Paper Review

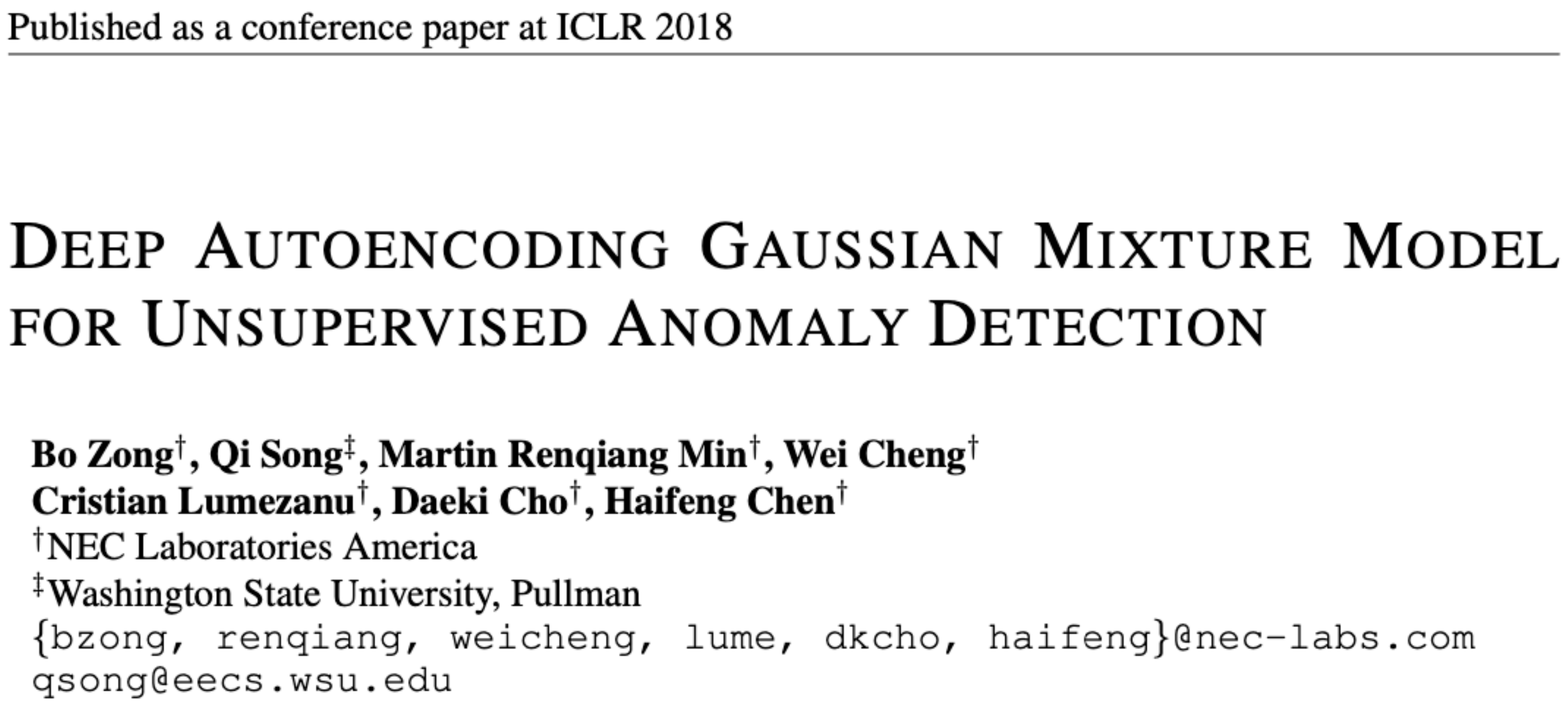
Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection

Boostcamp AI Tech 2021

Seung Woo Hong

Contents

Original Paper


Published as a conference paper at ICLR 2018

DEEP AUTOENCODING GAUSSIAN MIXTURE MODEL FOR UNSUPERVISED ANOMALY DETECTION

Bo Zong[†], **Qi Song**[‡], **Martin Renqiang Min**[†], **Wei Cheng**[†]
Cristian Lumezanu[†], **Daeki Cho**[†], **Haifeng Chen**[†]
[†]NEC Laboratories America
[‡]Washington State University, Pullman
{bzong, renqiang, weicheng, lume, dkcho, haifeng}@nec-labs.com
qsong@eecs.wsu.edu

Anomaly Detection

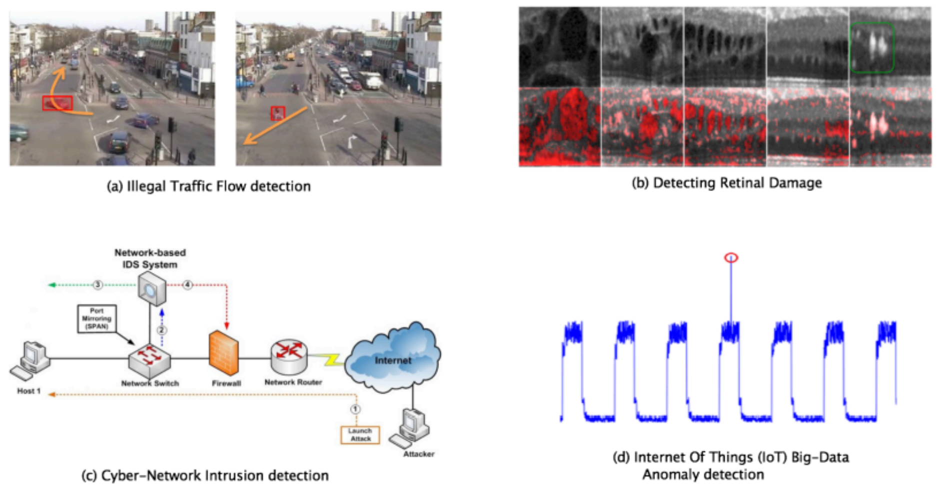

(a) Illegal Traffic Flow detection
(b) Detecting Retinal Damage
Network-based IDS System
Port Mirroring (SPAN)
Host 1
Network Switch
Firewall
Network Router
Internet
Launch Attack
Attacker
(c) Cyber–Network Intrusion detection
(d) Internet Of Things (IoT) Big–Data Anomaly detection

Figure 2: Applications Deep learning-based anomaly detection algorithms.
(a) Video Surveillance, Image Analysis: Illegal Traffic detection Xie et al. [2017], (b) Health-care: Detecting Retinal Damage Schlegl et al. [2017]
(c) Networks: Cyber-intrusion detection Javaid et al. [2016] (d) Sensor Networks: Internet of Things (IoT) big-data anomaly detection Mohammadi et al. [2017]

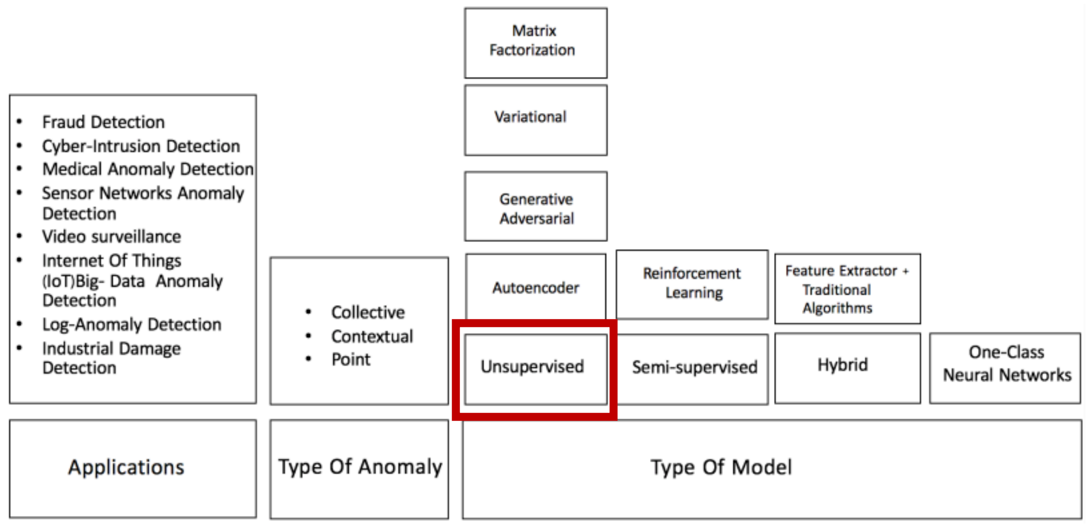


Figure 5: Key components associated with deep learning-based anomaly detection technique.

The core of <u>unsupervised</u> anomaly detection - density estimation: given a lot of input samples, anomalies are those ones residing in low probability density areas.

