AutoEncoder Model Documentation for Anomaly Detection

1. Introduction

An **AutoEncoder (AE)** is a type of artificial neural network used for unsupervised learning, commonly used for tasks like anomaly detection, dimensionality reduction, or feature learning. In the context of anomaly detection, an AutoEncoder learns to compress and reconstruct input data. When trained on normal data, the model will perform well in reconstructing similar data. For outliers or anomalies, the reconstruction error will be significantly higher, making these data points detectable as anomalies.

This document provides a detailed explanation of the AutoEncoder model used for anomaly detection, including the architecture, logic, and step-by-step code explanation.

Key Concepts

- **AutoEncoder Structure**: The model consists of two parts, an encoder and a decoder.
  - **Encoder**: Compresses the input data into a lower-dimensional latent space.
  - **Decoder**: Reconstructs the input data from the latent space.
- **Anomaly Detection**: The reconstruction error is used to detect anomalies. If a data point has a high reconstruction error, it is flagged as an anomaly.

2. Architecture

The AutoEncoder model follows the **Encoder-Decoder** architecture:

1. **Encoder**: A series of fully connected layers progressively reduce the dimensionality of the input data.
2. **Latent Space**: After the encoder, the data is compressed into a latent representation (a lower-dimensional space).
3. **Decoder**: The decoder attempts to reconstruct the original data from the latent space.
4. **Loss Function**: The model uses Mean Squared Error (MSE) loss, which measures the difference between the input data and the reconstructed output.

Detailed Components

- **Encoder**: A series of layers that apply linear transformations followed by activation functions (ReLU), batch normalization, and dropout layers to prevent overfitting.

- **Decoder**: Similar to the encoder but in reverse order. The final layer of the decoder outputs the same dimensionality as the input.
- **Activation Functions**: ReLU activation is used in both encoder and decoder layers.
- **Loss Function**: Mean Squared Error (MSE) is used to measure the reconstruction loss between the original data and its reconstruction.

3. Explaining the Logic

The model follows the logic outlined below:

1. Data Preprocessing

Before training the model, the data must be preprocessed:

- **Numerical Data**: Directly used in the model.
- **String Data**: Converted into their lengths as a simple encoding strategy. Alternatively, more sophisticated encoding methods could be used.
- **Mixed Data**: Data that combines various features, converted into scores for input to the model.

2. AutoEncoder Training Process

- **Input Data**: The data is passed through the encoder, which compresses it into a latent representation.
- **Latent Space**: The encoder's output (latent space) is passed to the decoder.
- **Reconstruction**: The decoder reconstructs the input data, which is compared to the original data to compute the loss.
- **Loss Calculation**: The reconstruction error is calculated using MSE, and the model is trained to minimize this error.
- **Anomaly Detection**: After training, the reconstruction error is used to detect anomalies. If a data point has a high reconstruction error, it is flagged as an anomaly.

3. Model Training

- **Epoch Loop**: The training is done over multiple epochs (50 in this case). Each epoch involves:
  - Feeding data into the encoder.
  - Passing the latent representation through the decoder.
  - Calculating the reconstruction error.
  - Backpropagating the error to adjust the weights of the model using the Adam optimizer.
- **Optimization**: The model uses the **Adam optimizer** to update the weights based on the calculated gradients.
- **Loss Function**: The model minimizes the reconstruction error using the **Mean Squared Error (MSE)** loss.

4. Saving the Model

After training, the model's learned parameters (weights) are saved to a file using `torch.save()`. This allows for later use in inference or further training.

4. Code Explanation

The code consists of several files and functions that work together to define, train, and save the AutoEncoder model.

