OTDD - OOD Report

H Anantha Krishnan[*]

July 2025

Abstract

An OOD detection framework has been implemented using the Optimal Transport Dataset Distance (OTDD) approach on the HisarMod2019.A dataset. The task is to detect OOD samples (here, analog modulation class) by calibrating a threshold between feature representations of in distribution (ID) and test datasets. A 4 layer CNN is trained to classify known modulation families (FSK, PAM, PSK, QAM). Features are extracted from the penultimate layer of the trained model, and OTDD is used to compute a distance metric between batches of test features and a cached reference set from the training data. The OTDD computation integrates both feature level and label level information by solving an optimal transport problem, the basis of OTDD. A distance threshold is calibrated(also tuned) using a subset of ID and OOD data, beyond which samples are classified as OOD. The OOD detection strategy is then evaluated on a mixed test set for reviewing the efficacy of the method.

1 Detailed Analysis of the code and working

The code can be partitioned into the following categories:

- class HisarModDataset(Dataset):
- class HisarModCNN(nn.Module):
- class FeatureExtractor(nn.Module):
- def prepare_ otdd_datasets():
- def train_with_progress(model, train_loader, val_loader, criterion, optimizer, device, epochs=10):
- def validate(model, loader, criterion, device):
- class OTDDOODDetector:
- def evaluate_mixed_test_set(detector, test_loader, batch_size=100):

[*]Corresponding author: h.ananthakrishnan2004@gmail.com. **IIT Hyderabad**

- def main():

I will now attempt to explain the contents of each of these classes/definitions along with their corresponding code snippets:

1.1 class HisarModDataset(Dataset)

```
class HisarModDataset(Dataset):
    #Data, labels converted to pytorch tensors if numpy arrays, no
        transformation applied
    def __init__(self, data, labels, transform=None):
        self.data = torch.from_numpy(data).float() if isinstance(
            data, np.ndarray) else data.float()
        self.labels = torch.from_numpy(labels).long() if isinstance
            (labels, np.ndarray) else labels.long()
        self.targets = self.labels
        self.transform = transform
    #returns length of dataset
    def __len__(self): return len(self.data)
    #returns sample at index idx, applies transformation if
        specified, return a tuple of (data, label)
    def __getitem__(self, idx):
        x, y = self.data[idx].clone(), self.labels[idx].clone()
        if self.transform: x = self.transform(x)
        return x, y
```

This class preprocesses the data by wrapping the modulation data and labels into a PyTorch compatible dataset format. We are not applying any transform on the data, e.g.: Noise injection. Getitem returns the idx-th sample tuple (input and label).

1.2 class HisarModCNN(nn.Module):

```
class HisarModCNN(nn.Module):
    def __init__(self, num_classes=4):
        super().__init__()
        self.noise = nn.Identity()
        self.conv1 = nn.Conv2d(1, 256, kernel_size=(1,3), padding
            =(0,1))
        self.pool1 = nn.MaxPool2d((1,2))
        self.dropout1 = nn.Dropout(0.5)
        self.conv2 = nn.Conv2d(256, 128, kernel_size=(1,3), padding
            =(0,1))
        self.pool2 = nn.MaxPool2d((1,2))
        self.dropout2 = nn.Dropout(0.5)
        self.conv3 = nn.Conv2d(128, 64, kernel_size=(1,3), padding
            =(0,1))
        self.pool3 = nn.MaxPool2d((1,2))
        self.dropout3 = nn.Dropout(0.5)
        self.conv4 = nn.Conv2d(64, 64, kernel_size=(1,3), padding
            =(0,1))
        self.pool4 = nn.MaxPool2d((1,2))
        self.dropout4 = nn.Dropout(0.5)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(8192, 128)
        self.fc2 = nn.Linear(128, num_classes)
    def forward(self, x):
        x = x.unsqueeze(1)
        x = self.noise(x)
        x = F.relu(self.conv1(x)); x = self.pool1(x); x = self.
            dropout1(x)
        x = F.relu(self.conv2(x)); x = self.pool2(x); x = self.
            dropout2(x)
        x = F.relu(self.conv3(x)); x = self.pool3(x); x = self.
            dropout3(x)
        x = F.relu(self.conv4(x)); x = self.pool4(x); x = self.
            dropout4(x)
        x = self.flatten(x); x = F.relu(self.fc1(x)); return self.
            fc2(x)
    def get_features(self, x):
        x = x.unsqueeze(1); x = self.noise(x)
        x = F.relu(self.conv1(x)); x = self.pool1(x); x = self.
            dropout1(x)
        x = F.relu(self.conv2(x)); x = self.pool2(x); x = self.
            dropout2(x)
        x = F.relu(self.conv3(x)); x = self.pool3(x); x = self.
            dropout3(x)
        x = F.relu(self.conv4(x)); x = self.pool4(x); x = self.
            dropout4(x)
        x = self.flatten(x); x = F.relu(self.fc1(x)); return x
```

