEPOINT_FLAG_SEPARATOR,    # Paragraph Separator
```

```
    "Zs": CODEPOINT_FLAG_SEPARATOR,    # Space Separator
}


codepoint_flags = array.array('H', [CODEPOINT_FLAG_UNDEFINED]) * MAX_CODEPOINTS
table_whitespace = []
table_lowercase = []
table_uppercase = []
table_nfd = []

for (cpt, cpt_lower, cpt_upper, categ, bidir) in unicode_data_iter():
    # convert codepoint to unicode character
    char = chr(cpt)

    # codepoint category flags
    codepoint_flags[cpt] = UNICODE_CATEGORY_TO_FLAG[categ]

    # lowercase conversion
    if cpt_lower:
        table_lowercase.append((cpt, cpt_lower))

    # uppercase conversion
    if cpt_upper:
        table_uppercase.append((cpt, cpt_upper))

    # NFD normalization
    norm = ord(unicodedata.normalize('NFD', char)[0])
    if cpt != norm:
        table_nfd.append((cpt, norm))


# whitespaces, see "<White_Space>" https://www.unicode.org/Public/UCD/latest/ucd/PropList.txt
table_whitespace.extend(range(0x0009, 0x000D + 1))
table_whitespace.extend(range(0x2000, 0x200A + 1))
table_whitespace.extend([0x0020, 0x0085, 0x00A0, 0x1680, 0x2028, 0x2029, 0x202F, 0x205F, 0x3000])


# sort by codepoint
table_whitespace.sort()
table_lowercase.sort()
table_uppercase.sort()
table_nfd.sort()


# group ranges with same flags
ranges_flags: list[tuple[int, int]] = [(0, codepoint_flags[0])]  # start, flags
for codepoint, flags in enumerate(codepoint_flags):
    if flags != ranges_flags[-1][1]:
        ranges_flags.append((codepoint, flags))
ranges_flags.append((MAX_CODEPOINTS, 0x0000))


# group ranges with same nfd
ranges_nfd: list[tuple[int, int, int]] = [(0, 0, 0)]  # start, last, nfd
```

```
for codepoint, norm in table_nfd:
    start = ranges_nfd[-1][0]
    if ranges_nfd[-1] != (start, codepoint - 1, norm):
        ranges_nfd.append(None)  # type: ignore[arg-type]  # dummy, will be replaced below
        start = codepoint
    ranges_nfd[-1] = (start, codepoint, norm)


# Generate 'unicode-data.cpp':
#   python ./scripts//gen-unicode-data.py > unicode-data.cpp

def out(line=""):
    print(line, end='\n')  # noqa


out("""\
// generated with scripts/gen-unicode-data.py

#include "unicode-data.h"

#include <cstdint>
#include <vector>
#include <unordered_map>
#include <unordered_set>
""")

out("const  std::vector<std::pair<uint32_t,  uint16_t>>  unicode_ranges_flags  =  {    //  start,  flags  //
last=next_start-1")
for codepoint, flags in ranges_flags:
    out("{0x%06X, 0x%04X}," % (codepoint, flags))
out("};\n")

out("const std::unordered_set<uint32_t> unicode_set_whitespace = {")
for codepoint in table_whitespace:
    out("0x%06X," % codepoint)
out("};\n")

out("const std::unordered_map<uint32_t, uint32_t> unicode_map_lowercase = {")
for tuple_lw in table_lowercase:
    out("{0x%06X, 0x%06X}," % tuple_lw)
out("};\n")

out("const std::unordered_map<uint32_t, uint32_t> unicode_map_uppercase = {")
for tuple_up in table_uppercase:
    out("{0x%06X, 0x%06X}," % tuple_up)
out("};\n")

out("const std::vector<range_nfd> unicode_ranges_nfd = {  // start, last, nfd")
for triple in ranges_nfd:
    out("{0x%06X, 0x%06X, 0x%06X}," % triple)
out("};\n")

==== gen_logic.py ====
"""Tree based data generation script for logic programs."""
```

```
import argparse
import random as R

# Symbol Pool
CONST_SYMBOLS = "abcdefghijklmnopqrstuvwxyz"
VAR_SYMBOLS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
PRED_SYMBOLS = "abcdefghijklmnopqrstuvwxyz"
EXTRA_SYMBOLS = "-,()"

CHARS = sorted(list(set(CONST_SYMBOLS+VAR_SYMBOLS+PRED_SYMBOLS+EXTRA_SYMBOLS)))
# Reserve 0 for padding
CHAR_IDX = dict((c, i+1) for i, c in enumerate(CHARS))
IDX_CHAR = [0]
IDX_CHAR.extend(CHARS)

# Predicate Templates
FACT_T = "{}."
RULE_T = "{}:-{}."
PRED_T = "{}({})"
ARG_SEP = ','
PRED_SEP = ';'
NEG_PREFIX = '-'
TARGET_T = "? {} {}"

# pylint: disable=line-too-long,too-many-arguments,too-many-statements

def r_string(symbols, length):
  """Return random sequence from given symbols."""
  return ''.join(R.choice(symbols)
                 for _ in range(length))

def r_symbols(size, symbols, length, used=None):
  """Return unique random from given symbols."""
  if length == 1 and not used:
    return R.sample(symbols, size)
  rset, used = set(), set(used or [])
  while len(rset) < size:
    s = r_string(symbols, R.randint(1, length))
    if s not in used:
      rset.add(s)
  return list(rset)

def r_consts(size, used=None):
  """Return size many unique constants."""
  return r_symbols(size, CONST_SYMBOLS, ARGS.constant_length, used)

def r_vars(size, used=None):
  """Return size many unique variables."""
  return r_symbols(size, VAR_SYMBOLS, ARGS.variable_length, used)

def r_preds(size, used=None):
  """Return size many unique predicates."""
  return r_symbols(size, PRED_SYMBOLS, ARGS.predicate_length, used)
```