4.1 Data Loading

```
def load_data(file_path):
    with open(file_path, 'r') as file:
        data = json.load(file)

    # Example preprocessing steps, modify as needed for your use case
    numeric_data = np.array(data['numeric_data'])
    string_data = np.array([len(s) for s in data['string_data']])  # Example: convert strings to their
lengths
    mixed_data_scores = np.array([item['scores'] for item in data['mixed_data']])

    # Concatenate or otherwise combine your data as needed
    combined_data = np.concatenate([numeric_data, string_data, mixed_data_scores.flatten()])
    combined_data = combined_data.reshape(-1, 1)  # Reshape as needed for your model

    return combined_data
```

Explanation: This function loads the data from a JSON file and processes it into a form suitable for training:

- **Numeric Data**: Loaded directly.
- **String Data**: Converted to lengths.
- **Mixed Data**: Scores are extracted.
- **Preprocessing**: Data is concatenated and reshaped for model input.

4.2 AutoEncoder Model

```
class AutoEncoder(nn.Module):

    def __init__(self, contamination=0.05, epoch_num=50, batch_size=64, feature_size=128,
hidden_neuron_list=[64, 32]):

        super(AutoEncoder, self).__init__()

        self.contamination = contamination

        self.epoch_num = epoch_num

        self.batch_size = batch_size

        self.feature_size = feature_size
        self.hidden_neuron_list = hidden_neuron_list
        # Build the encoder and decoder
        self.encoder = self._build_encoder()
        self.decoder = self._build_decoder()
```

Explanation:

- The model's initialization method takes multiple parameters, including:
  - `contamination`: The expected proportion of anomalies in the dataset.
  - `epoch_num`: Number of epochs for training.
  - `batch_size`: Size of data batches during training.
  - `feature_size`: Number of features in the input data.
  - `hidden_neuron_list`: Defines the number of neurons in each hidden layer of the encoder and decoder.

4.3 Training the Model

```
def fit(self, X_train):
    criterion = nn.MSELoss()  # Loss function: Mean Squared Error
    optimizer = torch.optim.Adam(self.parameters(), lr=0.001)  # Adam optimizer


    for epoch in range(self.epoch_num):
        self.train()  # Set the model to training mode
        running_loss = 0.0
        for i in range(0, len(X_train), self.batch_size):
            inputs = torch.tensor(X_train[i:i+self.batch_size], dtype=torch.float32)


            # Forward pass
            encoded = self.encoder(inputs)
            decoded = self.decoder(encoded)


            # Compute the loss
            loss = criterion(decoded, inputs)
            # Backward pass

            optimizer.zero_grad()  # Clear the previous gradients

            loss.backward()  # Backpropagate the loss

            optimizer.step()  # Update the model's parameters


            running_loss += loss.item()


        # Print the average loss per epoch

        print(f"Epoch {epoch+1}/{self.epoch_num}, Loss: {running_loss / len(X_train)}")


    print("Training complete")
```

Explanation:

- **Loss Function**: The model uses **Mean Squared Error** (MSE) to measure the reconstruction error.
- **Optimizer**: Adam optimizer is used for training.
- **Training Loop**: For each epoch, data is fed through the encoder and decoder, and the reconstruction error is minimized using backpropagation.

4.4 Saving the Model

```
torch.save(self.state_dict(), 'autoencoder.pth')
```

Explanation: After training, the model’s learned parameters (weights) are saved to the file autoencoder.pth for future use.

5. Expected Output

During the training process, the output will look like the following:

```
Epoch 1/50, Loss: 1.6490967273712158

Epoch 2/50, Loss: 1.5856103897094727

Epoch 3/50, Loss: 1.35365891456604

...

Epoch 50/50, Loss: 0.09393051266670227

Model saved to models\autoencoder.pth
```