Problem Statement

- Higher input dimensionality = more difficult density estimation
- <u>Curse of dimensionality!</u>

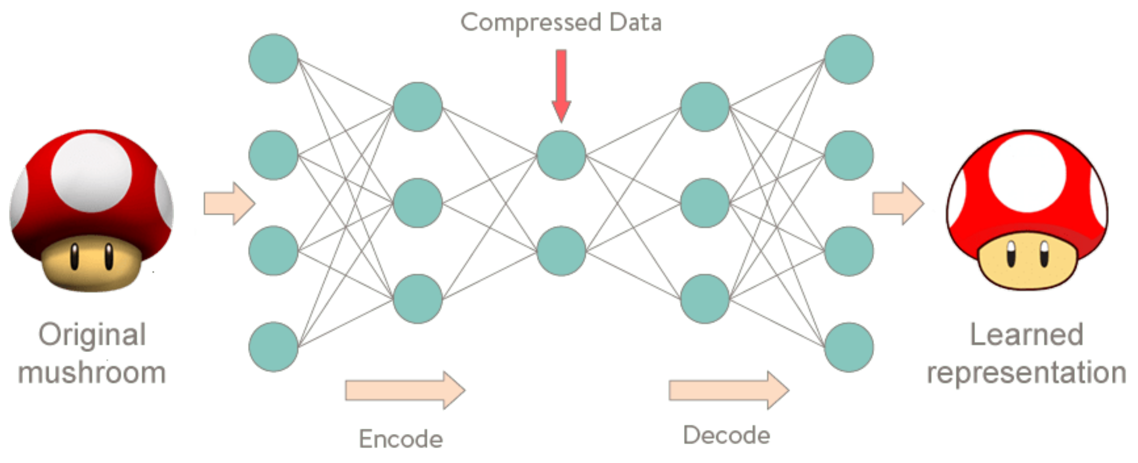
- So… dimensionality reduction <u>before</u> density estimation?
- The key information for anomaly detection could be removed, *when the two steps are separately learned*
- <u>Bad performance</u>

Dimensionality reduction + Density estimation

Existing Methods (1)

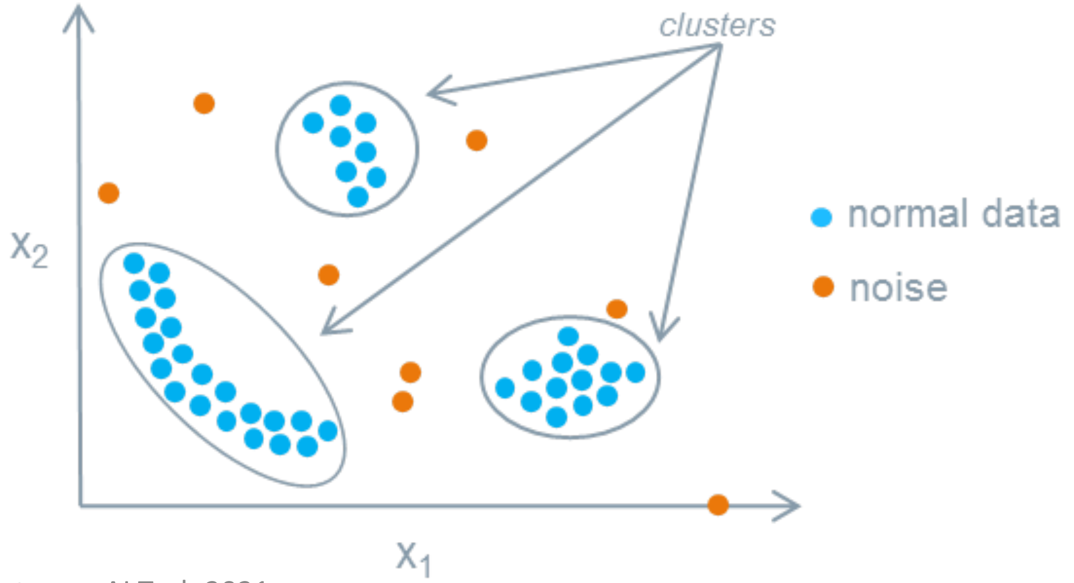
- **Reconstruction based methods**
- Assume that anomalies are incompressible and thus cannot be effectively reconstructed from low-dimensional projections

- Limitations:
  - Only conduct from a single aspect, <u>reconstruction error</u>
  - A significant amount of anomalous samples could also lurk with a normal level of error, due to high model complexity or noisy samples



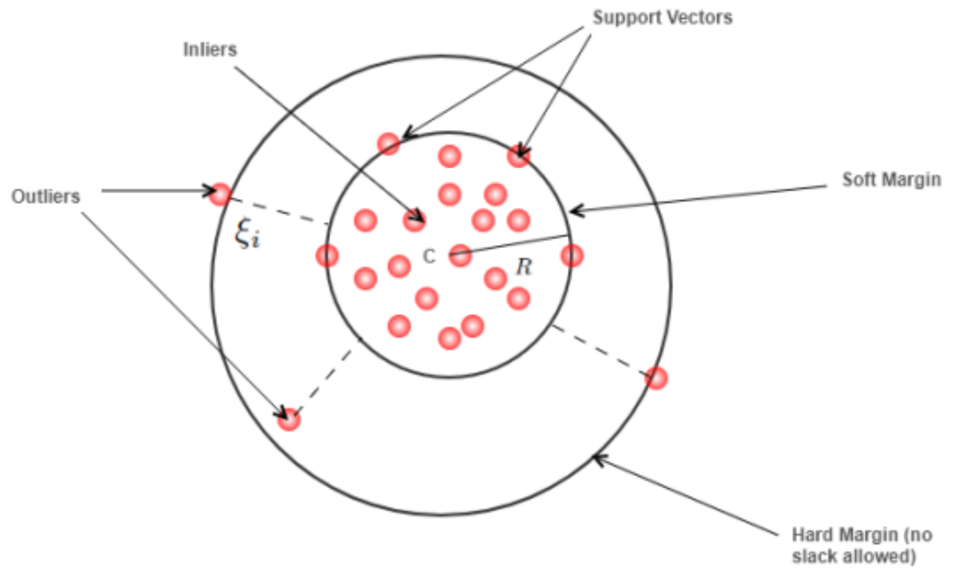
Existing Methods (2)

- **Clustering analysis**
- Gaussian Models, Gaussian Mixture Models, K-means …
- Difficult to directly apply such methods due to the curse of dimensionality



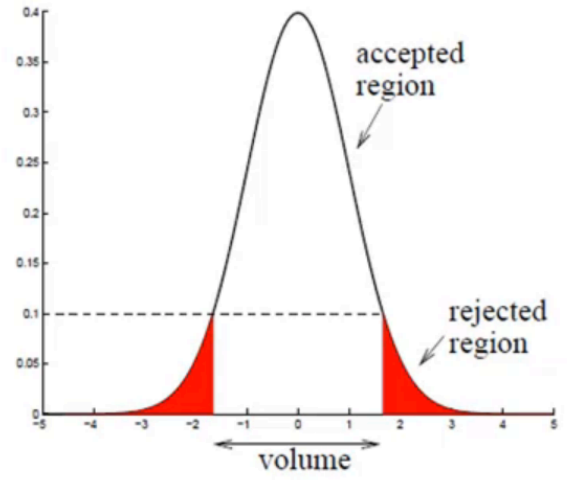
Existing Methods (3)

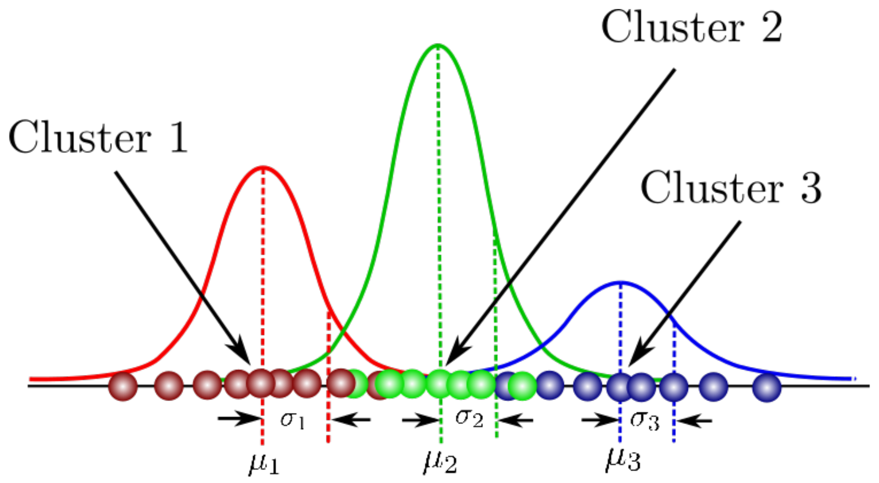
- **One-class classification approaches**
- A discriminative boundary surrounding the normal instances is learned by algorithms
- One-class SVM, Deep SVM, …
- Again, curse of dimensionality