```
def write_p(pred):
  """Format single predicate tuple into string."""
  return PRED_T.format(pred[0], ARG_SEP.join(pred[1:]))

def write_r(preds):
  """Convert rule predicate tuple into string."""
  head = write_p(preds[0])
  # Is it just a fact
  if len(preds) == 1:
    return FACT_T.format(head)
  # We have a rule
  return RULE_T.format(head, PRED_SEP.join([write_p(p) for p in preds[1:]]))

def output(context, targets):
  """Print the context and given targets."""
  # context: [[('p', 'a', 'b')], ...]
  # targets: [(('p', 'a', 'b'), 1), ...]
  if ARGS.shuffle_context:
    R.shuffle(context)
  print('\n'.join([write_r(c) for c in context]))
  for t, v in targets:
    print(TARGET_T.format(write_r([t]), v))

def cv_mismatch(consts):
  """Returns a possible mismatching variable binding for given constants."""
  if len(consts) <= 1 or len(set(consts)) == 1:
    return list()
  # We know some constant is different
  # [a,b,a,c] -> [X,Y,Y,Z]
  # [a,b] -> [X,X] are mismatches
  # assign same variables to different constants
  vs = r_vars(len(consts)-1) # [X,Y,Z,..]
  for i, c in enumerate(consts[1:]):
    if c != consts[0]:
      # we haven't seen it before
      vs.insert(i+1,vs[0])
      break
  assert len(vs) == len(consts)
  return vs

def cv_match(consts):
  """Returns a possible matching variable binding for given constants."""
  if len(consts) <= 1:
    return r_vars(len(consts))
  # We want to *randomly* assing the same variable to same constants
  # [a,a,b] -> [X,Y,Z] -> [X,X,Y]
  vs = r_vars(len(consts))
  cvmap = dict()
  for i, c in enumerate(consts):
    if c in cvmap:
      if R.random() < 0.5:
        vs[i] = cvmap[c] # assign the same variable
      # otherwise get a unique variable
    else:
```

```
      cvmap[c] = vs[i]
  assert len(vs) == len(consts)
  return vs

def generate(depth=0, context=None, target=None, success=None,
             upreds=None, uconsts=None, stats=None):
  """Generate tree based logic program."""
  ctx = context or list()
  args = target[1:] if target else [r_consts(1)[0] for _ in range(ARGS.arity)]
  t = target or [r_preds(1)[0]] + [R.choice(args) for _ in range(R.randint(1, ARGS.arity))]
  arity = len(t[1:])
  succ = success if success is not None else R.choice((True, False))
  upreds = upreds or set([t[0]])
  uconsts = uconsts or set(t[1:])
  stats = stats or dict()

  # Create rule OR branching
  num_rules = R.randint(1, ARGS.max_or_branch)
  stats.setdefault('or_num', list()).append(num_rules)
  # If the rule succeeds than at least one branch must succeed
  succs = [R.choice((True, False)) for _ in range(num_rules)] \
          if succ else [False]*num_rules # otherwise all branches must fail
  if succ and not any(succs):
    # Ensure at least one OR branch succeeds
    succs[R.randrange(len(succs))] = True
  # Rule depths randomised between 0 to max depth
  depths = [R.randint(0, depth) for _ in range(num_rules)]
  if max(depths) != depth:
    depths[R.randrange(num_rules)] = depth
  # print("HERE:", num_rules, succs, depths, t)

  # Generate OR branches
  is_tadded = False
  for child_depth, child_succ in zip(depths, succs):
    # Base case
    if child_depth == 0:
      if R.random() < 0.20:
        # The constant doesn't match
        args = t[1:]
        args[R.randrange(len(args))] = r_consts(1, uconsts)[0]
        uconsts.update(args)
        ctx.append([[t[0]] + args])
      if R.random() < 0.20:
        # The predicate doesn't match
        p = r_preds(1, upreds)[0]
        upreds.add(p)
        ctx.append([[p,] + t[1:]])
      if R.random() < 0.20:
        # The arity doesn't match
        ctx.append([[t[0]] + t[1:] + [R.choice(t[1:] + r_consts(arity))]])
      if R.random() < 0.20:
        # The variables don't match
        vs = cv_mismatch(t[1:])
        if vs:
```

```
          ctx.append([[t[0]] + vs])
      # The predicate doesn't appear at all
      if child_succ:
        if R.random() < 0.5:
          # p(X). case
          ctx.append([[t[0]] + cv_match(t[1:])])
        elif not is_tadded:
          # ground case
          ctx.append([t])
          is_tadded = True
      continue
    # Recursive case
    num_body = R.randint(1, ARGS.max_and_branch)
    stats.setdefault('body_num', list()).append(num_body)
    negation = [R.choice((True, False)) for _ in range(num_body)] \
               if ARGS.negation else [False]*num_body
    # Compute recursive success targets
    succ_targets = [R.choice((True, False)) for _ in range(num_body)] \
                   if not child_succ else [not n for n in negation]
    if not child_succ:
      # Ensure a failed target
      ri = R.randrange(len(succ_targets))
      # succeeding negation fails this, vice versa
      succ_targets[ri] = negation[ri]
    # Create rule
    body_preds = r_preds(num_body, upreds)
    upreds.update(body_preds)
    lit_vars = cv_match(t[1:])
    if not child_succ and R.random() < 0.5:
      # Fail due to variable pattern mismatch
      vs = cv_mismatch(t[1:])
      if vs:
        lit_vars = vs
        succ_targets = [R.choice((True, False)) for _ in range(num_body)]
    lit_vars.extend([r_vars(1)[0] for _ in range(ARGS.unbound_vars)])
    rule = [[t[0]]+lit_vars[:arity]]
    vcmap = {lit_vars[i]:t[i+1] for i in range(arity)}
    # Compute child targets
    child_targets = list()
    for i in range(num_body):
      R.shuffle(lit_vars)
      child_arity = R.randint(1, arity)
      pred = [body_preds[i]] + lit_vars[:child_arity]
      rule.append([(NEG_PREFIX if negation[i] else "") + pred[0]] + pred[1:])
      vs = [vcmap.get(v, r_consts(1, uconsts)[0]) for v in lit_vars[:child_arity]]
      child_targets.append([pred[0]]+vs)
    ctx.append(rule)
    # Recurse
    for child_t, s in zip(child_targets, succ_targets):
      generate(child_depth-1, ctx, child_t, s, upreds, uconsts, stats)
  return ctx, [(t, int(succ))], stats

if __name__ == '__main__':
  # Arguments
```

```
  parser = argparse.ArgumentParser(description="Generate logic program data.")
  parser.add_argument("-d", "--depth", default=0, type=int, help="The depth of the logic program.")
    parser.add_argument("-mob", "--max_or_branch", default=1, type=int, help="Upper bound on number of branches.")
    parser.add_argument("-mab", "--max_and_branch", default=1, type=int, help="Upper bound on number of branches.")
  parser.add_argument("-s", "--size", default=1, type=int, help="Number of programs to generate.")
  # Configuration parameters
  parser.add_argument("-uv", "--unbound_vars", default=0, type=int, help="Number of unbound variables.")
  parser.add_argument("-ar", "--arity", default=2, type=int, help="Upper bound on arity of literals.")
  parser.add_argument("-n", "--negation", action="store_true", help="Use negation by failure.")
  parser.add_argument("-cl", "--constant_length", default=2, type=int, help="Length of constants.")
  parser.add_argument("-vl", "--variable_length", default=1, type=int, help="Length of variables.")
  parser.add_argument("-pl", "--predicate_length", default=2, type=int, help="Length of predicates.")
  parser.add_argument("-sf", "--shuffle_context", action="store_true", help="Shuffle context before output.")
  ARGS = parser.parse_args()

  for _ in range(ARGS.size):
    context_out, targets_out, _ = generate(depth=ARGS.depth)
    output(context_out, targets_out)

