AI Capstone
Project1 Report

110550093 蔡師睿

1 Introduction

Anomaly detection, an essential task in data analysis, plays a crucial role in various fields. The goal of anomaly detection is to identify observations or patterns within data that significantly diverge from what is considered normal or anticipated. This is important as these anomalies may represent critical events, errors, or abnormalities that require attention.

In this project, I aim to apply anomaly detection to animal images, and thus, I conduct several experiments and compare the performance of two supervised models and one unsupervised model.

2 Dataset

The dataset is shared with my classmate whose student number is 110550110. This dataset consists of two types of images: non-real world animal images and real world animal images. Both these images are downloaded from google images, where we select the images one by one to construct the source of our dataset.

2.1 Source

For the non-real world animal images, they include drawing animals, cartoon animals, AI generated animals, and non-animal images. The total amount of non-real world animal images is 100. While for the real world animal images, they are all real world animals and the total amount of them is 200.

2.2 Train-test Split

We randomly split 80% of the real world animal images and non-real world animal images as our training set, and the rest of 20% are our test set. More details for training or evaluation on the dataset can be found in Section 4.3.

2.3 Preprocessing

We first convert all the downloaded images to JPG file, and then resize them to a size 32*32 pixels. After that, the images are placed into their corresponding folders and renamed using numbers.

3 Methods

3.1 Supervised Methods 1 : K-Nearest Neighbor[1][2]

K-nearest neighbors (KNN) is a simple yet powerful supervised learning algorithm. In our method, KNN calculates the squared Euclidean distance between each element of training set and each element of test set. Then, the prediction of a data point is determined by the majority class of its K nearest neighbors in the feature space, which is calculated based on the squared Euclidean distance. The choice of K is a hyperparameter that affects the performance of this algorithm.

3.2 Supervised Methods 2 : ResNet18[3]

ResNet18 was introduced by Microsoft Research in 2015 and is part of the family of ResNet models. Instead of constructing ResNet18, I directly use the models and its pre-trained weights provided in torchvision library.

3.2.1 Model Architecture

Different from the Fig. 1, there is no softmax layer before the output. Besides, I change the output of FC layer from 1000 to 2 which is the number of classes.
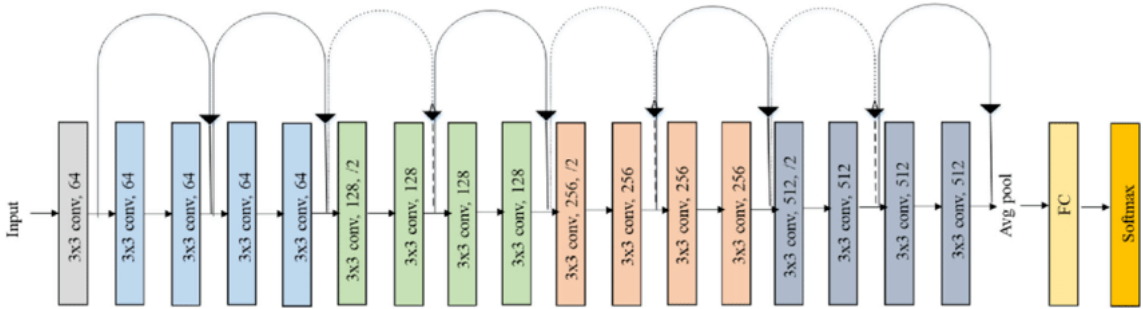

Figure 1: ResNet18 architecture[4]

3.2.2 Loss Function

Cross-entropy loss $\mathcal{L}_{CE}$:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{N}[y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

, where y_i represents the ground truth label, p_i represents the predicted label, and N represents the total number of data.

3.3 Unsupervised Method : Fully Connected AutoEncoder[5]

A fully connected autoencoder is a type of neural network architecture used for unsupervised learning. It consists of an encoder and a decoder, where each layer of the encoder and decoder is fully connected.

3.3.1 Model Architecture

The output of an autoencoder consists of the reconstructed images, which are flattened, enabling the measurement of the reconstruction loss between the flattened ground truth images and the output. Consequently, in the evaluation stage, we can use the reconstruction loss of each output as its anomaly score.
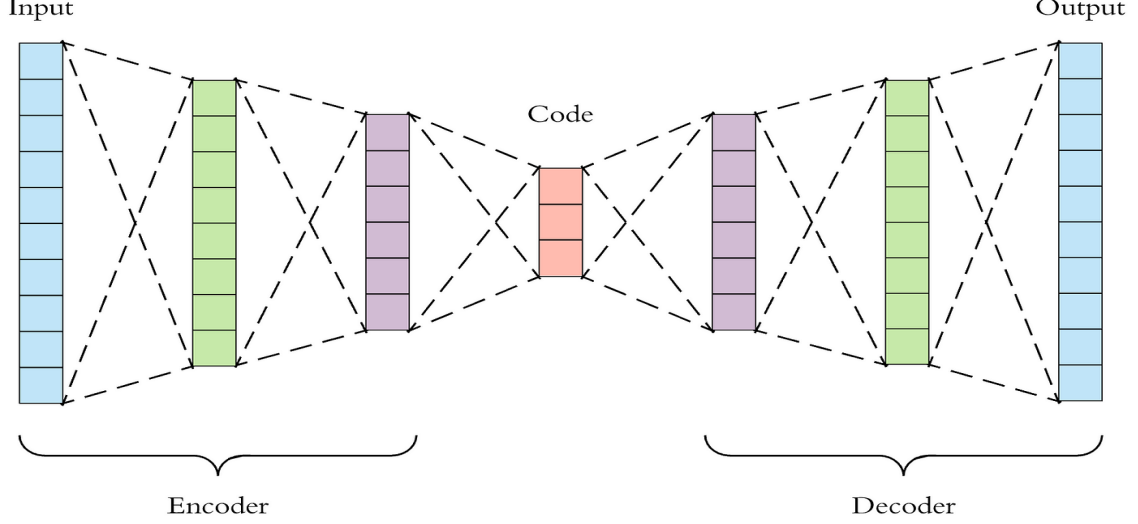

Figure 2: Fully connected autoencoder architecture[6]

3.3.2 Loss Function

Mean square error $\mathcal{L}_{MSE}$ as the reconstruction loss :

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1} (y_i - p_i)^2$$

, where y_i represents a pixel in the ground truth image, p_i represents a pixel in the generated image, and N represents the sum of the pixels for all data points.

4 Implementation Details

4.1 Evaluation Metrics

4.1.1 Supervised Methods

In our supervised models, I employ various evaluation metrics to assess their performance. The following are the four evaluation metrics utilized along with their respective formulas :
1. Accuracy

$$accuracy = \frac{1}{n} \sum_{i=1}^{n} f(x), \; where \; f(x) = \begin{cases} 1, & if \; x = label \\ 0, & otherwise \end{cases}$$

2. Precision (TP represents true positives and FP represents false positives)

$$precision = \frac{TP}{(TP + FP)}$$

3. Recall (TP represents true positives and FN represents false negatives)

$$recall = \frac{TP}{(TP + FN)}$$

4. F1-score

$$F1\text{-}score = \frac{2 * precision * recall}{(precision + recall)}$$

4.1.2 Unsupervised Methods

Since the output of my unsupervised model is anomaly score, I employ ROC curve and ROC-AUC score to assess its performance. The ROC curve plots the true positive rate against the false positive rate for different threshold values, while ROC-AUC score aggregates the area under the ROC curve, providing insight into the model's overall effectiveness.

4.2 Hyperparameters

The random seed is set to 42 to make the results reproducible. In KNN, the parameter K is set to 3, and the rationale behind this choice can be found in Section 5.2. I implemented ResNet18 and Autoencoder using PyTorch, and for ResNet18, I utilized pre-trained weights trained on ImageNet. For fair comparison, I train these two networks with a batch size of 16 on the GPU using Adam, and the learning rate is set to 10^{-4}. I found that training the autoencoder is challenging, and I'm concerned about overfitting on ResNet18. Therefore, I use 50 epochs for training ResNet18 and 100 epochs for training the autoencoder.

