PCA + MLP vs End-to-End CNN for Object Localization

ECE-4563 Machine Learning Project — NYU Tandon

Group member: Yanbo (Bob) Wang, Peijie (Max) Ma

Project Goal

This project investigates **two fundamentally different learning paradigms** for object localization:

- **Classical pipeline:** PCA for dimensionality reduction followed by an **MLP regressor**
- **End-to-end deep learning:** A **CNN regressor** trained directly on images

Both models predict a **bounding box** around a digit placed on a synthetic MNIST canvas.

Task Definition

- **Input:** 64×64 grayscale image
- **Output:** Bounding box parameters
  [(x_c, y_c, w, h)] where all values are **normalized to [0,1]**

Evaluation Metrics

- **Mean Squared Error (MSE)** — regression accuracy
- **Intersection-over-Union (IoU)** — localization quality

A **K-Fold Cross-Validation** procedure is used to rigorously optimize PCA dimensionality.

In [1]:
```
# --- 1) Imports and Environment Setup ---
import os
import random
import numpy as np
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
```

```
from torchvision.datasets import MNIST
from torchvision import transforms
```

1.1 Reproducibility and Device Configuration

To ensure **scientific reproducibility**, we fix all random seeds across:

- Python random
- NumPy
- PyTorch (CPU & GPU)

The code automatically detects and uses a **GPU if available**.

In [2]:
```
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

set_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
print("-" * 50)
```

```
Device: cpu
--------------------------------------------------
```

2. Core Utilities: Bounding Boxes and Data Generation

This section defines:

- Bounding box conversions
- IoU computation
- Synthetic MNIST-on-Canvas generation

Each digit is randomly **scaled and placed** on a 64×64 blank canvas, creating a **controlled localization task.**

In [3]:
```
def bbox_xywh_to_xyxy(b):
    """ (xc,yc,w,h) normalized -> (x1,y1,x2,y2) normalized """
    xc, yc, w, h = b.T
    x1 = xc - w / 2
    y1 = yc - h / 2
    x2 = xc + w / 2
    y2 = yc + h / 2
    return np.stack([x1, y1, x2, y2], axis=1)

"""
```

Intersection-over-Union (IoU)

IoU measures the **overlap ratio** between predicted and ground-truth boxes:
$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}$

It is the **standard localization metric** in computer vision. """

In [4]:
```python
def compute_iou(a, b):
    ax1, ay1, ax2, ay2 = a.T
    bx1, by1, bx2, by2 = b.T

    ix1 = np.maximum(ax1, bx1)
    iy1 = np.maximum(ay1, by1)
    ix2 = np.minimum(ax2, bx2)
    iy2 = np.minimum(ay2, by2)

    iw = np.maximum(0, ix2 - ix1)
    ih = np.maximum(0, iy2 - iy1)
    inter = iw * ih

    area_a = (ax2 - ax1) * (ay2 - ay1)
    area_b = (bx2 - bx1) * (by2 - by1)
    union = area_a + area_b - inter + 1e-9
    return inter / union
```

Synthetic MNIST-on-Canvas Generator

Each sample:

- Randomly rescales an MNIST digit
- Randomly places it on a blank canvas
- Computes its bounding box label

In [5]:
```python
def generate_sample(digit, canvas_size=64):
    canvas = np.zeros((canvas_size, canvas_size), dtype=np.float32)
    scale = np.random.uniform(0.6, 1.0)
    size = int(round(28 * scale))
    size = max(8, size)
    resized = digit[:size, :size]

    top = np.random.randint(0, canvas_size - size)
    left = np.random.randint(0, canvas_size - size)

    canvas[top:top+size, left:left+size] = resized

    xc = (left + size / 2) / canvas_size
    yc = (top + size / 2) / canvas_size
    w = size / canvas_size
    h = size / canvas_size

    return canvas, np.array([xc, yc, w, h], dtype=np.float32)
```

2.1 Dataset Construction

We generate a **large synthetic dataset** using MNIST digits. This avoids annotation noise while preserving realism.

In [6]:
```
def load_dataset(n_samples=8000, data_dir="./data"):
    print(f"Loading/Generating {n_samples} samples...")
    mnist = MNIST(data_dir, train=True, download=True, transform=transforms.ToTe

    X = np.zeros((n_samples, 1, 64, 64), dtype=np.float32)
    y = np.zeros((n_samples, 4), dtype=np.float32)

    for i in range(n_samples):
        img, _ = mnist[random.randint(0, len(mnist) - 1)]
        digit = img.squeeze().numpy().astype(np.float32)
        canvas, bbox = generate_sample(digit)
        X[i, 0] = canvas
        y[i] = bbox
    return X, y
```

3. Dataset Wrappers and Model Definitions

We define **two learning pipelines**:

Model 1 — PCA + MLP

- Flatten image
- Standardize
- Reduce dimension via PCA
- Predict bounding box using MLP

Model 2 — CNN

- End-to-end learning from raw pixels
- Convolutional feature extraction
- Fully-connected regression head

In [7]:
```
class ImageDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X).float()
        self.y = torch.from_numpy(y).float()
    def __len__(self): return len(self.X)
    def __getitem__(self, i): return self.X[i], self.y[i]
class PCADataset(Dataset):
    def __init__(self, Z, y):
        self.Z = torch.from_numpy(Z).float()
        self.y = torch.from_numpy(y).float()
    def __len__(self): return len(self.Z)
    def __getitem__(self, i): return self.Z[i], self.y[i]
```

MLP Regressor Architecture

- Input: PCA features
- Two hidden layers
- Sigmoid output to enforce valid box range

In [8]:
```
class MLPRegressor(nn.Module):
    def __init__(self, in_features=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_features, 256),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 4),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.net(x)
```

CNN Regressor Architecture

- Progressive spatial downsampling
- Adaptive pooling for shape invariance
- Fully connected regression head