This class defines a deep CNN for modulation classification. It includes four convolutional layers with ReLU activations, max pooling, dropout regularization, and two fully connected layers.
The model outputs class logits and also provides an intermediate feature extractor for OOD detection.
No noise is applied on the network(Identity).
For the first convolutional layer -

1. Input channels = 1
2. Output channels = 256 filters
3. Kernel slides horizontally over the signal.

Also, Pooling reduces dimensionality and dropout randomly drops 50% of activations for regularization. This pattern repeats 4 times, each with smaller number of filters ($256 \rightarrow 128 \rightarrow 64 \rightarrow 64$), MaxPooling halves the width every time and also there's dropout .

We now flatten the tensor in 2D, which is initially of size (batch_size, 8192). Then it is passed on to fully connected layers 1 and 2 where the final FC layer is to output logits for each class (size=4). The model is then forward passed, which gives us the logits.

We use get_features to extract the underlying data features from fc1 (fully connected layer 1). Note that this process is done to perform OTDD on a feature space with these features as points in that space.
This is preferred to fc2 as it loses a lot more features than fc1 (128>4), which might lead to a poor estimate during OOD.
The entire class returns the logits and features.

1.3 class FeatureExtractor(nn.Module):

```
class FeatureExtractor(nn.Module):
    def __init__(self, base_model):
        super().__init__()
        self.base_model = base_model
    def forward(self, x):
        with torch.no_grad():
            return self.base_model.get_features(x)
```

This wrapper class extracts intermediate feature representations from a trained `HisarModCNN` model. It bypasses gradient computation and returns activations from the penultimate layer for OTDD based OOD detection. It stores the model so we can reuse its internal layers for feature extraction.