4.3 Dataset Trained and Evaluated on Different Methods

We can simply view the anomaly detection task in supervised learning as a binary classification problem. Therefore, the supervised methods train with the entire training set with complete labeling. Unlike supervised methods, unsupervised method trains only with real-world animal images. For evaluation, both supervised and unsupervised learning utilize the same test split to assess their performance.

5 Experiments

5.1 Data Augmentation

I import transforms from torchvision library to perform data augmentation, and apply $transforms.RandomHorizontalFlip()$, $transforms.RandomRotation(45)$, and $transforms.RandomCrop(32, padding = 4)$ to all three methods. We can see that data augmentation definitely contribute to making deep learning-based methods more robust, but there isn't the improvement in KNN. In my point of view, the possible reason is that the squared Euclidean distance changes significantly during data augmentation, leading to inaccuracies. The following are the quantitative comparisons:

Supervised	Aug.	Accuracy(%)	Precision(%)	Recall(%)	F1-score(%)
KNN	✗	**70.00**	**58.33**	**35.00**	**43.75**
KNN	✓	68.33	53.85	**35.00**	42.42

Supervised	Aug.	Accuracy(%)	Precision(%)	Recall(%)	F1-score(%)
ResNet18	✗	83.33	70.83	**85.00**	77.27
ResNet18	✓	**86.67**	**87.50**	70.00	**77.78**

Table 1: Quantitative comparison of the effect of data augmentation on the two supervised methods

Unsupervised	Aug.	ROC-AUC Score(%)
Autoencoder	✗	72.00
Autoencoder	✓	**75.00**

Table 2: Quantitative comparison of the effect of data augmentation on the unsupervised method
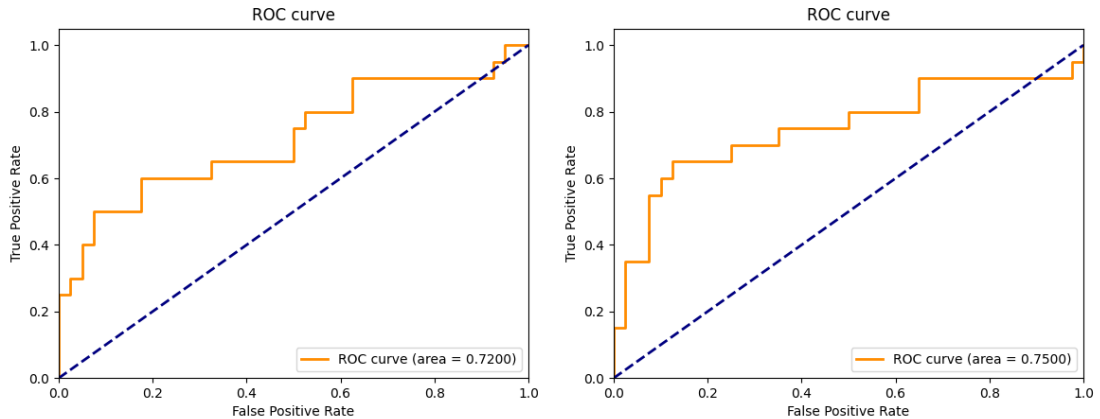

Figure 3: ROC curve comparing the autoencoder with and without data augmentation. The one on the left is without data augmentation, while the one on the right is with data augmentation.

5.2 Find Best K in KNN via Cross Validation

K is a hyperparameter in KNN that significantly impacts the algorithm's performance. Consequently, I have decided to determine the optimal K value within the range of 1 to 10.
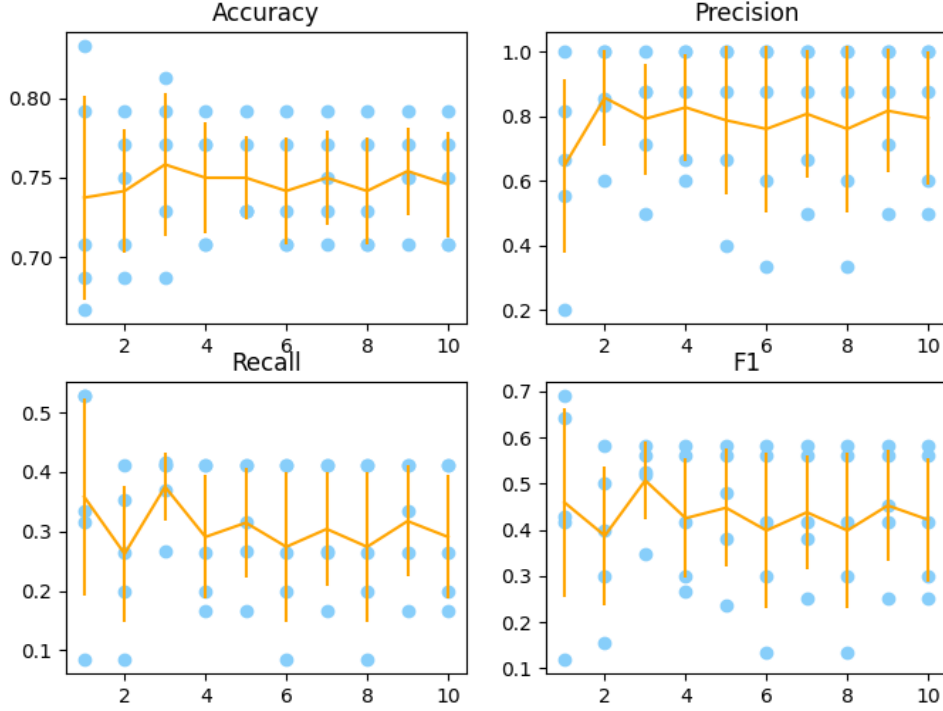
Figure 4: I compare K from 1 to 10, and the result indicates that KNN performs well in most of evaluation metrics as $K = 3$.

5.3 PCA + SMOTE on KNN

Since the input is high-dimensional data, I try to use a dimensionality reduction technique called PCA to project it to a lower dimensional space. PCA can help to remove noise and redundant information from the data, also reducing the computational complexity in KNN.

Other than that, there is an imbalance in our dataset, which can lead to several issues, such as bias in the model and poor generalization. To address this problem, I apply SMOTE to oversample the minority class. This helps to balance the class distribution in the dataset.

Method	Accuracy(%)	Precision(%)	Recall(%)	F1-score(%)
KNN	70.00	58.33	35.00	43.75
PCA + KNN	**73.33**	**59.09**	65.00	**61.90**
PCA + SMOTE + KNN	66.67	50.00	**80.00**	61.54

Table 3: Quantitative comparison between using PCA and SMOTE

According to Table 3, there is a significant improvement when using PCA as preprocessing alone, but the performance of using PCA and SMOTE together doesn't improve as expected. I think that the possible reason is overfitting, because the synthetic samples synthesized from 60 instances to 180 instances may not accurately represent the true distribution of the minority class.

5.4 Different Experiments on ResNet18

Due to the effectiveness of ResNet18, I devised another experiment involving it. First, I adjusted last layer of ResNet18 to output a single scalar value, serving as anomaly score, and then altered the loss function to be $\mathcal{L}_{MSE}$. Next, I established a threshold of 0.5 during the evaluation stage. If the output score surpasses the threshold, it will be categorized as an anomaly; otherwise, it will be labeled as normal. All other settings remain unchanged, including the training set, test set, and hyperparameters.

As shown in Table 4, ResNet18 demonstrates better performance in the classification task compared to the regression task.

