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.

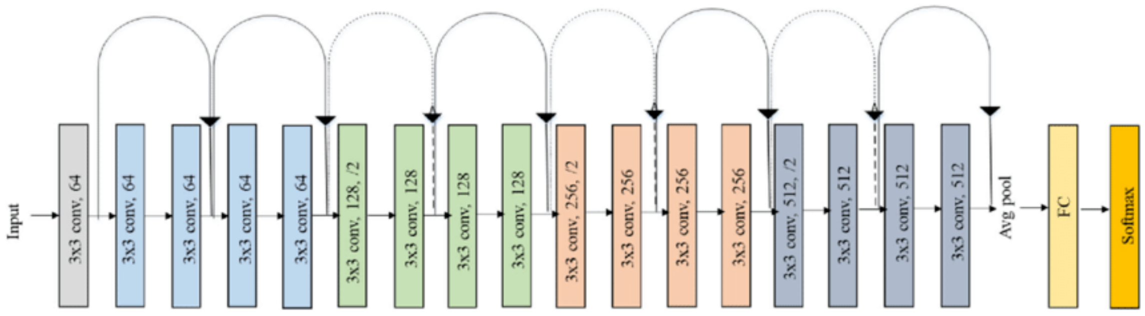


Figure 1: ResNet18 architecture[4]

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.

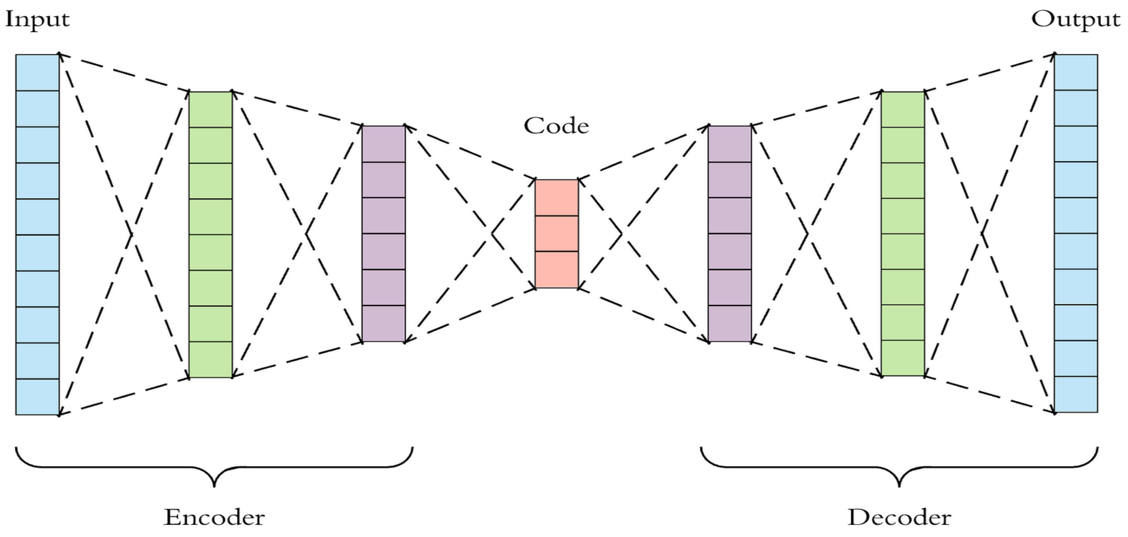


Figure 2: Fully connected autoencoder architecture[6]

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
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

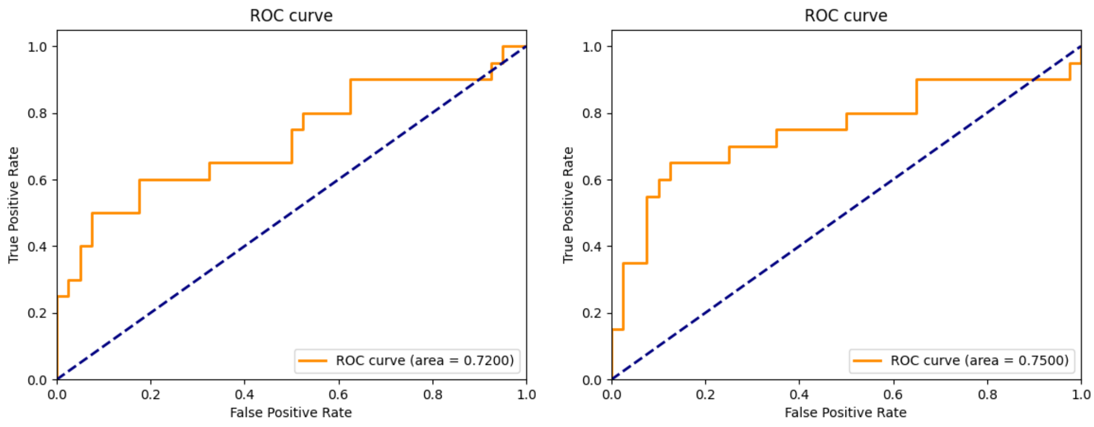


Figure 3: ROC curve comparing the autoencoder with and without data augmentation. The one on the left is without data augmentation, while the one on the right is with data augmentation.

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.