In [9]:
```
class CNNRegressor(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 16, 5, stride=2, padding=2),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d((4, 4))
        )
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 4 * 4, 128),
            nn.ReLU(),
            nn.Linear(128, 4),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.fc(self.conv(x))
```

4. Training and Evaluation Logic

Both models:

- Use **Mean Squared Error loss**
- Are evaluated using **IoU on the test set**

In [10]:
```python
def train_one_epoch(model, loader, optimizer, device):
    model.train()
    loss_fn = nn.MSELoss()
    total_loss = 0.0

    for x, y in loader:
        x, y = x.to(device), y.to(device)
        optimizer.zero_grad()
        loss = loss_fn(model(x), y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * x.size(0)

    return total_loss / len(loader.dataset)
```

In [11]:
```python
@torch.no_grad()
def evaluate(model, loader, device):
    model.eval()
    preds, gts = [], []

    for x, y in loader:
        pred = model(x.to(device)).cpu().numpy()
        preds.append(pred)
        gts.append(y.numpy())

    P = np.vstack(preds)
    G = np.vstack(gts)

    mse = float(np.mean((P - G) ** 2))
    iou = float(compute_iou(
        bbox_xywh_to_xyxy(P),
        bbox_xywh_to_xyxy(G)
    ).mean())
    return mse, iou, P, G
```

5. Main Execution

PCA Component Optimization (K-Fold CV) and CNN Training

This section contains the **core experimental pipeline** of the project.

We perform:

- **(A) PCA + MLP pipeline** with rigorous **K-Fold Cross-Validation**
- **(B) End-to-End CNN training** on the same data split

A strict **pool / test split** is enforced to avoid data leakage.

In [12]:
```python
# --- 5) Main Execution: (A) PCA Component Optimization (K-Fold CV) + (B) CNN Tr
```

```
from sklearn.model_selection import KFold
```

5.0 Experimental Configuration

Key hyperparameters used consistently across all experiments:

- **N_SAMPLES**: Total synthetic dataset size
- **PCA_GRID**: Candidate PCA dimensions
- **CV_EPOCHS**: Short training during cross-validation
- **FINAL_EPOCHS**: Full training for final evaluation
- **N_SPLITS**: Number of folds in K-Fold CV

In [13]:
```
# ========================
# 5.0) Configuration
# ========================
N_SAMPLES = 8000
BATCH_SIZE = 128

PCA_GRID = [16, 32, 64, 128, 256, 512]

CV_EPOCHS = 5       # epochs per fold during CV (speed)
FINAL_EPOCHS = 10   # epochs for final models

N_SPLITS = 5        # k-fold splits
```

5.1 Data Loading and Pool/Test Split

We split the dataset into:

- **Pool set (80%)**: used for PCA cross-validation and final training
- **Test set (20%)**: strictly held out for final evaluation

The test set is **never used** during PCA selection.

In [14]:
```
# 5.1) Data Loading + Pool/Test Split

X, y = load_dataset(n_samples=N_SAMPLES)

# Hold-out TEST set (never used during CV)
X_pool, X_test, y_pool, y_test = train_test_split(X, y, test_size=0.2, random_st

print(f"Pool/Test: {len(X_pool)} / {len(X_test)}")

# Pre-flatten pool once for PCA pipeline in CV
X_pool_flat = X_pool.reshape(len(X_pool), -1).astype(np.float32)
y_pool = y_pool.astype(np.float32)
y_test = y_test.astype(np.float32)
```

```
Loading/Generating 8000 samples...
Pool/Test: 6400 / 1600
```

5.2 PCA Component Optimization via K-Fold Cross-Validation (PCA + MLP)

Why K-Fold CV?

- Prevents **overfitting** to a single split
- Provides **mean ± std** performance estimates
- Ensures **no preprocessing leakage**

Important:
StandardScaler and PCA are fit only on the training split of each fold.

In [15]:
```python
# 5.2) (A) PCA Components Optimization via K-Fold CV (PCA+MLP)

def run_cv_for_k(Xflat, y, k, n_splits=5, seed=42):
    """
    Returns mean_mse, std_mse, mean_iou, std_iou over folds for given PCA compon
    No leakage: scaler + PCA are fit ONLY on each fold's training split.
    """
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=seed)

    fold_mses, fold_ious = [], []

    for fold, (tr_idx, va_idx) in enumerate(kf.split(Xflat), start=1):
        Xtr, Xva = Xflat[tr_idx], Xflat[va_idx]
        ytr, yva = y[tr_idx], y[va_idx]

        # Fit preprocessing on fold-train only
        scaler = StandardScaler()
        Xtr_s = scaler.fit_transform(Xtr)
        Xva_s = scaler.transform(Xva)

        pca = PCA(n_components=k, random_state=seed)
        Ztr = pca.fit_transform(Xtr_s).astype(np.float32)
        Zva = pca.transform(Xva_s).astype(np.float32)

        # Train MLP on PCA features
        mlp = MLPRegressor(in_features=k).to(device)
        opt = optim.Adam(mlp.parameters(), lr=1e-3)

        tr_loader = DataLoader(PCADataset(Ztr, ytr), batch_size=BATCH_SIZE, shuf
        va_loader = DataLoader(PCADataset(Zva, yva), batch_size=BATCH_SIZE, shuf

        for _ in range(CV_EPOCHS):
            train_one_epoch(mlp, tr_loader, opt, device)

        mse, iou, _, _ = evaluate(mlp, va_loader, device)
        fold_mses.append(mse)
        fold_ious.append(iou)

    return (float(np.mean(fold_mses)), float(np.std(fold_mses)),
            float(np.mean(fold_ious)), float(np.std(fold_ious)))
```

Cross-Validation Results over PCA_GRID

We evaluate each PCA dimension using:

- **Mean IoU** (primary selection metric)

- **Mean MSE** (regression accuracy)

The best PCA dimension is selected **by highest mean IoU.**

In [16]:
```
print(f"(A) PCA Optimization: {N_SPLITS}-Fold CV over PCA_GRID")
cv_results = []  # (k, mean_mse, std_mse, mean_iou, std_iou)

for k in PCA_GRID:
    mean_mse, std_mse, mean_iou, std_iou = run_cv_for_k(
        X_pool_flat, y_pool, k, n_splits=N_SPLITS, seed=42
    )
    cv_results.append((k, mean_mse, std_mse, mean_iou, std_iou))
    print(f"k={k:4d} | CV IoU={mean_iou:.4f} ± {std_iou:.4f} | CV MSE={mean_mse:
```

```
(A) PCA Optimization: 5-Fold CV over PCA_GRID
k=  16 | CV IoU=0.6346 ± 0.0069 | CV MSE=0.003049 ± 0.000156
k=  32 | CV IoU=0.7525 ± 0.0030 | CV MSE=0.001253 ± 0.000038
k=  64 | CV IoU=0.7561 ± 0.0044 | CV MSE=0.001204 ± 0.000070
k= 128 | CV IoU=0.7471 ± 0.0043 | CV MSE=0.001254 ± 0.000045
k= 256 | CV IoU=0.7400 ± 0.0059 | CV MSE=0.001349 ± 0.000074
k= 512 | CV IoU=0.7324 ± 0.0048 | CV MSE=0.001427 ± 0.000054
```

PCA Dimension Selection

The optimal PCA dimension is chosen as:

$$k^* = \arg\max_k \overline{\text{IoU}}_k$$

In [17]:
```
# Choose best k by mean IoU
best = max(cv_results, key=lambda t: t[3])  # mean_iou
best_k, best_mse, best_mse_std, best_iou, best_iou_std = best


print(f"Best k by CV mean IoU: k={best_k} (IoU={best_iou:.4f} ± {best_iou_std:.4
      f"MSE={best_mse:.6f} ± {best_mse_std:.6f})")
```

```
Best k by CV mean IoU: k=64 (IoU=0.7561 ± 0.0044, MSE=0.001204 ± 0.000070)
```

Cross-Validation Performance Curves

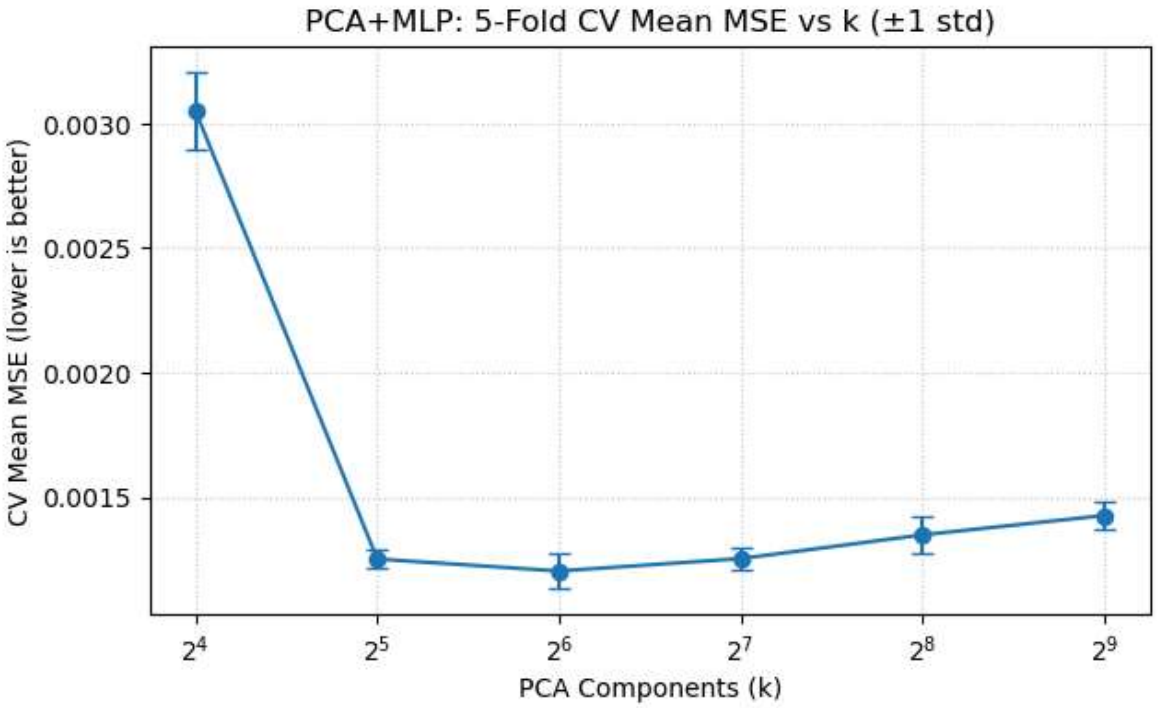
We visualize:

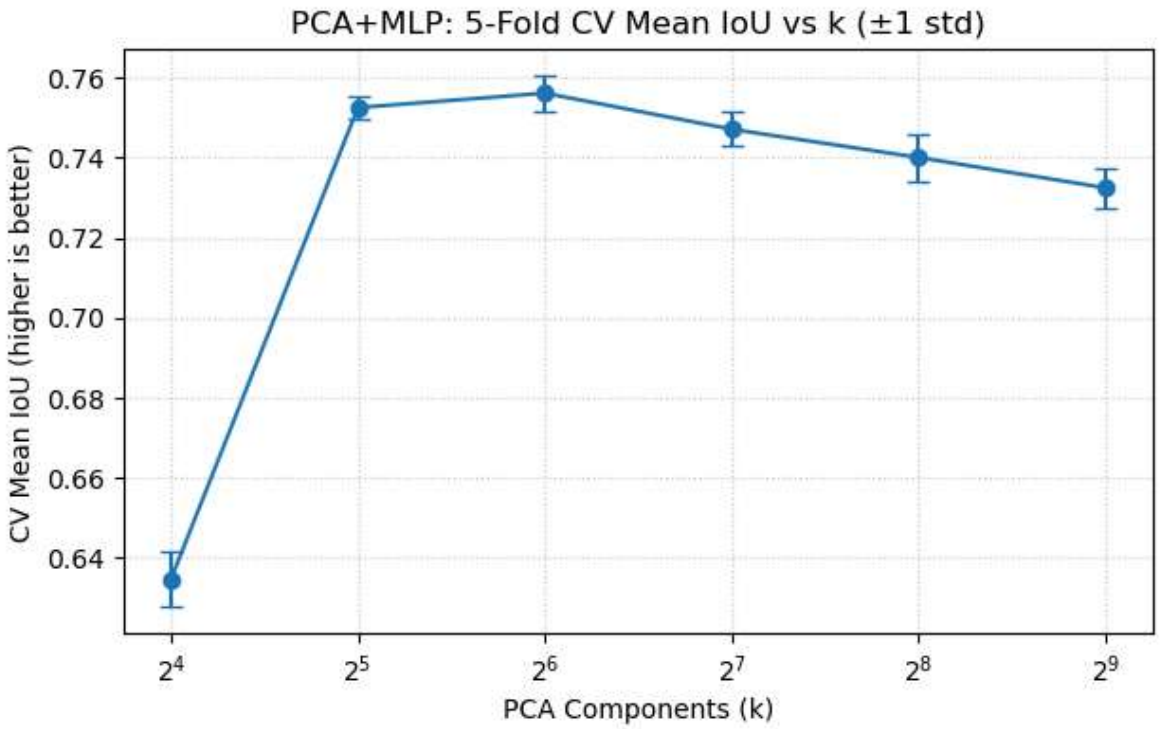
- **MSE vs PCA dimension**
- **IoU vs PCA dimension**

Error bars represent **±1 standard deviation** across folds.

In [18]:
```
# Plot CV curves with variation bars
cv_sorted = sorted(cv_results, key=lambda t: t[0])
ks       = [t[0] for t in cv_sorted]
mses     = [t[1] for t in cv_sorted]
mses_std = [t[2] for t in cv_sorted]
ious     = [t[3] for t in cv_sorted]
ious_std = [t[4] for t in cv_sorted]

plt.figure(figsize=(7, 4))
```

```
plt.errorbar(ks, mses, yerr=mses_std, marker="o", capsize=4)
plt.xlabel("PCA Components (k)")
plt.ylabel("CV Mean MSE (lower is better)")
plt.title(f"PCA+MLP: {N_SPLITS}-Fold CV Mean MSE vs k (±1 std)")
plt.grid(True, linestyle=":", alpha=0.6)
plt.xscale("log", base=2)
plt.show()

plt.figure(figsize=(7, 4))
plt.errorbar(ks, ious, yerr=ious_std, marker="o", capsize=4)
plt.xlabel("PCA Components (k)")
plt.ylabel("CV Mean IoU (higher is better)")
plt.title(f"PCA+MLP: {N_SPLITS}-Fold CV Mean IoU vs k (±1 std)")
plt.grid(True, linestyle=":", alpha=0.6)
plt.xscale("log", base=2)
plt.show()
```





5.3 Final Training and Evaluation: PCA + MLP

Using the **optimal PCA dimension**, we:

- Retrain on the **entire pool set**
- Evaluate once on the **held-out test set**

In [19]:
```
# 5.3) Final Train/Eval for PCA+MLP using best_k (on TEST)

print(f"Final PCA+MLP Training on FULL POOL with k={best_k}")
# Fit scaler + PCA on full pool
scaler_final = StandardScaler()
X_pool_s = scaler_final.fit_transform(X_pool_flat)
X_test_s = scaler_final.transform(X_test.reshape(len(X_test), -1))

pca_final = PCA(n_components=best_k, random_state=42)
Z_pool = pca_final.fit_transform(X_pool_s).astype(np.float32)
Z_test = pca_final.transform(X_test_s).astype(np.float32)

mlp = MLPRegressor(in_features=best_k).to(device)
mlp_opt = optim.Adam(mlp.parameters(), lr=1e-3)

mlp_train_loader = DataLoader(PCADataset(Z_pool, y_pool), batch_size=BATCH_SIZE,
mlp_test_loader  = DataLoader(PCADataset(Z_test, y_test), batch_size=BATCH_SIZE,

for ep in range(1, FINAL_EPOCHS + 1):
    loss = train_one_epoch(mlp, mlp_train_loader, mlp_opt, device)
    if ep % 2 == 0:
        print(f"[MLP] Epoch {ep:02d}/{FINAL_EPOCHS} | loss={loss:.6f}")

mlp_mse, mlp_iou, mlp_P, mlp_G = evaluate(mlp, mlp_test_loader, device)

print(f"FINAL PCA+MLP (k={best_k}) on TEST | MSE: {mlp_mse:.6f} | Mean IoU: {mlp
```

```
Final PCA+MLP Training on FULL POOL with k=64
[MLP] Epoch 02/10 | loss=0.002298
[MLP] Epoch 04/10 | loss=0.001499
[MLP] Epoch 06/10 | loss=0.001258
[MLP] Epoch 08/10 | loss=0.001136
[MLP] Epoch 10/10 | loss=0.001023
FINAL PCA+MLP (k=64) on TEST | MSE: 0.000952 | Mean IoU: 0.7827
```

5.4 End-to-End CNN Training

We now train a **CNN regressor directly on raw images**, using:

- The **same pool/test split**
- The **same optimizer and loss**

This enables a **fair comparison** with PCA+MLP.