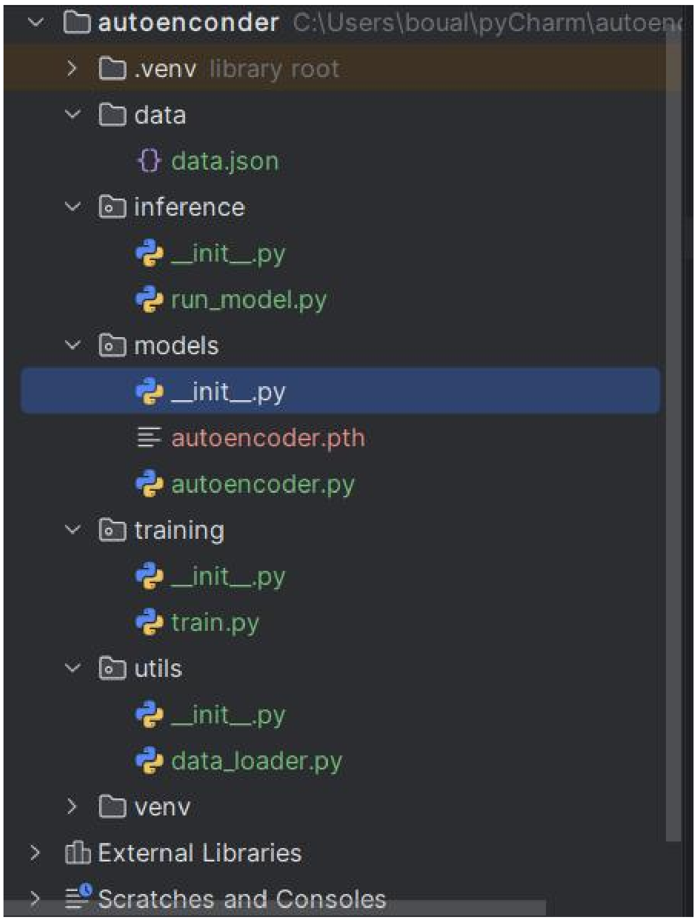
Explanation:

- **Epochs**: The model trains for 50 epochs (as specified), printing the loss at each epoch.
- **Loss**: The loss decreases over time as the model learns to reconstruct the input data more accurately.
- **Model Saved**: After training, the model's weights are saved to `autoencoder.pth`.

6. Conclusion

This AutoEncoder model is designed for **anomaly detection** by learning to compress and reconstruct data. The reconstruction error is used to identify data points that are different from the rest of the dataset. The detailed architecture, training process, and anomaly detection logic are essential to understanding how the model works and how it can be applied to real-world datasets.

SCREENSHOTS AND FULL CODE ;



```
autoenconder  C:\Users\boual\pyCharm\autoenc
  .venv  library root
  data
    data.json
  inference
    __init__.py
    run_model.py
  models
    __init__.py
    autoencoder.pth
    autoencoder.py
  training
    __init__.py
    train.py
  utils
    __init__.py
    data_loader.py
  venv
External Libraries
Scratches and Consoles
```

Run_model.py:

```
import os

import torch

import numpy as np

from models.autoencoder import AutoEncoderModel

from utils.data_loader import load_data


# Load data

data_path = os.path.join('data', 'data.json')

X_test = load_data(data_path)


# Load the trained model

model_path = os.path.join('models', 'autoencoder.pth')

feature_size = X_test.shape[1]

model = AutoEncoderModel(feature_size)

model.load_state_dict(torch.load(model_path))

model.eval()


# Predict anomalies

X_test_tensor = torch.from_numpy(X_test).float()

with torch.no_grad():

    reconstructions = model(X_test_tensor)

scores = np.mean((X_test_tensor.numpy() - reconstructions.numpy()) ** 2, axis=1)


# Assuming contamination is 0.05 as in training

threshold = np.percentile(scores, 95)

predictions = (scores > threshold).astype(int)
```

```
print("Anomaly scores:\n", scores)

print("Predictions:\n", predictions)

print("Threshold:\n", threshold)
```