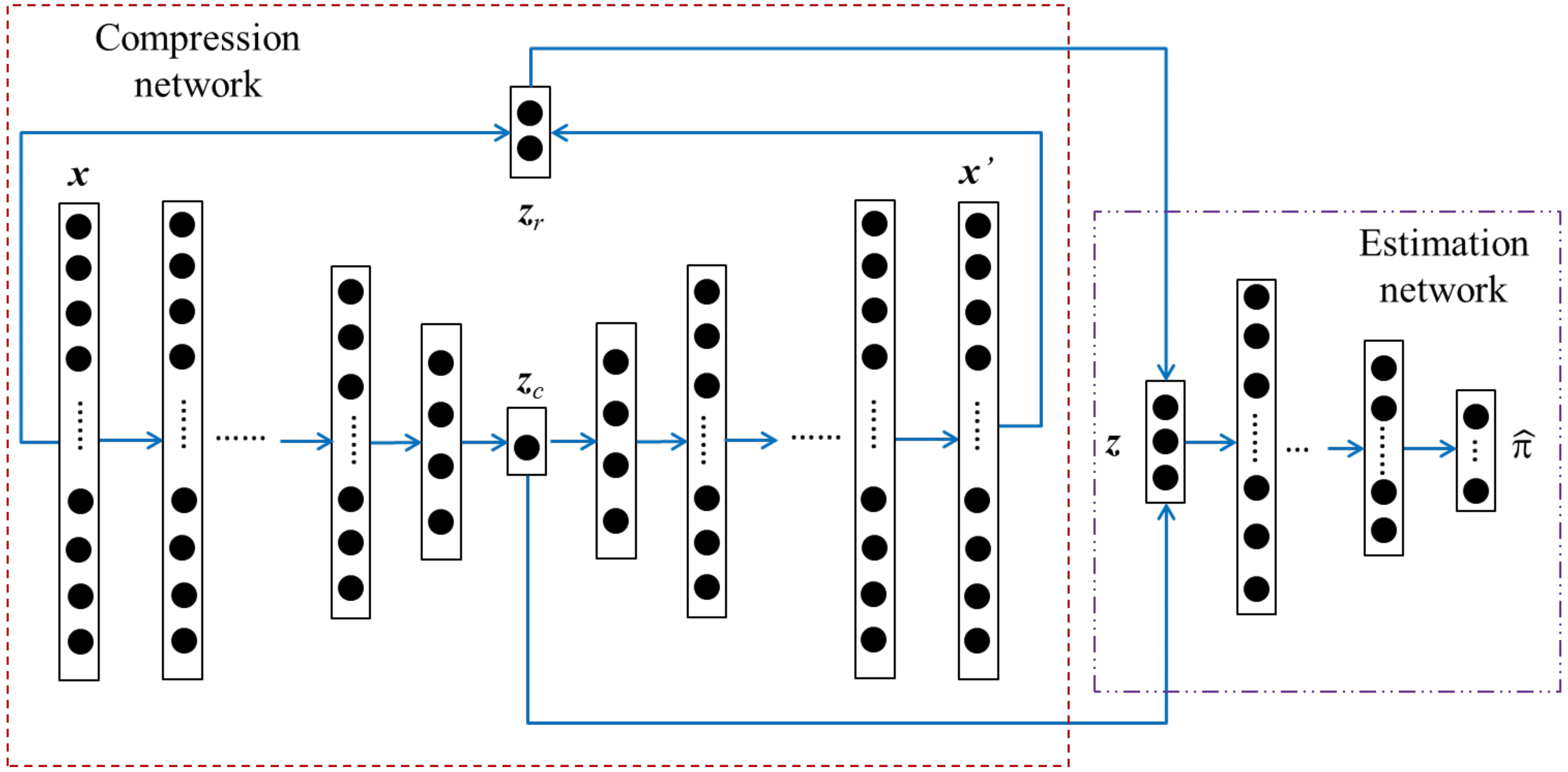
Gaussian Mixture Model

- Assumption: The observed data are drawn from a mixture of Gaussian distribution
- A Gaussian Mixture is a function that is comprised of several (K) Gaussians
- Parameters
  - Mean (μ) - defines its center
  - Covariance (Σ) – defines its width
  - Mixing probability (π) – defines how big or small the Gaussian function will be
- EM algorithm
- Only from normal data



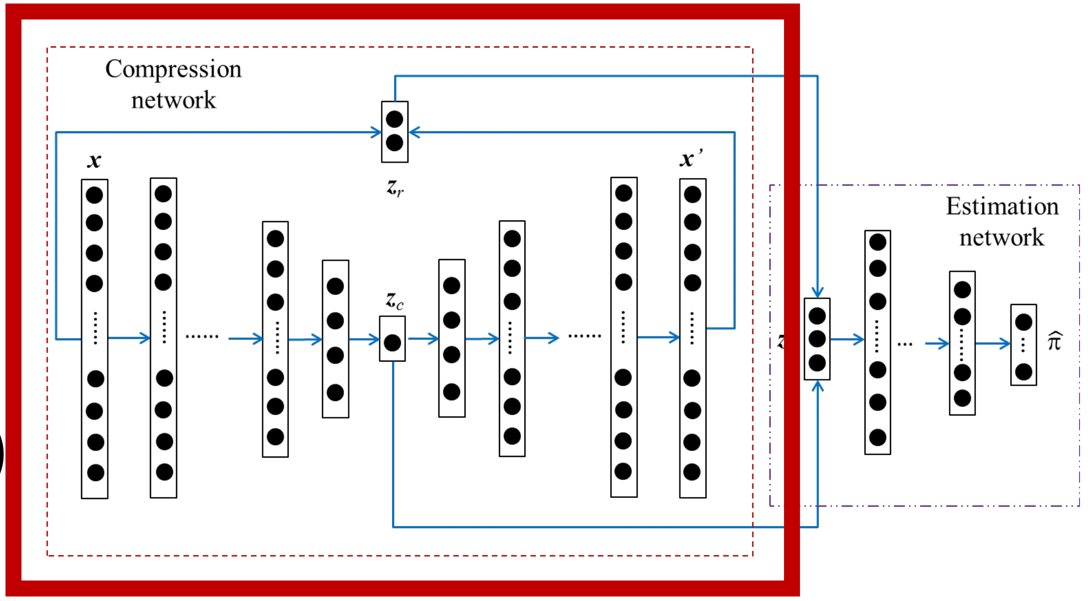


DAGMM



DAGMM – Compression Network

- Encoded to the reduced low-dimensional representation
  - zc = h(x; θe)
- Reconstructed counterpart of x (decoding)
  - x' = g(zc; θd)
- Reconstruction error
  - zr =f(x,x')
  - Considers multiple distance metrics
- Z to feed the estimation network
  - z = [zc,zr]
    (zc, zr, cosine similarity concatenated)



DAGMM – Estimation Network

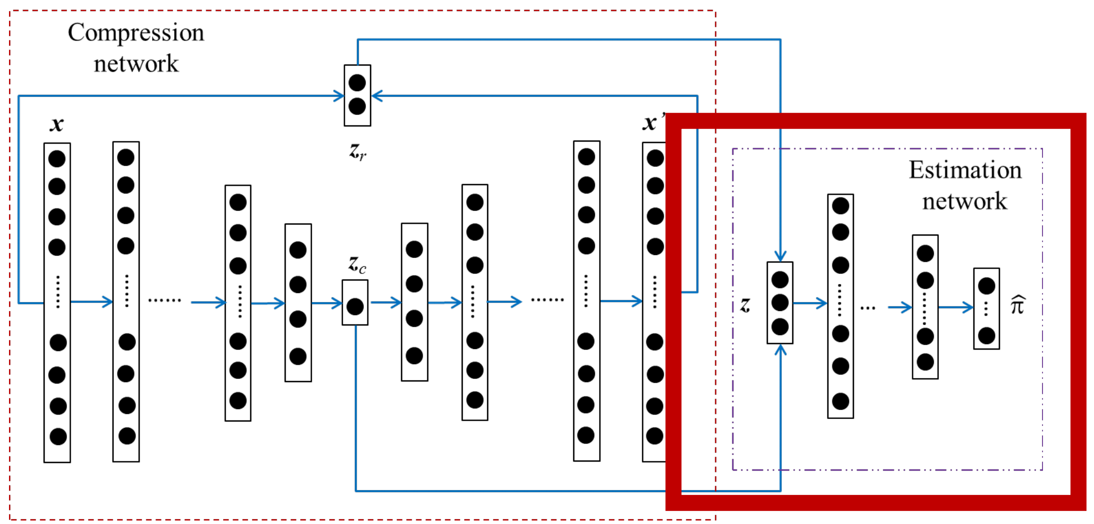
- Performs density estimation under the framework of GMM
- Estimates the parameters of GMM
- MLN predicts the mixture membership for each sample
- (Testing) Straightforward estimation of sample energy with learned GMM parameters

$$\mathbf{p} = MLN(\mathbf{z}; \theta_m) \qquad \hat{\gamma} = \text{softmax}(\mathbf{p})$$

$$\hat{\phi}_k = \sum_{i=1}^{N} \frac{\hat{\gamma}_{ik}}{N}, \qquad \hat{\mu}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik}\mathbf{z}_i}{\sum_{i=1}^{N} \hat{\gamma}_{ik}},$$

$$\hat{\mathbf{\Sigma}}_k = \frac{\sum_{i=1}^{N} \hat{\gamma}_{ik}(\mathbf{z}_i - \hat{\mu}_k)(\mathbf{z}_i - \hat{\mu}_k)^T}{\sum_{i=1}^{N} \hat{\gamma}_{ik}}.$$

$$E(\mathbf{z}) = -\log\left(\sum_{k=1}^{K} \hat{\phi}_k \frac{\exp\left(-\frac{1}{2}(\mathbf{z}-\hat{\mu}_k)^T \hat{\mathbf{\Sigma}}_k^{-1}(\mathbf{z}-\hat{\mu}_k)\right)}{\sqrt{|2\pi\hat{\mathbf{\Sigma}}_k|}}\right)$$



