Graph Attention Autoencoder for MultiOmics Integration,

Risk Stratification and Biomarker Identification in Cancer

File: graph_autoencoder.py

First 75 lines:

```
import torch
import torch.nn.functional as F
from torch_geometric.data import Data, Batch
from torch_geometric.nn import GCNConv, GAE, GATv2Conv
from torch.utils.data import DataLoader
from torch.optim import Adam
from torch_geometric.utils import negative_sampling
from torch.nn.functional import cosine_similarity
from torch.optim import AdamW  # NEW: Import AdamW
from torch.optim.lr_scheduler import StepLR

def collate_graph_data(batch):
    return Batch.from_data_list(batch)

@staticmethod
def create_data_loader(train_data, batch_size=1, shuffle=True):
    graph_data = list(train_data.values())
    return DataLoader(graph_data, batch_size=batch_size, shuffle=shuffle, collate_fn=collate_graph_data)

class GATv2Encoder(torch.nn.Module):
    def __init__(self, in_channels, edge_attr_channels, out_channels, heads=1, concat=True):
        super(GATv2Encoder, self).__init__()
        self.conv1 = GATv2Conv(in_channels, out_channels, heads=heads, concat=concat, edge_dim=edge_attr_channels, add_self_loops=False)
        self.attention_weights1 = None;

    def forward(self, x, edge_index, edge_attr):
```

```
        x, _a1_ = self.conv1(x, edge_index, edge_attr, return_attention_weights=True)
        # x = x.relu()
        self.attention_weights1 = _a1_
        return x


class GATv2Decoder(torch.nn.Module):
    def __init__(self, in_channels, original_feature_size):
        super(GATv2Decoder, self).__init__()
        self.edge_weight_predictor = torch.nn.Sequential(
            torch.nn.Linear(2 * in_channels, 128),  # First linear layer
            torch.nn.ReLU(),                        # Activation function
            torch.nn.Linear(128, 1)                 # Output layer
        )
        self.fc = torch.nn.Linear(in_channels, original_feature_size)

    def forward(self, z, sigmoid=True):
        x_reconstructed = self.fc(z)
        return x_reconstructed

    def predict_edge_weights(self, z, edge_index):
        edge_embeddings = torch.cat([z[edge_index[0]], z[edge_index[1]]], dim=-1)
        return self.edge_weight_predictor(edge_embeddings)



def graph_reconstruction_loss(pred_features, true_features):
    node_loss = F.mse_loss(pred_features, true_features)
    return node_loss


def edge_reconstruction_loss(z, pos_edge_index, neg_edge_index=None):
    # Get the positive edge logits (inner products)
```

```
    pos_logits = (z[pos_edge_index[0]] * z[pos_edge_index[1]]).sum(dim=-1)
    pos_loss = F.binary_cross_entropy_with_logits(pos_logits, torch.ones_like(pos_logits))

    # If negative samples are not provided, generate them
    if neg_edge_index is None:
        neg_edge_index = negative_sampling(pos_edge_index, z.size(0))

    # Get the negative edge logits (inner products)
    neg_logits = (z[neg_edge_index[0]] * z[neg_edge_index[1]]).sum(dim=-1)
    neg_loss = F.binary_cross_entropy_with_logits(neg_logits, torch.zeros_like(neg_logits))

    return pos_loss + neg_loss

def edge_weight_reconstruction_loss(pred_weights, true_weights):
    pred_weights = pred_weights.squeeze(-1)
    return F.mse_loss(pred_weights, true_weights)

class GraphAutoencoder:
    def __init__(self, in_channels, edge_attr_channels, out_channels, original_feature_size, learning_rate=0.01):
```