Autoenconder.py

```
import torch

from torch import nn, optim

from torch.utils.data import DataLoader, Dataset

import numpy as np

from pyod.models.base_dl import BaseDeepLearningDetector

from pyod.utils.stat_models import pairwise_distances_no_broadcast

from pyod.utils.torch_utility import LinearBlock




class LargeDataset(Dataset):

    def __init__(self, data):

        self.data = data


    def __len__(self):

        return len(self.data)


    def __getitem__(self, idx):

        return self.data[idx]
```

```
class AutoEncoder(BaseDeepLearningDetector):

    def __init__(self, contamination=0.1, preprocessing=True, lr=1e-3, epoch_num=10,
batch_size=32,

                 optimizer_name='adam', device=None, random_state=42, use_compile=False,
compile_mode='default',

                 verbose=1, optimizer_params=None, hidden_neuron_list=None,

                 hidden_activation_name='relu', batch_norm=True, dropout_rate=0.2):

        if optimizer_params is None:

            optimizer_params = {'weight_decay': 1e-5}

        if hidden_neuron_list is None:

            hidden_neuron_list = [64, 32]

        super(AutoEncoder, self).__init__(contamination=contamination,
preprocessing=preprocessing, lr=lr,

                                          epoch_num=epoch_num,

                                          batch_size=batch_size, optimizer_name=optimizer_name,
criterion_name='mse',

                                          device=device, random_state=random_state, use_compile=use_compile,

                                          compile_mode=compile_mode, verbose=verbose,
optimizer_params=optimizer_params)

        self.hidden_neuron_list = hidden_neuron_list

        self.hidden_activation_name = hidden_activation_name

        self.batch_norm = batch_norm

        self.dropout_rate = dropout_rate

        self.model = None

        self.optimizer = None

        self.criterion = None

        self.decision_scores_ = None

        self.feature_size = None  # Initialize feature_size to None
    def build_model(self):

        self.model = AutoEncoderModel(self.feature_size,
hidden_neuron_list=self.hidden_neuron_list,

                                    hidden_activation_name=self.hidden_activation_name,
batch_norm=self.batch_norm,

                                    dropout_rate=self.dropout_rate)



    def training_forward(self, batch_data):

        x = batch_data

        x = x.to(self.device)

        self.optimizer.zero_grad()

        x_recon = self.model(x)

        loss = self.criterion(x_recon, x)

        loss.backward()

        self.optimizer.step()

        return loss.item()



    def evaluating_forward(self, batch_data):

        x = batch_data

        x_gpu = x.to(self.device)

        x_recon = self.model(x_gpu)

        score = pairwise_distances_no_broadcast(x.numpy(), x_recon.cpu().numpy())

        return score



    def fit(self, X, y=None):

        X = self._preprocess_data(X)

        self._set_n_classes(y)
        self.device = self._get_device()

        # Set feature_size based on input data dimensions
        self.feature_size = X.shape[1]  # Assuming X is a 2D array with shape (num_samples,
num_features)

        self.build_model()
        self.model.to(self.device)
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr, **self.optimizer_params)
        self.criterion = nn.MSELoss()
        dataset = LargeDataset(torch.from_numpy(X).float())
        train_loader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True,
num_workers=0)

        for epoch in range(self.epoch_num):
            self.model.train()
            train_loss = 0.0
            for batch in train_loader:
                batch_data = batch.to(self.device)
                loss = self.training_forward(batch_data)
                train_loss += loss
            if self.verbose:
                print(f'Epoch {epoch + 1}/{self.epoch_num}, Loss: {train_loss / len(train_loader)}')
        self.model.eval()
        with torch.no_grad():
            X_tensor = torch.from_numpy(X).float().to(self.device)
            reconstructions = self.model(X_tensor)
            self.decision_scores_ = pairwise_distances_no_broadcast(X, reconstructions.cpu().numpy())

        self._process_decision_scores()


    def decision_function(self, X):

        X = self._preprocess_data(X)

        X_tensor = torch.from_numpy(X).float().to(self.device)

        self.model.eval()

        with torch.no_grad():

            reconstructions = self.model(X_tensor)

            scores = pairwise_distances_no_broadcast(X, reconstructions.cpu().numpy())

        return scores


    def _preprocess_data(self, X):

        # Assuming you need to normalize or scale your data

        # Replace with actual preprocessing steps as needed

        return (X - X.mean(axis=0)) / X.std(axis=0)


    def _get_device(self):

        return torch.device("cuda" if torch.cuda.is_available() else "cpu")




class AutoEncoderModel(nn.Module):

    def __init__(self, feature_size, hidden_neuron_list=None, hidden_activation_name='relu',
batch_norm=True,

                 dropout_rate=0.2):

        if hidden_neuron_list is None:

            hidden_neuron_list = [64, 32]
        super(AutoEncoderModel, self).__init__()

        self.feature_size = feature_size

        self.hidden_neuron_list = hidden_neuron_list

        self.hidden_activation_name = hidden_activation_name

        self.batch_norm = batch_norm

        self.dropout_rate = dropout_rate

        self.encoder = self._build_encoder()

        self.decoder = self._build_decoder()



    def _build_encoder(self):

        encoder_layers = []

        last_neuron_size = self.feature_size

        for neuron_size in self.hidden_neuron_list:

            encoder_layers.append(

                LinearBlock(last_neuron_size, neuron_size,
activation_name=self.hidden_activation_name,

                            batch_norm=self.batch_norm, dropout_rate=self.dropout_rate))

            last_neuron_size = neuron_size

        return nn.Sequential(*encoder_layers)



    def _build_decoder(self):

        decoder_layers = []

        last_neuron_size = self.hidden_neuron_list[-1]

        for neuron_size in reversed(self.hidden_neuron_list[:-1]):

            decoder_layers.append(

                LinearBlock(last_neuron_size, neuron_size,
activation_name=self.hidden_activation_name,
                        batch_norm=self.batch_norm, dropout_rate=self.dropout_rate))
            last_neuron_size = neuron_size
        decoder_layers.append(nn.Linear(last_neuron_size, self.feature_size))
        return nn.Sequential(*decoder_layers)

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

Train. py