1.4 def prepare_ otdd_datasets():

```
def prepare_otdd_datasets():
    """
    Prepares datasets for OTDD-based OOD detection.

    - Training: non-analog only (FSK, PAM, PSK, QAM)
    - Validation: non-analog only
    - Test: remaining non-analog + all analog (OOD)
    - OOD: analog only
    - Threshold: 10% training + 10% OOD
    """
    np.random.seed(42)

    # Load data and map to families
    data = np.load('train_data_reshaped_1024.npy')
    labels = np.load('train_labels_continuous.npy')
    family_labels = np.array([family_map[label] for label in labels
        ])

    # Boolean masks
    in_dist_mask = family_labels != 0  # FSK, PAM, PSK, QAM
    ood_mask = family_labels == 0      # Analog only

    # Filter and remap in distribution labels (0 to 3)
    in_dist_data = data[in_dist_mask]
    in_dist_labels = family_labels[in_dist_mask] - 1

    ood_data = data[ood_mask]
    ood_labels = family_labels[ood_mask]  # Remain 0
```

```
# === Split in-distribution: 60% train, 20% val, 20% test ===
total_in = len(in_dist_data)
indices = np.random.permutation(total_in)

n_train = int(0.6 * total_in)
n_val = int(0.2 * total_in)

train_idx = indices[:n_train]
val_idx = indices[n_train:n_train + n_val]
test_in_idx = indices[n_train + n_val:]

train_data, train_labels = in_dist_data[train_idx],
    in_dist_labels[train_idx]
val_data, val_labels = in_dist_data[val_idx], in_dist_labels[
    val_idx]
test_in_data, test_in_labels = in_dist_data[test_in_idx],
    in_dist_labels[test_in_idx]

# === Final test set: remaining in-dist + all OOD (analog) ===
test_data = np.concatenate([test_in_data, ood_data])
test_labels = np.concatenate([test_in_labels, ood_labels])

# === Threshold set: 10% train + 10% OOD ===
#Enforce 10% ood length on 10% of train length, not clear if
    otdd needs same length
n_thresh = int(0.1 * len(ood_data))
thresh_train_idx = np.random.choice(len(train_data), n_thresh,
    replace=False)
thresh_ood_idx = np.random.choice(len(ood_data), n_thresh,
    replace=False)

thresh_id_data = train_data[thresh_train_idx]
thresh_id_labels = train_labels[thresh_train_idx]

thresh_ood_data = ood_data[thresh_ood_idx]
thresh_ood_labels = ood_labels[thresh_ood_idx]

print(f"Dataset splits according to OTDD OOD strategy:")
print(f"  Training set (no analog): {len(train_data)}")
print(f"  Validation set (no analog): {len(val_data)}")
print(f"  Test set (mixed): {len(test_data)} (in-dist: {len(
    test_in_data)}, analog: {len(ood_data)})")
print(f"  OOD set (analog only): {len(ood_data)}")
print(f"  Threshold computation set: {len(thresh_id_data) + len
    (thresh_ood_data)} , split as {len(thresh_id_data)} train +
     {len(thresh_ood_data)} OOD")

return (
    HisarModDataset(train_data, train_labels),
    HisarModDataset(val_data, val_labels),
    HisarModDataset(test_data, test_labels),
    HisarModDataset(ood_data, ood_labels),
    HisarModDataset(thresh_id_data, thresh_id_labels),
    HisarModDataset(thresh_ood_data, thresh_ood_labels)
)
```

A point i would like to make here is that im using only the train dataset file and partitioning it for all the purposes.

1. Training: non-analog only (FSK, PAM, PSK, QAM) - 0.6(train_data)
2. Validation: non-analog only - 0.2(train_data)
3. Test: remaining non-analog(0.2(train_data)) + all analog (OOD)
4. OOD: analog only
5. Threshold: 10% training + 10% OOD (Split is equal samples here)

Potential issues I noticed when adopting this split:
Possible slight overlap between Test and Threshold sets which might lead to overfitting but it is small in this case. To circumvent this i can directly use Test_data which i plan on implementing soon. Below is the sample sizes of the above split observed when running the code:

Dataset splits according to OTDD OOD strategy:
Training set (no analog): 228000
Validation set (no analog): 76000
Test set (mixed): 216000 (in-dist: 76000, analog: 140000)
OOD set (analog only): 140000
Threshold computation set: 28000 , split as 14000 train + 14000 OOD

1.5 def train_with_progress(model, train_loader, val_loader, criterion, optimizer, device, epochs=10):

```
def train_with_progress(model, train_loader, val_loader, criterion,
     optimizer, device, epochs=1):
    model.to(device); best_acc = 0.0
    for epoch in range(epochs):
        model.train(); correct, total, loss_sum = 0, 0, 0.0
        for batch_idx, (x, y) in enumerate(train_loader):
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            out = model(x)
            loss = criterion(out, y)
            loss.backward(); optimizer.step()
            loss_sum += loss.item() * x.size(0)
            correct += (out.argmax(1) == y).sum().item(); total +=
                y.size(0)
            if (batch_idx + 1) % 100 == 0:
                current_acc = correct / total
                print(f"[Training] Epoch {epoch+1}, Batch {
                    batch_idx+1}/{len(train_loader)}, Accuracy: {
                    current_acc:.4f}")
        val_loss, val_acc = validate(model, val_loader, criterion,
            device)
        print(f"Epoch {epoch+1}: Final Train Acc: {correct/total:.4
            f}, Val Acc: {val_acc:.4f}")
        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), "
                hisarmod_otdd_best_model.pth")
            print(" Model checkpoint saved")
```