==== generate_cu_files.py ====
#!/usr/bin/env python3

from glob import glob
import os

TYPES_KV = ["GGML_TYPE_Q4_0", "GGML_TYPE_Q4_1", "GGML_TYPE_Q5_0", "GGML_TYPE_Q5_1", "GGML_TYPE_Q8_0", "GGML_TYPE_F16"]

SOURCE_FATTN_VEC = """// This file has been autogenerated by generate_cu_files.py, do not edit manually.

#include "../fattn-vec-f{vkq_size}.cuh"

DECL_FATTN_VEC_F{vkq_size}_CASE({head_size}, {type_k}, {type_v});
"""

SOURCE_FATTN_MMA_START = """// This file has been autogenerated by generate_cu_files.py, do not edit manually.

#include "../fattn-mma-f16.cuh"

"""

SOURCE_FATTN_MMA_CASE = "DECL_FATTN_MMA_F16_CASE({head_size}, {ncols1}, {ncols2});\n"

TYPES_MMQ = [
    "GGML_TYPE_Q4_0", "GGML_TYPE_Q4_1", "GGML_TYPE_Q5_0", "GGML_TYPE_Q5_1", "GGML_TYPE_Q8_0",
    "GGML_TYPE_Q2_K", "GGML_TYPE_Q3_K", "GGML_TYPE_Q4_K", "GGML_TYPE_Q5_K", "GGML_TYPE_Q6_K",
    "GGML_TYPE_IQ2_XXS", "GGML_TYPE_IQ2_XS", "GGML_TYPE_IQ2_S", "GGML_TYPE_IQ3_XXS", "GGML_TYPE_IQ3_S",
    "GGML_TYPE_IQ1_S", "GGML_TYPE_IQ4_NL", "GGML_TYPE_IQ4_XS"
]

SOURCE_MMQ = """// This file has been autogenerated by generate_cu_files.py, do not edit manually.
```

```
#include "../mmq.cuh"

DECL_MMQ_CASE({type});
"""


def get_short_name(long_quant_name):
    return long_quant_name.replace("GGML_TYPE_", "").lower()


def get_head_sizes(type_k, type_v):
    if type_k == "GGML_TYPE_F16" and type_v == "GGML_TYPE_F16":
        return [64, 128, 256]
    if type_k == "GGML_TYPE_F16":
        return [64, 128]
    return [128]


for filename in glob("*.cu"):
    os.remove(filename)

for vkq_size in [16, 32]:
    for type_k in TYPES_KV:
        for type_v in TYPES_KV:
            for head_size in get_head_sizes(type_k, type_v):
                                                                                                with
open(f"fattn-vec-f{vkq_size}-instance-hs{head_size}-{get_short_name(type_k)}-{get_short_name(type_v)}.cu", "w")
as f:
                        f.write(SOURCE_FATTN_VEC.format(vkq_size=vkq_size, head_size=head_size, type_k=type_k,
type_v=type_v))

for ncols in [8, 16, 32, 64, 128]:
    for ncols2 in [1, 2, 4, 8]:
        ncols1 = ncols // ncols2
        if ncols == 128:
            continue  # Too much register pressure.
        with open(f"fattn-mma-f16-instance-ncols1_{ncols1}-ncols2_{ncols2}.cu", "w") as f:
            f.write(SOURCE_FATTN_MMA_START)

            for head_size in [64, 80, 96, 112, 128, 256]:
                if ncols == 128 and head_size == 256:
                    continue  # Needs too much shared memory.
                f.write(SOURCE_FATTN_MMA_CASE.format(ncols1=ncols1, ncols2=ncols2, head_size=head_size))

for type in TYPES_MMQ:
    with open(f"mmq-instance-{get_short_name(type)}.cu", "w") as f:
        f.write(SOURCE_MMQ.format(type=type))

==== get_chat_template.py ====
#!/usr/bin/env python
'''
  Fetches the Jinja chat template of a HuggingFace model.
  If a model has multiple chat templates, you can specify the variant name.
```

```
  Syntax:
    ./scripts/get_chat_template.py model_id [variant]

  Examples:
    ./scripts/get_chat_template.py CohereForAI/c4ai-command-r-plus tool_use
    ./scripts/get_chat_template.py microsoft/Phi-3.5-mini-instruct
'''

import json
import re
import sys


def get_chat_template(model_id, variant=None):
    try:
        # Use huggingface_hub library if available.
        # Allows access to gated models if the user has access and ran `huggingface-cli login`.
        from huggingface_hub import hf_hub_download
        with open(hf_hub_download(repo_id=model_id, filename="tokenizer_config.json"), encoding="utf-8") as f:
            config_str = f.read()
    except ImportError:
        import requests
        assert re.match(r"^[\w.-]+/[\w.-]+$", model_id), f"Invalid model ID: {model_id}"
        response = requests.get(f"https://huggingface.co/{model_id}/resolve/main/tokenizer_config.json")
        if response.status_code == 401:
                  raise Exception('Access to this model is gated, please request access, authenticate with
`huggingface-cli login` and make sure to run `pip install huggingface_hub`')
        response.raise_for_status()
        config_str = response.text

    try:
        config = json.loads(config_str)
    except json.JSONDecodeError:
        # Fix https://huggingface.co/NousResearch/Meta-Llama-3-8B-Instruct/blob/main/tokenizer_config.json
        # (Remove extra '}' near the end of the file)
            config = json.loads(re.sub(r'\}([\n\s]*\}[\n\s]*\],[\n\s]*"clean_up_tokenization_spaces")', r'\1',
config_str))

    chat_template = config['chat_template']
    if isinstance(chat_template, str):
        return chat_template
    else:
        variants = {
            ct['name']: ct['template']
            for ct in chat_template
        }

        def format_variants():
            return ', '.join(f'"{v}"' for v in variants.keys())

        if variant is None:
            if 'default' not in variants:
                raise Exception(f'Please specify a chat template variant (one of {format_variants()})')
            variant = 'default'
```