In [20]:
```
cnn = CNNRegressor().to(device)
cnn_opt = optim.Adam(cnn.parameters(), lr=1e-3)

cnn_train_loader = DataLoader(ImageDataset(X_pool, y_pool), batch_size=BATCH_SIZ
cnn_test_loader  = DataLoader(ImageDataset(X_test, y_test), batch_size=BATCH_SIZ

for ep in range(1, FINAL_EPOCHS + 1):
    loss = train_one_epoch(cnn, cnn_train_loader, cnn_opt, device)
    if ep % 2 == 0:
        print(f"[CNN] Epoch {ep:02d}/{FINAL_EPOCHS} | loss={loss:.6f}")

cnn_mse, cnn_iou, cnn_P, cnn_G = evaluate(cnn, cnn_test_loader, device)

print(f"FINAL CNN on TEST | MSE: {cnn_mse:.6f} | Mean IoU: {cnn_iou:.4f}")


# For your visualization section (use TEST set)
Xts = X_test
yts = y_test
Zts = Z_test
```

```
[CNN] Epoch 02/10 | loss=0.002049
[CNN] Epoch 04/10 | loss=0.001097
[CNN] Epoch 06/10 | loss=0.000941
[CNN] Epoch 08/10 | loss=0.000831
[CNN] Epoch 10/10 | loss=0.000798
FINAL CNN on TEST | MSE: 0.000778 | Mean IoU: 0.7962
```

5.5 Model Architecture Inspection (Sanity Check)

Before visualization, we inspect the **final trained models** to verify:

- CNN spatial dimension changes across convolutional layers
- Flattened feature size consistency
- MLP hidden layer widths after PCA compression

This step **does not affect training or evaluation**, and is used purely for model interpretability and correctness verification.

In [21]:
```
def print_cnn_shapes_once(model, loader, device):
    """
    Prints intermediate tensor shapes for ONE batch using forward hooks.
    Call this near the end (before visualization) so it doesn't spam training lo
    """
    model.eval()

    # pick one batch
    x, _ = next(iter(loader))
    x = x.to(device)

    print("\n CNN Layer Output Shapes (one batch) ")
    print("Legend:")
    print("  B = batch size")
    print("  C = channels (feature maps)")
    print("  H = height (pixels)")
    print("  W = width  (pixels)")
    print("  F = flattened feature length\n")

    hooks = []
    def hook_fn(name):
        def _hook(module, inp, out):
            if torch.is_tensor(out):
                print(f"{name:<25} -> {tuple(out.shape)}")
        return _hook

    # Register hooks on key layers
    hooks.append(model.conv[0].register_forward_hook(hook_fn("conv1 (1->16, s2)"
    hooks.append(model.conv[2].register_forward_hook(hook_fn("conv2 (16->32, s2)
    hooks.append(model.conv[4].register_forward_hook(hook_fn("conv3 (32->64, s2)
    hooks.append(model.conv[6].register_forward_hook(hook_fn("adaptive_avg_pool(
    hooks.append(model.fc[0].register_forward_hook(hook_fn("flatten")))
    hooks.append(model.fc[1].register_forward_hook(hook_fn("fc1 (1024->128)")))
    hooks.append(model.fc[3].register_forward_hook(hook_fn("fc2 (128->4)")))

    print(f"input                     -> {tuple(x.shape)}")
    with torch.no_grad():
        _ = model(x)

    # Remove hooks
    for h in hooks:
        h.remove()

def print_mlp_hidden_units(model):
    """
    Prints the hidden units / layer widths of an MLPRegressor built from nn.Sequ
    Example output: in_features -> 256 -> 256 -> 4
    """
    linears = [m for m in model.modules() if isinstance(m, nn.Linear)]
    if not linears:
        print("No nn.Linear layers found.")
        return

    sizes = [linears[0].in_features] + [l.out_features for l in linears]
    print("\nMLP Hidden Units (Layer Widths)")
    print(" -> ".join(map(str, sizes)))
```

```
    print("(Format: input_dim -> hidden1 -> hidden2 -> ... -> output_dim)")

print_cnn_shapes_once(cnn, cnn_test_loader, device)
print_mlp_hidden_units(mlp)
```

```
 CNN Layer Output Shapes (one batch)
Legend:
  B = batch size
  C = channels (feature maps)
  H = height (pixels)
  W = width  (pixels)
  F = flattened feature length

input                    -> (128, 1, 64, 64)
conv1 (1->16, s2)         -> (128, 16, 32, 32)
conv2 (16->32, s2)        -> (128, 32, 16, 16)
conv3 (32->64, s2)        -> (128, 64, 8, 8)
adaptive_avg_pool(4x4)    -> (128, 64, 4, 4)
flatten                   -> (128, 1024)
fc1 (1024->128)           -> (128, 128)
fc2 (128->4)              -> (128, 4)

MLP Hidden Units (Layer Widths)
64 -> 256 -> 256 -> 4
(Format: input_dim -> hidden1 -> hidden2 -> ... -> output_dim)
```