Method	Output type	Accuracy(%)	Precision(%)	Recall(%)	F1-score(%)
ResNet18	anomaly score	76.67	60.71	**85.00**	70.83
ResNet18	binary labels	**86.67**	**87.50**	70.00	**77.78**

Table 4: Quantitative comparison between the different output types in ResNet18

5.5 t-SNE Visualization on ResNet18

I apply t-SNE to visualize the distribution of test set in different layers of the ResNet18. Fig. 5 indicates that the input test set is initially clustered together in the 2-dimensional space, but as it passes through the layers of ResNet18, it separates into two distinct parts successfully.
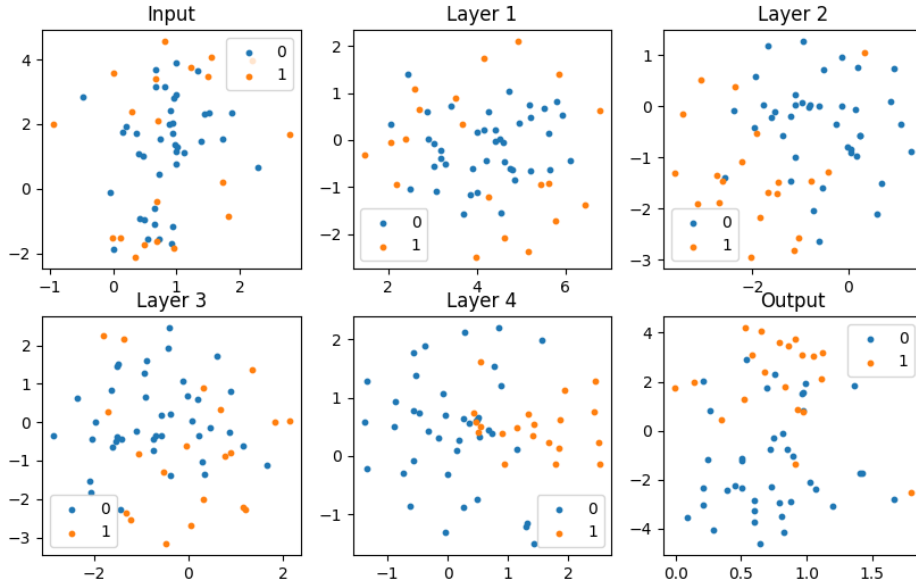

Figure 5: Visualizing t-SNE on each layer of ResNet18
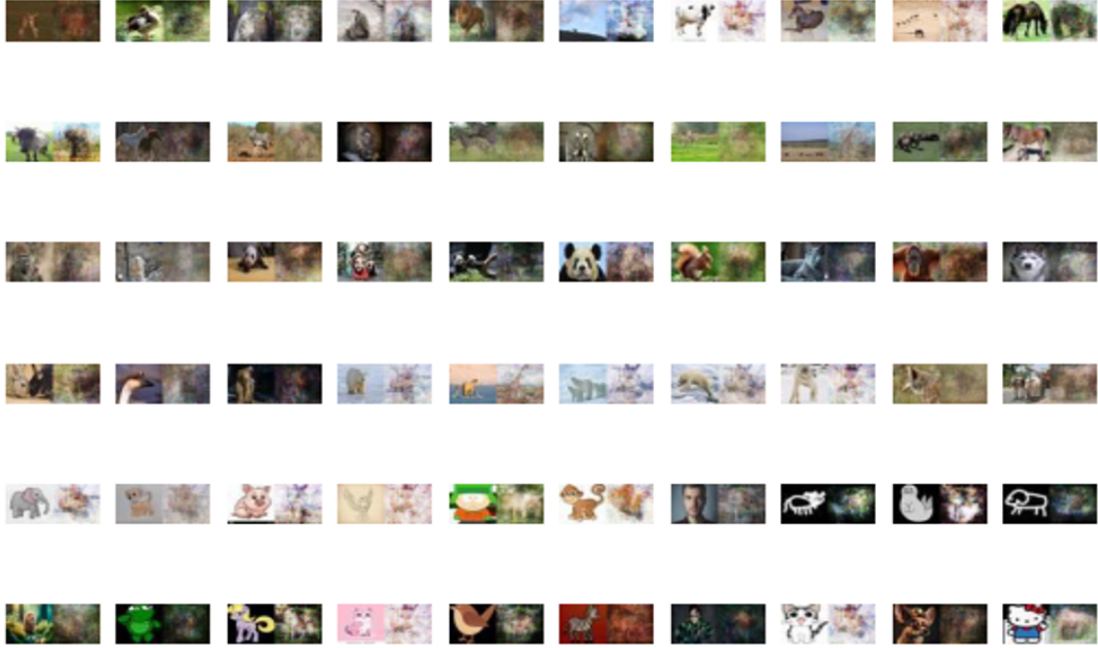
5.6 Visual Results of Autoencoder

Figure 6: Visual Results of autoencoder evaluated on the test set. The ground truth image (left) is concatenated with the reconstructed image (right).

According to Fig. 6, the reconstructed images appear blurred and noisy. This may be attributed to the challenges involved in training the autoencoder, as well as the insufficiency of our dataset for this task.

6 Discussion

6.1 Experiments results

To my surprise, the anomaly detection achieves a high accuracy in autoencoder, despite the reconstructed images not appearing to be of high quality. It seems that the autoencoder could potentially make more accurate predictions if our dataset contained a larger number of real-world animal images.

In supervised learning, as the task is simplified to binary classification, we expect the accuracy to be better than in unsupervised learning. Indeed, ResNet18 performs well, but I believe its accuracy should be higher than 90%. The accuracy of KNN is approximately only 70%, primarily because high-dimensional data may not be well-suited for this algorithm.

6.2 Factors that affect the performance

As mentioned in the previous sections, the main factor influencing performance is the dataset. After collecting the dataset, it should be analyzed more carefully, involving several steps to gain insights into its characteristics, structure, and potential patterns. Moreover, the dataset we collected is too small, which makes training the autoencoder more challenging. The imbalanced data also leads to potential

problems in supervised methods, including overfitting on the majority class, poor generalization, and biased models.

Another factor is the simplicity of the network architecture in our fully connected autoencoder. It is difficult for the model to learn to generalize the characteristics of real-world animal images with only a few linear layers in both the encoder and decoder. However, I encountered some issues using other autoencoder. The issue can be referred to in Section 6.4 of the report.

6.3   More experiments if available

If more time were available, I would prioritize conducting thorough data analysis and cleaning the dataset to reduce noise and ensure its quality. Then, I would like to collect more anomaly data to assess the robustness of the autoencoder in anomaly detection.

6.4   Failure case in the experiments

At the beginning, I use variational autoencoder (VAE) as my unsupervised method. However, when I attempted to visualize the reconstructed results, I discovered that the output for all the test set samples failed to reconstruct properly. Below are the cases of failure in reconstruction.

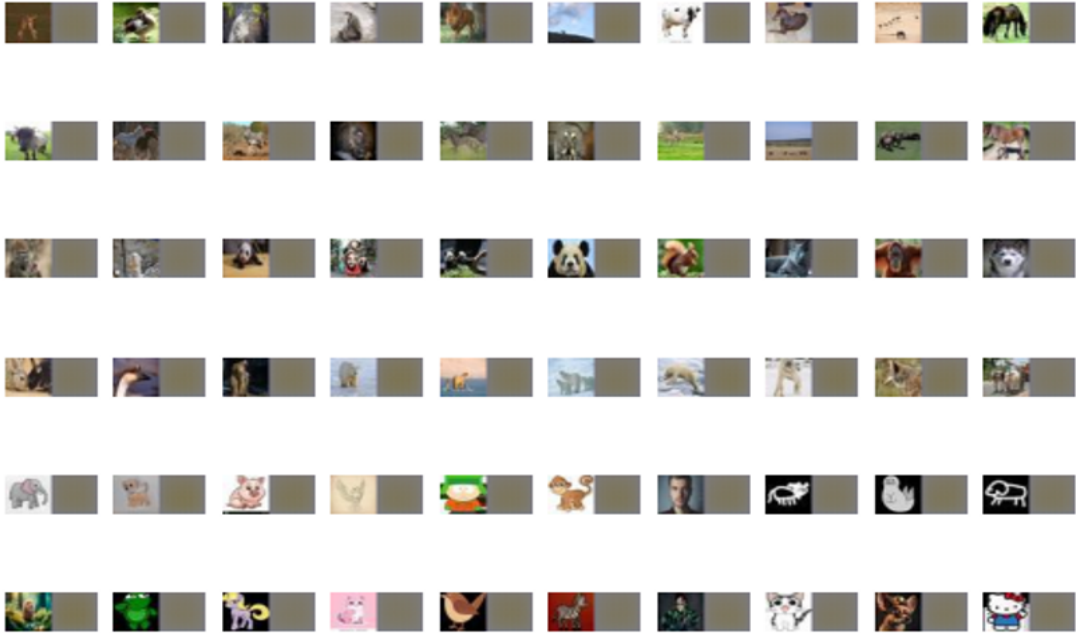Figure 7: Failure cases in VAE. The ground truth image (left) is concatenated with the reconstructed image (right).

