Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection

```
import numpy as np
import pandas as pd
import torch
from data_loader import *
from main import *
from tqdm import tqdm
```

KDD Cup 1999 Data (10% subset)

This is the data set used for The Third International Know ledge Discovery and Data Mining Tools Competition, w hich w as held in conjunction w ith KDD-99 The Fifth International Conference on Know ledge Discovery and Data Mining. The competition task w as to build a netw ork intrusion detector, a predictive model capable of distinguishing betw een "bad" connections, called intrusions or attacks, and "good" normal connections. This database contains a standard set of data to be audited, w hich includes a w ide variety of intrusions simulated in a military netw ork environment.

```
data = pd.read_csv("kddcup.data_10_percent_corrected", header=None,names=['duration', 'protocol_type', 'service', 'flag',
```

Pre-processing

Follow ing the paper, since the "normal" only comprises of approximately 20% of the entries, the "normal" data w ere considered as anomalies instead.

```
data.loc[data["type"] != "normal.", 'type'] = 0
data.loc[data["type"] == "normal.", 'type'] = 1
```

Next, the categorical variables are converted to a one hot encoding representation. My implementation is a bit different from the original paper in this aspect. Since I am only using the 10% subset to generate the columns, I get 118 features instead of 120 as reported in the paper.

```
one_hot_protocol = pd.get_dummies(data["protocol_type"])
one_hot_service = pd.get_dummies(data["service"])
one_hot_flag = pd.get_dummies(data["flag"])

data = data.drop("protocol_type",axis=1)
data = data.drop("service",axis=1)
data = data.drop("flag",axis=1)

data = pd.concat([one_hot_protocol, one_hot_service, one_hot_flag, data],axis=1)
data.head()
```

	icmp	tcp	udp	IRC	X11	Z39_50	auth	bgp	courier	csnet_ns	...
0	0	1	0	0	0	0	0	0	0	0	...
1	0	1	0	0	0	0	0	0	0	0	...

2	0	1	0	0	0	0	0	0	0	0	...
3	0	1	0	0	0	0	0	0	0	0	...
4	0	1	0	0	0	0	0	0	0	0	...

5 rows × 119 columns

```
proportions = data["type"].value_counts()
print(proportions)
print("Anomaly Percentage",proportions[1] / proportions.sum())
```

```
0    396743
1     97278
Name: type, dtype: int64
Anomaly Percentage 0.19691065764410826
```

```
#proportions_alfa = data["type"].value_counts(normalize=True)
#print(proportions_alfa)
```

```
0    0.803089
1    0.196911
Name: type, dtype: float64
```

Normalize all the numeric variables.

```
cols_to_norm = ["duration", "src_bytes", "dst_bytes", "wrong_fragment", "urgent",
            "hot", "num_failed_logins", "num_compromised", "num_root",
            "num_file_creations", "num_shells", "num_access_files", "count", "srv_count",
            "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate", "same_srv_rate",
            "diff_srv_rate", "srv_diff_host_rate", "dst_host_count", "dst_host_srv_count", "dst_host_same_srv_rate",
            "dst_host_diff_srv_rate", "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
            "dst_host_serror_rate", "dst_host_srv_serror_rate", "dst_host_rerror_rate", "dst_host_srv_rerror_rate" ]

#data.loc[:, cols_to_norm] = (data[cols_to_norm] - data[cols_to_norm].mean()) / data[cols_to_norm].std()
min_cols = data.loc[data["type"]==0 , cols_to_norm].min()
max_cols = data.loc[data["type"]==0 , cols_to_norm].max()

data.loc[:, cols_to_norm] = (data[cols_to_norm] - min_cols) / (max_cols - min_cols)
```

I saved the preprocessed data into a numpy file format and load it using the pytorch data loader.

```
np.savez_compressed("kdd_cup",kdd=data.values())
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-10-400cfb624d16> in <module>
----> 1 np.savez_compressed("kdd_cup",kdd=data.values())
```

```
TypeError: 'numpy.ndarray' object is not callable
```

I initially implemented this to be ran in the command line and use argparse to get the hyperparameters. To make it runnable in a jupyter notebook, I had to create a dummy class for the hyperparameters.

```
class hyperparams():
    def __init__(self, config):
        self.__dict__.update(**config)
defaults = {
    'lr' : 1e-4,
    'num_epochs' : 200,
    'batch_size' : 1024,
    'gmm_k' : 4,
    'lambda_energy' : 0.1,
    'lambda_cov_diag' : 0.005,
    'pretrained_model' : None,
    'mode' : 'train',
    'use_tensorboard' : False,
    'data_path' : 'kdd_cup.npz',

    'log_path' : './dagmm/logs',
    'model_save_path' : './dagmm/models',
    'sample_path' : './dagmm/samples',
    'test_sample_path' : './dagmm/test_samples',
    'result_path' : './dagmm/results',

    'log_step' : 194//4,
    'sample_step' : 194,
    'model_save_step' : 194,
}
```

```
solver = main(hyperparams(defaults))
accuracy, precision, recall, f_score = solver.test()
```

I copy pasted the testing code here in the notebook so we could play around the results.