```
            sys.stderr.write(f'Note: picked "default" chat template variant (out of {format_variants()})\n')
        elif variant not in variants:
            raise Exception(f"Variant {variant} not found in chat template (found {format_variants()})")

        return variants[variant]


def main(args):
    if len(args) < 1:
        raise ValueError("Please provide a model ID and an optional variant name")
    model_id = args[0]
    variant = None if len(args) < 2 else args[1]

    template = get_chat_template(model_id, variant)
    sys.stdout.write(template)


if __name__ == '__main__':
    main(sys.argv[1:])

==== gguf - Copy.py ====
# This file left for compatibility. If you want to use the GGUF API from Python
# then don't import gguf/gguf.py directly. If you're looking for examples, see the
# examples/ directory for gguf-py

import importlib
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent))

# Compatibility for people trying to import gguf/gguf.py directly instead of as a package.
importlib.invalidate_caches()
import gguf  # noqa: E402

importlib.reload(gguf)

==== gguf.py ====
# This file left for compatibility. If you want to use the GGUF API from Python
# then don't import gguf/gguf.py directly. If you're looking for examples, see the
# examples/ directory for gguf-py

import importlib
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent))

# Compatibility for people trying to import gguf/gguf.py directly instead of as a package.
importlib.invalidate_caches()
import gguf  # noqa: E402