I eventually found out that the problem is in the loss function. In addition to reconstruct loss, I introduced KL divergence to constrain the distribution between input and output. Here is the explanation of KL divergence $\mathcal{D}_{KL}$:

$$\mathcal{D}_{KL}(p\|q) = \sum_i p_i log \frac{p_i}{q_i}$$

, where p and q represent the probability distribution of the true labels and the predicted labels, respectively. Additionally, p_i and q_i are the probability of the i-th data point in the true distribution and the predicted distribution, respectively.

After my observations, I noticed that only the KL divergence decreased during training. The problem still existed even though I changed the VAE into β-VAE, where the KL divergence is multiplied by a hyperparameter β to reduce its importance during training. Therefore, I quite the VAE network and use a fully connected autoencoder for the whole experiments.

6.5 Learned from the experiments

In my previous research and project experiences, I consistently relied on available datasets to train models. However, after completing this project, I realized the significant importance of data engineering. There are plenty of noise in the raw data, so it is extremely crucial to have a clean dataset. Without a clean dataset, the experiments and performance evaluations in machine learning or deep learning may easily fail or become impossible to conduct.

Moreover, in the field of anomaly detection, anomaly data may vary significantly across various aspects, making it challenging to collect all possible anomalies. Therefore, unsupervised learning, which merely requires feeding normal data, is more valuable for this task. Based on our experiments, it shows that a simple yet effective autoencoder is able to perform well. In our future work, we can explore increasing our real-world dataset and adopt other more powerful models, such as different types of autoencoders, diffusion models, or SOTA models in anomaly detection, to enhance the ability to detect anomalies.

References

[1] `https://web.eecs.umich.edu/~justincj/slides/eecs498/WI2022/598_WI2022_lecture02.pdf`

[2] `https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/assignment1.html`

[3] `https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html`

[4] A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State fMRI and Residual Neural Networks - Scientific Figure on ResearchGate. Available from: `https://www.researchgate.net/figure/Original-ResNet-18-Architecture_fig1_336642248` [accessed 17 Mar, 2024]

[5] `https://speech.ee.ntu.edu.tw/~hylee/ml/ml2023-course-data/HW08.pdf`

[6] `https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798`

Appendix

Github Link : https://github.com/Ray-1026/AI-Capstone/tree/main/Project1
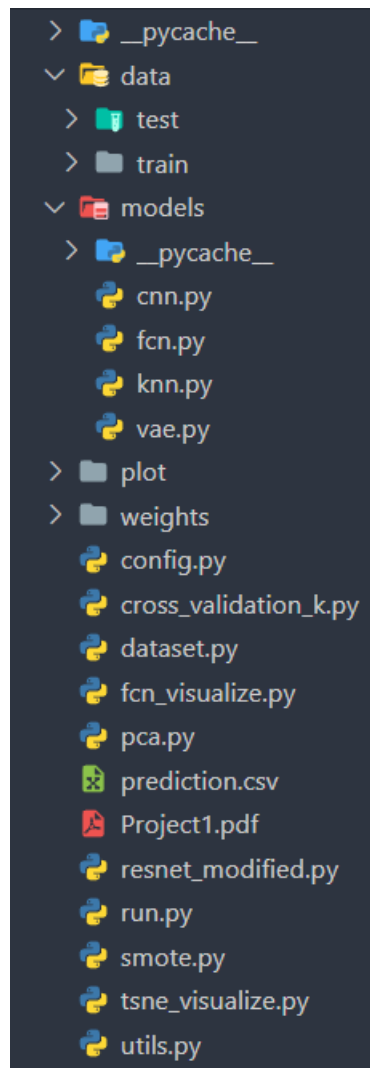
Figure 8: Directory Structure

models/cnn.py

```
1  from torch import nn
2  import torchvision.models as models
3
4
5  class Resnet(nn.Module):
6      def __init__(self):
7          super(Resnet, self).__init__()
8          self.resnet = models.resnet18(pretrained=True)
9          self.resnet.fc = nn.Linear(512, 2)  # change the last
               layer to output 2 values
10
11     def forward(self, x):
12         x = self.resnet(x)
13         return x
```

```python
1  import torch
2  from torch import nn
3
4
5  class fcn_autoencoder(nn.Module):
6      def __init__(self):
7          super(fcn_autoencoder, self).__init__()
8          self.encoder1 = nn.Sequential(
9              nn.Linear(32 * 32 * 3, 1024),
10             nn.ReLU(),
11             nn.Linear(1024, 256),
12             nn.ReLU(),
13             nn.Linear(256, 64),
14         )
15
16         self.decoder = nn.Sequential(
17             nn.Linear(64, 256),
18             nn.ReLU(),
19             nn.Linear(256, 512),
20             nn.ReLU(),
21             nn.Linear(512, 1024),
22             nn.ReLU(),
23             nn.Linear(1024, 32 * 32 * 3),
24             nn.Tanh(),
25         )
26
27     def forward(self, x, train=True):
28         x1 = self.encoder1(x)
29
30         # add gaussian noise during training
31         if train:
32             x1 = x1 + torch.randn(x1.size()).cuda() * 0.05
33
34         out = self.decoder(x1)
35         return out
```

```python
1  import torch
2
3
4  class KNN:
5      def __init__(self, x_train, y_train, K=1):
6          self.x_train = x_train
7          self.y_train = y_train
8          self.K = K
9
10     def predict(self, x_test):
11         dists = self.compute_distance(self.x_train, x_test)
12         return self.predict_labels(dists, self.y_train)
13
14     def compute_distance(self, x_train, x_test):
15         """
```

```
16          Compute squared Euclidean distance using vectorization
17          """
18          num_train = x_train.shape[0]
19          num_test = x_test.shape[0]
20          x_train_flat = x_train.view(num_train, -1)
21          x_test_flat = x_test.view(num_test, -1)
22
23          dists = x_train.new_zeros(num_test, num_train)
24          dists = (
25              torch.sum(x_train_flat**2, dim=1, keepdim=True)
26              + torch.sum(x_test_flat**2, dim=1)
27              - 2 * torch.mm(x_train_flat, x_test_flat.t())
28          )
29          return dists
30
31      def predict_labels(self, dists, y_train):
32          """
33          Find the k nearest neighbors and predict the labels
34          """
35          _, num_test = dists.shape
36          y_pred = torch.zeros(num_test, dtype=torch.int64)
37
38          for i in range(num_test):
39              top_k = torch.topk(dists[:, i], self.K, largest=
                    False)
40              top_k_labels = y_train[top_k.indices]
41              y_pred[i] = torch.bincount(top_k_labels).argmax()
42
43          return y_pred
```