6. Visualization and Qualitative Evaluation

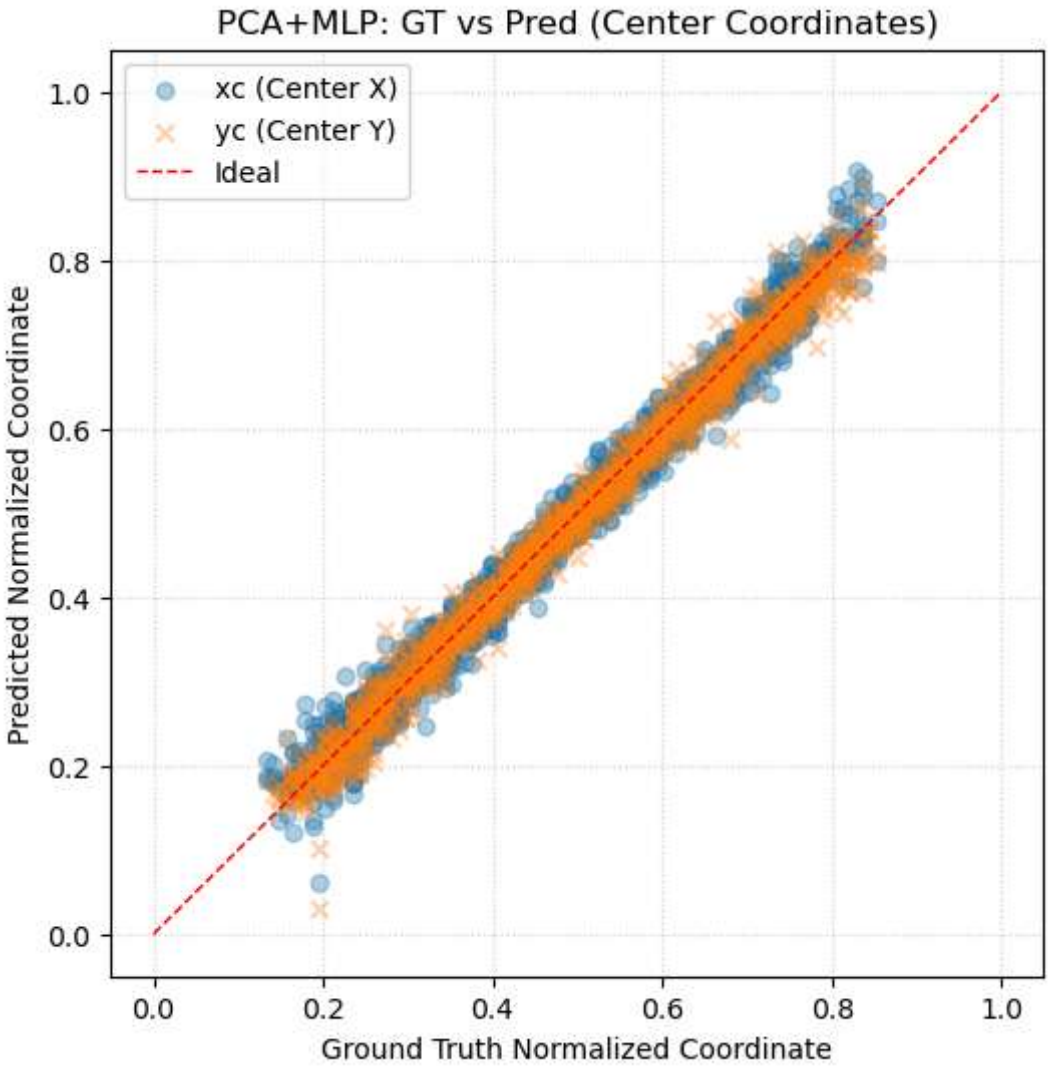
This section provides **model interpretability and qualitative validation** via:

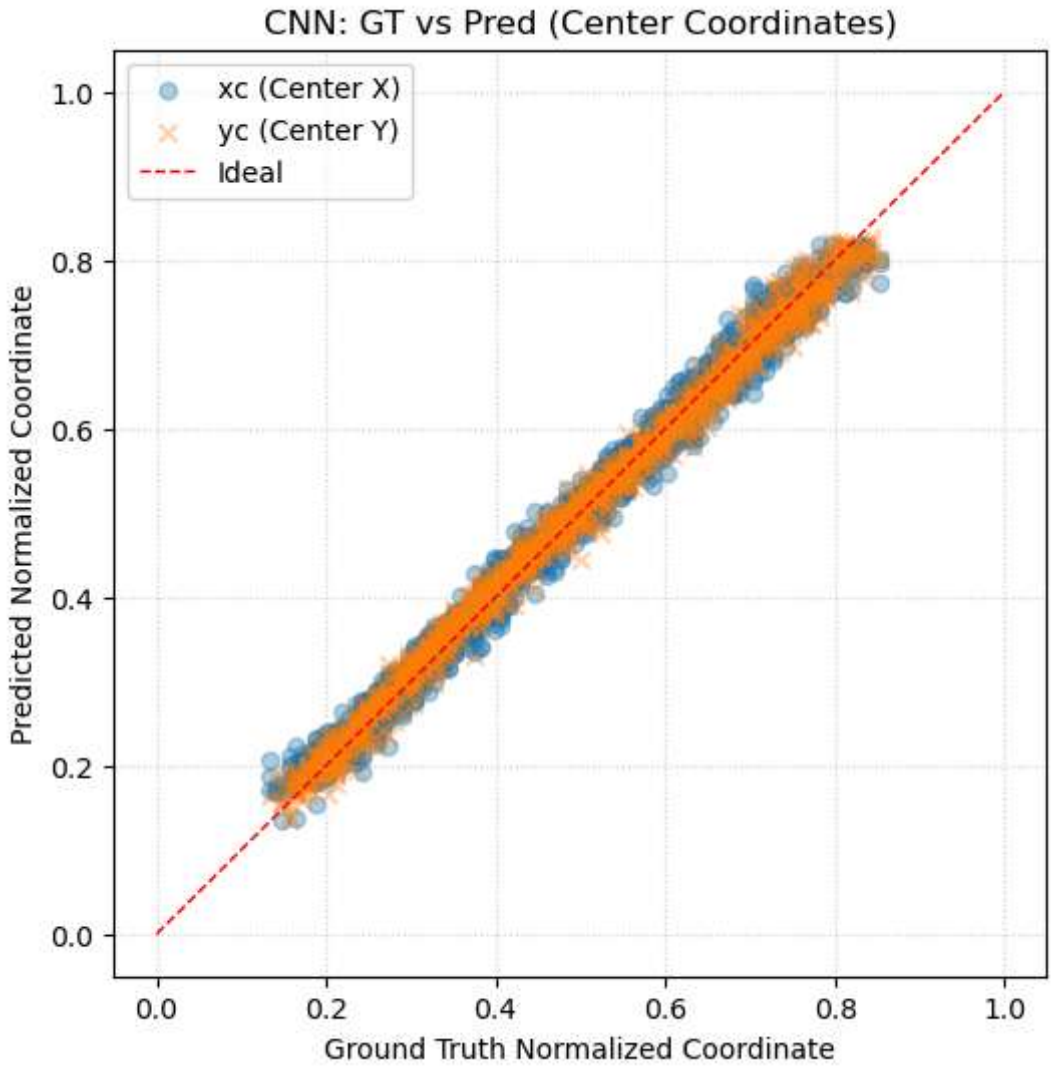
1. **Scatter plots**: GT vs predicted box centers
2. **Image overlays**: Bounding boxes on test images with IoU scores