importlib.reload(gguf)
```

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

import logging
import argparse
import os
import sys
from tqdm import tqdm
from pathlib import Path

import numpy as np

# Necessary to load the local gguf package
if "NO_LOCAL_GGUF" not in os.environ and (Path(__file__).parent.parent.parent.parent / 'gguf-py').exists():
    sys.path.insert(0, str(Path(__file__).parent.parent.parent))

import gguf

logger = logging.getLogger("gguf-convert-endian")


def convert_byteorder(reader: gguf.GGUFReader, args: argparse.Namespace) -> None:
    file_endian = reader.endianess.name
    if reader.byte_order == 'S':
        host_endian = 'BIG' if file_endian == 'LITTLE' else 'LITTLE'
    else:
        host_endian = file_endian
    order = host_endian if args.order == "native" else args.order.upper()
    logger.info(f"* Host is {host_endian} endian, GGUF file seems to be {file_endian} endian")
    if file_endian == order:
        logger.info(f"* File is already {order} endian. Nothing to do.")
        sys.exit(0)
    logger.info("* Checking tensors for conversion compatibility")
    for tensor in reader.tensors:
        if tensor.tensor_type not in (
            gguf.GGMLQuantizationType.F32,
            gguf.GGMLQuantizationType.F16,
            gguf.GGMLQuantizationType.Q8_0,
            gguf.GGMLQuantizationType.Q4_K,
            gguf.GGMLQuantizationType.Q6_K,
        ):
            raise ValueError(f"Cannot handle type {tensor.tensor_type.name} for tensor {repr(tensor.name)}")
    logger.info(f"* Preparing to convert from {file_endian} to {order}")
    if args.dry_run:
        return
    logger.warning("*** Warning *** Warning *** Warning **")
    logger.warning("* This conversion process may damage the file. Ensure you have a backup.")
    if order != host_endian:
         logger.warning("* Requested endian differs from host, you will not be able to load the model on this
machine.")
    logger.warning("* The file will be modified immediately, so if conversion fails or is interrupted")
    logger.warning("* the file will be corrupted. Enter exactly YES if you are positive you want to proceed:")
    response = input("YES, I am sure> ")
```

```
if response != "YES":
    logger.warning("You didn't enter YES. Okay then, see ya!")
    sys.exit(0)
logger.info(f"* Converting fields ({len(reader.fields)})")
for idx, field in enumerate(reader.fields.values()):
    logger.info(f"- {idx:4}: Converting field {repr(field.name)}, part count: {len(field.parts)}")
    for part in field.parts:
        part.byteswap(inplace=True)
logger.info(f"* Converting tensors ({len(reader.tensors)})")

for idx, tensor in enumerate(pbar := tqdm(reader.tensors, desc="Converting tensor")):
    log_message = (
        f"Converting tensor {repr(tensor.name)}, "
        f"type={tensor.tensor_type.name}, "
        f"elements={tensor.n_elements} "
    )

    # Byte-swap each part of the tensor's field
    for part in tensor.field.parts:
        part.byteswap(inplace=True)

    # Byte-swap tensor data if necessary
    if tensor.tensor_type == gguf.GGMLQuantizationType.Q8_0:
        # Handle Q8_0 tensor blocks (block_q8_0)
        # Specific handling of block_q8_0 is required.
        # Each block_q8_0 consists of an f16 delta (scaling factor) followed by 32 int8 quantizations.

        block_size = 34 # 34 bytes = <f16 delta scaling factor> + 32 * <int8 quant>

        n_blocks = len(tensor.data) // block_size
        for block_num in (inner_pbar := tqdm(range(n_blocks), desc="Byte-swapping Blocks", leave=False)):
            block_offs = block_num * block_size

            # Byte-Swap f16 sized delta field
            delta = tensor.data[block_offs:block_offs + 2].view(dtype=np.uint16)
            delta.byteswap(inplace=True)

            # Byte-Swap Q8 weights
            if block_num % 100000 == 0:
                inner_pbar.set_description(f"Byte-swapping Blocks [{(n_blocks - block_num) // n_blocks}]")

    elif tensor.tensor_type == gguf.GGMLQuantizationType.Q4_K:
        # Handle Q4_K tensor blocks (block_q4_k)
        # Specific handling of block_q4_k is required.
        # Each block_q4_k consists of 2 f16 values followed by 140 int8 values.

        # first flatten structure
        newshape = 1
        for i in tensor.data.shape:
            newshape *= i

        tensor.data.resize(newshape)

        block_size = 144
```

```
            n_blocks = len(tensor.data) // block_size
            for block_num in (inner_pbar := tqdm(range(n_blocks), desc="Byte-swapping Blocks", leave=False)):
                block_offs = block_num * block_size

                # Byte-Swap f16 sized fields
                delta = tensor.data[block_offs:block_offs + 2].view(dtype=np.uint16)
                delta.byteswap(inplace=True)

                delta = tensor.data[block_offs + 2:block_offs + 4].view(dtype=np.uint16)
                delta.byteswap(inplace=True)

                # Byte-Swap
                if block_num % 100000 == 0:
                    inner_pbar.set_description(f"Byte-swapping Blocks [{(n_blocks - block_num) // n_blocks}]")

        elif tensor.tensor_type == gguf.GGMLQuantizationType.Q6_K:
            # Handle Q6_K tensor blocks (block_q6_k)
            # Specific handling of block_q6_k is required.
            # Each block_q6_k consists of 208 int8 values followed by 1 f16 value.

            # first flatten structure
            newshape = 1
            for i in tensor.data.shape:
                newshape *= i

            tensor.data.resize(newshape)

            block_size = 210
            n_blocks = len(tensor.data) // block_size
            for block_num in (inner_pbar := tqdm(range(n_blocks), desc="Byte-swapping Blocks", leave=False)):
                block_offs = block_num * block_size

                # Byte-Swap f16 sized field
                delta = tensor.data[block_offs + 208:block_offs + 210].view(dtype=np.uint16)
                delta.byteswap(inplace=True)

                # Byte-Swap
                if block_num % 100000 == 0:
                    inner_pbar.set_description(f"Byte-swapping Blocks [{(n_blocks - block_num) // n_blocks}]")

        else:
            # Handle other tensor types
            tensor.data.byteswap(inplace=True)

        pbar.set_description(log_message)

    logger.info("* Completion")


def main() -> None:
    parser = argparse.ArgumentParser(description="Convert GGUF file byte order")
    parser.add_argument(
        "model", type=str,
        help="GGUF format model filename",
```

```
    )
    parser.add_argument(
        "order", type=str, choices=['big', 'little', 'native'],
        help="Requested byte order",
    )
    parser.add_argument(
        "--dry-run", action="store_true",
        help="Don't actually change anything",
    )
    parser.add_argument("--verbose", action="store_true", help="increase output verbosity")

    args = parser.parse_args(None if len(sys.argv) > 1 else ["--help"])

    logging.basicConfig(level=logging.DEBUG if args.verbose else logging.INFO)

    logger.info(f'* Loading: {args.model}')
    reader = gguf.GGUFReader(args.model, 'r' if args.dry_run else 'r+')
    convert_byteorder(reader, args)


if __name__ == "__main__":
    main()

==== gguf_dump - Copy.py ====
#!/usr/bin/env python3
from __future__ import annotations

import logging
import argparse
import os
import re
import sys
from pathlib import Path
from typing import Any

# Necessary to load the local gguf package
if "NO_LOCAL_GGUF" not in os.environ and (Path(__file__).parent.parent.parent.parent / 'gguf-py').exists():
    sys.path.insert(0, str(Path(__file__).parent.parent.parent))

from gguf import GGUFReader, GGUFValueType, ReaderTensor  # noqa: E402

logger = logging.getLogger("gguf-dump")


def get_file_host_endian(reader: GGUFReader) -> tuple[str, str]:
    file_endian = reader.endianess.name
    if reader.byte_order == 'S':
        host_endian = 'BIG' if file_endian == 'LITTLE' else 'LITTLE'
    else:
        host_endian = file_endian
    return (host_endian, file_endian)


# For more information about what field.parts and field.data represent,
```

```