<center>models/vae.py</center>

```
1  import torch
2  from torch import nn
3  from torch.autograd import Variable
4
5
6  class VAE(nn.Module):
7      def __init__(self):
8          super(VAE, self).__init__()
9          self.encoder = nn.Sequential(
10             nn.Conv2d(3, 12, 4, stride=2, padding=1),
11             nn.ReLU(),
12             nn.Conv2d(12, 24, 4, stride=2, padding=1),
13             nn.ReLU(),
14             nn.Conv2d(24, 48, 4, stride=2, padding=1),
15             nn.ReLU(),
16         )
17         self.enc_out_1 = nn.Sequential(
18             nn.Conv2d(48, 24, 4, stride=2, padding=1),
19             nn.ReLU(),
20             nn.Conv2d(24, 24, 4, stride=2, padding=1),
21             nn.ReLU(),
22         )
```

```python
23          self.enc_out_2 = nn.Sequential(
24              nn.Conv2d(48, 24, 4, stride=2, padding=1),
25              nn.ReLU(),
26              nn.Conv2d(24, 24, 4, stride=2, padding=1),
27              nn.ReLU(),
28          )
29          self.decoder = nn.Sequential(
30              nn.ConvTranspose2d(24, 24, 4, stride=2, padding=1),
31              nn.ReLU(),
32              nn.ConvTranspose2d(24, 48, 4, stride=2, padding=1),
33              nn.ReLU(),
34              nn.ConvTranspose2d(48, 24, 4, stride=2, padding=1),
35              nn.ReLU(),
36              nn.ConvTranspose2d(24, 12, 4, stride=2, padding=1),
37              nn.ReLU(),
38              nn.ConvTranspose2d(12, 3, 4, stride=2, padding=1),
39              nn.Tanh(),
40          )
41
42      def encode(self, x):
43          h1 = self.encoder(x)
44          return self.enc_out_1(h1), self.enc_out_2(h1)
45
46      def reparametrize(self, mu, logvar):
47          std = logvar.mul(0.5).exp_()
48          if torch.cuda.is_available():
49              eps = torch.cuda.FloatTensor(std.size()).normal_()
50          else:
51              eps = torch.FloatTensor(std.size()).normal_()
52          eps = Variable(eps)
53
54          return eps.mul(std).add_(mu)
55
56      def decode(self, z):
57          return self.decoder(z)
58
59      def forward(self, x, train=True):
60          mu, logvar = self.encode(x)
61          z = self.reparametrize(mu, logvar)
62          return self.decode(z), mu, logvar
63
64
65  def loss_vae(recon_x, x, mu, logvar, criterion):
66      """
67      recon_x: generating images
68      x: origin images
69      mu: latent mean
70      logvar: latent log variance
71      """
72      mse = criterion(recon_x, x)
73      KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).
            add_(logvar)
74      KLD = torch.sum(KLD_element).mul_(-0.5)
```

```
75        return mse + KLD
```

config.py

```
1  # hyperparameters and configurations
2  num_epochs = 50   # 50 for resnet, 100 for vae
3  batch_size = 16
4  learning_rate = 1e-4
5
6  # for KNN
7  K = 3
8
9  model_type = "cnn"   # choose model type from ["cnn", "vae", "knn
       ", "fcn"]
10
11 # output filename
12 output_filename = "prediction.csv"
```

cross_validation.py

```
1  import numpy as np
2  import os
3  import matplotlib.pyplot as plt
4
5  import config as cfg
6  from models.knn import KNN
7  from dataset import KNNDataset
8  from utils import same_seeds, supervised_score, save_result
9  from sklearn.model_selection import KFold
10
11 same_seeds(42)
12
13
14 def plot_metrics(accs, pres, recs, f1s, Ks):
15     """
16     visualize results of cross validation
17     """
18     plt.figure(figsize=(8, 6))
19     met_name = ["Accuracy", "Precision", "Recall", "F1"]
20     for i, met in enumerate([accs, pres, recs, f1s]):
21         plt.subplot(2, 2, i + 1)
22         ks, means, stds = [], [], []
23         for k, m in met.items():
24             plt.scatter([k] * len(m), m, color="lightskyblue")
25             ks.append(k)
26             means.append(np.mean(m))
27             stds.append(np.std(m))
28         plt.errorbar(ks, means, yerr=stds, color="orange")
29         plt.title(f"{met_name[i]}")
30     plt.savefig("plot/knn_cross_validation.png")
31     plt.show()
32
33
34 def knn_cross_validation():
```

```python
35      """
36      find the best K for KNN model using cross validation
37      """
38      Ks = range(1, 11)
39      accs, pres, recs, f1s = (
40          {k: [] for k in Ks},
41          {k: [] for k in Ks},
42          {k: [] for k in Ks},
43          {k: [] for k in Ks},
44      )
45      x_train, y_train, x_test, y_test = KNNDataset(
46          os.path.join("data", "train"), os.path.join("data", "
                test")
47      )
48
49      kf = KFold(n_splits=5, shuffle=True)
50      for train_index, test_index in kf.split(x_train):
51          # split train and validation set
52          x_train_fold, x_val_fold = x_train[train_index], x_train
                [test_index]
53          y_train_fold, y_val_fold = y_train[train_index], y_train
                [test_index]
54
55          for K in Ks:
56              model = KNN(x_train_fold, y_train_fold, K=K)
57              y_pred = model.predict(x_val_fold)
58              save_result(y_val_fold.cpu().numpy(), y_pred.cpu().
                    numpy())
59              ret = supervised_score(cfg.output_filename,
                    print_result=False)
60              accs[K].append(ret[0])
61              pres[K].append(ret[1])
62              recs[K].append(ret[2])
63              f1s[K].append(ret[3])
64
65      plot_metrics(accs, pres, recs, f1s, Ks)
66
67      # find the best K
68      best_K = 0
69      best_acc = 0
70      for K in Ks:
71          acc = np.mean(accs[K])
72          prec = np.mean(pres[K])
73          rec = np.mean(recs[K])
74          f1 = np.mean(f1s[K])
75          print(
76              f"K: {K:2d}, Accuracy: {acc:.4f}, Precision: {prec
                    :.4f}, Recall: {rec:.4f}, F1: {f1:.4f}"
77          )
78          if acc > best_acc:
79              best_acc = acc
80              best_K = K
81
```

```
82      print(f"best K: {best_K}", end=", ")
83
84      # test the best K
85      model = KNN(x_train, y_train, K=best_K)
86      y_pred = model.predict(x_test)
87      save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
88      supervised_score(cfg.output_filename, print_result=True)
89
90
91  if __name__ == "__main__":
92      knn_cross_validation()
```