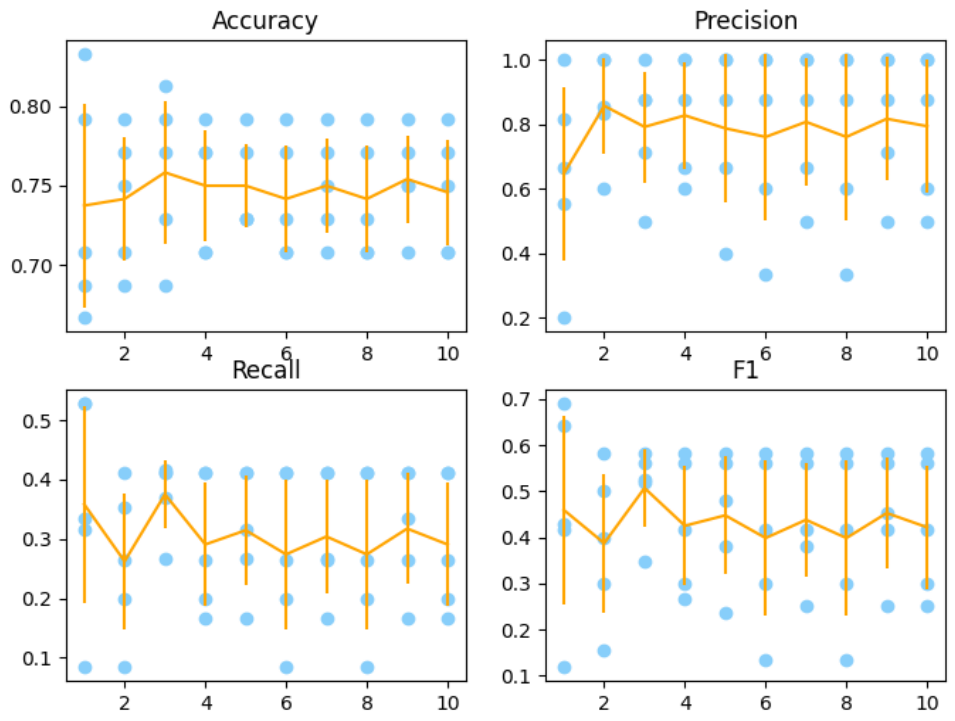


Figure 4: I compare K from 1 to 10, and the result indicates that KNN performs well in most of evaluation metrics as $K = 3$.

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.