Incrementally compute for the GMM parameters across all training data for a better estimate

```
solver.data_loader.dataset.mode="train"
solver.dagmm.eval()
N = 0
mu_sum = 0
cov_sum = 0
gamma_sum = 0

for it, (input_data, labels) in enumerate(solver.data_loader):
    input_data = solver.to_var(input_data)
    enc, dec, z, gamma = solver.dagmm(input_data)
    phi, mu, cov = solver.dagmm.compute_gmm_params(z, gamma)

    batch_gamma_sum = torch.sum(gamma, dim=0)

    gamma_sum += batch_gamma_sum
    mu_sum += mu * batch_gamma_sum.unsqueeze(-1) # keep sums of the numerator only
    cov_sum += cov * batch_gamma_sum.unsqueeze(-1).unsqueeze(-1) # keep sums of the numerator only

    N += input_data.size(0)

train_phi = gamma_sum / N
```

```
train_mu = mu_sum / gamma_sum.unsqueeze(-1)
train_cov = cov_sum / gamma_sum.unsqueeze(-1).unsqueeze(-1)

print("N:",N)
print("phi :\n",train_phi)
print("mu :\n",train_mu)
print("cov :\n",train_cov)
```

```
train_energy = []
train_labels = []
train_z = []
for it, (input_data, labels) in enumerate(solver.data_loader):
    input_data = solver.to_var(input_data)
    enc, dec, z, gamma = solver.dagmm(input_data)
    sample_energy, cov_diag = solver.dagmm.compute_energy(z, phi=train_phi, mu=train_mu, cov=train_cov, size_average=False

    train_energy.append(sample_energy.data.cpu().numpy())
    train_z.append(z.data.cpu().numpy())
    train_labels.append(labels.numpy())


train_energy = np.concatenate(train_energy,axis=0)
train_z = np.concatenate(train_z,axis=0)
train_labels = np.concatenate(train_labels,axis=0)
```

Compute the energy of every sample in the test data

```
solver.data_loader.dataset.mode="test"
test_energy = []
test_labels = []
test_z = []
for it, (input_data, labels) in enumerate(solver.data_loader):
    input_data = solver.to_var(input_data)
    enc, dec, z, gamma = solver.dagmm(input_data)
    sample_energy, cov_diag = solver.dagmm.compute_energy(z, size_average=False)
    test_energy.append(sample_energy.data.cpu().numpy())
    test_z.append(z.data.cpu().numpy())
    test_labels.append(labels.numpy())


test_energy = np.concatenate(test_energy,axis=0)
test_z = np.concatenate(test_z,axis=0)
test_labels = np.concatenate(test_labels,axis=0)
```

```
combined_energy = np.concatenate([train_energy, test_energy], axis=0)
combined_z = np.concatenate([train_z, test_z], axis=0)
combined_labels = np.concatenate([train_labels, test_labels], axis=0)
```

Compute for the threshold energy. Following the paper I just get the highest 20% and treat it as an anomaly. That corresponds to setting the threshold at the 80th percentile.

```
thresh = np.percentile(combined_energy, 100 - 20)
print("Threshold :", thresh)
```

```
pred = (test_energy>thresh).astype(int)
gt = test_labels.astype(int)
```

```
from sklearn.metrics import precision_recall_fscore_support as prf, accuracy_score
```

```
accuracy = accuracy_score(gt,pred)
precision, recall, f_score, support = prf(gt, pred, average='binary')
```

```
print("Accuracy : {:0.4f}, Precision : {:0.4f}, Recall : {:0.4f}, F-score : {:0.4f}".format(accuracy,precision, recall, f_
```

Visualizing the z space

It's a little different from the paper's figure but I assume that's because of the small changes in my implementation.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
%matplotlib notebook
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(test_z[:,1],test_z[:,0], test_z[:,2], c=test_labels.astype(int))
ax.set_xlabel('Encoded')
ax.set_ylabel('Euclidean')
ax.set_zlabel('Cosine')
plt.show()
```