Last 75 lines:

```
            pred_edge_weights = self.Gdecoder.predict_edge_weights(z, data.edge_index)
            edge_weight_loss = edge_weight_reconstruction_loss(pred_edge_weights, data.edge_attr)
            loss = (loss_weight_node * node_loss) + (loss_weight_edge * edge_loss) + (loss_weight_edge_attr * edge_weight_loss)
            print(f"node_loss: {node_loss}, edge_loss: {edge_loss:.4f}, edge_weight_loss: {edge_weight_loss:.4f}, cosine_similarity: {cos_sim:.4f}")
            loss.backward()
            self.optimizer.step()
```

```
            total_loss += loss.item()

        avg_loss, avg_cosine_similarity = total_loss / len(data_loader), total_cosine_similarity / len(data_loader)
        return avg_loss, avg_cosine_similarity  # Return both the average loss and average cosine similarity


    def fit(self, train_loader, validation_loader, epochs):
        train_losses = []
        val_losses = []

        for epoch in range(1, epochs + 1):
            train_loss, train_cosine_similarity = self.train(train_loader)  # Unpack the tuple
            torch.cuda.empty_cache()
            val_loss, val_cosine_similarity = self.validate(validation_loader)  # Unpack the tuple

            print(f"Epoch: {epoch}, Train Loss: {train_loss:.4f}, Train Cosine Similarity: {train_cosine_similarity:.4f}, Validation Loss: {val_loss:.4f}, Validation Cosine Similarity: {val_cosine_similarity:.4f}")

            # NEW: Step the learning rate scheduler
            self.scheduler.step()

        return train_losses, val_losses


    def validate(self, validation_loader):
        self.gae.eval()  # set the model to evaluation mode
        total_loss = 0
        total_cosine_similarity = 0

        with torch.no_grad():  # No gradient computation during validation
```

```
            for data in validation_loader:
                data = data.to(self.device)
                z = self.gae(data.x, data.edge_index, data.edge_attr)
                x_reconstructed = self.Gdecoder(z)
                node_loss = graph_reconstruction_loss(x_reconstructed, data.x)
                edge_loss = edge_reconstruction_loss(z, data.edge_index)

                # Calculate cosine similarity as you do in the train method
                cos_sim = cosine_similarity(x_reconstructed, data.x, dim=-1).mean()
                total_cosine_similarity += cos_sim.item()  # Aggregate for all batches

                loss = node_loss + edge_loss
                total_loss += loss.item()

        avg_loss = total_loss / len(validation_loader)
        avg_cosine_similarity = total_cosine_similarity / len(validation_loader)  # Calculate average cosine similarity

        return avg_loss, avg_cosine_similarity  # Return both the average loss and average cosine similarity



    def evaluate(self, test_loader):
        self.gae.eval()  # Set the model to evaluation mode
        total_loss = 0
        total_accuracy = 0
        # torch.cuda.empty_cache()
        with torch.no_grad():  # No gradient computation during evaluation
            for data in test_loader:
                data = data.to(self.device)
                z = self.gae(data.x, data.edge_index, data.edge_attr)
```

```
        x_reconstructed = self.Gdecoder(z)
        node_loss = graph_reconstruction_loss(x_reconstructed, data.x)
        edge_loss = edge_reconstruction_loss(z, data.edge_index)

        loss = node_loss + edge_loss
        total_loss += loss.item()

    avg_loss = total_loss / len(test_loader)
    avg_accuracy = total_accuracy / len(test_loader)  # Calculate average accuracy

    return avg_loss, avg_accuracy
```

File: GraphAnalysis.py

First 75 lines:

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from lifelines.statistics import logrank_test
from itertools import combinations
import matplotlib.pyplot as plt
from yellowbrick.cluster import KElbowVisualizer
import pandas as pd
import seaborn as sns
from lifelines import KaplanMeierFitter
import matplotlib.cm as cm
import itertools
import torch
```