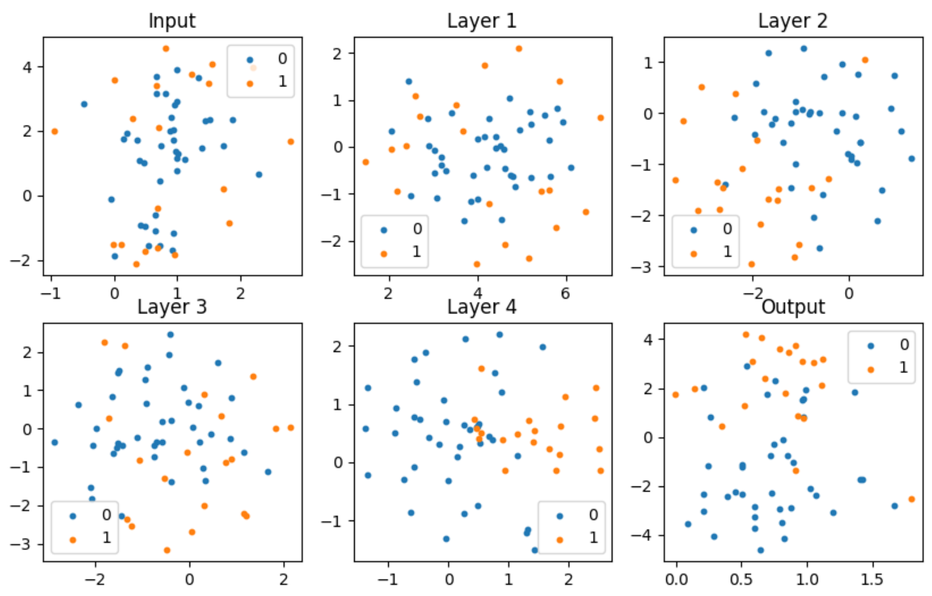


Figure 5: Visualizing t-SNE on each layer of ResNet18

5.6 Visual Results of Autoencoder

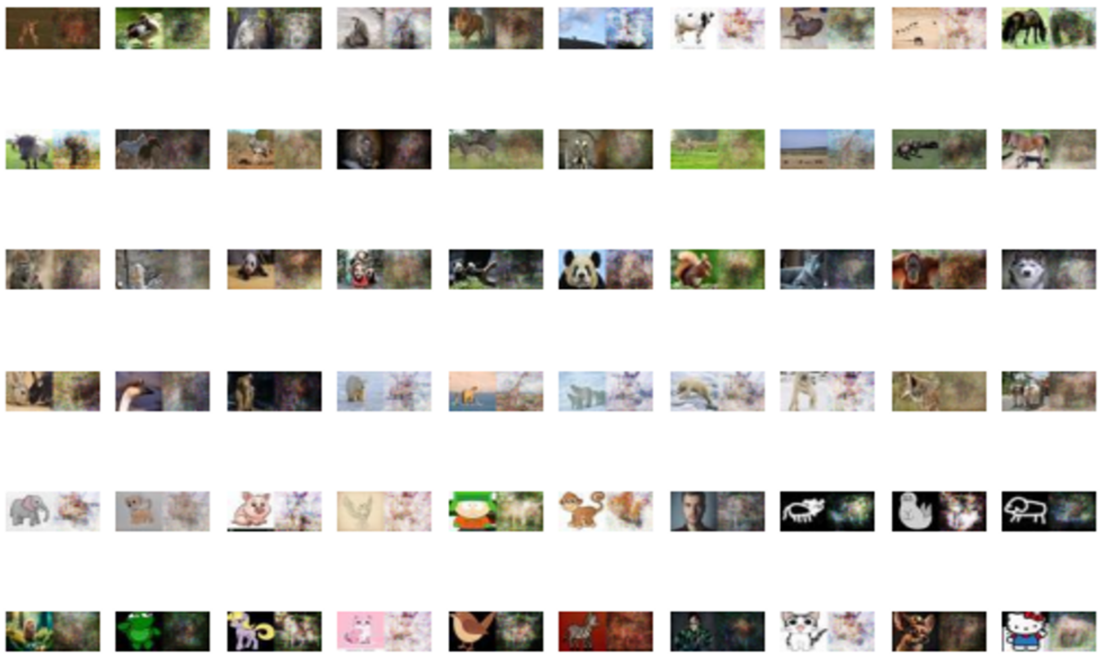


Figure 6: Visual Results of autoencoder evaluated on the test set. The ground truth image (left) is concatenated with the reconstructed image (right).

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.

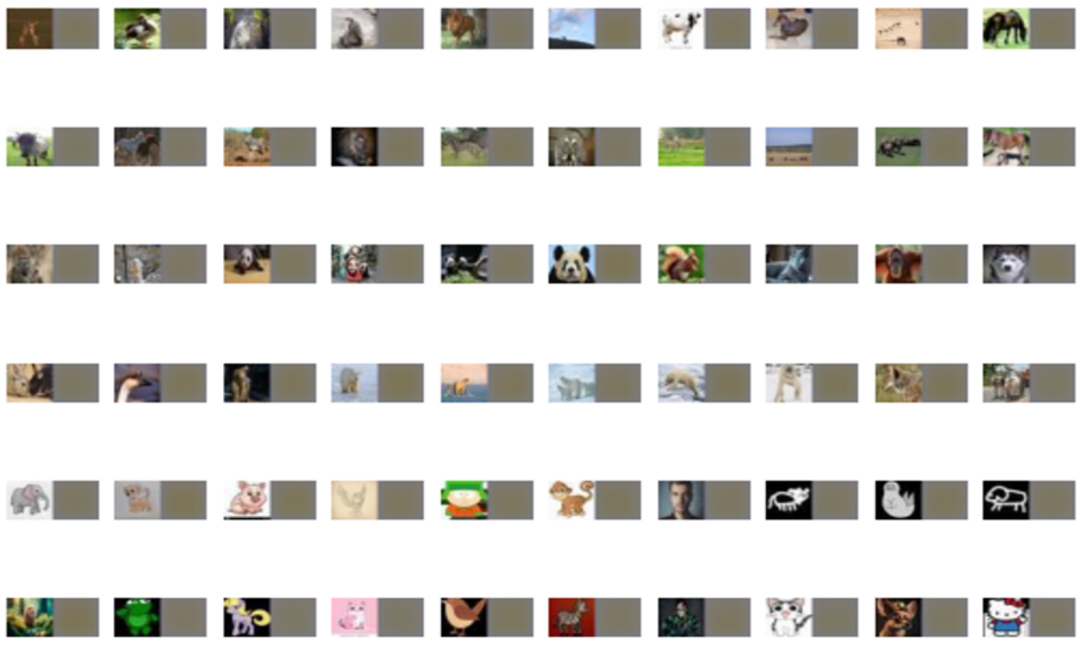


Figure 7: Failure cases in VAE. The ground truth image (left) is concatenated with the reconstructed image (right).

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`

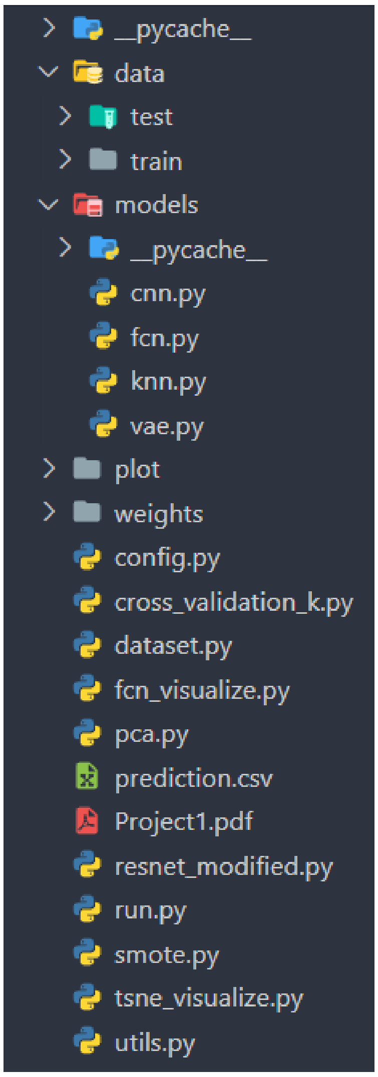
[4] A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State fMRI and Residual Neural Networks - Scientific Figure on ResearchGate. Available from: `https://www.researchgate.net/figure/Original-ResNet-18-Architecture_fig1_336642248`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



```
> __pycache__
v data
    > test
    > train
v models
    > __pycache__
      cnn.py
      fcn.py
      knn.py
      vae.py
> plot
> weights
  config.py
  cross_validation_k.py
  dataset.py
  fcn_visualize.py
  pca.py
  prediction.csv
  Project1.pdf
  resnet_modified.py
  run.py
  smote.py
  tsne_visualize.py
  utils.py
```