<div align="center">dataset.py</div>

```
1   import torch
2   from torch.utils.data import Dataset
3   from torchvision import transforms
4   from PIL import Image
5   import os
6
7
8   class UnsupervisedDataset(Dataset):
9       """
10      Custom dataset for unsupervised learning (VAE, Fully
            Connected Autoencoder)
11      """
12
13      def __init__(self, root, train=True):
14          self.root = root
15          self.train_transform = transforms.Compose(
16              [
17                  transforms.ToTensor(),
18                  transforms.Lambda(lambda x: x.to(torch.float32))
                        ,
19                  transforms.Lambda(lambda x: 2.0 * x - 1.0),
20                  transforms.RandomHorizontalFlip(),
21                  transforms.RandomRotation(45),
22                  transforms.RandomCrop(32, padding=4),
23              ]
24          )
25          self.test_transform = transforms.Compose(
26              [
27                  transforms.ToTensor(),
28                  transforms.Lambda(lambda x: x.to(torch.float32))
                        ,
29                  transforms.Lambda(lambda x: 2.0 * x - 1.0),
30              ]
31          )
32
33          self.train = train
34
35          if self.train:
36              normal = os.listdir(os.path.join(self.root, "normal"
                    ))
```

```
37            self.images = [os.path.join(self.root, "normal", img
                  ) for img in normal]
38        else:
39            normal = os.listdir(os.path.join(self.root, "normal"
                  ))
40            abnormal = os.listdir(os.path.join(self.root, "
                  abnormal"))
41            self.images = [os.path.join(self.root, "normal", img
                  ) for img in normal] + [
42                os.path.join(self.root, "abnormal", img) for img
                      in abnormal
43            ]
44            self.labels = [0] * len(normal) + [1] * len(abnormal
                  )
45
46    def __len__(self):
47        return len(self.images)
48
49    def __getitem__(self, idx):
50        image = Image.open(self.images[idx])
51        if self.train:
52            image = self.train_transform(image)
53        else:
54            image = self.test_transform(image)
55
56        if self.train:
57            return image
58        else:
59            return image, self.labels[idx], self.images[idx]
60
61
62 class SupervisedDataset(Dataset):
63    """
64    Custom dataset for supervised learning (ResNet)
65    """
66
67    def __init__(self, root, train=True):
68        self.root = root
69        self.train_transform = transforms.Compose(
70            [
71                transforms.ToTensor(),
72                transforms.Lambda(lambda x: x.to(torch.float32))
                      ,
73                transforms.Lambda(lambda x: 2.0 * x - 1.0),
74                transforms.RandomHorizontalFlip(),
75                transforms.RandomRotation(45),
76                transforms.RandomCrop(32, padding=4),
77            ]
78        )
79        self.test_transform = transforms.Compose(
80            [
81                transforms.ToTensor(),
82                transforms.Lambda(lambda x: x.to(torch.float32))
```

```python
                  ,
                  transforms.Lambda(lambda x: 2.0 * x - 1.0),
              ]
          )

          self.train = train

          normal = os.listdir(os.path.join(self.root, "normal"))
          abnormal = os.listdir(os.path.join(self.root, "abnormal"
              ))
          self.images = [os.path.join(self.root, "normal", img)
              for img in normal] + [
                os.path.join(self.root, "abnormal", img) for img in
                    abnormal
          ]
          self.labels = [0] * len(normal) + [1] * len(abnormal)

      def __len__(self):
          return len(self.images)

      def __getitem__(self, idx):
          image = Image.open(self.images[idx])
          if self.train:
              image = self.train_transform(image)
          else:
              image = self.test_transform(image)

          if self.train:
              return image, self.labels[idx]
          else:
              return image, self.labels[idx], self.images[idx]


def KNNDataset(train_path, test_path, aug=False):
    """
    Custom dataset for KNN model
    """
    x_train = [
        os.path.join(train_path, "normal", i)
        for i in os.listdir(os.path.join(train_path, "normal"))
    ] + [
        os.path.join(train_path, "abnormal", i)
        for i in os.listdir(os.path.join(train_path, "abnormal")
            )
    ]
    y_train = [0] * len(os.listdir(os.path.join(train_path, "
        normal"))) + [1] * len(
        os.listdir(os.path.join(train_path, "abnormal"))
    )

    x_test = [
        os.path.join(test_path, "normal", i)
        for i in os.listdir(os.path.join(test_path, "normal"))
```

```
130        ] + [
131            os.path.join(test_path, "abnormal", i)
132            for i in os.listdir(os.path.join(test_path, "abnormal"))
133        ]
134        y_test = [0] * len(os.listdir(os.path.join(test_path, "
                normal"))) + [1] * len(
135            os.listdir(os.path.join(test_path, "abnormal"))
136        )
137
138        if aug:
139            train_transform = transforms.Compose(
140                [
141                    transforms.ToTensor(),
142                    transforms.RandomHorizontalFlip(),
143                    transforms.RandomRotation(45),
144                    transforms.RandomCrop(32, padding=4),
145                ]
146            )
147        else:
148            train_transform = transforms.Compose(
149                [
150                    transforms.ToTensor(),
151                ]
152            )
153
154        test_transform = transforms.Compose(
155            [
156                transforms.ToTensor(),
157            ]
158        )
159
160        x_train = [Image.open(i) for i in x_train]
161        x_train = torch.stack([train_transform(i) for i in x_train])
162        y_train = torch.tensor(y_train)
163        x_test = [Image.open(i) for i in x_test]
164        x_test = torch.stack([test_transform(i) for i in x_test])
165        y_test = torch.tensor(y_test)
166
167        return x_train, y_train, x_test, y_test
```

<center>fcn_visualize.py</center>

```
1  import numpy as np
2  import torch
3  import matplotlib.pyplot as plt
4  from torch.utils.data import DataLoader, SequentialSampler
5
6  from models.fcn import fcn_autoencoder
7  from models.vae import VAE
8  from dataset import UnsupervisedDataset
9  from utils import same_seeds
10
11 same_seeds(42)
12
```

```
13
14  def autoencoder_visual():
15      """
16      Visualize the reconstructed images from Fully Connected
            Autoencoder
17      """
18      model = fcn_autoencoder().cuda()
19      model.load_state_dict(torch.load("weights/best_fcn.pth"))
20      model.eval()
21
22      # dataset & dataloader
23      dataset = UnsupervisedDataset("data/test", train=False)
24      dataloader = DataLoader(
25          dataset,
26          batch_size=60,
27          sampler=SequentialSampler(dataset),
28      )
29
30      with torch.no_grad():
31          for i, (data, label, name) in enumerate(dataloader):
32              inputs = data.cuda()
33              outputs = model(inputs.view(inputs.size(0), -1),
                    train=False)
34              outputs = outputs.view(outputs.size(0), 3, 32, 32)
35
36              # outputs, _, _ = model(inputs, train=False)
37
38              # plot images
39              for i in range(60):
40                  plt.subplot(6, 10, i + 1)
41                  img1 = inputs[i].permute(1, 2, 0).cpu().numpy()
                        * 0.5 + 0.5
42                  img2 = (outputs[i].permute(1, 2, 0).cpu().numpy
                        () + 1) * 0.5
43                  img_cat = np.concatenate([img1, img2], axis=1)
44                  plt.imshow(img_cat)
45                  plt.axis("off")
46              plt.savefig("plot/fcn_visualize.png")
47              plt.show()
48              break
49
50
51  if __name__ == "__main__":
52      autoencoder_visual()
```

---

pca.py

```
1  import torch
2  import os
3  from sklearn.decomposition import PCA
4
5  import config as cfg
6  from models.knn import KNN
7  from dataset import KNNDataset
```

```python
 8  from utils import same_seeds, supervised_score, save_result
 9
10
11  same_seeds(42)
12
13
14  def KNN_PCA():
15      """
16      PCA + KNN
17      """
18      x_train, y_train, x_test, y_test = KNNDataset(
19          os.path.join("data", "train"), os.path.join("data", "
               test")
20      )
21
22      # use PCA to reduce dimension to 2
23      pca = PCA(n_components=2)
24      x_train_pca = pca.fit_transform(x_train.view(x_train.size(0)
               , -1))
25      x_test_pca = pca.fit_transform(x_test.view(x_test.size(0),
               -1))
26      x_train_pca = torch.tensor(x_train_pca)
27      x_test_pca = torch.tensor(x_test_pca)
28
29      model = KNN(x_train_pca, y_train, K=cfg.K)
30      y_pred = model.predict(x_test_pca)
31      save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
32      supervised_score(cfg.output_filename)
33
34
35  if __name__ == "__main__":
36      KNN_PCA()
```