```
class GraphAnalysis:
    def __init__(self, EXTRACTER):
        self.extracter = EXTRACTER
        self.process()

    def process(self):
        latent_features_list = list(self.extracter.latent_feat_dict.values())
        patient_list = list(self.extracter.latent_feat_dict.keys())
        latentF = torch.stack(latent_features_list, dim=0)
        self.latentF = np.squeeze(latentF.numpy())
        self.pIDs = patient_list
        self.df = pd.DataFrame(columns=['PC1','PC2','tX','tY','groups'], index=self.pIDs)
        self.clnc_df = pd.read_csv('./data/survival.hnsc_data.csv').set_index('PatientID')
        self.df = self.df.join(self.clnc_df)

    def pca_tsne(self):
        pca = PCA(n_components=2)
        X_pca = pca.fit_transform(self.latentF)
        self.df['PC1'] = X_pca[:,0]
        self.df['PC2'] = X_pca[:,1]
        tsne = TSNE(n_components=2)
        X_tsne = tsne.fit_transform(self.latentF)
        self.df['tX'] = X_tsne[:,0]
        self.df['tY'] = X_tsne[:,1]

    def find_optimal_clusters(self, min_clusters=2, max_clusters=11, save_path='./results/kelbow'):
        model = KMeans(random_state=42)
        visualizer = KElbowVisualizer(model, k=(min_clusters, max_clusters))
        visualizer.fit(self.latentF)
        visualizer.show()
```

```
    fig = visualizer.ax.get_figure()
    fig.savefig(save_path + ".png", dpi=150)
    fig.savefig(save_path + ".jpeg", format="jpeg", dpi=150)
    self.optimal_clusters = visualizer.elbow_value_

def cluster_data(self):
    if self.optimal_clusters is None:
        raise ValueError("Please run 'find_optimal_clusters' method before clustering the data.")
    kmeans = KMeans(n_clusters=self.optimal_clusters, random_state=0).fit(self.latentF)
    self.labels = kmeans.labels_
    self.df['groups'] = self.labels
    self.generate_color_list_based_on_median_survival()

def cluster_data2(self, kclust):
    kmeans = KMeans(n_clusters=kclust, random_state=0).fit(self.latentF)
    self.labels = kmeans.labels_
    self.df['groups'] = self.labels
    self.generate_color_list_based_on_median_survival()

def visualize_clusters(self):
    plt.figure(figsize=(20,8))
    plt.subplot(1,2,1)
    sns.scatterplot(data=self.df, x='PC1', y='PC2', hue='groups', palette=self.color_list)
    plt.subplot(1,2,2)
    sns.scatterplot(data=self.df, x='tX', y='tY', hue='groups', palette=self.color_list)

def save_visualize_clusters(self):
    plt.figure(figsize=(10,8))
    sns.scatterplot(data=self.df, x='PC1', y='PC2', hue='groups', palette=self.color_list)
    plt.savefig('./results/temp_pca.jpeg', dpi=300)
```

Last 75 lines:

```
        groups = self.df['groups'].unique()
        significant_pairs = []
        for pair in itertools.combinations(groups, 2):
            group_a = self.df[self.df['groups'] == pair[0]]
            group_b = self.df[self.df['groups'] == pair[1]]
            results = logrank_test(group_a['Overall Survival (Months)'], group_b['Overall Survival (Months)'], group_a['Overall Survival Status'], group_b['Overall Survival Status'])
            if results.p_value < alpha:
                significant_pairs.append(pair)
        self.significant_pairs = significant_pairs
        return self.significant_pairs

    def generate_summary_table(self):
        groups = self.df['groups'].unique()
        summary_table = pd.DataFrame(columns=['Total number of patients', 'Alive', 'Deceased', 'Median survival time'], index=groups)
        for group in groups:
            group_data = self.df[self.df['groups'] == group]
            total_patients = len(group_data)
            alive = len(group_data[group_data['Overall Survival Status'] == 0])
            deceased = len(group_data[group_data['Overall Survival Status'] == 1])
            kmf = KaplanMeierFitter()
            kmf.fit(group_data['Overall Survival (Months)'], group_data['Overall Survival Status'])
            median_survival_time = kmf.median_survival_time_
            summary_table.loc[group] = [total_patients, alive, deceased, median_survival_time]
        return summary_table

    def plot_kaplan_meier(self, plot_for_groups=True, name='temp_k5'):
        kmf = KaplanMeierFitter()
```