Energy Function

- Maps each point of an input space to a single scalar (= energy)
- When modeling X alone within an unsupervised learning setting, lower energy is attributed to the data manifold
- Known data -> lower energy
- Unknown data - > higher energy

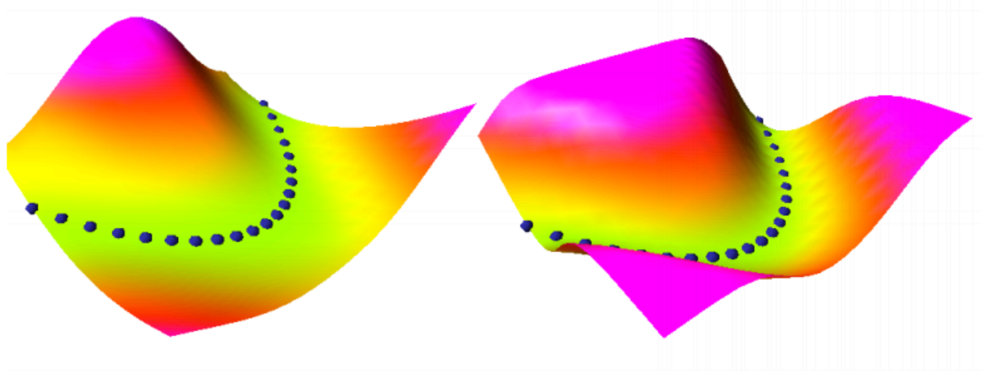


Figure 1: Two energy surfaces in X, Y space obtained by training two neural nets to compute the function $Y = X^2 - 1/2$. The blue dots represent a subset of the training samples. In the left diagram, the energy is quadratic in Y, therefore its exponential is integrable over Y. This model is equivalent to a probabilistic Gaussian model of $P(Y|X)$. The right diagram uses a non-quadratic saturated energy whose exponential is not integrable over Y. This model is not normalizable, and therefore has no probabilistic counterpart.

DAGMM – Objective Function

$$J(\theta_e, \theta_d, \theta_m) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}_i, \mathbf{x}'_i) + \frac{\lambda_1}{N}\sum_{i=1}^{N} E(\mathbf{z}_i) + \lambda_2 P(\hat{\mathbf{\Sigma}}).$$

- Reconstruction error - L(xi,x'i)
- Sample energy - E(zi)
  - By <u>minimizing</u> the sample energy, we look for the best combination of compression and estimation networks that <u>maximize the likelihood to observe input samples.</u>
- Solving singularity problem - P (Σ)
  - Penalize small values on the diagonal entries
  - Cholesky decomposition

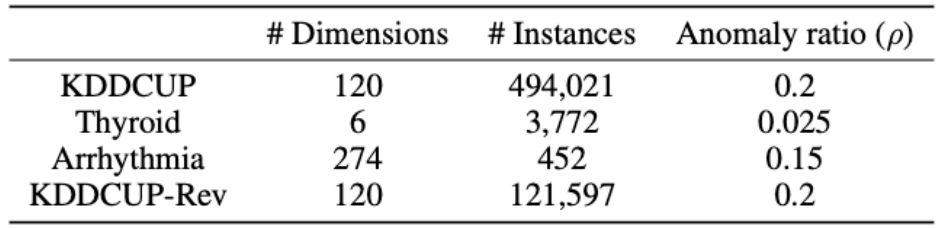
$$P(\hat{\mathbf{\Sigma}}) = \sum_{k=1}^{K} \sum_{j=1}^{d} \frac{1}{\hat{\mathbf{\Sigma}}_{kjj}},$$

DAGMM – Key Points

- Preserves the key information of an input sample in a low-dimensional space (z = [zc, zr])
- GMM – learned by alternating EM algorithms, where it is hard to perform joint optimization of dimensionality reduction & density estimation
- DAGMM – simultaneously minimizes reconstruction error & sample energy (joint train)
- End-to-end training – pre-training limits the potential to adjust the dimensionality reduction behavior

PyTorch Implementation – Data & Setting

- Thyroid Dataset
- Reconstruction error = $\frac{\|X-\acute{X}\|_2}{\|X\|_2}$
- N mixture components = 2
- Compression network
  - FC(6, 12, tanh) – FC(12, 4, tanh) – FC(4, 1, none) – FC(1, 4, tanh)– FC(4, 12, tanh) – FC(12, 6, none)
- Estimation network
  - FC(3, 10, tanh) – Dropout(0.5) – FC(10, 2, softmax)
- Adam optimizer, learning rate = 0.0001
- Training epochs = 20000, Batch size = 1024
- Lambda 1 = 0.1 , Lambda 2 = 0.005
- Metric – avg. precision, recall and F1
- Threshold = 0.025 (=anomaly ratio)
- Test size = 50%



	# Dimensions	# Instances	Anomaly ratio (ρ)
KDDCUP	120	494,021	0.2
Thyroid	6	3,772	0.025
Arrhythmia	274	452	0.15
KDDCUP-Rev	120	121,597	0.2

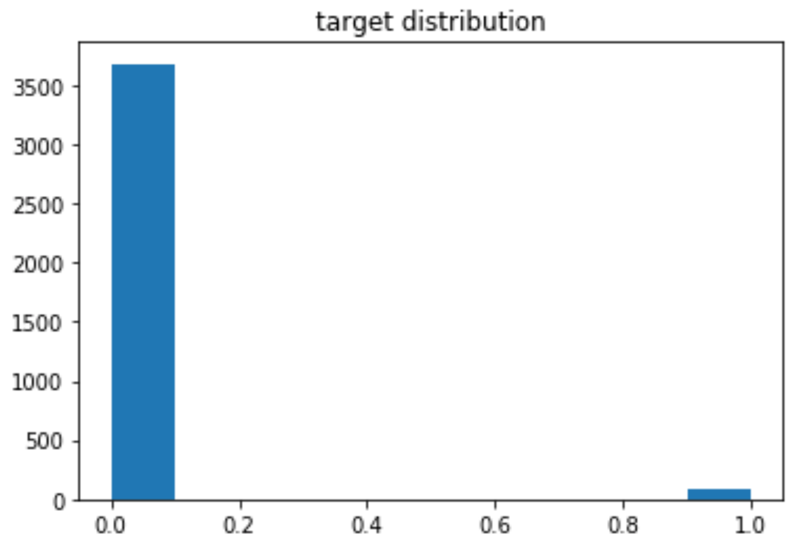
Table 1: Statistics of the public benchmark datasets

PyTorch Implementation – Thyroid Data

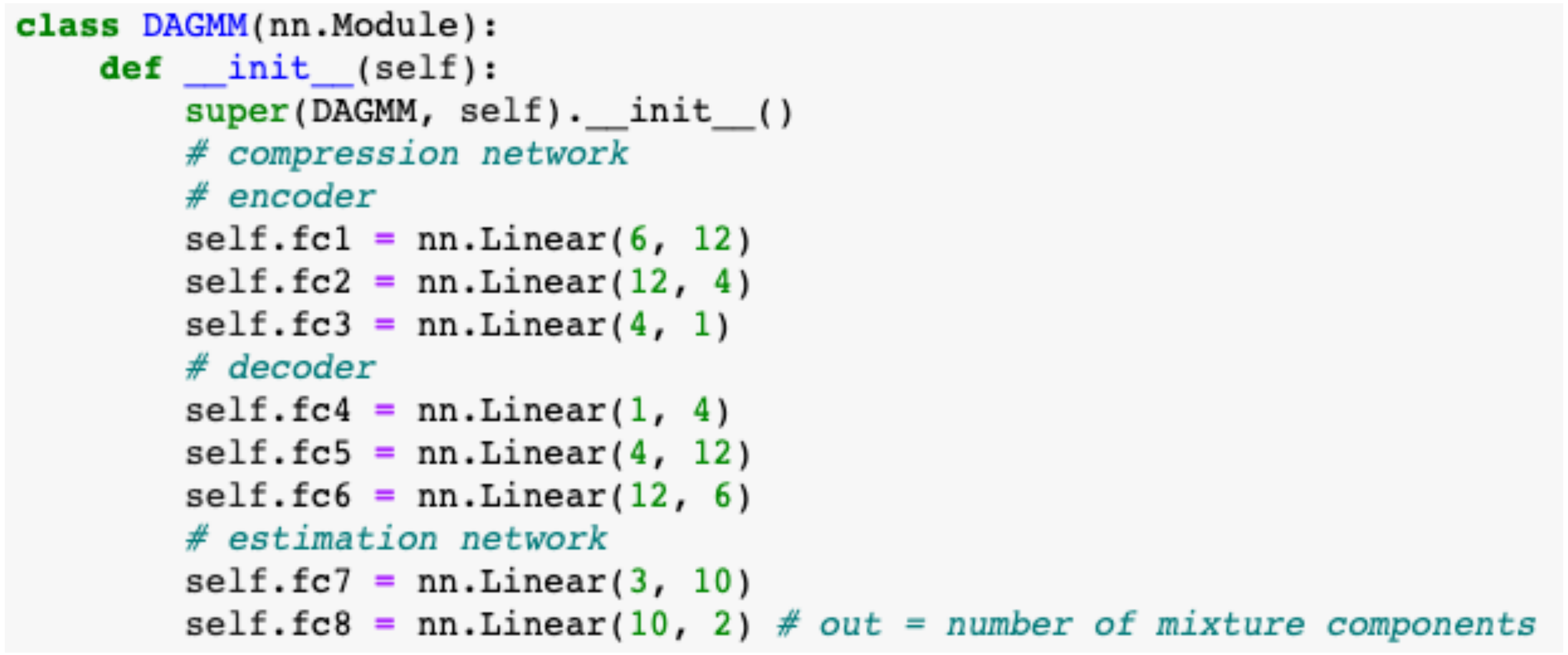
```python
# load 'Thyroid' dataset

from tqdm import tqdm_notebook
from scipy import io
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

mat_file = io.loadmat('thyroid.mat')

X = mat_file['X']
y = mat_file['y']

print('X Shape: ', X.shape)

display(pd.DataFrame(X).head(3))

plt.hist(y)
plt.title('target distribution')
plt.show()
```

```
X Shape:  (3772, 6)
```