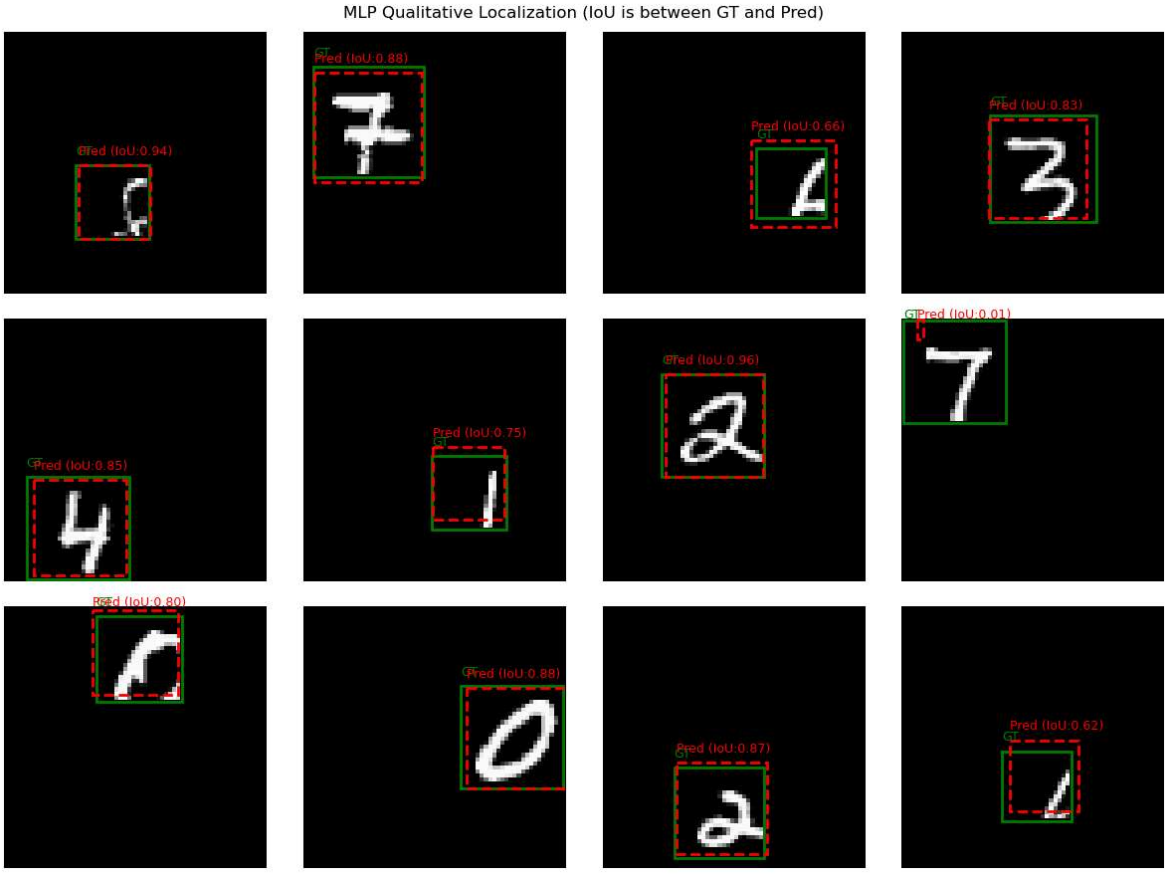
These visualizations are essential for understanding *where* models succeed or fail.