```
        plt.figure(figsize=(8, 6))
        plt.grid(False)
        if plot_for_groups:
            groups = sorted(self.df['groups'].unique())
            for i, group in enumerate(groups):
                group_data = self.df[self.df['groups'] == group]
                kmf.fit(group_data['Overall Survival (Months)'], group_data['Overall Survival Status'], label=f'Group
{group}')
                kmf.plot(ci_show=False, linewidth=2, color=self.color_list[group])
            plt.title("Kaplan-Meier Curves for Each Group")
        else:
            kmf.fit(self.df['Overall Survival (Months)'], self.df['Overall Survival Status'], label='All Data')
            kmf.plot(ci_show=False, linewidth=2, color='black')
            plt.title("Kaplan-Meier Curve for All Data")
        plt.gca().set_facecolor('#f5f5f5')
        plt.grid(color='lightgrey', linestyle='-', linewidth=0.5)
        plt.xlabel("Overall Survival (Months)", fontweight='bold')
        plt.ylabel("Survival Probability", fontweight='bold')
        plt.legend()
        plt.savefig('./results/{}_plan_meir.jpeg'.format(name), dpi=300)
        plt.savefig('./results/{}_plan_meir.png'.format(name), dpi=300)
        plt.show()

    def club_two_groups(self, primary_group, secondary_group):
        self.df.loc[self.df['groups'] == secondary_group, 'groups'] = primary_group
        unique_groups = sorted(self.df['groups'].unique())
        mapping = {old: new for new, old in enumerate(unique_groups)}
        self.df['groups'] = self.df['groups'].map(mapping)
        self.generate_color_list_based_on_median_survival()
        self.summary_table = self.generate_summary_table()
```

```
    def plot_median_survival_bar(self, name='temp_k5'):
        summary_df = self.generate_summary_table()
        summary_df['group'] = summary_df.index
        max_val = summary_df["Median survival time"].replace(np.inf, np.nan).max()
        summary_df["Display Median"] = summary_df["Median survival time"].replace(np.inf, max_val * 1.1)
        summary_df = summary_df.sort_index()
        colors = [self.color_list[group] for group in summary_df.index]
        num_groups = len(summary_df)
        plt.figure(figsize=(6, num_groups * 0.8))
        plt.grid(False)
        sns.barplot(data=summary_df, y='group', x="Display Median", palette=colors, orient="h", order=summary_df.index)
        plt.xlabel("Median Survival Time (Months)")
        plt.ylabel("Groups")
        plt.title("Median Survival Time by Group")
        plt.tight_layout()
        plt.savefig('./results/{}_median_survival.jpeg'.format(name), dpi=300)
        plt.savefig('./results/{}_median_survival.png'.format(name), dpi=300)
        plt.show()
```

File: Attention_Extracter.py

First 75 lines:

```
import torch
import pickle
import numpy as np

class Attention_Extracter:
    def __init__(self, graph_data_dict_path, encoder_model, gpu=False):
```

```
    self.torch_device = 'cuda' if gpu else 'cpu'

    self.graph_data_dict = torch.load(graph_data_dict_path)
    self.encoder_model = encoder_model
    self.encoder_model.to(self.torch_device)
    self.encoder_model.eval()
    self.latent_feat_dict, self.attention_scores1 = self.extract_latent_attention_features()

def extract_latent_attention_features(self):
    latent_features = {}
    attention_scores1 = {}

    with torch.no_grad():
        for graph_id, data in self.graph_data_dict.items():
            data = data.to(self.torch_device)
            z, attention_weights = self.encoder_model(data.x, data.edge_index, data.edge_attr)
            latent_features[graph_id] = z.cpu()