	0	1	2	3	4	5
0	0.774194	0.001132	0.137571	0.275701	0.295775	0.236066
1	0.247312	0.000472	0.279886	0.329439	0.535211	0.173770
2	0.494624	0.003585	0.222960	0.233645	0.525822	0.124590

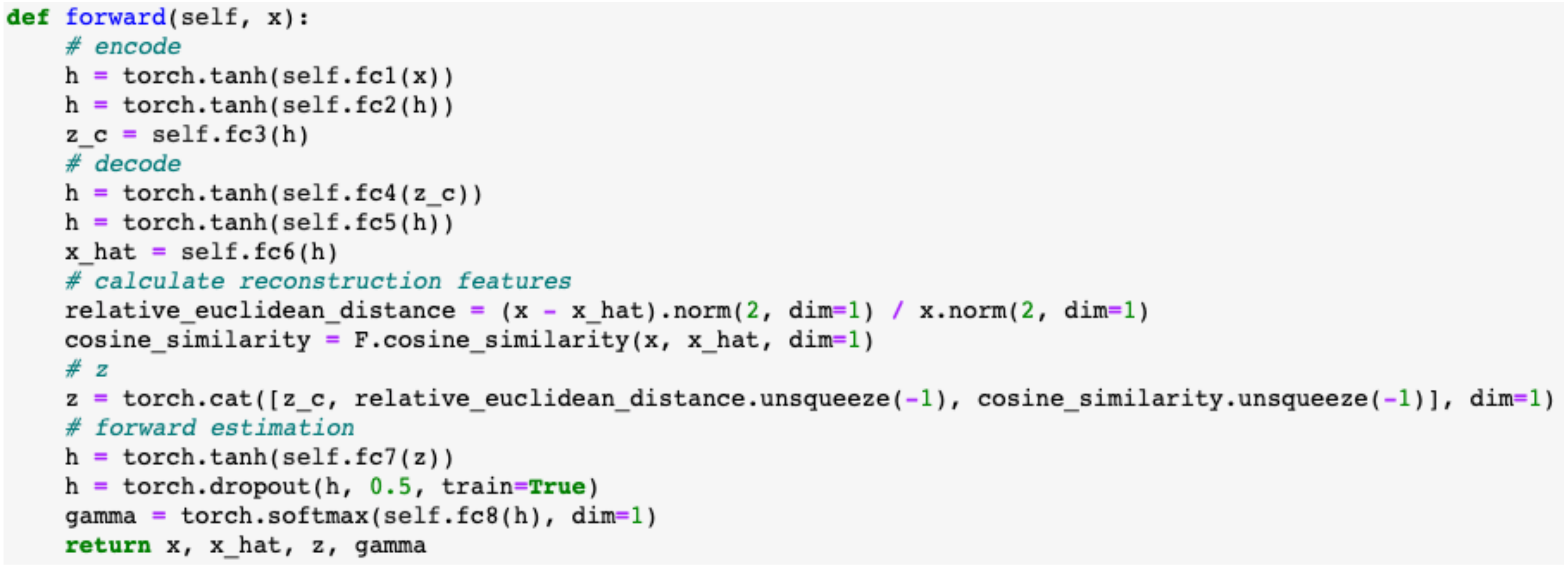


PyTorch Implementation - Network



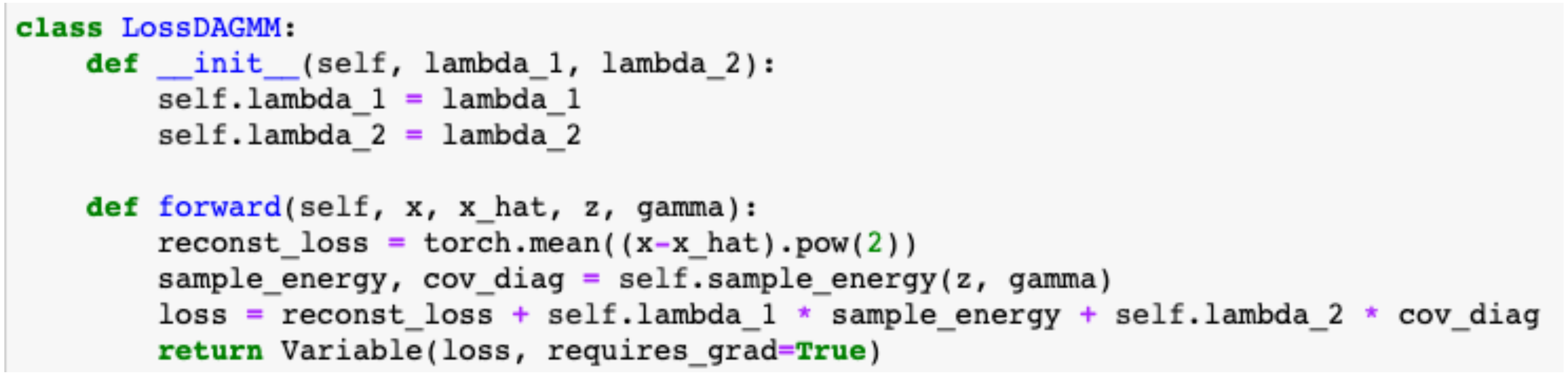
```
class DAGMM(nn.Module):
    def __init__(self):
        super(DAGMM, self).__init__()
        # compression network
        # encoder
        self.fc1 = nn.Linear(6, 12)
        self.fc2 = nn.Linear(12, 4)
        self.fc3 = nn.Linear(4, 1)
        # decoder
        self.fc4 = nn.Linear(1, 4)
        self.fc5 = nn.Linear(4, 12)
        self.fc6 = nn.Linear(12, 6)
        # estimation network
        self.fc7 = nn.Linear(3, 10)
        self.fc8 = nn.Linear(10, 2) # out = number of mixture components
```

PyTorch Implementation - Network



```python
def forward(self, x):
    # encode
    h = torch.tanh(self.fc1(x))
    h = torch.tanh(self.fc2(h))
    z_c = self.fc3(h)
    # decode
    h = torch.tanh(self.fc4(z_c))
    h = torch.tanh(self.fc5(h))
    x_hat = self.fc6(h)
    # calculate reconstruction features
    relative_euclidean_distance = (x - x_hat).norm(2, dim=1) / x.norm(2, dim=1)
    cosine_similarity = F.cosine_similarity(x, x_hat, dim=1)
    # z
    z = torch.cat([z_c, relative_euclidean_distance.unsqueeze(-1), cosine_similarity.unsqueeze(-1)], dim=1)
    # forward estimation
    h = torch.tanh(self.fc7(z))
    h = torch.dropout(h, 0.5, train=True)
    gamma = torch.softmax(self.fc8(h), dim=1)
    return x, x_hat, z, gamma
```