Sends the tuple to appropriate device (Cpu/Gpu) and commences training by:

1. Clearing old gradients per iteration
2. Running forward pass
3. Computing loss
4. Computing gradients via back prop
5. Updates model parameters
6. Computes accuracy metrics
7. Runs on validation set
8. Saves model

1.6 def validate(model, loader, criterion, device):

```
def validate(model, loader, criterion, device):
    model.eval(); total, correct, loss_sum = 0, 0, 0.0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            out = model(x); loss = criterion(out, y)
            loss_sum += loss.item() * x.size(0)
            correct += (out.argmax(1) == y).sum().item(); total +=
                y.size(0)
    return loss_sum/total, correct/total
```

This definition is the 'validation' part used in the previous definition of training. It just loads to tuple onto the device and computes output, loss and appropriate accuracy metric.

1.7 class OTDDOODDetector:

```
class OTDDOODDetector:
    def __init__(self, model, device):
        self.device = device
        self.model = model.to(device)
        self.feature_extractor = FeatureExtractor(model).to(device)
            .eval()
        self.threshold = None
        self.reference_features = None
        self.reference_labels = None
```

```
def cache_reference_features(self, train_loader, num_samples
    =2000):
    """Cache training reference features for consistent OTDD
        comparison"""
    print(f" Caching {num_samples} training reference features
        ...")
    features, labels, count = [], [], 0
    for x, y in train_loader:
        x, y = x.to(self.device), y.to(self.device)
        with torch.no_grad():
            feat = self.feature_extractor(x)
        for i in range(feat.size(0)):
            if count >= num_samples: break
            features.append(feat[i]); labels.append(y[i]);
                count += 1
        if count >= num_samples: break
    self.reference_features = torch.stack(features)
    self.reference_labels = torch.stack(labels)
    print(f" Cached {len(self.reference_features)} reference
        features")
```

```
def extract_features(self, loader, maxsamples=2000):
    features, labels, count = [], [], 0
    for x, y in loader:
        x, y = x.to(self.device), y.to(self.device)
        with torch.no_grad():
            feat = self.feature_extractor(x)
        for i in range(feat.size(0)):
            if count >= maxsamples: break
            features.append(feat[i]); labels.append(y[i]);
                count += 1
        if count >= maxsamples: break
    return torch.stack(features), torch.stack(labels)
```

```
def make_feature_dataset(self, features, labels):
    class FeatureDataset(Dataset):
        def __init__(self, feat, lab):
            self.data = feat.cpu(); self.targets = lab.cpu()
        def __len__(self): return len(self.data)
        def __getitem__(self, idx): return self.data[idx], self
            .targets[idx]
    return FeatureDataset(features, labels)
```

```
def compute_feature_distance(self, features1, labels1,
    features2, labels2):
    d1 = self.make_feature_dataset(features1, labels1)
    d2 = self.make_feature_dataset(features2, labels2)
    l1 = DataLoader(d1, batch_size=32, shuffle=False)
    l2 = DataLoader(d2, batch_size=32, shuffle=False)
    identity_embedding = nn.Identity().to(self.device)
    cost = FeatureCost(
        src_embedding=identity_embedding,
        tgt_embedding=identity_embedding,
        src_dim=(128,), tgt_dim=(128,), p=2, device=self.device
    )
    dist = DatasetDistance(l1, l2, inner_ot_method='exact',
        feature_cost=cost, device=self.device)
    return float(dist.distance(maxsamples=min(len(features1),
        len(features2))))




def calibrate_threshold(self, train_loader, ood_loader,
    maxsamples=5000):
    """
    Calibrate threshold using pure ID vs OOD comparison:
    Reference: subset of training data (ID only)
    Comparison: OOD data only
    """
    print(" Calibrating OTDD OOD detection threshold...")

    # Cache reference features from training set (ID only)
    self.cache_reference_features(train_loader, num_samples=
        maxsamples)

    # Extract features from OOD set only
    print(" Extracting OOD features for threshold calibration
        ...")
    ood_feat, ood_lab = self.extract_features(ood_loader,
        maxsamples)

    # Compute OTDD distance between reference (training ID) and
         pure OOD
    print(" Computing OTDD distance for threshold calibration
        ...")
    distance = self.compute_feature_distance(
        self.reference_features[:maxsamples], self.
            reference_labels[:maxsamples],
        ood_feat, ood_lab
    )

    # Set threshold based on this ID vs OOD distance
    # You can use the computed distance directly or apply a
        factor
    self.threshold = distance * 0.6  # must tune wrt mean and
        std deviation instead of hard-assignment
    # Or keep your hardcoded approach: self.threshold = 50
```

```
        print(f" Threshold Calibration Complete:")
        print(f"   Reference set size (ID): {len(self.
            reference_features)}")
        print(f"   OOD comparison set size: {len(ood_feat)}")
        print(f"   Computed ID vs OOD distance: {distance:.4f}")
        print(f"   Threshold set to: {self.threshold:.4f}")

        return self.threshold




    def detect(self, test_loader, maxsamples=100):
        """Detect OOD by comparing test batch to cached reference
            features"""
        if self.reference_features is None:
            raise ValueError("Must call calibrate_threshold() first
                 to cache reference features")

        test_feat, test_lab = self.extract_features(test_loader,
            maxsamples)
        ref_samples = min(maxsamples, len(self.reference_features))

        distance = self.compute_feature_distance(
            self.reference_features[:ref_samples], self.
                reference_labels[:ref_samples],
            test_feat, test_lab
        )

        return distance > self.threshold, distance
```

