H-SGE: Hybrid Scene Graph Enrichment for Small Handgun Detection

================================================================

"H-SGE: A Hybrid Model Based on Scene Graph Enrichment"

Authors: Nasreen Jawaid, Najma Imtiaz Ali, Imtiaz Ali Korejo,

Imtiaz Ali Brohi, Noor Hafeizah Binti Hassan

Main Components:

1. GAN-based Image Enhancement

2. Scene Graph Generation

3. Knowledge Graph Integration

4. Multi-YOLO Detection

5. Weighted Fusion

Requirements:

- torch>=1.9.0

- torchvision>=0.10.0

- opencv-python>=4.5.0

- ultralytics>=8.0.0

- numpy>=1.21.0

- matplotlib>=3.3.0

- networkx>=2.6.0

- scipy>=1.7.0

"""

import torch

import torch.nn as nn

import torch.nn.functional as F

import cv2

import numpy as np

import networkx as nx

from typing import Dict, List, Tuple, Optional

import json

import logging

from pathlib import Path

import yaml

from dataclasses import dataclass

from ultralytics import YOLO

import torchvision.transforms as transforms

from torch.utils.data import Dataset, DataLoader

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class HSGEConfig:

"""Configuration class for H-SGE framework"""

# Model paths

yolov5\_path: str = "yolov5s.pt"

yolov7\_path: str = "yolov7.pt"

yolo10\_path: str = "yolo10n.pt"

yolo11\_path: str = "yolo11n.pt"

gan\_model\_path: str = "models/gan\_enhancer.pth"

# Detection parameters

confidence\_threshold: float = 0.25

iou\_threshold: float = 0.45

kg\_threshold: float = 0.7

# Image processing

input\_size: Tuple[int, int] = (640, 640)

# Knowledge graph parameters

spatial\_weight: float = 0.3

contextual\_weight: float = 0.4

temporal\_weight: float = 0.3

# Training parameters

batch\_size: int = 16

learning\_rate: float = 1e-4

epochs: int = 100

class GANEnhancer(nn.Module):

"""GAN-based image enhancement for small object visibility"""

def \_\_init\_\_(self, input\_channels=3, output\_channels=3):

super(GANEnhancer, self).\_\_init\_\_()

# Encoder

self.encoder = nn.Sequential(

nn.Conv2d(input\_channels, 64, 4, 2, 1),

nn.LeakyReLU(0.2, inplace=True),

nn.Conv2d(64, 128, 4, 2, 1),

nn.BatchNorm2d(128),

nn.LeakyReLU(0.2, inplace=True),

nn.Conv2d(128, 256, 4, 2, 1),

nn.BatchNorm2d(256),

nn.LeakyReLU(0.2, inplace=True),

nn.Conv2d(256, 512, 4, 2, 1),

nn.BatchNorm2d(512),

nn.LeakyReLU(0.2, inplace=True),

)

# Decoder

self.decoder = nn.Sequential(

nn.ConvTranspose2d(512, 256, 4, 2, 1),

nn.BatchNorm2d(256),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(256, 128, 4, 2, 1),

nn.BatchNorm2d(128),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(128, 64, 4, 2, 1),

nn.BatchNorm2d(64),

nn.ReLU(inplace=True),

nn.ConvTranspose2d(64, output\_channels, 4, 2, 1),

nn.Tanh()

)

def forward(self, x):

encoded = self.encoder(x)

enhanced = self.decoder(encoded)

return enhanced

class SceneGraphGenerator:

"""Scene graph generation for spatial-semantic relationships"""

def \_\_init\_\_(self):

self.object\_classes = [

'person', 'handgun', 'vehicle', 'building', 'furniture',

'electronics', 'outdoor', 'indoor', 'public\_space', 'private\_space'

]

self.relationships = [

'holding', 'near', 'in', 'on', 'next\_to', 'behind',

'in\_front\_of', 'inside', 'outside', 'threatening'

]

def generate\_scene\_graph(self, detections: List[Dict], image\_info: Dict) -> nx.DiGraph:

"""Generate scene graph from object detections"""

G = nx.DiGraph()

# Add nodes (objects)

for i, detection in enumerate(detections):

G.add\_node(i,

class\_name=detection['class'],

confidence=detection['confidence'],

bbox=detection['bbox'],

features=detection.get('features', None))