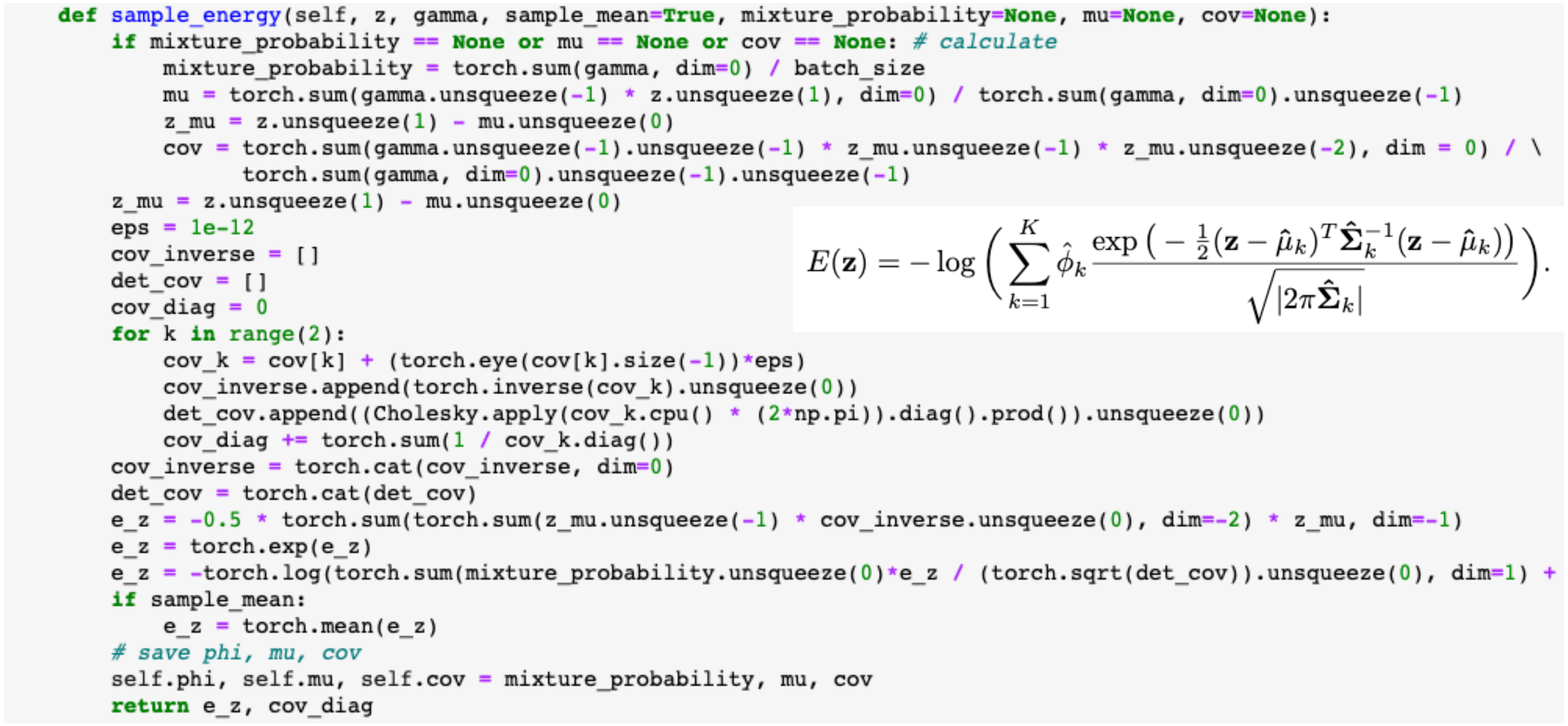
PyTorch Implementation – Loss Function



```python
class LossDAGMM:
    def __init__(self, lambda_1, lambda_2):
        self.lambda_1 = lambda_1
        self.lambda_2 = lambda_2

    def forward(self, x, x_hat, z, gamma):
        reconst_loss = torch.mean((x-x_hat).pow(2))
        sample_energy, cov_diag = self.sample_energy(z, gamma)
        loss = reconst_loss + self.lambda_1 * sample_energy + self.lambda_2 * cov_diag
        return Variable(loss, requires_grad=True)
```

$$J(\theta_e, \theta_d, \theta_m) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}_i, \mathbf{x}_i') + \frac{\lambda_1}{N}\sum_{i=1}^{N} E(\mathbf{z}_i) + \lambda_2 P(\hat{\mathbf{\Sigma}}).$$

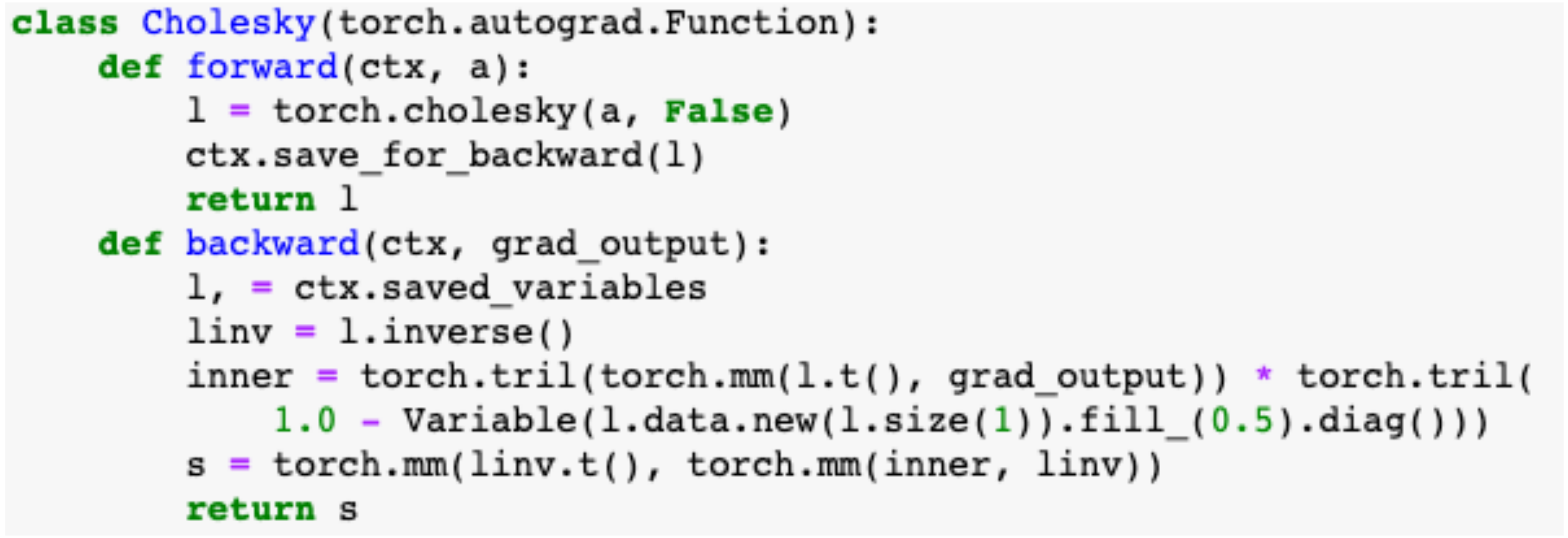
PyTorch Implementation – Loss Function



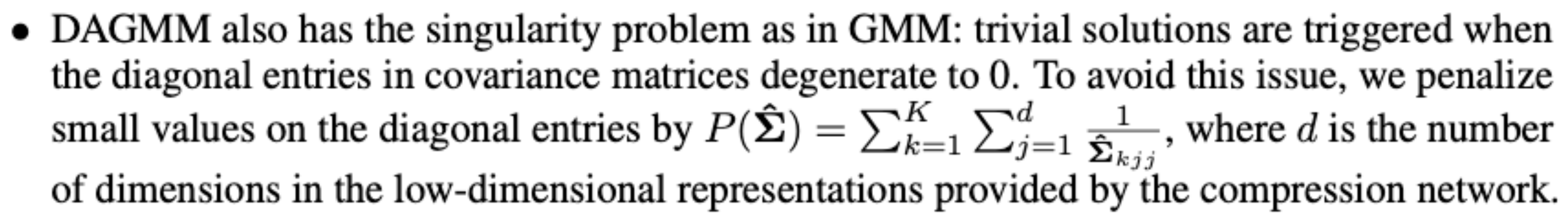
```python
def sample_energy(self, z, gamma, sample_mean=True, mixture_probability=None, mu=None, cov=None):
    if mixture_probability == None or mu == None or cov == None: # calculate
        mixture_probability = torch.sum(gamma, dim=0) / batch_size
        mu = torch.sum(gamma.unsqueeze(-1) * z.unsqueeze(1), dim=0) / torch.sum(gamma, dim=0).unsqueeze(-1)
        z_mu = z.unsqueeze(1) - mu.unsqueeze(0)
        cov = torch.sum(gamma.unsqueeze(-1).unsqueeze(-1) * z_mu.unsqueeze(-1) * z_mu.unsqueeze(-2), dim = 0) / \
              torch.sum(gamma, dim=0).unsqueeze(-1).unsqueeze(-1)
    z_mu = z.unsqueeze(1) - mu.unsqueeze(0)
    eps = 1e-12
    cov_inverse = []
    det_cov = []
    cov_diag = 0
    for k in range(2):
        cov_k = cov[k] + (torch.eye(cov[k].size(-1))*eps)
        cov_inverse.append(torch.inverse(cov_k).unsqueeze(0))
        det_cov.append((Cholesky.apply(cov_k.cpu() * (2*np.pi)).diag().prod()).unsqueeze(0))
        cov_diag += torch.sum(1 / cov_k.diag())
    cov_inverse = torch.cat(cov_inverse, dim=0)
    det_cov = torch.cat(det_cov)
    e_z = -0.5 * torch.sum(torch.sum(z_mu.unsqueeze(-1) * cov_inverse.unsqueeze(0), dim=-2) * z_mu, dim=-1)
    e_z = torch.exp(e_z)
    e_z = -torch.log(torch.sum(mixture_probability.unsqueeze(0)*e_z / (torch.sqrt(det_cov)).unsqueeze(0), dim=1) +
    if sample_mean:
        e_z = torch.mean(e_z)
    # save phi, mu, cov
    self.phi, self.mu, self.cov = mixture_probability, mu, cov
    return e_z, cov_diag
```

$$E(\mathbf{z}) = -\log\left(\sum_{k=1}^{K}\hat{\phi}_k \frac{\exp\left(-\frac{1}{2}(\mathbf{z}-\hat{\mu}_k)^T\hat{\mathbf{\Sigma}}_k^{-1}(\mathbf{z}-\hat{\mu}_k)\right)}{\sqrt{|2\pi\hat{\mathbf{\Sigma}}_k|}}\right).$$