```
import os
import torch
from models.autoencoder import AutoEncoder
from utils.data_loader import load_data

if __name__ == "__main__":
    # Load data
    data_path = os.path.join('data', 'data.json')
    X_train = load_data(data_path)

    # Initialize and train the model
    clf = AutoEncoder(contamination=0.05, epoch_num=50, batch_size=64)
    clf.fit(X_train)

    # Save the model
    model_save_path = os.path.join('models', 'autoencoder.pth')

    torch.save(clf.model.state_dict(), model_save_path)

    print(f"Model saved to {model_save_path}")
```

Data_loader.py

```
import json

import numpy as np

import os

import torch

from models.autoencoder import AutoEncoder


# Data loading function

def load_data(file_path):

    with open(file_path, 'r') as file:

        data = json.load(file)


    # Example preprocessing steps, modify as needed for your use case

    numeric_data = np.array(data['numeric_data'])

    string_data = np.array([len(s) for s in data['string_data']])  # Example: convert strings to their lengths

    mixed_data_scores = np.array([item['scores'] for item in data['mixed_data']])


    # Concatenate or otherwise combine your data as needed

    combined_data = np.concatenate([numeric_data, string_data, mixed_data_scores.flatten()])

    combined_data = combined_data.reshape(-1, 1)  # Reshape as needed for your model


    return combined_data
```

```
# Main script

if __name__ == "__main__":

    # Load data

    data_path = os.path.join('data', 'data.json')

    X_train = load_data(data_path)


    # Initialize and train the model

    clf = AutoEncoder(contamination=0.05, epoch_num=50, batch_size=64)

    clf.fit(X_train)


    # Save the model

    model_save_path = os.path.join('models', 'autoencoder.pth')

    torch.save(clf.model.state_dict(), model_save_path)

    print(f"Model saved to {model_save_path}")
```