# Add edges (relationships)

for i in range(len(detections)):

for j in range(i+1, len(detections)):

relationship = self.\_compute\_spatial\_relationship(

detections[i]['bbox'],

detections[j]['bbox']

)

if relationship:

G.add\_edge(i, j, relationship=relationship)

return G

def \_compute\_spatial\_relationship(self, bbox1: List[float], bbox2: List[float]) -> Optional[str]:

"""Compute spatial relationship between two bounding boxes"""

x1, y1, w1, h1 = bbox1

x2, y2, w2, h2 = bbox2

# Calculate centers and distances

center1 = (x1 + w1/2, y1 + h1/2)

center2 = (x2 + w2/2, y2 + h2/2)

distance = np.sqrt((center1[0] - center2[0])\*\*2 + (center1[1] - center2[1])\*\*2)

# Determine relationships based on spatial proximity and relative positions

if distance < 50: # Close proximity

if abs(center1[1] - center2[1]) < 20: # Same horizontal level

return 'next\_to'

elif center1[1] < center2[1]:

return 'above'

else:

return 'below'

elif distance < 100:

return 'near'

return None

class KnowledgeGraph:

"""Knowledge graph for contextual validation"""

def \_\_init\_\_(self, kg\_file: str = None):

self.graph = nx.Graph()

self.threat\_patterns = {}

self.contextual\_rules = {}

if kg\_file and Path(kg\_file).exists():

self.load\_knowledge\_graph(kg\_file)

else:

self.\_initialize\_default\_kg()

def \_initialize\_default\_kg(self):

"""Initialize default knowledge graph with handgun-specific patterns"""

# Threat scenarios

threat\_scenarios = [

('person', 'holding', 'handgun', 0.9),

('handgun', 'in', 'public\_space', 0.8),

('person', 'threatening\_pose', 'handgun', 0.85),

('handgun', 'near', 'crowd', 0.75),

('concealed', 'handgun', 'suspicious\_behavior', 0.7)

]

for scenario in threat\_scenarios:

if len(scenario) == 4:

entity1, relation, entity2, weight = scenario

self.graph.add\_edge(entity1, entity2,

relation=relation, weight=weight)

# Contextual rules for validation

self.contextual\_rules = {

'high\_risk\_locations': ['bank', 'school', 'airport', 'government\_building'],

'suspicious\_behaviors': ['aggressive\_pose', 'concealment', 'rapid\_movement'],

'innocent\_contexts': ['toy\_gun', 'police\_officer', 'security\_guard']

}

def validate\_detection(self, scene\_graph: nx.DiGraph, detection\_node: int) -> float:

"""Validate detection using knowledge graph reasoning"""

base\_confidence = 0.5

contextual\_boost = 0.0

# Get detection info

detection\_data = scene\_graph.nodes[detection\_node]

# Check spatial relationships

for neighbor in scene\_graph.neighbors(detection\_node):

neighbor\_data = scene\_graph.nodes[neighbor]

edge\_data = scene\_graph.edges[detection\_node, neighbor]

relationship = edge\_data.get('relationship', '')

# Apply knowledge graph rules

if self.\_matches\_threat\_pattern(detection\_data, neighbor\_data, relationship):

contextual\_boost += 0.2

# Check scene context

scene\_context = self.\_analyze\_scene\_context(scene\_graph)

if scene\_context.get('high\_risk\_environment', False):

contextual\_boost += 0.15

return min(base\_confidence + contextual\_boost, 1.0)

def \_matches\_threat\_pattern(self, det1: Dict, det2: Dict, relationship: str) -> bool:

"""Check if detection matches known threat patterns"""

# Check if pattern exists in knowledge graph

entity1 = det1.get('class\_name', '')

entity2 = det2.get('class\_name', '')

if self.graph.has\_edge(entity1, entity2):

edge\_data = self.graph.edges[entity1, entity2]

return edge\_data.get('relation', '') == relationship

return False

def \_analyze\_scene\_context(self, scene\_graph: nx.DiGraph) -> Dict:

"""Analyze overall scene context for threat assessment"""

context = {'high\_risk\_environment': False}

# Count people and objects

person\_count = sum(1 for \_, data in scene\_graph.nodes(data=True)

if data.get('class\_name') == 'person')