---

resnet_modified.py

```python
 1  import numpy as np
 2  import torch
 3  import os
 4  import torchvision.models as models
 5  from torch import nn
 6  from torch.optim import Adam
 7  from tqdm import tqdm
 8  from torch.utils.data import DataLoader, RandomSampler,
        SequentialSampler
 9
10  import config as cfg
11  from dataset import SupervisedDataset
12  from utils import same_seeds, supervised_score, save_result
13
14
15  same_seeds(42)
16
17
18  class Resnet_M(nn.Module):
```

```python
19      def __init__(self):
20          super(Resnet_M, self).__init__()
21          self.resnet = models.resnet18(pretrained=True)
22          self.resnet.fc = nn.Linear(512, 1)  # change the last
                layer to output 1 value
23
24      def forward(self, x):
25          x = self.resnet(x)
26          return x
27
28
29  def train():
30      if not os.path.exists("weights"):
31          os.mkdir("weights")
32
33      model = Resnet_M().cuda()
34
35      train_dataset = SupervisedDataset("data/train")
36      train_loader = DataLoader(
37          train_dataset,
38          batch_size=cfg.batch_size,
39          sampler=RandomSampler(train_dataset),
40      )
41
42      criterion = nn.MSELoss()
43      optimizer = Adam(model.parameters(), lr=cfg.learning_rate)
44
45      best_loss = np.inf
46
47      for epoch in range(cfg.num_epochs):
48          model.train()
49          train_loss = []
50
51          with tqdm(total=len(train_loader), unit="batch") as
                tqdm_bar:
52              tqdm_bar.set_description(f"Epoch {epoch+1:02d}/{cfg.
                    num_epochs}")
53              for data in train_loader:
54                  images, labels = data
55                  images = images.float().cuda()
56                  labels = labels.cuda().view(-1, 1).float()
57
58                  output = model(images)
59
60                  loss = criterion(output, labels)
61                  train_loss.append(loss.item())
62
63                  optimizer.zero_grad()
64                  loss.backward()
65                  optimizer.step()
66
67                  tqdm_bar.set_postfix(
68                      loss=f"{sum(train_loss)/len(train_loss):.5f}
```

```python
                              ",
                  )
                  tqdm_bar.update()

          if np.mean(train_loss) < best_loss:
              best_loss = np.mean(train_loss)
              torch.save(model.state_dict(), f"weights/
                  best_resnet_m.pth")


def test():
    model = Resnet_M().cuda()
    model.load_state_dict(torch.load(f"weights/best_resnet_m.pth
        "))
    model.eval()

    test_dataset = SupervisedDataset("data/test", train=False)
    test_loader = DataLoader(
        test_dataset,
        batch_size=cfg.batch_size,
        sampler=SequentialSampler(test_dataset),
    )

    threshold = 0.5  # threshold for evaluating anomality

    anomality, names, labels = list(), list(), list()
    with torch.no_grad():
        for data, label, name in test_loader:
            img = data.float().cuda()

            output = model(img)
            loss = output > threshold

            anomality.append(loss)
            names.extend([n.split("test\\")[-1] for n in name])
            labels.extend(label)

    anomality = torch.cat(anomality, axis=0)
    anomality = torch.sqrt(anomality).reshape(len(test_dataset),
        1).cpu().numpy()

    save_result(np.array(labels), anomality.flatten())
    supervised_score(cfg.output_filename)


if __name__ == "__main__":
    train()
    test()
```

run.py

```python
import numpy as np
import torch
import os
```

```python
 4  from torch import nn
 5  from torch.optim import Adam
 6  from tqdm import tqdm
 7  from torch.utils.data import DataLoader, RandomSampler,
        SequentialSampler
 8
 9  import config as cfg
10  from models.cnn import Resnet
11  from models.knn import KNN
12  from models.vae import VAE, loss_vae
13  from models.fcn import fcn_autoencoder
14  from dataset import UnsupervisedDataset, SupervisedDataset,
        KNNDataset
15  from utils import same_seeds, unsupervised_score,
        supervised_score, save_result
16
17  same_seeds(42)
18
19  model_classes = {
20      "cnn": Resnet(),
21      "vae": VAE(),
22      "fcn": fcn_autoencoder(),
23  }  # dictionary of models
24
25
26  def train():
27      if cfg.model_type == "knn":
28          return
29
30      if not os.path.exists("weights"):
31          os.mkdir("weights")
32
33      # model
34      model = model_classes[cfg.model_type].cuda()
35
36      # dataset & dataloader
37      if cfg.model_type == "cnn":
38          train_dataset = SupervisedDataset("data/train")
39      else:
40          train_dataset = UnsupervisedDataset("data/train")
41      train_loader = DataLoader(
42          train_dataset,
43          batch_size=cfg.batch_size,
44          sampler=RandomSampler(train_dataset),
45      )
46
47      # loss & optimizer
48      if cfg.model_type == "cnn":
49          criterion = nn.CrossEntropyLoss()
50      else:
51          criterion = nn.MSELoss()
52      optimizer = Adam(model.parameters(), lr=cfg.learning_rate)
53
```

```python
54     best_loss = np.inf
55
56     for epoch in range(cfg.num_epochs):
57         model.train()
58         train_loss = []
59
60         with tqdm(total=len(train_loader), unit="batch") as
           tqdm_bar:
61             tqdm_bar.set_description(f"Epoch {epoch+1:02d}/{cfg.
               num_epochs}")
62             for data in train_loader:
63                 if cfg.model_type != "cnn":
64                     images = data.float().cuda()
65
66                     if cfg.model_type == "fcn":
67                         images = images.view(images.size(0), -1)
68                 else:
69                     images, labels = data
70                     images = images.float().cuda()
71                     labels = labels.cuda()
72
73                 output = model(images)
74
75                 # different models use different criterion
76                 if cfg.model_type == "vae":
77                     loss = loss_vae(output[0], images, output
                       [1], output[2], criterion)
78                 elif cfg.model_type == "fcn":
79                     loss = criterion(output, images)
80                 else:
81                     loss = criterion(output, labels)
82
83                 train_loss.append(loss.item())
84
85                 optimizer.zero_grad()
86                 loss.backward()
87                 optimizer.step()
88
89                 tqdm_bar.set_postfix(
90                     loss=f"{sum(train_loss)/len(train_loss):.5f}
                       ",
91                 )
92                 tqdm_bar.update()
93
94         # save best model
95         if np.mean(train_loss) < best_loss:
96             best_loss = np.mean(train_loss)
97             torch.save(model.state_dict(), f"weights/best_{cfg.
               model_type}.pth")
98
99
100 def test():
101     if cfg.model_type == "knn":
```

```
102        x_train, y_train, x_test, y_test = KNNDataset(
103            os.path.join("data", "train"), os.path.join("data",
                    "test")
104        )
105        model = KNN(x_train, y_train, K=cfg.K)
106        y_pred = model.predict(x_test)
107        save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
108    else:
109        model = model_classes[cfg.model_type].cuda()
110
111        # load weights
112        model.load_state_dict(torch.load(f"weights/best_{cfg.
                model_type}.pth"))
113
114        # dataset & dataloader
115        if cfg.model_type == "cnn":
116            test_dataset = SupervisedDataset("data/test", train=
                    False)
117        else:
118            test_dataset = UnsupervisedDataset("data/test",
                    train=False)
119        test_loader = DataLoader(
120            test_dataset,
121            batch_size=cfg.batch_size,
122            sampler=SequentialSampler(test_dataset),
123        )
124
125        # loss
126        eval_loss = nn.MSELoss(reduction="none")
127        model.eval()
128
129        anomality, names, labels = list(), list(), list()
130        with torch.no_grad():
131            for data, label, name in test_loader:
132                img = data.float().cuda()
133
134                if cfg.model_type == "cnn":
135                    output = model(img)
136                    loss = (output.argmax(1)).int()
137                elif cfg.model_type == "fcn":
138                    img = img.view(img.size(0), -1)
139                    output = model(img, train=False)
140                    loss = eval_loss(output, img).sum(1)
141                else:
142                    output = model(img, train=False)
143                    output = output[0]
144                    loss = eval_loss(output, img).sum([1, 2, 3])
145
146                anomality.append(loss)
147                names.extend([n.split("test\\")[-1] for n in
                    name])
148                labels.extend(label)
149
```

```
150            # calculate anomaly score
151            anomality = torch.cat(anomality, axis=0)
152            anomality = torch.sqrt(anomality).reshape(len(
                   test_dataset), 1).cpu().numpy()
153
154            save_result(np.array(labels), anomality.flatten())
155
156        if cfg.model_type != "vae" and cfg.model_type != "fcn":
157            supervised_score(cfg.output_filename)
158        else:
159            unsupervised_score(cfg.output_filename)
160
161
162 if __name__ == "__main__":
163        train()
164        test()
```

```
 1 import torch
 2 import os
 3 from imblearn.over_sampling import SMOTE
 4 from sklearn.decomposition import PCA
 5
 6 import config as cfg
 7 from models.knn import KNN
 8 from dataset import KNNDataset
 9 from utils import same_seeds, supervised_score, save_result
10
11 same_seeds(42)
12
13
14 def smote():
15     """
16     this function is used to apply SMOTE to oversampling and PCA
           to reduce dimension
17     """
18     x_train, y_train, x_test, y_test = KNNDataset(
19         os.path.join("data", "train"), os.path.join("data", "
               test")
20     )
21
22     # use PCA to reduce dimension
23     pca = PCA(n_components=2)
24     x_train_pca = pca.fit_transform(x_train.view(x_train.size(0)
           , -1))
25     x_test_pca = pca.fit_transform(x_test.view(x_test.size(0),
           -1))
26
27     # use SMOTE to oversampling
28     sm = SMOTE()
29     x_train_pca, y_train = sm.fit_resample(x_train_pca, y_train)
30
31     x_train_pca = torch.tensor(x_train_pca)
```

```
32        y_train = torch.tensor(y_train)
33        x_test_pca = torch.tensor(x_test_pca)
34
35        model = KNN(x_train_pca, y_train, K=cfg.K)
36        y_pred = model.predict(x_test_pca)
37        save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
38        supervised_score(cfg.output_filename)
39
40
41   if __name__ == "__main__":
42        smote()
```