Figure 8: Directory Structure

models/cnn.py

```python
from torch import nn
import torchvision.models as models


class Resnet(nn.Module):
    def __init__(self):
        super(Resnet, self).__init__()
        self.resnet = models.resnet18(pretrained=True)
        self.resnet.fc = nn.Linear(512, 2)  # change the last layer to output 2 values

    def forward(self, x):
        x = self.resnet(x)
        return x
```

models/fcn.py

```python
import torch
from torch import nn


class fcn_autoencoder(nn.Module):
    def __init__(self):
        super(fcn_autoencoder, self).__init__()
        self.encoder1 = nn.Sequential(
            nn.Linear(32 * 32 * 3, 1024),
            nn.ReLU(),
            nn.Linear(1024, 256),
            nn.ReLU(),
            nn.Linear(256, 64),
        )

        self.decoder = nn.Sequential(
            nn.Linear(64, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 32 * 32 * 3),
            nn.Tanh(),
        )

    def forward(self, x, train=True):
        x1 = self.encoder1(x)

        # add gaussian noise during training
        if train:
            x1 = x1 + torch.randn(x1.size()).cuda() * 0.05

        out = self.decoder(x1)
        return out
```

models/knn.py

```python
import torch


class KNN:
    def __init__(self, x_train, y_train, K=1):
        self.x_train = x_train
        self.y_train = y_train
        self.K = K

    def predict(self, x_test):
        dists = self.compute_distance(self.x_train, x_test)
        return self.predict_labels(dists, self.y_train)

    def compute_distance(self, x_train, x_test):
        """
        Compute squared Euclidean distance using vectorization
        """
        num_train = x_train.shape[0]
        num_test = x_test.shape[0]
        x_train_flat = x_train.view(num_train, -1)
        x_test_flat = x_test.view(num_test, -1)

        dists = x_train.new_zeros(num_test, num_train)
        dists = (
            torch.sum(x_train_flat**2, dim=1, keepdim=True)
            + torch.sum(x_test_flat**2, dim=1)
            - 2 * torch.mm(x_train_flat, x_test_flat.t())
        )
        return dists

    def predict_labels(self, dists, y_train):
        """
        Find the k nearest neighbors and predict the labels
        """
        _, num_test = dists.shape
        y_pred = torch.zeros(num_test, dtype=torch.int64)

        for i in range(num_test):
            top_k = torch.topk(dists[:, i], self.K, largest=
               False)
            top_k_labels = y_train[top_k.indices]
            y_pred[i] = torch.bincount(top_k_labels).argmax()

        return y_pred
```

models/vae.py

```
import torch
from torch import nn
from torch.autograd import Variable


class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 12, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(12, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(24, 48, 4, stride=2, padding=1),
            nn.ReLU(),
        )
        self.enc_out_1 = nn.Sequential(
            nn.Conv2d(48, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(24, 24, 4, stride=2, padding=1),
            nn.ReLU(),
        )
        self.enc_out_2 = nn.Sequential(
            nn.Conv2d(48, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(24, 24, 4, stride=2, padding=1),
            nn.ReLU(),
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(24, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(24, 48, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(48, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(24, 12, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(12, 3, 4, stride=2, padding=1),
            nn.Tanh(),
        )

    def encode(self, x):
        h1 = self.encoder(x)
        return self.enc_out_1(h1), self.enc_out_2(h1)

    def reparametrize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        if torch.cuda.is_available():
            eps = torch.cuda.FloatTensor(std.size()).normal_()
        else:
            eps = torch.FloatTensor(std.size()).normal_()
        eps = Variable(eps)

        return eps.mul(std).add_(mu)

    def decode(self, z):
        return self.decoder(z)

    def forward(self, x, train=True):
        mu, logvar = self.encode(x)
        z = self.reparametrize(mu, logvar)
        return self.decode(z), mu, logvar


def loss_vae(recon_x, x, mu, logvar, criterion):
    """
    recon_x: generating images
    x: origin images
    mu: latent mean
    logvar: latent log variance
    """
    mse = criterion(recon_x, x)
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).
       add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)
```

```python
    return mse + KLD
```

config.py

```python
# hyperparameters and configurations
num_epochs = 50  # 50 for resnet, 100 for vae
batch_size = 16
learning_rate = 1e-4

# for KNN
K = 3

model_type = "cnn"  # choose model type from ["cnn", "vae", "knn
    ", "fcn"]

# output filename
output_filename = "prediction.csv"
```

