Contractive Autoencoder (CAE)

Overview

Contractive Autoencoders (CAE) are a variation of autoencoders that add a penalty term to the loss function to enforce a more robust representation. The key difference in CAE is the introduction of a **Jacobian penalty**, which regularizes the encoder's output with respect to its input, making the learned representations less sensitive to small changes in the input.

In this implementation, we utilize PyTorch to define and train a Contractive Autoencoder for unsupervised feature learning on a synthetic dataset.

Why Use Contractive Autoencoders?

CAE offers several benefits over standard autoencoders:

- **Robust Representations**: The Jacobian penalty helps in learning representations that are more stable to input perturbations, leading to better generalization.
- **Regularization**: The regularization term reduces the risk of overfitting, making it more effective for sparse datasets or smaller datasets.
- **Feature Learning**: It can be used for learning useful features from data in an unsupervised manner.

Prerequisites

- **Python 3.x**: Ensure Python 3.x is installed.
- **PyTorch**: Install PyTorch compatible with your system.
- **scikit-learn**: For dataset generation and preprocessing.

  ```
  pip install scikit-learn
  ```

- **matplotlib**: For plotting results (optional).

  ```
  pip install matplotlib
  ```

Files Included

- `contractive_autoencoder.py` : Contains the implementation of the Contractive Autoencoder model.
- `train.py` : Script to train the Contractive Autoencoder model on the synthetic dataset.
- `plot_results.py` : Script to visualize the training results (optional).

Code Description

1. **Data Generation and Preprocessing**:

```
 from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
import numpy as np

X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)
X_train = (X_train - mean) / std
X_test = (X_test - mean) / std
```

This segment generates a synthetic dataset with 1,000 samples and 20 features. The data is then standardized by subtracting the mean and dividing by the standard deviation.

2. **Contractive Autoencoder Model Definition**:

```
 import torch
import torch.nn as nn
import torch.optim as optim

class ContractiveAutoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(ContractiveAutoencoder, self).__init__()
        self.encoder = nn.Linear(input_dim, hidden_dim)
        self.decoder = nn.Linear(hidden_dim, input_dim)

    def forward(self, x):
        encoded = torch.relu(self.encoder(x))
        decoded = torch.sigmoid(self.decoder(encoded))
        return decoded

    def jacobian_penalty(self, x):
        x.requires_grad_()
        encoded = self.encoder(x)
        jacobian = torch.autograd.grad(encoded, x, grad_outputs=torch.ones(encoded.
        return torch.sum(jacobian ** 2)
```

The `ContractiveAutoencoder` class defines an encoder-decoder structure. The `jacobian_penalty` method computes the Jacobian penalty, which regularizes the model by penalizing large changes in the encoded representation with respect to the input.

3. **Model Training**:

```
 model = ContractiveAutoencoder(input_dim=X_train.shape[1], hidden_dim=8)
optimizer = optim.Adam(model.parameters(), lr=0.001)
loss_fn = nn.MSELoss()

for epoch in range(50):
    optimizer.zero_grad()
    y_pred = model(torch.tensor(X_train, dtype=torch.float32))
    reconstruction_loss = loss_fn(y_pred, torch.tensor(X_train, dtype=torch.float32
    jacobian_loss = model.jacobian_penalty(torch.tensor(X_train, dtype=torch.float3
    loss = reconstruction_loss + 0.01 * jacobian_loss
    loss.backward()
    optimizer.step()
```

```
    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/50], Loss: {loss.item():.4f}")
```

This block trains the model using Adam optimizer and Mean Squared Error loss. The Jacobian penalty term is added to the reconstruction loss to improve robustness. The loss is computed at every epoch, and the optimizer updates the model's weights accordingly.

4. **Model Evaluation** (optional):

```
from sklearn.metrics import mean_squared_error

y_pred = model(torch.tensor(X_test, dtype=torch.float32))
mse = mean_squared_error(X_test, y_pred.detach().numpy())
print(f"Test MSE: {mse:.4f}")
```

After training, the model's performance is evaluated on the test set by computing the Mean Squared Error (MSE) between the true test data and the model’s reconstructed output.

5. **Visualization** (optional):

```
import matplotlib.pyplot as plt

# Plot reconstructed data vs. original data
plt.plot(X_test[0], label='Original')
plt.plot(y_pred[0].detach().numpy(), label='Reconstructed')
plt.legend()
plt.title('Original vs Reconstructed Data')
plt.show()
```

This block visualizes the original and reconstructed data for a sample from the test set, providing insight into the model’s ability to reconstruct inputs.

Expected Outputs

- **Training Progress**: The console will display the loss at every 10th epoch, indicating the model's convergence.
- **Test Performance**: After training, the MSE on the test set will be printed, reflecting the model’s generalization capability.
- **Visualization**: A plot comparing the original and reconstructed data will be displayed, helping to assess the quality of the learned representations.

Use Cases

- **Unsupervised Feature Learning**: Learn compact representations of data without labeled information.
- **Dimensionality Reduction**: Contractive Autoencoders can be used for reducing the dimensionality of high-dimensional data.
- **Noise Robustness**: The Jacobian penalty helps the model generalize better by making it more robust to small variations in the input.

Future Enhancements

1. **Hyperparameter Tuning**: Experiment with different architectures, such as deeper networks or variational autoencoders, to further enhance model performance.
2. **Advanced Regularization**: Explore other regularization techniques like dropout, L2 regularization, or adversarial training to improve robustness.
3. **Data Augmentation**: Introduce techniques like dropout or noise injection to the training data to make the model more robust to variations in input data.
4. **Real-World Applications**: Test the model on real-world datasets, such as image or text data, to assess its effectiveness in practical applications.
5. **Visualization of Latent Space**: Use t-SNE or PCA to visualize the learned latent space and the separability of different data points.

References

- Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive Auto-Encoders: Explicit Invariance During Feature Extraction. *Proceedings of the 28th International Conference on Machine Learning (ICML)*. Link.
- Bengio, Y., et al. (2013). Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*. Link.

Feel free to adapt or expand upon any sections as needed!