# please see the comments in the modify_gguf.py example.
def dump_metadata(reader: GGUFReader, args: argparse.Namespace) -> None:
    host_endian, file_endian = get_file_host_endian(reader)
    print(f'* File is {file_endian} endian, script is running on a {host_endian} endian host.')  # noqa: NP100
    print(f'* Dumping {len(reader.fields)} key/value pair(s)')  # noqa: NP100
    for n, field in enumerate(reader.fields.values(), 1):
        if not field.types:
            pretty_type = 'N/A'
        elif field.types[0] == GGUFValueType.ARRAY:
            nest_count = len(field.types) - 1
            pretty_type = '[' * nest_count + str(field.types[-1].name) + ']' * nest_count
        else:
            pretty_type = str(field.types[-1].name)

        log_message = f'  {n:5}: {pretty_type:10} | {len(field.data):8} | {field.name}'
        if field.types:
            curr_type = field.types[0]
            if curr_type == GGUFValueType.STRING:
                content = field.contents()
                if len(content) > 60:
                    content = content[:57] + '...'
                log_message += ' = {0}'.format(repr(content))
            elif curr_type in reader.gguf_scalar_to_np:
                log_message += ' = {0}'.format(field.contents())
            else:
                content = repr(field.contents(slice(6)))
                if len(field.data) > 6:
                    content = content[:-1] + ', ...]'
                log_message += ' = {0}'.format(content)
        print(log_message)  # noqa: NP100
    if args.no_tensors:
        return
    print(f'* Dumping {len(reader.tensors)} tensor(s)')  # noqa: NP100
    for n, tensor in enumerate(reader.tensors, 1):
        prettydims = ', '.join('{0:5}'.format(d) for d in list(tensor.shape) + [1] * (4 - len(tensor.shape)))
        print(f'  {n:5}: {tensor.n_elements:10} | {prettydims} | {tensor.tensor_type.name:7} | {tensor.name}')
# noqa: NP100


def dump_metadata_json(reader: GGUFReader, args: argparse.Namespace) -> None:
    import json
    host_endian, file_endian = get_file_host_endian(reader)
    metadata: dict[str, Any] = {}
    tensors: dict[str, Any] = {}
    result = {
        "filename": args.model,
        "endian": file_endian,
        "metadata": metadata,
        "tensors": tensors,
    }
    for idx, field in enumerate(reader.fields.values()):
        curr: dict[str, Any] = {
            "index": idx,
            "type": field.types[0].name if field.types else 'UNKNOWN',
```

```
            "offset": field.offset,
        }
        metadata[field.name] = curr
        if field.types[:1] == [GGUFValueType.ARRAY]:
            curr["array_types"] = [t.name for t in field.types][1:]
            if not args.json_array:
                continue
            curr["value"] = field.contents()
        else:
            curr["value"] = field.contents()
    if not args.no_tensors:
        for idx, tensor in enumerate(reader.tensors):
            tensors[tensor.name] = {
                "index": idx,
                "shape": tensor.shape.tolist(),
                "type": tensor.tensor_type.name,
                "offset": tensor.field.offset,
            }
    json.dump(result, sys.stdout)


def markdown_table_with_alignment_support(header_map: list[dict[str, str]], data: list[dict[str, Any]]):
    # JSON to Markdown table formatting: https://stackoverflow.com/a/72983854/2850957

    # Alignment Utility Function
    def strAlign(padding: int, alignMode: str | None, strVal: str):
        if alignMode == 'center':
            return strVal.center(padding)
        elif alignMode == 'right':
            return strVal.rjust(padding - 1) + ' '
        elif alignMode == 'left':
            return ' ' + strVal.ljust(padding - 1)
        else: # default left
            return ' ' + strVal.ljust(padding - 1)

    def dashAlign(padding: int, alignMode: str | None):
        if alignMode == 'center':
            return ':' + '-' * (padding - 2) + ':'
        elif alignMode == 'right':
            return '-' * (padding - 1) + ':'
        elif alignMode == 'left':
            return ':' + '-' * (padding - 1)
        else: # default left
            return '-' * (padding)

    # Calculate Padding For Each Column Based On Header and Data Length
    rowsPadding = {}
    for index, columnEntry in enumerate(header_map):
          padCount = max([len(str(v)) for d in data for k, v in d.items() if k == columnEntry['key_name']],
default=0) + 2
        headerPadCount = len(columnEntry['header_name']) + 2
        rowsPadding[index] = headerPadCount if padCount <= headerPadCount else padCount

    # Render Markdown Header
```

```
    rows = []
                                rows.append('|'.join(strAlign(rowsPadding[index],       columnEntry.get('align'),
str(columnEntry['header_name'])) for index, columnEntry in enumerate(header_map)))
       rows.append('|'.join(dashAlign(rowsPadding[index], columnEntry.get('align')) for index, columnEntry in
enumerate(header_map)))

    # Render Tabular Data
    for item in data:
                                   rows.append('|'.join(strAlign(rowsPadding[index],    columnEntry.get('align'),
str(item[columnEntry['key_name']])) for index, columnEntry in enumerate(header_map)))

    # Convert Tabular String Rows Into String
    tableString = ""
    for row in rows:
        tableString += f'|{row}|\n'

    return tableString


def element_count_rounded_notation(count: int) -> str:
    if count > 1e15 :
        # Quadrillion
        scaled_amount = count * 1e-15
        scale_suffix = "Q"
    elif count > 1e12 :
        # Trillions
        scaled_amount = count * 1e-12
        scale_suffix = "T"
    elif count > 1e9 :
        # Billions
        scaled_amount = count * 1e-9
        scale_suffix = "B"
    elif count > 1e6 :
        # Millions
        scaled_amount = count * 1e-6
        scale_suffix = "M"
    elif count > 1e3 :
        # Thousands
        scaled_amount = count * 1e-3
        scale_suffix = "K"
    else:
        # Under Thousands
        scaled_amount = count
        scale_suffix = ""
    return f"{'~' if count > 1e3 else ''}{round(scaled_amount)}{scale_suffix}"


def translate_tensor_name(name):
    words = name.split(".")

    # Source: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md#standardized-tensor-names
    abbreviation_dictionary = {
        'token_embd': 'Token embedding',
        'pos_embd': 'Position embedding',
```

```
        'output_norm': 'Output normalization',
        'output': 'Output',
        'attn_norm': 'Attention normalization',
        'attn_norm_2': 'Attention normalization',
        'attn_qkv': 'Attention query-key-value',
        'attn_q': 'Attention query',
        'attn_k': 'Attention key',
        'attn_v': 'Attention value',
        'attn_output': 'Attention output',
        'ffn_norm': 'Feed-forward network normalization',
        'ffn_up': 'Feed-forward network "up"',
        'ffn_gate': 'Feed-forward network "gate"',
        'ffn_down': 'Feed-forward network "down"',
        'ffn_gate_inp': 'Expert-routing layer for the Feed-forward network in Mixture of Expert models',
        'ffn_gate_exp': 'Feed-forward network "gate" layer per expert in Mixture of Expert models',
        'ffn_down_exp': 'Feed-forward network "down" layer per expert in Mixture of Expert models',
        'ffn_up_exp': 'Feed-forward network "up" layer per expert in Mixture of Expert models',
        'ssm_in': 'State space model input projections',
        'ssm_conv1d': 'State space model rolling/shift',
        'ssm_x': 'State space model selective parametrization',
        'ssm_a': 'State space model state compression',
        'ssm_d': 'State space model skip connection',
        'ssm_dt': 'State space model time step',
        'ssm_out': 'State space model output projection',
        'blk': 'Block',
        'enc': 'Encoder',
        'dec': 'Decoder',
    }

    expanded_words = []
    for word in words:
        word_norm = word.strip().lower()
        if word_norm in abbreviation_dictionary:
            expanded_words.append(abbreviation_dictionary[word_norm].title())
        else:
            expanded_words.append(word.title())

    return ' '.join(expanded_words)


def dump_markdown_metadata(reader: GGUFReader, args: argparse.Namespace) -> None:
    host_endian, file_endian = get_file_host_endian(reader)
    markdown_content = ""
    markdown_content += f'# {args.model} - GGUF Internal File Dump\n\n'
    markdown_content += f'- Endian: {file_endian} endian\n'
    markdown_content += '\n'
    markdown_content += '## Key Value Metadata Store\n\n'
    markdown_content += f'There are {len(reader.fields)} key-value pairs in this file\n'
    markdown_content += '\n'

    kv_dump_table: list[dict[str, str | int]] = []
    for n, field in enumerate(reader.fields.values(), 1):
        if not field.types:
            pretty_type = 'N/A'
```

```
        elif field.types[0] == GGUFValueType.ARRAY:
            nest_count = len(field.types) - 1
            pretty_type = '[' * nest_count + str(field.types[-1].name) + ']' * nest_count
        else:
            pretty_type = str(field.types[-1].name)

        def escape_markdown_inline_code(value_string):
            # Find the longest contiguous sequence of backticks in the string then
            # wrap string with appropriate number of backticks required to escape it
            max_backticks = max((len(match.group(0)) for match in re.finditer(r'`+', value_string)), default=0)
            inline_code_marker = '`' * (max_backticks + 1)

            # If the string starts or ends with a backtick, add a space at the beginning and end
            if value_string.startswith('`') or value_string.endswith('`'):
                value_string = f" {value_string} "

            return f"{inline_code_marker}{value_string}{inline_code_marker}"

        total_elements = len(field.data)
        value = ""
        if len(field.types) == 1:
            curr_type = field.types[0]
            if curr_type == GGUFValueType.STRING:
                truncate_length = 60
                value_string = str(bytes(field.parts[-1]), encoding='utf-8')
                if len(value_string) > truncate_length:
                    head = escape_markdown_inline_code(value_string[:truncate_length // 2])
                    tail = escape_markdown_inline_code(value_string[-truncate_length // 2:])
                    value = "{head}...{tail}".format(head=head, tail=tail)
                else:
                    value = escape_markdown_inline_code(value_string)
            elif curr_type in reader.gguf_scalar_to_np:
                value = str(field.parts[-1][0])
        else:
            if field.types[0] == GGUFValueType.ARRAY:
                curr_type = field.types[1]
                array_elements = []

                if curr_type == GGUFValueType.STRING:
                    render_element = min(5, total_elements)
                    for element_pos in range(render_element):
                        truncate_length = 30
                           value_string = str(bytes(field.parts[-1 - (total_elements - element_pos - 1) * 2]),
encoding='utf-8')
                        if len(value_string) > truncate_length:
                            head = escape_markdown_inline_code(value_string[:truncate_length // 2])
                            tail = escape_markdown_inline_code(value_string[-truncate_length // 2:])
                            value = "{head}...{tail}".format(head=head, tail=tail)
                        else:
                            value = escape_markdown_inline_code(value_string)
                        array_elements.append(value)

                elif curr_type in reader.gguf_scalar_to_np:
                    render_element = min(7, total_elements)
```

```
                    for element_pos in range(render_element):
                        array_elements.append(str(field.parts[-1 - (total_elements - element_pos - 1)][0]))

                value = f'[ {", ".join(array_elements).strip()}{", ..." if total_elements > len(array_elements) else ""} ]'

                    kv_dump_table.append({"n":n,   "pretty_type":pretty_type,   "total_elements":total_elements, "field_name":field.name, "value":value})

    kv_dump_table_header_map = [
        {'key_name':'n',                'header_name':'POS',      'align':'right'},
        {'key_name':'pretty_type',      'header_name':'TYPE',     'align':'left'},
        {'key_name':'total_elements',   'header_name':'Count',    'align':'right'},
        {'key_name':'field_name',       'header_name':'Key',      'align':'left'},
        {'key_name':'value',            'header_name':'Value',    'align':'left'},
    ]

    markdown_content += markdown_table_with_alignment_support(kv_dump_table_header_map, kv_dump_table)

    markdown_content += "\n"

    if not args.no_tensors:
        # Group tensors by their prefix and maintain order
        tensor_prefix_order: list[str] = []
        tensor_name_to_key: dict[str, int] = {}
        tensor_groups: dict[str, list[ReaderTensor]] = {}
        total_elements = sum(tensor.n_elements for tensor in reader.tensors)

        # Parsing Tensors Record
        for key, tensor in enumerate(reader.tensors):
            tensor_components = tensor.name.split('.')

            # Classify Tensor Group
            tensor_group_name = "base"
            if tensor_components[0] == 'blk':
                tensor_group_name = f"{tensor_components[0]}.{tensor_components[1]}"
            elif tensor_components[0] in ['enc', 'dec'] and tensor_components[1] == 'blk':
                tensor_group_name = f"{tensor_components[0]}.{tensor_components[1]}.{tensor_components[2]}"
            elif tensor_components[0] in ['enc', 'dec']:
                tensor_group_name = f"{tensor_components[0]}"

            # Check if new Tensor Group
            if tensor_group_name not in tensor_groups:
                tensor_groups[tensor_group_name] = []
                tensor_prefix_order.append(tensor_group_name)

            # Record Tensor and Tensor Position
            tensor_groups[tensor_group_name].append(tensor)
            tensor_name_to_key[tensor.name] = key

        # Tensors Mapping Dump
                markdown_content  +=  f'##  Tensors  Overview  {element_count_rounded_notation(total_elements)} Elements\n\n'
        markdown_content += f'Total number of elements in all tensors: {total_elements} Elements\n'
```

```
        markdown_content += '\n'

        for group in tensor_prefix_order:
            tensors = tensor_groups[group]
            group_elements = sum(tensor.n_elements for tensor in tensors)
                                markdown_content  +=  f"-  [{translate_tensor_name(group)}  Tensor  Group  -
{element_count_rounded_notation(group_elements)} Elements](#{group.replace('.', '_')})\n"

        markdown_content += "\n"

        markdown_content += "### Tensor Data Offset\n"
        markdown_content += '\n'
        markdown_content += 'This table contains the offset and data segment relative to start of file\n'
        markdown_content += '\n'

        tensor_mapping_table: list[dict[str, str | int]] = []
        for key, tensor in enumerate(reader.tensors):
            data_offset_pretty = '{0:#16x}'.format(tensor.data_offset)
            data_size_pretty = '{0:#16x}'.format(tensor.n_bytes)
                                        tensor_mapping_table.append({"t_id":key,    "layer_name":tensor.name,
"data_offset":data_offset_pretty, "data_size":data_size_pretty})

        tensors_mapping_table_header_map = [
            {'key_name':'t_id',         'header_name':'T_ID',               'align':'right'},
            {'key_name':'layer_name',   'header_name':'Tensor Layer Name',  'align':'left'},
            {'key_name':'data_offset',  'header_name':'Data Offset (B)',    'align':'right'},
            {'key_name':'data_size',    'header_name':'Data Size (B)',      'align':'right'},
        ]

                markdown_content  +=  markdown_table_with_alignment_support(tensors_mapping_table_header_map,
tensor_mapping_table)
        markdown_content += "\n"

        for group in tensor_prefix_order:
            tensors = tensor_groups[group]
            group_elements = sum(tensor.n_elements for tensor in tensors)
            group_percentage = group_elements / total_elements * 100
                markdown_content += f"### <a name=\"{group.replace('.', '_')}\">{translate_tensor_name(group)}