There are multiple definitions within the class OTDDOOD Detector

1.7.1 def __init__

Initializes the OOD detector by assigning the model and computation device. It sets up a feature extractor (in eval mode) to extract intermediate representations from the model. It also initializes placeholders for the detection threshold and reference features that will be used for comparing new samples.

1.7.2 def cache_reference_features(self, train_loader, num_samples=2000):

This method extracts and stores features from up to numsamples ID training samples using the feature extractor. These ”reference features” are cached once so they can be reused later during OOD detection without recomputing them.

1.7.3 def extract_features(self, loader, maxsamples=2000):

Calls feature extractor and stores them, this is used for repeated computation of the test features which keep changes for every batch.

1.7.4 def make_feature_dataset(self, features, labels):

This method creates a lightweight PyTorch Dataset from given feature vectors and labels. It defines an internal class FeatureDataset with basic functionality:
–init: Stores the features and labels (moved to CPU).
–len: Returns the dataset size.
–getitem: Returns a (feature, label) pair for a given index.

Purpose: Converts raw feature tensors into a form that can be used with a DataLoader.

1.7.5 compute_feature_distance(self, features1, labels1, features2, labels2)

This method computes the Optimal Transport Dataset Distance (OTDD) between two sets of features:
– Converts both feature-label pairs into datasets using make_feature_dataset.
– Wraps them in DataLoaders.
– Sets up an identity embedding (i.e. no transformation of features).
– Defines a FeatureCost object using L2 distance (p=2).
– Initializes a DatasetDistance object using exact OT.
– Calls .distance() to compute the OTDD between the two datasets, based on a capped sample size.
Also, src and target dims are 128 because this is what fc1 layer in the nn outputs, and this is where we extract the features from.

1.7.6 def calibrate_threshold(self, train_loader, ood_loader, maxsamples=5000):

Calls compute_feature_distance to quanitfy distance between both thresholding datasets, and implements a threshold metric, in this case 0.6* distance. Since the batches are well separable this thresholding lands a 100% accuracy, but ideally this has to be computed based on the mean and variance of the distances across batches till the max samples, this will work better during testing.