# Check for high-risk indicators

if person\_count > 5: # Crowded area

context['high\_risk\_environment'] = True

return context

def load\_knowledge\_graph(self, filepath: str):

"""Load knowledge graph from file"""

with open(filepath, 'r') as f:

kg\_data = json.load(f)

for edge in kg\_data.get('edges', []):

self.graph.add\_edge(

edge['source'],

edge['target'],

\*\*edge.get('attributes', {})

)

class MultiYOLODetector:

"""Multi-YOLO detection with ensemble fusion"""

def \_\_init\_\_(self, config: HSGEConfig):

self.config = config

self.models = {}

self.\_load\_models()

def \_load\_models(self):

"""Load all YOLO variants"""

model\_configs = {

'yolov5': self.config.yolov5\_path,

'yolov7': self.config.yolov7\_path,

'yolo10': self.config.yolo10\_path,

'yolo11': self.config.yolo11\_path

}

for name, path in model\_configs.items():

try:

if Path(path).exists():

self.models[name] = YOLO(path)

logger.info(f"Loaded {name} from {path}")

else:

# Load pretrained model

self.models[name] = YOLO(Path(path).name)

logger.info(f"Loaded pretrained {name}")

except Exception as e:

logger.warning(f"Failed to load {name}: {e}")

def detect(self, image: np.ndarray) -> Dict[str, List[Dict]]:

"""Run detection with all available YOLO models"""

detections = {}

for name, model in self.models.items():

try:

results = model(image,

conf=self.config.confidence\_threshold,

iou=self.config.iou\_threshold)

detections[name] = self.\_parse\_results(results[0])

except Exception as e:

logger.error(f"Detection failed for {name}: {e}")

detections[name] = []

return detections

def \_parse\_results(self, result) -> List[Dict]:

"""Parse YOLO results into standard format"""

detections = []

if result.boxes is not None:

boxes = result.boxes.xyxy.cpu().numpy()

confidences = result.boxes.conf.cpu().numpy()

classes = result.boxes.cls.cpu().numpy()

for box, conf, cls in zip(boxes, confidences, classes):

x1, y1, x2, y2 = box

detections.append({

'class': int(cls),

'confidence': float(conf),

'bbox': [float(x1), float(y1),

float(x2-x1), float(y2-y1)]

})

return detections

class WeightedFusion:

"""Weighted fusion for multi-model ensemble"""

def \_\_init\_\_(self, model\_weights: Dict[str, float] = None):

self.model\_weights = model\_weights or {

'yolov5': 0.2,

'yolov7': 0.25,

'yolo10': 0.25,

'yolo11': 0.3

}

def fuse\_detections(self,

multi\_detections: Dict[str, List[Dict]],

kg\_scores: Dict[str, float] = None) -> List[Dict]:

"""Fuse detections from multiple models with knowledge graph validation"""

all\_detections = []

# Collect all detections with model weights

for model\_name, detections in multi\_detections.items():

weight = self.model\_weights.get(model\_name, 0.25)

for det in detections:

det\_copy = det.copy()

det\_copy['model'] = model\_name

det\_copy['model\_weight'] = weight

det\_copy['weighted\_confidence'] = det['confidence'] \* weight

# Add knowledge graph score if available

if kg\_scores and model\_name in kg\_scores:

det\_copy['kg\_score'] = kg\_scores[model\_name]

det\_copy['final\_confidence'] = (

det\_copy['weighted\_confidence'] \* 0.7 +

det\_copy['kg\_score'] \* 0.3

)

else:

det\_copy['final\_confidence'] = det\_copy['weighted\_confidence']

all\_detections.append(det\_copy)

# Apply Non-Maximum Suppression

fused\_detections = self.\_weighted\_nms(all\_detections)

return fused\_detections

def \_weighted\_nms(self, detections: List[Dict], iou\_threshold: float = 0.5) -> List[Dict]:

"""Weighted Non-Maximum Suppression"""

if not detections:

return []

# Sort by final confidence

detections.sort(key=lambda x: x['final\_confidence'], reverse=True)

keep = []

while detections:

current = detections.pop(0)

keep.append(current)