cross_validation.py

```python
import numpy as np
import os
import matplotlib.pyplot as plt

import config as cfg
from models.knn import KNN
from dataset import KNNDataset
from utils import same_seeds, supervised_score, save_result
from sklearn.model_selection import KFold

same_seeds(42)


def plot_metrics(accs, pres, recs, f1s, Ks):
    """
    visualize results of cross validation
    """
    plt.figure(figsize=(8, 6))
    met_name = ["Accuracy", "Precision", "Recall", "F1"]
    for i, met in enumerate([accs, pres, recs, f1s]):
        plt.subplot(2, 2, i + 1)
        ks, means, stds = [], [], []
        for k, m in met.items():
            plt.scatter([k] * len(m), m, color="lightskyblue")
            ks.append(k)
            means.append(np.mean(m))
            stds.append(np.std(m))
        plt.errorbar(ks, means, yerr=stds, color="orange")
        plt.title(f"{met_name[i]}")
    plt.savefig("plot/knn_cross_validation.png")
    plt.show()


def knn_cross_validation():
    """
    find the best K for KNN model using cross validation
    """
    Ks = range(1, 11)
    accs, pres, recs, f1s = (
        {k: [] for k in Ks},
        {k: [] for k in Ks},
        {k: [] for k in Ks},
        {k: [] for k in Ks},
    )
    x_train, y_train, x_test, y_test = KNNDataset(
        os.path.join("data", "train"), os.path.join("data", "test")
    )

    kf = KFold(n_splits=5, shuffle=True)
    for train_index, test_index in kf.split(x_train):
        # split train and validation set
        x_train_fold, x_val_fold = x_train[train_index], x_train[test_index]
        y_train_fold, y_val_fold = y_train[train_index], y_train[test_index]

        for K in Ks:
            model = KNN(x_train_fold, y_train_fold, K=K)
            y_pred = model.predict(x_val_fold)
            save_result(y_val_fold.cpu().numpy(), y_pred.cpu().numpy())
            ret = supervised_score(cfg.output_filename, print_result=False)
            accs[K].append(ret[0])
            pres[K].append(ret[1])
            recs[K].append(ret[2])
            f1s[K].append(ret[3])

    plot_metrics(accs, pres, recs, f1s, Ks)

    # find the best K
    best_K = 0
    best_acc = 0
    for K in Ks:
        acc = np.mean(accs[K])
        prec = np.mean(pres[K])
        rec = np.mean(recs[K])
        f1 = np.mean(f1s[K])
        print(
            f"K: {K:2d}, Accuracy: {acc:.4f}, Precision: {prec:.4f}, Recall: {rec:.4f}, F1: {f1:.4f}"
        )
        if acc > best_acc:
            best_acc = acc
            best_K = K

    print(f"best K: {best_K}", end=", ")

    # test the best K
    model = KNN(x_train, y_train, K=best_K)
    y_pred = model.predict(x_test)
    save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
    supervised_score(cfg.output_filename, print_result=True)


if __name__ == "__main__":
    knn_cross_validation()
```

dataset.py

```
import torch
from torch.utils.data import Dataset
from torchvision import transforms
from PIL import Image
import os


class UnsupervisedDataset(Dataset):
    """
    Custom dataset for unsupervised learning (VAE, Fully
       Connected Autoencoder)
    """

    def __init__(self, root, train=True):
        self.root = root
        self.train_transform = transforms.Compose(
            [
                transforms.ToTensor(),
                transforms.Lambda(lambda x: x.to(torch.float32))
                    ,
                transforms.Lambda(lambda x: 2.0 * x - 1.0),
                transforms.RandomHorizontalFlip(),
                transforms.RandomRotation(45),
                transforms.RandomCrop(32, padding=4),
            ]
        )
        self.test_transform = transforms.Compose(
            [
                transforms.ToTensor(),
                transforms.Lambda(lambda x: x.to(torch.float32))
                    ,
                transforms.Lambda(lambda x: 2.0 * x - 1.0),
            ]
        )

        self.train = train

        if self.train:
            normal = os.listdir(os.path.join(self.root, "normal"
                ))
```

```python
            self.images = [os.path.join(self.root, "normal", img) for img in normal]
        else:
            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
        else:
            return image, self.labels[idx], self.images[idx]


class SupervisedDataset(Dataset):
    """
    Custom dataset for supervised learning (ResNet)
    """

    def __init__(self, root, train=True):
        self.root = root
        self.train_transform = transforms.Compose(
            [
                transforms.ToTensor(),
                transforms.Lambda(lambda x: x.to(torch.float32)),
                transforms.Lambda(lambda x: 2.0 * x - 1.0),
                transforms.RandomHorizontalFlip(),
                transforms.RandomRotation(45),
                transforms.RandomCrop(32, padding=4),
            ]
        )
        self.test_transform = transforms.Compose(
            [
                transforms.ToTensor(),
                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"))
    ] + [
        os.path.join(test_path, "abnormal", i)
        for i in os.listdir(os.path.join(test_path, "abnormal"))
    ]
    y_test = [0] * len(os.listdir(os.path.join(test_path, "
       normal"))) + [1] * len(
        os.listdir(os.path.join(test_path, "abnormal"))
    )

    if aug:
        train_transform = transforms.Compose(
            [
                transforms.ToTensor(),
                transforms.RandomHorizontalFlip(),
                transforms.RandomRotation(45),
                transforms.RandomCrop(32, padding=4),
            ]
        )
    else:
        train_transform = transforms.Compose(
            [
                transforms.ToTensor(),
            ]
        )

    test_transform = transforms.Compose(
        [
            transforms.ToTensor(),
        ]
    )

    x_train = [Image.open(i) for i in x_train]
    x_train = torch.stack([train_transform(i) for i in x_train])
    y_train = torch.tensor(y_train)
    x_test = [Image.open(i) for i in x_test]
    x_test = torch.stack([test_transform(i) for i in x_test])
    y_test = torch.tensor(y_test)

    return x_train, y_train, x_test, y_test
```