1.7.7 def detect(self, test_loader, maxsamples=100):

Returns a boolean whether the batch is OOD or not

1.8 def evaluate_mixed_test_set(detector, test_loader, batch_size=100):

```python
def evaluate_mixed_test_set(detector, test_loader, batch_size=100):
    """
    Evaluate OOD detection on mixed test set (contains both ID and
        OOD samples)
    """
    print("\n===  Mixed Test Set OOD Detection Evaluation ===")
```

```
test_dataset = test_loader.dataset
max_batches = int(np.ceil(len(test_dataset) / batch_size))

all_distances = []
all_predictions = []
all_true_labels = []  # 0 for ID, 1 for OOD (analog)

print(f"Processing {max_batches} batches from mixed test set...
    ")

for i in range(max_batches):
    indices = list(range(i*batch_size, min((i+1)*batch_size,
        len(test_dataset))))
    subset = torch.utils.data.Subset(test_dataset, indices)
    subset_loader = DataLoader(subset, batch_size=len(indices),
         shuffle=False)

    # Get true labels for this batch
    true_batch_labels = []
    for idx in indices:
        sample_label = test_dataset.targets[idx].item()
        true_batch_labels.append(1 if sample_label == 0 else 0)
              # 1 if analog (OOD), 0 if others (ID)

    # Majority vote for batch label (since OTDD works on
        batches)
    batch_is_ood = sum(true_batch_labels) > len(
        true_batch_labels) // 2

    # Run OOD detection
    is_ood_pred, distance = detector.detect(subset_loader,
        maxsamples=len(indices))

    all_distances.append(distance)
    all_predictions.append(1 if is_ood_pred else 0)
    all_true_labels.append(1 if batch_is_ood else 0)

    if (i + 1) % 50 == 0 or i == max_batches - 1:
        current_acc = sum(p == t for p, t in zip(
            all_predictions, all_true_labels)) / len(
            all_predictions)
        print(f"   Processed {i+1}/{max_batches} batches,
            Accuracy: {current_acc:.4f} (D={distance:.2f})")

# Calculate final metrics
all_distances = np.array(all_distances)
all_predictions = np.array(all_predictions)
all_true_labels = np.array(all_true_labels)

# Compute confusion matrix elements
tp = sum((p == 1) and (t == 1) for p, t in zip(all_predictions,
     all_true_labels))
tn = sum((p == 0) and (t == 0) for p, t in zip(all_predictions,
     all_true_labels))
fp = sum((p == 1) and (t == 0) for p, t in zip(all_predictions,
     all_true_labels))
fn = sum((p == 0) and (t == 1) for p, t in zip(all_predictions,
     all_true_labels))

tpr = tp / (tp + fn) if (tp + fn) > 0 else 0
fpr = fp / (fp + tn) if (fp + tn) > 0 else 0
tnr = tn / (tn + fp) if (tn + fp) > 0 else 0
fnr = fn / (fn + tp) if (fn + tp) > 0 else 0
accuracy = (tp + tn) / (tp + tn + fp + fn)

print(f"\n Final Mixed Test Set OOD Detection Performance:")
print(f"   True Positive Rate (OOD Detection): {tpr:.4f} ({tp
    }/{tp + fn})")
print(f"   False Positive Rate (ID as OOD): {fpr:.4f} ({fp}/{fp
     + tn})")
print(f"   True Negative Rate (ID Detection): {tnr:.4f} ({tn}/{
    tn + fp})")
print(f"   False Negative Rate (OOD as ID): {fnr:.4f} ({fn}/{fn
     + tp})")
print(f"   Overall Accuracy: {accuracy:.4f}")
print(f"   Average Distance: {all_distances.mean():.4f} +-{
    all_distances.std():.4f}")
print(f"   Threshold: {detector.threshold:.4f}")

return {
    'tpr': tpr, 'fpr': fpr, 'tnr': tnr, 'fnr': fnr, 'accuracy':
         accuracy,
    'distances': all_distances, 'threshold': detector.threshold
}
```

Evaluates the OOD detection performance of the OTDD detector on a mixed test set that contains both in distribution (ID) and OOD samples.