<div align="center">tsne__visualize.py</div>

```
 1   # draw t-SNE
 2   import matplotlib.pyplot as plt
 3   import numpy as np
 4   import torch
 5   from tqdm import tqdm
 6   from torch.utils.data import DataLoader, SequentialSampler
 7   from sklearn.manifold import TSNE
 8
 9   import config as cfg
10   from models.cnn import Resnet
11   from dataset import SupervisedDataset
12   from utils import same_seeds
13
14   same_seeds(42)
15
16
17   def tSNE_visualize():
18       """
19       Visualize the t-SNE of the features from each layer
20       """
21       model = Resnet().cuda()
22       model.load_state_dict(torch.load(f"weights/best_cnn.pth"))
23       model.eval()
24
25       print(model)  # print model architecture
26
27       test_dataset = SupervisedDataset("data/test")
28       test_loader = DataLoader(
29           test_dataset,
30           batch_size=cfg.batch_size,
31           sampler=SequentialSampler(test_dataset),
32       )
33
34       def hook(module, input, output):
35           """
36           hook for extracting features
37           """
38           temp.append(output.clone().detach())
39
40       handles = []
```

```python
41     handles.append(model.resnet.layer1.register_forward_hook(
           hook))
42     handles.append(model.resnet.layer2.register_forward_hook(
           hook))
43     handles.append(model.resnet.layer3.register_forward_hook(
           hook))
44     handles.append(model.resnet.layer4.register_forward_hook(
           hook))
45     handles.append(model.resnet.fc.register_forward_hook(hook))
46
47     features_tsnes = []
48
49     # input
50     features = []
51     labels = []
52     for img, label in tqdm(test_loader):
53         with torch.no_grad():
54             logits = img.view(img.size()[0], -1)
55             labels.extend(label.numpy())
56             logits = np.squeeze(logits.cpu().numpy())
57             features.extend(logits)
58     features = np.array(features)
59     features_tsnes.append(TSNE(n_components=2).fit_transform(
           features))
60
61     # layers 1-4 and output
62     for handle in handles:
63         features = []
64         labels = []
65         with torch.no_grad():
66             for img, label in tqdm(test_loader):
67                 temp = []
68                 _ = model(img.cuda())
69                 logits = temp[0].view(temp[0].size()[0], -1)
70                 labels.extend(label.numpy())
71                 logits = np.squeeze(logits.cpu().numpy())
72                 features.extend(logits)
73         handle.remove()
74
75         features = np.array(features)
76         features_tsnes.append(TSNE(n_components=2).fit_transform
           (features))
77
78     # draw t-SNE
79     plt.figure(figsize=(10, 6))
80     for i in range(len(features_tsnes)):
81         plt.subplot(2, 3, i + 1)
82         for label in np.unique(labels):
83             plt.scatter(
84                 features_tsnes[i][labels == label, 0],
85                 features_tsnes[i][labels == label, 1],
86                 label=label,
87                 s=10,
```

```
 88                  )
 89              if i == 0:
 90                  plt.title("Input")
 91              elif i == 5:
 92                  plt.title("Output")
 93              else:
 94                  plt.title(f"Layer {i}")
 95          plt.legend()
 96      plt.savefig("plot/tsne.png")
 97      plt.show()
 98
 99
100  if __name__ == "__main__":
101      tSNE_visualize()
```

---

utils.py

```
 1  import numpy as np
 2  import torch
 3  import pandas as pd
 4  import matplotlib.pyplot as plt
 5  import config as cfg
 6  from sklearn.metrics import (
 7      roc_auc_score,
 8      precision_score,
 9      recall_score,
10      f1_score,
11      roc_curve,
12  )
13  from sklearn.preprocessing import MinMaxScaler
14
15
16  def same_seeds(myseed):
17      """
18      Set the random seeds to make the experiments reproducible.
19      """
20      torch.backends.cudnn.deterministic = True
21      torch.backends.cudnn.benchmark = False
22      np.random.seed(myseed)
23      torch.manual_seed(myseed)
24      if torch.cuda.is_available():
25          torch.cuda.manual_seed_all(myseed)
26
27
28  def save_result(y_test, y_pred):
29      """
30      Save the results to a csv file.
31      """
32      result = pd.DataFrame(
33          {
34              "label": y_test,
35              "score": y_pred,
36          }
37      )
```

```python
38      result = result.sort_values(by="score", ascending=False)
39      result.to_csv(cfg.output_filename, index=False)
40
41
42  def unsupervised_score(file_name):
43      """
44      Calculate the unsupervised evaluation metrics.
45      """
46      csv_file = file_name
47      df = pd.read_csv(csv_file)
48      df["label"] = df["label"].astype(int)
49      df["score"] = MinMaxScaler().fit_transform(df[["score"]])
50
51      y_true = df["label"]
52      y_pred = df["score"]
53
54      roc_auc = roc_auc_score(y_true, y_pred)
55      print(f"roc_auc: {roc_auc:.4f}")
56
57      # draw ROC curve
58      fpr, tpr, _ = roc_curve(y_true, y_pred)
59      plt.figure()
60      lw = 2
61      plt.plot(
62          fpr, tpr, color="darkorange", lw=lw, label="ROC curve (
              area = %0.4f)" % roc_auc
63      )
64      plt.plot([0, 1], [0, 1], color="navy", lw=lw, linestyle="--"
          )
65      plt.xlim([0.0, 1.0])
66      plt.ylim([0.0, 1.05])
67      plt.xlabel("False Positive Rate")
68      plt.ylabel("True Positive Rate")
69      plt.title("ROC curve")
70      plt.legend(loc="lower right")
71      plt.savefig(f"plot/{cfg.model_type}_roc.png")
72      plt.show()
73
74
75  def supervised_score(file_name, print_result=True):
76      """
77      Calculate the supervised evaluation metrics.
78      """
79      csv_file = file_name
80      df = pd.read_csv(csv_file)
81      y_true = df["label"]
82      y_pred = df["score"]
83
84      # precision, recall, F1-score
85      precision = precision_score(y_true, y_pred)
86      recall = recall_score(y_true, y_pred)
87      f1 = f1_score(y_true, y_pred)
88
```

```
89      if print_result:
90          print(
91              f"acc: {np.mean((y_true == y_pred)):.4f}, precision:
                    {precision:.4f}, recall: {recall:.4f}, f1: {f1
                    :.4f}"
92          )
93      else:
94          return np.mean((y_true == y_pred)), precision, recall,
                f1
```