# Remove overlapping detections

detections = [det for det in detections

if self.\_calculate\_iou(current['bbox'], det['bbox']) < iou\_threshold]

return keep

def \_calculate\_iou(self, box1: List[float], box2: List[float]) -> float:

"""Calculate Intersection over Union"""

x1, y1, w1, h1 = box1

x2, y2, w2, h2 = box2

# Calculate intersection

xi1 = max(x1, x2)

yi1 = max(y1, y2)

xi2 = min(x1 + w1, x2 + w2)

yi2 = min(y1 + h1, y2 + h2)

if xi2 <= xi1 or yi2 <= yi1:

return 0.0

intersection = (xi2 - xi1) \* (yi2 - yi1)

union = w1 \* h1 + w2 \* h2 - intersection

return intersection / union if union > 0 else 0.0

class HSGEFramework:

"""Main H-SGE Framework integrating all components"""

def \_\_init\_\_(self, config: HSGEConfig):

self.config = config

# Initialize components

self.gan\_enhancer = GANEnhancer()

self.scene\_graph\_generator = SceneGraphGenerator()

self.knowledge\_graph = KnowledgeGraph()

self.multi\_yolo = MultiYOLODetector(config)

self.fusion = WeightedFusion()

# Load GAN model if available

self.\_load\_gan\_model()

def \_load\_gan\_model(self):

"""Load pre-trained GAN model"""

if Path(self.config.gan\_model\_path).exists():

try:

self.gan\_enhancer.load\_state\_dict(

torch.load(self.config.gan\_model\_path, map\_location='cpu')

)

self.gan\_enhancer.eval()

logger.info("Loaded GAN enhancer model")

except Exception as e:

logger.warning(f"Failed to load GAN model: {e}")

def detect\_handgun(self, image: np.ndarray) -> Dict:

"""Complete H-SGE detection pipeline"""

# Stage 1: GAN-based Enhancement

enhanced\_image = self.\_enhance\_image(image)

# Stage 2: Multi-YOLO Detection

multi\_detections = self.multi\_yolo.detect(enhanced\_image)

# Stage 3: Scene Graph Generation

all\_detections = []

for model\_dets in multi\_detections.values():

all\_detections.extend(model\_dets)

scene\_graph = self.scene\_graph\_generator.generate\_scene\_graph(

all\_detections, {'image\_shape': image.shape}

)

# Stage 4: Knowledge Graph Validation

kg\_scores = {}

for model\_name, detections in multi\_detections.items():

kg\_scores[model\_name] = 0.0

for i, det in enumerate(detections):

if det['class'] == 0: # Assuming class 0 is handgun

kg\_score = self.knowledge\_graph.validate\_detection(scene\_graph, i)

kg\_scores[model\_name] = max(kg\_scores[model\_name], kg\_score)

# Stage 5: Weighted Fusion

final\_detections = self.fusion.fuse\_detections(multi\_detections, kg\_scores)

return {

'detections': final\_detections,

'enhanced\_image': enhanced\_image,

'scene\_graph': scene\_graph,

'model\_scores': kg\_scores

}

def \_enhance\_image(self, image: np.ndarray) -> np.ndarray:

"""Apply GAN-based image enhancement"""

# Convert to tensor

transform = transforms.Compose([

transforms.ToPILImage(),

transforms.Resize(self.config.input\_size),

transforms.ToTensor(),

transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])

])

# Check image quality - skip enhancement for high-quality images

brightness = np.mean(cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY))

if brightness > 120: # High quality image

return image

try:

input\_tensor = transform(image).unsqueeze(0)

with torch.no\_grad():

enhanced\_tensor = self.gan\_enhancer(input\_tensor)

# Convert back to numpy

enhanced\_tensor = (enhanced\_tensor.squeeze(0) + 1) / 2 # Denormalize

enhanced\_image = enhanced\_tensor.permute(1, 2, 0).numpy()

enhanced\_image = (enhanced\_image \* 255).astype(np.uint8)

return enhanced\_image

except Exception as e:

logger.warning(f"GAN enhancement failed: {e}")

return image

# Evaluation and Training Functions

class HandgunDataset(Dataset):