fcn_visualize.py

```python
import numpy as np
import torch
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader, SequentialSampler

from models.fcn import fcn_autoencoder
from models.vae import VAE
from dataset import UnsupervisedDataset
from utils import same_seeds

same_seeds(42)

```

```python

def autoencoder_visual():
    """
    Visualize the reconstructed images from Fully Connected
       Autoencoder
    """
    model = fcn_autoencoder().cuda()
    model.load_state_dict(torch.load("weights/best_fcn.pth"))
    model.eval()

    # dataset & dataloader
    dataset = UnsupervisedDataset("data/test", train=False)
    dataloader = DataLoader(
        dataset,
        batch_size=60,
        sampler=SequentialSampler(dataset),
    )

    with torch.no_grad():
        for i, (data, label, name) in enumerate(dataloader):
            inputs = data.cuda()
            outputs = model(inputs.view(inputs.size(0), -1),
               train=False)
            outputs = outputs.view(outputs.size(0), 3, 32, 32)

            # outputs, _, _ = model(inputs, train=False)

            # plot images
            for i in range(60):
                plt.subplot(6, 10, i + 1)
                img1 = inputs[i].permute(1, 2, 0).cpu().numpy()
                   * 0.5 + 0.5
                img2 = (outputs[i].permute(1, 2, 0).cpu().numpy
                   () + 1) * 0.5
                img_cat = np.concatenate([img1, img2], axis=1)
                plt.imshow(img_cat)
                plt.axis("off")
            plt.savefig("plot/fcn_visualize.png")
            plt.show()
            break


if __name__ == "__main__":
    autoencoder_visual()
```

pca.py

```python
import torch
import os
from sklearn.decomposition import PCA

import config as cfg
from models.knn import KNN
from dataset import KNNDataset
from utils import same_seeds, supervised_score, save_result


same_seeds(42)


def KNN_PCA():
    """
    PCA + KNN
    """
    x_train, y_train, x_test, y_test = KNNDataset(
        os.path.join("data", "train"), os.path.join("data", "
            test")
    )

    # use PCA to reduce dimension to 2
    pca = PCA(n_components=2)
    x_train_pca = pca.fit_transform(x_train.view(x_train.size(0)
        , -1))
    x_test_pca = pca.fit_transform(x_test.view(x_test.size(0),
        -1))
    x_train_pca = torch.tensor(x_train_pca)
    x_test_pca = torch.tensor(x_test_pca)

    model = KNN(x_train_pca, y_train, K=cfg.K)
    y_pred = model.predict(x_test_pca)
    save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
    supervised_score(cfg.output_filename)


if __name__ == "__main__":
    KNN_PCA()
```

resnet_modified.py

```python
import numpy as np
import torch
import os
import torchvision.models as models
from torch import nn
from torch.optim import Adam
from tqdm import tqdm
from torch.utils.data import DataLoader, RandomSampler,
   SequentialSampler

import config as cfg
from dataset import SupervisedDataset
from utils import same_seeds, supervised_score, save_result


same_seeds(42)


class Resnet_M(nn.Module):
    def __init__(self):
        super(Resnet_M, self).__init__()
        self.resnet = models.resnet18(pretrained=True)
        self.resnet.fc = nn.Linear(512, 1)  # change the last
           layer to output 1 value

    def forward(self, x):
        x = self.resnet(x)
        return x


def train():
    if not os.path.exists("weights"):
        os.mkdir("weights")

    model = Resnet_M().cuda()

    train_dataset = SupervisedDataset("data/train")
    train_loader = DataLoader(
        train_dataset,
        batch_size=cfg.batch_size,
        sampler=RandomSampler(train_dataset),
    )

    criterion = nn.MSELoss()
    optimizer = Adam(model.parameters(), lr=cfg.learning_rate)

    best_loss = np.inf

    for epoch in range(cfg.num_epochs):
        model.train()
        train_loss = []

        with tqdm(total=len(train_loader), unit="batch") as
           tqdm_bar:
            tqdm_bar.set_description(f"Epoch {epoch+1:02d}/{cfg.
               num_epochs}")
            for data in train_loader:
                images, labels = data
                images = images.float().cuda()
                labels = labels.cuda().view(-1, 1).float()

                output = model(images)

                loss = criterion(output, labels)
                train_loss.append(loss.item())

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                tqdm_bar.set_postfix(
                    loss=f"{sum(train_loss)/len(train_loss):.5f}
```

```
                        ",
                )
                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

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