Tensor Group : {element_count_rounded_notation(group_elements)} Elements</a>\n\n"

            # Precalculate column sizing for visual consistency
            prettify_element_est_count_size: int = 1
            prettify_element_count_size: int = 1
            prettify_dimension_max_widths: dict[int, int] = {}
            for tensor in tensors:
                                    prettify_element_est_count_size  =  max(prettify_element_est_count_size,
len(str(element_count_rounded_notation(tensor.n_elements))))
                prettify_element_count_size = max(prettify_element_count_size, len(str(tensor.n_elements)))
                for i, dimension_size in enumerate(list(tensor.shape) + [1] * (4 - len(tensor.shape))):
                                prettify_dimension_max_widths[i]  =  max(prettify_dimension_max_widths.get(i,1),
len(str(dimension_size)))

            # Generate Tensor Layer Table Content
            tensor_dump_table: list[dict[str, str | int]] = []
```

```
            for tensor in tensors:
                                    human_friendly_name  =  translate_tensor_name(tensor.name.replace(".weight",
".(W)").replace(".bias", ".(B)"))
                    pretty_dimension = ' x '.join(f'{str(d):>{prettify_dimension_max_widths[i]}}' for i, d in
enumerate(list(tensor.shape) + [1] * (4 - len(tensor.shape))))
                                                                                                element_count_est     =
f"({element_count_rounded_notation(tensor.n_elements):>{prettify_element_est_count_size}})"
                                                                    element_count_string    =    f"{element_count_est}
{tensor.n_elements:>{prettify_element_count_size}}"
                type_name_string = f"{tensor.tensor_type.name}"
                    tensor_dump_table.append({"t_id":tensor_name_to_key[tensor.name], "layer_name":tensor.name,
"human_layer_name":human_friendly_name,                                        "element_count":element_count_string,
"pretty_dimension":pretty_dimension, "tensor_type":type_name_string})

            tensor_dump_table_header_map = [
                    {'key_name':'t_id',                'header_name':'T_ID',
'align':'right'},
                     {'key_name':'layer_name',          'header_name':'Tensor  Layer  Name',
'align':'left'},
                          {'key_name':'human_layer_name',  'header_name':'Human  Friendly  Tensor  Layer  Name',
'align':'left'},
                     {'key_name':'element_count',     'header_name':'Elements',
'align':'left'},
                     {'key_name':'pretty_dimension',  'header_name':'Shape',
'align':'left'},
                    {'key_name':'tensor_type',        'header_name':'Type',
'align':'left'},
            ]

                      markdown_content  +=  markdown_table_with_alignment_support(tensor_dump_table_header_map,
tensor_dump_table)

            markdown_content += "\n"
                                          markdown_content    +=    f"-    Total    elements    in    {group}:
({element_count_rounded_notation(group_elements):>4}) {group_elements}\n"
            markdown_content += f"- Percentage of total elements: {group_percentage:.2f}%\n"
            markdown_content += "\n\n"

    print(markdown_content)  # noqa: NP100


def main() -> None:
    parser = argparse.ArgumentParser(description="Dump GGUF file metadata")
    parser.add_argument("model",           type=str,            help="GGUF format model filename")
    parser.add_argument("--no-tensors", action="store_true", help="Don't dump tensor metadata")
    parser.add_argument("--json",       action="store_true", help="Produce JSON output")
     parser.add_argument("--json-array", action="store_true", help="Include full array values in JSON output
(long)")
    parser.add_argument("--data-offset",    action="store_true", help="Start of data offset")
    parser.add_argument("--data-alignment", action="store_true", help="Data alignment applied globally to data
field")
    parser.add_argument("--markdown",   action="store_true", help="Produce markdown output")
    parser.add_argument("--verbose",    action="store_true", help="increase output verbosity")
```

```
    args = parser.parse_args(None if len(sys.argv) > 1 else ["--help"])

    logging.basicConfig(level=logging.DEBUG if args.verbose else logging.INFO)

    if not args.json and not args.markdown and not args.data_offset and not args.data_alignment:
        logger.info(f'* Loading: {args.model}')

    reader = GGUFReader(args.model, 'r')

    if args.json:
        dump_metadata_json(reader, args)
    elif args.markdown:
        dump_markdown_metadata(reader, args)
    elif args.data_offset:
        print(reader.data_offset)  # noqa: NP100
    elif args.data_alignment:
        print(reader.alignment)  # noqa: NP100
    else:
        dump_metadata(reader, args)


if __name__ == '__main__':
    main()

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

import logging
import argparse
import os
import re
import sys
from pathlib import Path
from typing import Any

# Necessary to load the local gguf package
if "NO_LOCAL_GGUF" not in os.environ and (Path(__file__).parent.parent.parent.parent / 'gguf-py').exists():
    sys.path.insert(0, str(Path(__file__).parent.parent.parent))

from gguf import GGUFReader, GGUFValueType, ReaderTensor  # noqa: E402

logger = logging.getLogger("gguf-dump")


def get_file_host_endian(reader: GGUFReader) -> tuple[str, str]:
    file_endian = reader.endianess.name
    if reader.byte_order == 'S':
        host_endian = 'BIG' if file_endian == 'LITTLE' else 'LITTLE'
    else:
        host_endian = file_endian
    return (host_endian, file_endian)
```

```
# For more information about what field.parts and field.data represent,
# please see the comments in the modify_gguf.py example.
def dump_metadata(reader: GGUFReader, args: argparse.Namespace) -> None:
    host_endian, file_endian = get_file_host_endian(reader)
    print(f'* File is {file_endian} endian, script is running on a {host_endian} endian host.')  # noqa: NP100
    print(f'* Dumping {len(reader.fields)} key/value pair(s)')  # noqa: NP100
    for n, field in enumerate(reader.fields.values(), 1):
        if not field.types:
            pretty_type = 'N/A'
        elif field.types[0] == GGUFValueType.ARRAY:
            nest_count = len(field.types) - 1
            pretty_type = '[' * nest_count + str(field.types[-1].name) + ']' * nest_count
        else:
            pretty_type = str(field.types[-1].name)

        log_message = f'  {n:5}: {pretty_type:10} | {len(field.data):8} | {field.name}'
        if field.types:
            curr_type = field.types[0]
            if curr_type == GGUFValueType.STRING:
                content = field.contents()
                if len(content) > 60:
                    content = content[:57] + '...'
                log_message += ' = {0}'.format(repr(content))
            elif curr_type in reader.gguf_scalar_to_np:
                log_message += ' = {0}'.format(field.contents())
            else:
                content = repr(field.contents(slice(6)))
                if len(field.data) > 6:
                    content = content[:-1] + ', ...]'
                log_message += ' = {0}'.format(content)
        print(log_message)  # noqa: NP100
    if args.no_tensors:
        return
    print(f'* Dumping {len(reader.tensors)} tensor(s)')  # noqa: NP100
    for n, tensor in enumerate(reader.tensors, 1):
        prettydims = ', '.join('{0:5}'.format(d) for d in list(tensor.shape) + [1] * (4 - len(tensor.shape)))
        print(f'  {n:5}: {tensor.n_elements:10} | {prettydims} | {tensor.tensor_type.name:7} | {tensor.name}')
# noqa: NP100


def dump_metadata_json(reader: GGUFReader, args: argparse.Namespace) -> None:
    import json
    host_endian, file_endian = get_file_host_endian(reader)
    metadata: dict[str, Any] = {}
    tensors: dict[str, Any] = {}
    result = {
        "filename": args.model,
        "endian": file_endian,
        "metadata": metadata,
        "tensors": tensors,
    }
    for idx, field in enumerate(reader.fields.values()):
        curr: dict[str, Any] = {
            "index": idx,
```