"""Dataset class for handgun detection"""

def \_\_init\_\_(self, image\_dir: str, annotation\_file: str, transform=None):

self.image\_dir = Path(image\_dir)

self.transform = transform

# Load annotations

with open(annotation\_file, 'r') as f:

self.annotations = json.load(f)

def \_\_len\_\_(self):

return len(self.annotations)

def \_\_getitem\_\_(self, idx):

ann = self.annotations[idx]

# Load image

image\_path = self.image\_dir / ann['image\_file']

image = cv2.imread(str(image\_path))

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

if self.transform:

image = self.transform(image)

return {

'image': image,

'annotations': ann['annotations'],

'image\_id': ann.get('image\_id', idx)

}

def evaluate\_hsge(model: HSGEFramework, test\_loader: DataLoader) -> Dict:

"""Evaluate H-SGE framework on test dataset"""

model.gan\_enhancer.eval()

metrics = {

'precision': [],

'recall': [],

'f1\_score': [],

'map\_50': []

}

for batch in test\_loader:

images = batch['image']

annotations = batch['annotations']

for image, ann in zip(images, annotations):

# Convert tensor to numpy if needed

if torch.is\_tensor(image):

image = image.permute(1, 2, 0).numpy()

image = (image \* 255).astype(np.uint8)

# Run detection

results = model.detect\_handgun(image)

# Calculate metrics

batch\_metrics = calculate\_detection\_metrics(

results['detections'], ann

)

for key in metrics:

if key in batch\_metrics:

metrics[key].append(batch\_metrics[key])

# Average metrics

final\_metrics = {}

for key, values in metrics.items():

if values:

final\_metrics[key] = np.mean(values)

else:

final\_metrics[key] = 0.0

return final\_metrics

def calculate\_detection\_metrics(predictions: List[Dict], ground\_truth: List[Dict]) -> Dict:

"""Calculate detection metrics (precision, recall, F1, mAP)"""

if not predictions:

return {'precision': 0.0, 'recall': 0.0, 'f1\_score': 0.0}

# Simple IoU-based matching

tp = 0

fp = 0

fn = len(ground\_truth)

matched\_gt = set()

for pred in predictions:

best\_iou = 0.0

best\_gt\_idx = -1

for i, gt in enumerate(ground\_truth):

if i in matched\_gt:

continue

iou = calculate\_bbox\_iou(pred['bbox'], gt['bbox'])

if iou > best\_iou:

best\_iou = iou

best\_gt\_idx = i

if best\_iou > 0.5: # IoU threshold

tp += 1

fn -= 1

matched\_gt.add(best\_gt\_idx)

else:

fp += 1

precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0

recall = tp / (tp + fn) if (tp + fn) > 0 else 0.0

f1\_score = 2 \* (precision \* recall) / (precision + recall) if (precision + recall) > 0 else 0.0

return {

'precision': precision,

'recall': recall,

'f1\_score': f1\_score

}

def calculate\_bbox\_iou(box1: List[float], box2: List[float]) -> float:

"""Calculate IoU between two bounding boxes"""

x1, y1, w1, h1 = box1

x2, y2, w2, h2 = box2

# Convert to corner coordinates

x1\_max, y1\_max = x1 + w1, y1 + h1

x2\_max, y2\_max = x2 + w2, y2 + h2

# Calculate intersection

xi1 = max(x1, x2)

yi1 = max(y1, y2)

xi2 = min(x1\_max, x2\_max)

yi2 = min(y1\_max, y2\_max)

if xi2 <= xi1 or yi2 <= yi1:

return 0.0

intersection = (xi2 - xi1) \* (yi2 - yi1)

union = w1 \* h1 + w2 \* h2 - intersection

return intersection / union if union > 0 else 0.0

# Main execution example

if \_\_name\_\_ == "\_\_main\_\_":

# Configuration

config = HSGEConfig()

# Initialize H-SGE framework

hsge = HSGEFramework(config)

# Example usage

image\_path = "path/to/test/image.jpg"

if Path(image\_path).exists():

# Load test image

image = cv2.imread(image\_path)

image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Run detection

results = hsge.detect\_handgun(image)

print(f"Detected {len(results['detections'])} handguns")

for i, detection in enumerate(results['detections']):

print(f"Detection {i+1}: confidence={detection['final\_confidence']:.3f}")

else:

print("Please provide a valid image path for testing")