PyTorch Implementation – Loss Function

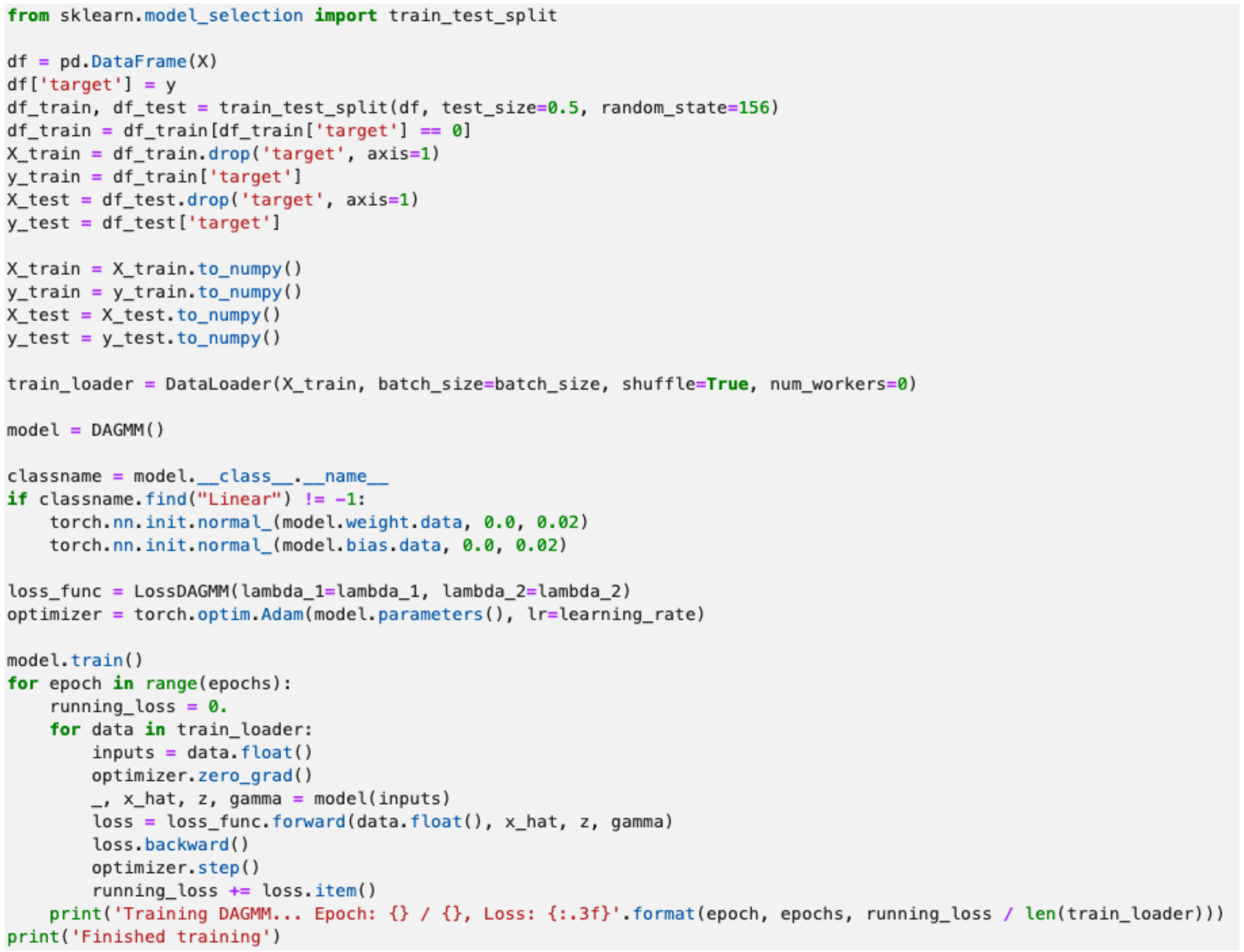


```
class Cholesky(torch.autograd.Function):
    def forward(ctx, a):
        l = torch.cholesky(a, False)
        ctx.save_for_backward(l)
        return l
    def backward(ctx, grad_output):
        l, = ctx.saved_variables
        linv = l.inverse()
        inner = torch.tril(torch.mm(l.t(), grad_output)) * torch.tril(
            1.0 - Variable(l.data.new(l.size(1)).fill_(0.5).diag()))
        s = torch.mm(linv.t(), torch.mm(inner, linv))
        return s
```



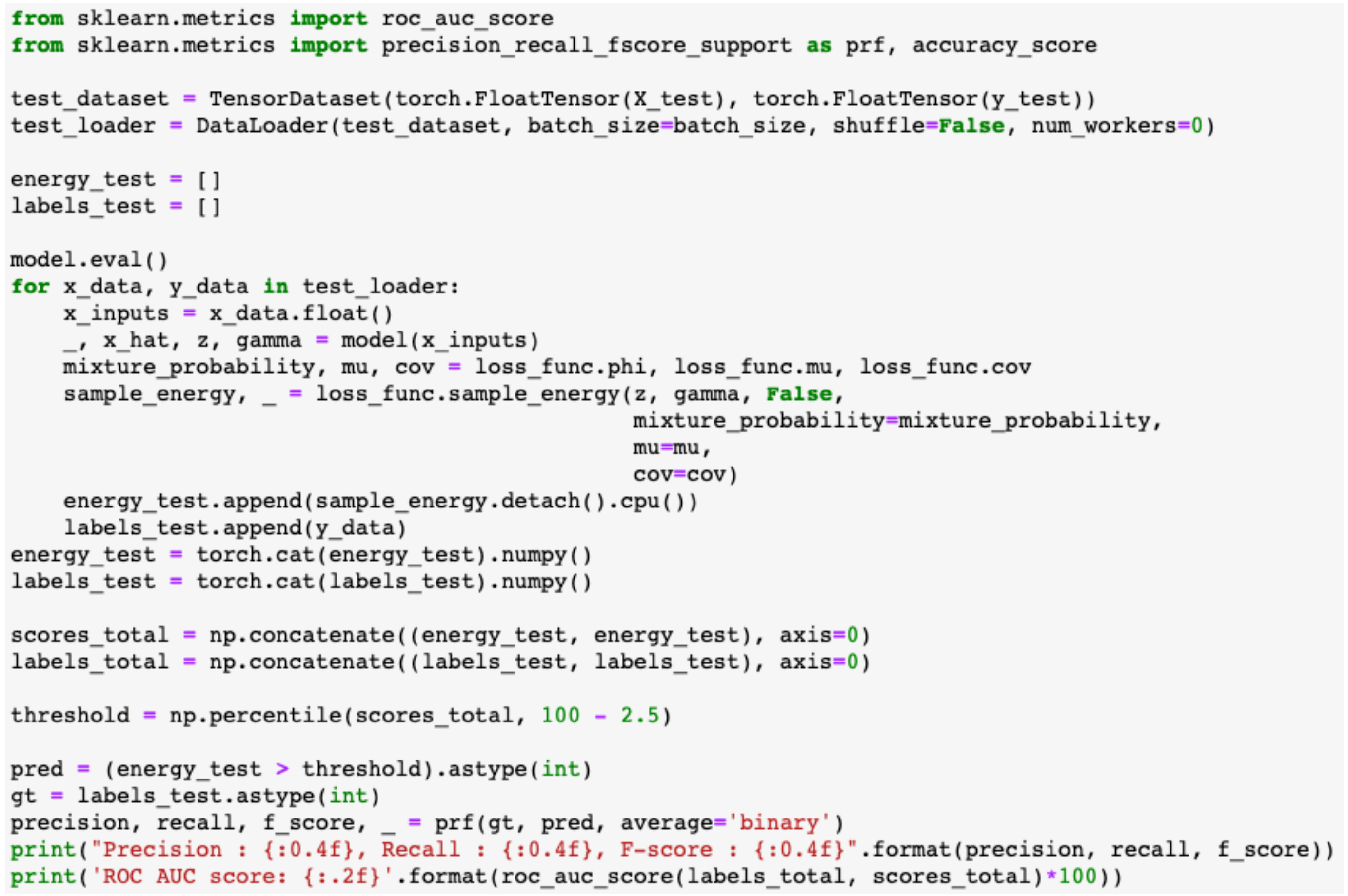
- DAGMM also has the singularity problem as in GMM: trivial solutions are triggered when the diagonal entries in covariance matrices degenerate to 0. To avoid this issue, we penalize small values on the diagonal entries by $P(\hat{\Sigma}) = \sum_{k=1}^{K} \sum_{j=1}^{d} \frac{1}{\hat{\Sigma}_{kjj}}$, where d is the number of dimensions in the low-dimensional representations provided by the compression network.