```python
from torch import nn
from torch.optim import Adam
from tqdm import tqdm
from torch.utils.data import DataLoader, RandomSampler,
    SequentialSampler

import config as cfg
from models.cnn import Resnet
from models.knn import KNN
from models.vae import VAE, loss_vae
from models.fcn import fcn_autoencoder
from dataset import UnsupervisedDataset, SupervisedDataset,
    KNNDataset
from utils import same_seeds, unsupervised_score,
    supervised_score, save_result

same_seeds(42)

model_classes = {
    "cnn": Resnet(),
    "vae": VAE(),
    "fcn": fcn_autoencoder(),
}  # dictionary of models


def train():
    if cfg.model_type == "knn":
        return

    if not os.path.exists("weights"):
        os.mkdir("weights")

    # model
    model = model_classes[cfg.model_type].cuda()

    # dataset & dataloader
    if cfg.model_type == "cnn":
        train_dataset = SupervisedDataset("data/train")
    else:
        train_dataset = UnsupervisedDataset("data/train")
    train_loader = DataLoader(
        train_dataset,
        batch_size=cfg.batch_size,
        sampler=RandomSampler(train_dataset),
    )

    # loss & optimizer
    if cfg.model_type == "cnn":
        criterion = nn.CrossEntropyLoss()
    else:
        criterion = nn.MSELoss()
    optimizer = Adam(model.parameters(), lr=cfg.learning_rate)

    best_loss = np.inf

    for epoch in range(cfg.num_epochs):
        model.train()
        train_loss = []

        with tqdm(total=len(train_loader), unit="batch") as
            tqdm_bar:
            tqdm_bar.set_description(f"Epoch {epoch+1:02d}/{cfg.
                num_epochs}")
            for data in train_loader:
                if cfg.model_type != "cnn":
                    images = data.float().cuda()

                    if cfg.model_type == "fcn":
                        images = images.view(images.size(0), -1)
                else:
                    images, labels = data
                    images = images.float().cuda()
                    labels = labels.cuda()

                output = model(images)

                # different models use different criterion
                if cfg.model_type == "vae":
                    loss = loss_vae(output[0], images, output
                        [1], output[2], criterion)
                elif cfg.model_type == "fcn":
                    loss = criterion(output, images)
                else:
                    loss = criterion(output, labels)

                train_loss.append(loss.item())

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                tqdm_bar.set_postfix(
                    loss=f"{sum(train_loss)/len(train_loss):.5f}
                        ",
                )
                tqdm_bar.update()

        # save best model
        if np.mean(train_loss) < best_loss:
            best_loss = np.mean(train_loss)
            torch.save(model.state_dict(), f"weights/best_{cfg.
                model_type}.pth")


def test():
    if cfg.model_type == "knn":
        x_train, y_train, x_test, y_test = KNNDataset(
            os.path.join("data", "train"), os.path.join("data", "test")
        )
        model = KNN(x_train, y_train, K=cfg.K)
        y_pred = model.predict(x_test)
        save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
    else:
        model = model_classes[cfg.model_type].cuda()

        # load weights
        model.load_state_dict(torch.load(f"weights/best_{cfg.model_type}.pth"))

        # dataset & dataloader
        if cfg.model_type == "cnn":
            test_dataset = SupervisedDataset("data/test", train=False)
        else:
            test_dataset = UnsupervisedDataset("data/test", train=False)
        test_loader = DataLoader(
            test_dataset,
            batch_size=cfg.batch_size,
            sampler=SequentialSampler(test_dataset),
        )

        # loss
        eval_loss = nn.MSELoss(reduction="none")
        model.eval()

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

                if cfg.model_type == "cnn":
                    output = model(img)
                    loss = (output.argmax(1)).int()
                elif cfg.model_type == "fcn":
                    img = img.view(img.size(0), -1)
                    output = model(img, train=False)
                    loss = eval_loss(output, img).sum(1)
                else:
                    output = model(img, train=False)
                    output = output[0]
                    loss = eval_loss(output, img).sum([1, 2, 3])

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

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

        save_result(np.array(labels), anomality.flatten())

    if cfg.model_type != "vae" and cfg.model_type != "fcn":
        supervised_score(cfg.output_filename)
    else:
        unsupervised_score(cfg.output_filename)


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

smote.py

```
import torch
import os
from imblearn.over_sampling import SMOTE
from sklearn.decomposition import PCA

import config as cfg
from models.knn import KNN
from dataset import KNNDataset
from utils import same_seeds, supervised_score, save_result

same_seeds(42)


def smote():
    """
    this function is used to apply SMOTE to oversampling and PCA
        to reduce dimension
    """
    x_train, y_train, x_test, y_test = KNNDataset(
        os.path.join("data", "train"), os.path.join("data", "
            test")
    )

    # use PCA to reduce dimension
    pca = PCA(n_components=2)
    x_train_pca = pca.fit_transform(x_train.view(x_train.size(0)
        , -1))
    x_test_pca = pca.fit_transform(x_test.view(x_test.size(0),
        -1))

    # use SMOTE to oversampling
    sm = SMOTE()
    x_train_pca, y_train = sm.fit_resample(x_train_pca, y_train)

    x_train_pca = torch.tensor(x_train_pca)
    y_train = torch.tensor(y_train)
    x_test_pca = torch.tensor(x_test_pca)

    model = KNN(x_train_pca, y_train, K=cfg.K)
    y_pred = model.predict(x_test_pca)
    save_result(y_test.cpu().numpy(), y_pred.cpu().numpy())
    supervised_score(cfg.output_filename)


if __name__ == "__main__":
    smote()
```