Some important observations and remarks i would like to make here:

1. Firstly, it is to be noted that the test set was not "shuffled" during the partition, i.e out of 2160 batches, around first 850 are continuously ID and the rest consecutive 1210 are continuously OOD samples. We will see how this impacts results

2. The code flags OOD batch wise through the detect class. Once the distance is computed between the batch and the reference it is compared with the threshold, and a boolean is returned whether it is an OOD or not

3. Why the batch wise assignment, wont it be more practical to do it sample wise? The answer i believe is yes, but there is a potential issue with implementing this. OTDD requires a feature reference space (readily available) and a "test" space. If i were to do a sample wise assignment, i would have to compare a huge distribution of features against one (OTDD is distance between two sets and not a sample wise metric as per my understanding). I therefore tried my best to incorporate batches with the best scenario to showcase its best performance.

4. Another drawback is the issue with first point, there is no mixing of ID and OOD signals during detection within the batches, which is very unlikely

in real life scenarios, this again brings us back to the 3rd point, where it is not feasible to do a sample wise detection, and hence we adopt a batch wise classification

The rest of the code is computing TN(true negative),FN,TP,FP for checking how OTDD gets confused. I have observed that in most cases OOD assigned as ID confusion is very low, and ID assigned as OOD is relatively higher (i had observed this during tuning my threshold function).

1.9 def main():

```
def main():
    device = torch.device("cuda" if torch.cuda.is_available() else
        "cpu")
    print(f"Using device: {device}")

    # Prepare datasets according to OTDD OOD strategy
    train_ds, val_ds, test_ds, ood_ds, threshold_id_ds,
        threshold_ood_ds = prepare_otdd_datasets()

    train_loader = DataLoader(train_ds, 64, shuffle=True)
    val_loader = DataLoader(val_ds, 64)
    test_loader = DataLoader(test_ds, 64)
    ood_loader = DataLoader(ood_ds, 64)
    threshold_id_loader = DataLoader(threshold_id_ds, 64)
    threshold_ood_loader = DataLoader(threshold_ood_ds, 64)

    # Train model (only on non-analog data)
    model = HisarModCNN(num_classes=4)
    optimizer = optim.Adam(model.parameters(), lr=1e-4)
    criterion = nn.CrossEntropyLoss()

    print("=== Training Phase ===")
    train_with_progress(model, train_loader, val_loader, criterion,
         optimizer, device)

    # Load best model
    model.load_state_dict(torch.load("hisarmod_otdd_best_model.pth"
        , map_location=device))
    model.eval().to(device)

    # Test in-distribution performance on validation set
    print("\n=== In-Distribution Validation Performance ===")
    val_loss, val_acc = validate(model, val_loader, criterion,
        device)
    print(f"Validation Accuracy (In-Distribution): {val_acc:.4f}")

    # Initialize and calibrate OTDD OOD detector
    print("\n=== OTDD OOD Detection Setup ===")
    detector = OTDDOODDetector(model, device)

    # UPDATED: Calibrate using pure ID vs OOD comparison
    threshold = detector.calibrate_threshold(threshold_id_loader,
        threshold_ood_loader, maxsamples=5000)
    # Evaluate on mixed test set
    metrics = evaluate_mixed_test_set(detector, test_loader,
        batch_size=100)

    print(f"\n OTDD OOD Detection Complete!")
    print(f" Your OTDD strategy successfully implemented with
        threshold: {threshold:.4f}")

if __name__ == '__main__':
    main()
```

Calls all of the above classes and definitions to ensure everything goes smoothly during runtime, nothing much of significance here, except that we use adam optimizer and cross entropy loss for the nn.
A last point i would like to make is that currently OTDD runs on CPU, not GPU and is slightly slow. I have tried multiple times to resolve the issues but i haven't been able to get it to run. But the training part runs on GPU.
The codes can be run from my repo: *link*