PyTorch Implementation - Train



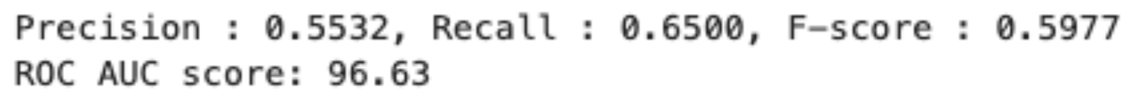
```python
from sklearn.model_selection import train_test_split

df = pd.DataFrame(X)
df['target'] = y
df_train, df_test = train_test_split(df, test_size=0.5, random_state=156)
df_train = df_train[df_train['target'] == 0]
X_train = df_train.drop('target', axis=1)
y_train = df_train['target']
X_test = df_test.drop('target', axis=1)
y_test = df_test['target']

X_train = X_train.to_numpy()
y_train = y_train.to_numpy()
X_test = X_test.to_numpy()
y_test = y_test.to_numpy()

train_loader = DataLoader(X_train, batch_size=batch_size, shuffle=True, num_workers=0)

model = DAGMM()

classname = model.__class__.__name__
if classname.find("Linear") != -1:
    torch.nn.init.normal_(model.weight.data, 0.0, 0.02)
    torch.nn.init.normal_(model.bias.data, 0.0, 0.02)

loss_func = LossDAGMM(lambda_1=lambda_1, lambda_2=lambda_2)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

model.train()
for epoch in range(epochs):
    running_loss = 0.
    for data in train_loader:
        inputs = data.float()
        optimizer.zero_grad()
        _, x_hat, z, gamma = model(inputs)
        loss = loss_func.forward(data.float(), x_hat, z, gamma)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('Training DAGMM... Epoch: {} / {}, Loss: {:.3f}'.format(epoch, epochs, running_loss / len(train_loader)))
print('Finished training')
```

PyTorch Implementation – Inference & Eval

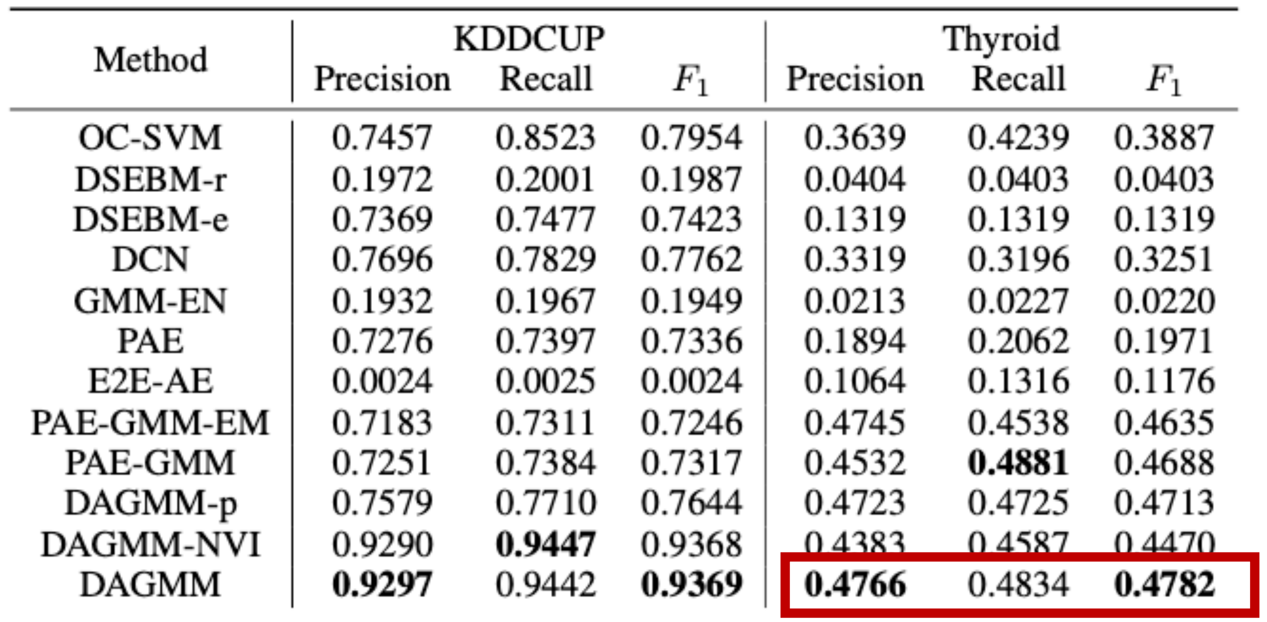


```python
from sklearn.metrics import roc_auc_score
from sklearn.metrics import precision_recall_fscore_support as prf, accuracy_score

test_dataset = TensorDataset(torch.FloatTensor(X_test), torch.FloatTensor(y_test))
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=0)

energy_test = []
labels_test = []

model.eval()
for x_data, y_data in test_loader:
    x_inputs = x_data.float()
    _, x_hat, z, gamma = model(x_inputs)
    mixture_probability, mu, cov = loss_func.phi, loss_func.mu, loss_func.cov
    sample_energy, _ = loss_func.sample_energy(z, gamma, False,
                                               mixture_probability=mixture_probability,
                                               mu=mu,
                                               cov=cov)
    energy_test.append(sample_energy.detach().cpu())
    labels_test.append(y_data)
energy_test = torch.cat(energy_test).numpy()
labels_test = torch.cat(labels_test).numpy()

scores_total = np.concatenate((energy_test, energy_test), axis=0)
labels_total = np.concatenate((labels_test, labels_test), axis=0)

threshold = np.percentile(scores_total, 100 - 2.5)

pred = (energy_test > threshold).astype(int)
gt = labels_test.astype(int)
precision, recall, f_score, _ = prf(gt, pred, average='binary')
print("Precision : {:0.4f}, Recall : {:0.4f}, F-score : {:0.4f}".format(precision, recall, f_score))
print('ROC AUC score: {:.2f}'.format(roc_auc_score(labels_total, scores_total)*100))
```

PyTorch Implementation – Result



```
Precision : 0.5532, Recall : 0.6500, F-score : 0.5977
ROC AUC score: 96.63
```



Method	KDDCUP			Thyroid		
	Precision	Recall	F_1	Precision	Recall	F_1
OC-SVM	0.7457	0.8523	0.7954	0.3639	0.4239	0.3887
DSEBM-r	0.1972	0.2001	0.1987	0.0404	0.0403	0.0403
DSEBM-e	0.7369	0.7477	0.7423	0.1319	0.1319	0.1319
DCN	0.7696	0.7829	0.7762	0.3319	0.3196	0.3251
GMM-EN	0.1932	0.1967	0.1949	0.0213	0.0227	0.0220
PAE	0.7276	0.7397	0.7336	0.1894	0.2062	0.1971
E2E-AE	0.0024	0.0025	0.0024	0.1064	0.1316	0.1176
PAE-GMM-EM	0.7183	0.7311	0.7246	0.4745	0.4538	0.4635
PAE-GMM	0.7251	0.7384	0.7317	0.4532	**0.4881**	0.4688
DAGMM-p	0.7579	0.7710	0.7644	0.4723	0.4725	0.4713
DAGMM-NVI	0.9290	**0.9447**	0.9368	0.4383	0.4587	0.4470
DAGMM	**0.9297**	0.9442	**0.9369**	**0.4766**	0.4834	**0.4782**

Conclusion

- Compression network to project samples into a low-dimensional space (with key information!)
- Estimation network to evaluate sample energy in the low-dimensional space under GMM
- Promising direction for unsupervised anomaly detection on high dimensional data

References

- Zong, B., Song, Q., Min, M. R., Cheng, W., Lumezanu, C., Cho, D., & Chen, H. (2018, February). Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *International Conference on Learning Representations*.
- Chalapathy, R., & Chawla, S. (2019). Deep learning for anomaly detection: A survey. arXiv preprint arXiv:1901.03407.
- Lecun, Yann & Huang, Fu. (2005). Loss Functions for Discriminative Training of Energy-Based Models.
- Zhao, J., Mathieu, M., & LeCun, Y. (2016). Energy-based generative adversarial network. arXiv preprint arXiv:1609.03126.
- https://www.youtube.com/watch?v=kKZM8bxwQbA&ab_channel=KoreaUnivDSBA
- https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95
- https://hoya012.github.io/blog/anomaly-detection-overview-1/
- https://kh-kim.github.io/blog/2019/12/15/Autoencoder-based-anomaly-detection.html
- http://jaejunyoo.blogspot.com/2018/02/energy-based-generative-adversarial-nets-1.html
- https://medium.com/@amaluddin11/credit-card-fraud-detection-7e47750db863
- (Implementation example – TF 1.x) https://github.com/tnakae/DAGMM
- (Implementation example – PyTorch) https://github.com/mperezcarrasco/PyTorch-DAGMM