            # Handling the case where attention_weights is a tuple or other data structure
            if isinstance(attention_weights, (list, tuple)):
                attention_scores1[graph_id] = [aw for aw in attention_weights]
            else:
                attention_scores1[graph_id] = attention_weights.cpu()

    return latent_features, attention_scores1

def load_edge_indices(self, glist_path, edge_matrix_path):
    with open(glist_path, 'rb') as f:
        glist = pickle.load(f)
```

```
        edge_matrix = np.load(edge_matrix_path)
        edge_matrix = torch.tensor(edge_matrix, dtype=torch.float)
        edge_index = torch.nonzero(edge_matrix, as_tuple=False).t().contiguous()
        edge_indices_dict = {}

        for i in range(edge_index.shape[1]):
            index1, index2 = edge_index[0, i].item(), edge_index[1, i].item()
            gene1, gene2 = glist[index1], glist[index2]
            edge_indices_dict[(index1, index2)] = (gene1, gene2)

        return edge_indices_dict
```

Last 75 lines:

```
import torch
import pickle
import numpy as np

class Attention_Extracter:
    def __init__(self, graph_data_dict_path, encoder_model, gpu=False):
        self.torch_device = 'cuda' if gpu else 'cpu'

        self.graph_data_dict = torch.load(graph_data_dict_path)
        self.encoder_model = encoder_model
        self.encoder_model.to(self.torch_device)
        self.encoder_model.eval()
        self.latent_feat_dict, self.attention_scores1 = self.extract_latent_attention_features()

    def extract_latent_attention_features(self):
        latent_features = {}
        attention_scores1 = {}
```

```
    with torch.no_grad():
        for graph_id, data in self.graph_data_dict.items():
            data = data.to(self.torch_device)
            z, attention_weights = self.encoder_model(data.x, data.edge_index, data.edge_attr)
            latent_features[graph_id] = z.cpu()

            # Handling the case where attention_weights is a tuple or other data structure
            if isinstance(attention_weights, (list, tuple)):
                attention_scores1[graph_id] = [aw for aw in attention_weights]
            else:
                attention_scores1[graph_id] = attention_weights.cpu()

    return latent_features, attention_scores1

def load_edge_indices(self, glist_path, edge_matrix_path):
    with open(glist_path, 'rb') as f:
        glist = pickle.load(f)

    edge_matrix = np.load(edge_matrix_path)
    edge_matrix = torch.tensor(edge_matrix, dtype=torch.float)
    edge_index = torch.nonzero(edge_matrix, as_tuple=False).t().contiguous()
    edge_indices_dict = {}

    for i in range(edge_index.shape[1]):
        index1, index2 = edge_index[0, i].item(), edge_index[1, i].item()
        gene1, gene2 = glist[index1], glist[index2]
        edge_indices_dict[(index1, index2)] = (gene1, gene2)

    return edge_indices_dict
```

Graph Attention Autoencoder for MultiOmics Integration,

Risk Stratification and Biomarker Identification in Cancer

File: GATv2EncoderModel.py

First 75 lines:

```
from transformers import PreTrainedModel
from OmicsConfig import OmicsConfig
from transformers import PretrainedConfig, PreTrainedModel
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GATv2Conv
from torch_geometric.data import Batch
from torch.utils.data import DataLoader
from torch.optim import AdamW
from torch_geometric.utils import negative_sampling
from torch.nn.functional import cosine_similarity
from torch.optim.lr_scheduler import StepLR


class GATv2EncoderModel(PreTrainedModel):
    config_class = OmicsConfig
    base_model_prefix = "gatv2_encoder"

    def __init__(self, config):
        super().__init__(config)
        self.layers = nn.ModuleList([
                GATv2Conv(config.in_channels if i == 0 else config.out_channels, config.out_channels, heads=1,
concat=True, edge_dim=config.edge_attr_channels, add_self_loops=False)
            for i in range(config.num_layers)
        ])
```

```
    def forward(self, x, edge_index, edge_attr):
        attention_weights = []
        for layer in self.layers:
            x, attn_weights = layer(x, edge_index, edge_attr, return_attention_weights=True)
            attention_weights.append(attn_weights)
        return x, attention_weights
```

Last 75 lines:

```
from transformers import PreTrainedModel
from OmicsConfig import OmicsConfig
from transformers import PretrainedConfig, PreTrainedModel
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GATv2Conv
from torch_geometric.data import Batch
from torch.utils.data import DataLoader
from torch.optim import AdamW
from torch_geometric.utils import negative_sampling
from torch.nn.functional import cosine_similarity
from torch.optim.lr_scheduler import StepLR



class GATv2EncoderModel(PreTrainedModel):
    config_class = OmicsConfig
    base_model_prefix = "gatv2_encoder"

    def __init__(self, config):
        super().__init__(config)
        self.layers = nn.ModuleList([
```

```
            GATv2Conv(config.in_channels if i == 0 else config.out_channels, config.out_channels, heads=1, concat=True, edge_dim=config.edge_attr_channels, add_self_loops=False)
            for i in range(config.num_layers)
        ])

    def forward(self, x, edge_index, edge_attr):
        attention_weights = []
        for layer in self.layers:
            x, attn_weights = layer(x, edge_index, edge_attr, return_attention_weights=True)
            attention_weights.append(attn_weights)
        return x, attention_weights
```

File: GATv2DecoderModel.py

First 75 lines:

```
from transformers import PreTrainedModel
from OmicsConfig import OmicsConfig
from transformers import PretrainedConfig, PreTrainedModel
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GATv2Conv
from torch_geometric.data import Batch
from torch.utils.data import DataLoader
from torch.optim import AdamW
from torch_geometric.utils import negative_sampling
from torch.nn.functional import cosine_similarity
from torch.optim.lr_scheduler import StepLR

from EdgeWeightPredictorModel import EdgeWeightPredictorModel
```

```
class GATv2DecoderModel(PreTrainedModel):
    config_class = OmicsConfig
    base_model_prefix = "gatv2_decoder"

    def __init__(self, config):
        super().__init__(config)
        self.layers = nn.ModuleList([
            nn.Linear(config.out_channels if i == 0 else config.out_channels, config.out_channels)
            for i in range(config.num_layers)
        ])
        self.fc = nn.Linear(config.out_channels, config.original_feature_size)
        self.edge_weight_predictor = EdgeWeightPredictorModel(config)

    def forward(self, z):
        for layer in self.layers:
            z = layer(z)
            z = F.relu(z)
        x_reconstructed = self.fc(z)
        return x_reconstructed

    def predict_edge_weights(self, z, edge_index):
        return self.edge_weight_predictor(z, edge_index)
```

Last 75 lines:

```
from transformers import PreTrainedModel
from OmicsConfig import OmicsConfig
from transformers import PretrainedConfig, PreTrainedModel
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from torch_geometric.nn import GATv2Conv
from torch_geometric.data import Batch
from torch.utils.data import DataLoader
from torch.optim import AdamW
from torch_geometric.utils import negative_sampling
from torch.nn.functional import cosine_similarity
from torch.optim.lr_scheduler import StepLR

from EdgeWeightPredictorModel import EdgeWeightPredictorModel

class GATv2DecoderModel(PreTrainedModel):
    config_class = OmicsConfig
    base_model_prefix = "gatv2_decoder"

    def __init__(self, config):
        super().__init__(config)
        self.layers = nn.ModuleList([
            nn.Linear(config.out_channels if i == 0 else config.out_channels, config.out_channels)
            for i in range(config.num_layers)
        ])
        self.fc = nn.Linear(config.out_channels, config.original_feature_size)
        self.edge_weight_predictor = EdgeWeightPredictorModel(config)

    def forward(self, z):
        for layer in self.layers:
            z = layer(z)
            z = F.relu(z)
        x_reconstructed = self.fc(z)
        return x_reconstructed
```

```
def predict_edge_weights(self, z, edge_index):
    return self.edge_weight_predictor(z, edge_index)
```