

Paper URL: <https://paperswithcode.com/paper/deep-anomaly-detection-with-deviation>

Dataset: Kaggle-Credit Card Fraud Dataset

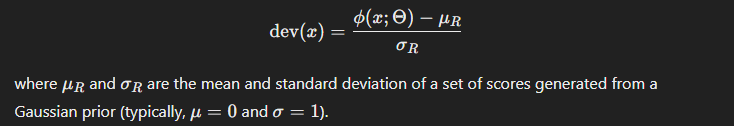
Task: Fraud Detection

Paper concept summary and method explanation:

**DevNet Architecture for Anomaly Detection**

**Key Ideas**

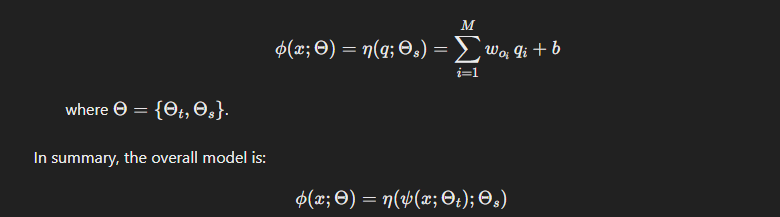
* **End-to-End Anomaly Score Learning:**  
  Instead of first learning a representation and then applying a separate anomaly detector, DevNet directly learns a function that maps an input data object xto a scalar anomaly score ϕ(x;Θ).
* **Leveraging a Few Labeled Anomalies:**  
  The framework is designed to use a very small number of labeled anomalies (even a few dozen) together with a large pool of unlabeled data (assumed to be normal) to guide the training.
* **Deviation Loss:**  
  DevNet employs a novel loss function—called deviation loss—that encourages the anomaly scores of anomalous instances to deviate (in the upper tail) significantly from a reference score.  
  The deviation is computed as:

  
The loss function is then defined (using a contrastive-like formulation) to penalize normal examples for deviating from μR​ while forcing the anomaly scores of anomalies to exceed a threshold (e.g., using a margin a).

**Network Architecture**

DevNet is typically implemented as a feed-forward neural network. The overall mapping is decomposed into two components:

1. **Feature Representation Learner ψ(⋅;Θt):**  
   This part of the network extracts features from the input. For tabular data, this could be a simple multilayer perceptron (MLP).  
   For example, you might use one or more hidden layers (the paper reports good performance with one hidden layer with 20 units, though variants with more layers were also tested).
2. **Anomaly Scoring Function η(⋅;Θs):**  
   A final linear layer takes the latent representation q=ψ(x;Θt) and computes a scalar anomaly score:

This design allows the network to be trained end-to-end with the deviation loss.

**Training with Deviation Loss**

**Deviation Loss Details**

* **Reference Score Generation:**  
  A reference anomaly score μR ​ is computed as the mean of anomaly scores drawn from a Gaussian prior N(μ,σ2) (typically, μ=0 and σ=1). This score acts as a baseline for “normal” behavior.
* **Loss Function:**  
  **A screenshot of a black screen

  AI-generated content may be incorrect.Training Strategy**
* **Data Composition:**  
  You use the few labeled anomalies (set K) and treat the remaining unlabeled data (set U) as normal.
* **Mini-batch Sampling:**  
  Each mini-batch is typically constructed by sampling an equal number of examples from K and U.
* **Optimization:**  
  The model is trained using stochastic gradient descent (or an optimizer like RMSprop) over multiple epochs.

**Integrating DevNet into an AutoML Pipeline**

To incorporate DevNet into your AutoML framework, you would:

1. **Implement the Model:**
   * **Architecture:** Build the network in your favorite deep learning framework (e.g., Keras, PyTorch). For instance, for tabular data, start with an MLP that has one hidden layer (or experiment with deeper architectures if you have enough labeled data).
   * **Custom Loss Function:** Define the deviation loss as described above. In Keras, you can write a custom loss function that takes the anomaly score ϕ(x;Θ) along with the computed μR and σR​ (generated from the Gaussian prior).
2. **Wrap the Model as an Estimator:**
   * If your AutoML system accepts custom models (e.g., via a scikit-learn wrapper), you can wrap the Keras model using keras.wrappers.scikit\_learn.KerasRegressor or a similar tool.
   * This lets your AutoML pipeline search over hyperparameters (e.g., number of hidden units, learning rate, margin a).
3. **Data Preparation:**
   * **Labeling:** Ensure you have a small set of labeled anomalies; treat the rest of your training data as normal.
   * **Preprocessing:** Standardize or normalize your input features as needed.
4. **Training and Evaluation:**
   * Use standard AutoML procedures (e.g., cross-validation, grid search, or Bayesian optimization) to tune the model’s hyperparameters.
   * Evaluate performance using metrics suited for anomaly detection (e.g., AUC-ROC and AUC-PR).

**Example Pseudocode (Using Keras)**

Below is a simplified pseudocode snippet to illustrate the implementation:

python

Copy

import numpy as np

from keras.models import Sequential

from keras.layers import Dense

from keras.optimizers import RMSprop

import keras.backend as K

# Define the custom deviation loss function

def deviation\_loss(a=5, mu=0.0, sigma=1.0, l=5000):

# For each batch, generate l samples from N(mu, sigma^2) and compute reference stats

def loss(y\_true, y\_pred):

# Generate l random samples from Gaussian prior

r\_samples = K.random\_normal(shape=(l,), mean=mu, stddev=sigma)

mu\_R = K.mean(r\_samples)

sigma\_R = K.std(r\_samples) + K.epsilon()

# Compute deviation: (score - reference) / std deviation

dev = (y\_pred - mu\_R) / sigma\_R

# y\_true is 1 for anomalies and 0 for normal

loss\_normal = (1 - y\_true) \* K.abs(dev)

loss\_anomaly = y\_true \* K.maximum(0.0, a - dev)

return K.mean(loss\_normal + loss\_anomaly)

return loss

# Define a simple MLP architecture

def build\_devnet(input\_dim):

model = Sequential()

model.add(Dense(20, activation='relu', input\_dim=input\_dim))

# The output layer produces a scalar anomaly score

model.add(Dense(1, activation='linear'))

model.compile(optimizer=RMSprop(lr=0.001), loss=deviation\_loss(a=5))

return model

# Example usage (assume X\_train is your training data and y\_train are labels: 1 for anomaly, 0 for normal)

input\_dim = X\_train.shape[1]

model = build\_devnet(input\_dim)

model.fit(X\_train, y\_train, epochs=50, batch\_size=32)

*Note:*

* The loss function above is a simplified version to illustrate the idea. In practice, you might need to adjust how you generate and integrate the reference score (µ\_R and σ\_R) within each mini-batch.
* The hyperparameters (e.g., number of epochs, batch size, number of hidden units) should be tuned as part of your AutoML process.

**Summary**

* **Architecture:** DevNet is a deep anomaly detection model that uses an MLP (or other suitable network) to learn a scalar anomaly score directly from input data.
* **Training:** The network is trained with a custom deviation loss that leverages a Gaussian prior to enforce that the anomaly scores of known anomalies are significantly higher than those of normal data.
* **Integration:** To include DevNet in your AutoML pipeline, implement the network with a custom loss function, wrap it as a scikit-learn estimator (if needed), and tune its hyperparameters using your AutoML framework.

This approach allows you to directly optimize anomaly scores in an end-to-end manner and efficiently incorporate even a small number of labeled anomalies into your detection system.