tsne_visualize.py

```
# draw t-SNE
import matplotlib.pyplot as plt
import numpy as np
import torch
from tqdm import tqdm
from torch.utils.data import DataLoader, SequentialSampler
from sklearn.manifold import TSNE

import config as cfg
from models.cnn import Resnet
from dataset import SupervisedDataset
from utils import same_seeds

same_seeds(42)


def tSNE_visualize():
    """
    Visualize the t-SNE of the features from each layer
    """
    model = Resnet().cuda()
    model.load_state_dict(torch.load(f"weights/best_cnn.pth"))
    model.eval()

    print(model)  # print model architecture

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

    def hook(module, input, output):
        """
        hook for extracting features
        """
        temp.append(output.clone().detach())

    handles = []
```

```python
    handles.append(model.resnet.layer1.register_forward_hook(hook))
    handles.append(model.resnet.layer2.register_forward_hook(hook))
    handles.append(model.resnet.layer3.register_forward_hook(hook))
    handles.append(model.resnet.layer4.register_forward_hook(hook))
    handles.append(model.resnet.fc.register_forward_hook(hook))

    features_tsnes = []

    # input
    features = []
    labels = []
    for img, label in tqdm(test_loader):
        with torch.no_grad():
            logits = img.view(img.size()[0], -1)
            labels.extend(label.numpy())
            logits = np.squeeze(logits.cpu().numpy())
            features.extend(logits)
    features = np.array(features)
    features_tsnes.append(TSNE(n_components=2).fit_transform(features))

    # layers 1-4 and output
    for handle in handles:
        features = []
        labels = []
        with torch.no_grad():
            for img, label in tqdm(test_loader):
                temp = []
                _ = model(img.cuda())
                logits = temp[0].view(temp[0].size()[0], -1)
                labels.extend(label.numpy())
                logits = np.squeeze(logits.cpu().numpy())
                features.extend(logits)
        handle.remove()

        features = np.array(features)
        features_tsnes.append(TSNE(n_components=2).fit_transform(features))

    # draw t-SNE
    plt.figure(figsize=(10, 6))
    for i in range(len(features_tsnes)):
        plt.subplot(2, 3, i + 1)
        for label in np.unique(labels):
            plt.scatter(
                features_tsnes[i][labels == label, 0],
                features_tsnes[i][labels == label, 1],
                label=label,
                s=10,
            )
        if i == 0:
            plt.title("Input")
        elif i == 5:
            plt.title("Output")
        else:
            plt.title(f"Layer {i}")
        plt.legend()
    plt.savefig("plot/tsne.png")
    plt.show()


if __name__ == "__main__":
    tSNE_visualize()
```

utils.py

```python
import numpy as np
import torch
import pandas as pd
import matplotlib.pyplot as plt
import config as cfg
from sklearn.metrics import (
    roc_auc_score,
    precision_score,
    recall_score,
    f1_score,
    roc_curve,
)
from sklearn.preprocessing import MinMaxScaler


def same_seeds(myseed):
    """
    Set the random seeds to make the experiments reproducible.
    """
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    np.random.seed(myseed)
    torch.manual_seed(myseed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(myseed)


def save_result(y_test, y_pred):
    """
    Save the results to a csv file.
    """
    result = pd.DataFrame(
        {
            "label": y_test,
            "score": y_pred,
        }
    )
    result = result.sort_values(by="score", ascending=False)
    result.to_csv(cfg.output_filename, index=False)


def unsupervised_score(file_name):
    """
    Calculate the unsupervised evaluation metrics.
    """
    csv_file = file_name
    df = pd.read_csv(csv_file)
    df["label"] = df["label"].astype(int)
    df["score"] = MinMaxScaler().fit_transform(df[["score"]])

    y_true = df["label"]
    y_pred = df["score"]

    roc_auc = roc_auc_score(y_true, y_pred)
    print(f"roc_auc: {roc_auc:.4f}")

    # draw ROC curve
    fpr, tpr, _ = roc_curve(y_true, y_pred)
    plt.figure()
    lw = 2
    plt.plot(
        fpr, tpr, color="darkorange", lw=lw, label="ROC curve (
           area = %0.4f)" % roc_auc
    )
    plt.plot([0, 1], [0, 1], color="navy", lw=lw, linestyle="--"
       )
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC curve")
    plt.legend(loc="lower right")
    plt.savefig(f"plot/{cfg.model_type}_roc.png")
    plt.show()


def supervised_score(file_name, print_result=True):
    """
    Calculate the supervised evaluation metrics.
    """
    csv_file = file_name
    df = pd.read_csv(csv_file)
    y_true = df["label"]
    y_pred = df["score"]

    # precision, recall, F1-score
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)

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