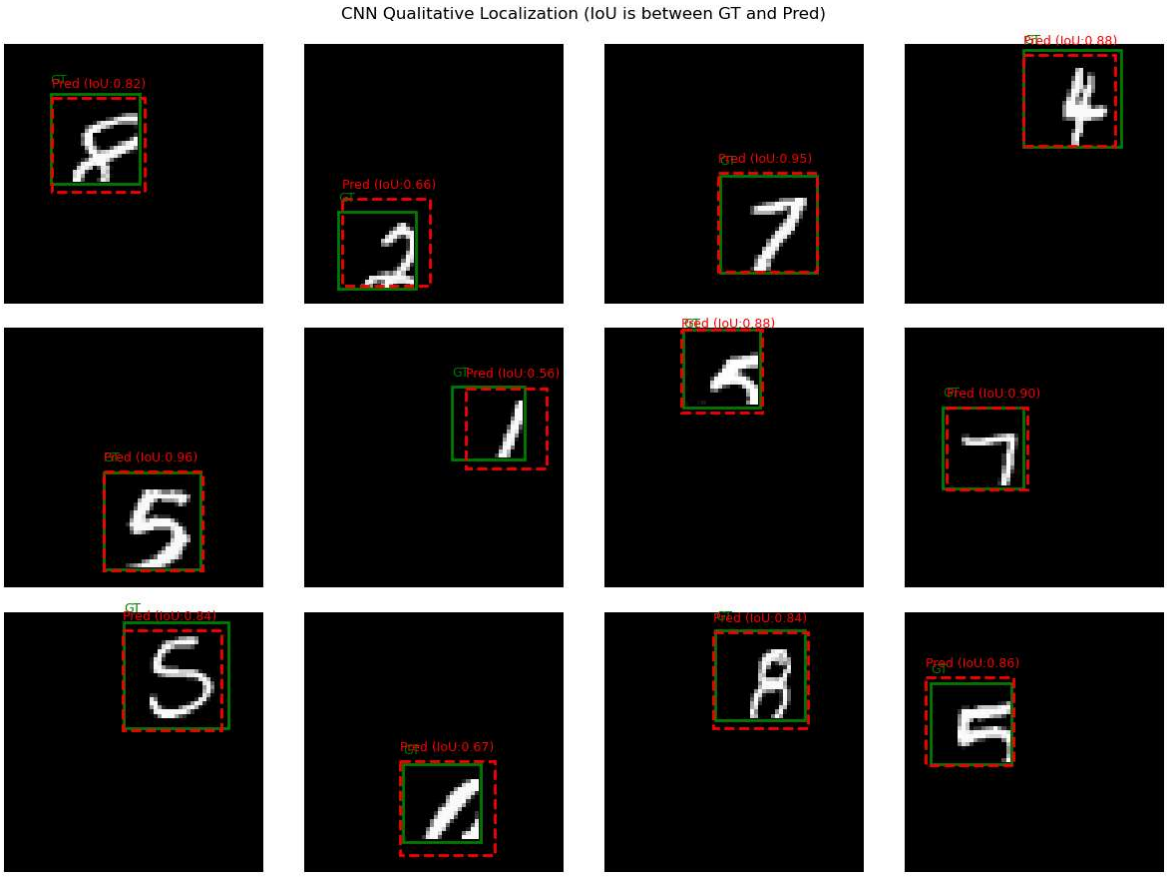
In [22]:
```
def scatter_gt_vs_pred(G, P, title):
    """ Visualization 1: Scatter plot of Ground Truth vs. Predicted values. """
    plt.figure(figsize=(6, 6))
    plt.scatter(G[:, 0], P[:, 0], alpha=0.35, label="xc (Center X)", marker="o")
    plt.scatter(G[:, 1], P[:, 1], alpha=0.35, label="yc (Center Y)", marker="x")
    plt.plot([0, 1], [0, 1], 'r--', linewidth=1, label="Ideal")
    plt.xlabel("Ground Truth Normalized Coordinate")
    plt.ylabel("Predicted Normalized Coordinate")
    plt.title(title)
    plt.legend()
    plt.grid(True, linestyle=':', alpha=0.5)
    plt.show()

def draw_bbox(ax, bbox_xyxy_norm, H=64, W=64, label="Pred", color='r', linestyle
    """ Helper to draw a single bounding box on a matplotlib axis. """
    x1, y1, x2, y2 = bbox_xyxy_norm
    x1p, y1p = x1 * W, y1 * H
    wp, hp = (x2 - x1) * W, (y2 - y1) * H
    rect = plt.Rectangle((x1p, y1p), wp, hp, fill=False, linewidth=2, color=colo
    ax.add_patch(rect)
    ax.text(x1p, max(0, y1p - 2), label, fontsize=9, va="bottom", color=color)
```

```
@torch.no_grad()
def visualize_predictions(model, X_images, Z_features, y_true, device, model_typ
    """ Visualization 2: Qualitative visualization of BBox predictions on test i
    model.eval()
    idxs = np.random.choice(len(X_images), size=min(n, len(X_images)), replace=F

    ncols = 4
    nrows = int(np.ceil(len(idxs) / ncols))
    plt.figure(figsize=(12, 3 * nrows))

    for i, idx in enumerate(idxs, start=1):
        img = X_images[idx, 0]
        gt = y_true[idx]

        if model_type == 'MLP':
            # Use PCA features Z for MLP
            data = torch.from_numpy(Z_features[idx:idx+1]).float().to(device)
        else:
            # Use raw image X for CNN
            data = torch.from_numpy(X_images[idx:idx+1]).float().to(device)

        pred = model(data).cpu().numpy()[0]

        gt_xyxy = bbox_xywh_to_xyxy(gt[None])[0]
        pr_xyxy = bbox_xywh_to_xyxy(pred[None])[0]
        iou = compute_iou(gt_xyxy[None], pr_xyxy[None])[0]

        ax = plt.subplot(nrows, ncols, i)
        ax.imshow(img, cmap="gray")
        draw_bbox(ax, gt_xyxy, label="GT", color='g', linestyle='-')
        draw_bbox(ax, pr_xyxy, label=f"Pred (IoU:{iou:.2f})", color='r', linesty
        ax.axis("off")

    plt.suptitle(f"{model_type} Qualitative Localization (IoU is between GT and
    plt.tight_layout()
    plt.show()


# Run all visualizations


scatter_gt_vs_pred(mlp_G, mlp_P, "PCA+MLP: GT vs Pred (Center Coordinates)")
scatter_gt_vs_pred(cnn_G, cnn_P, "CNN: GT vs Pred (Center Coordinates)")

visualize_predictions(mlp, Xts, Zts, yts, device, model_type='MLP', n=12)
visualize_predictions(cnn, Xts, None, yts, device, model_type='CNN', n=12)
```


PCA+MLP: GT vs Pred (Center Coordinates)
xc (Center X)
yc (Center Y)
Ideal
Predicted Normalized Coordinate
Ground Truth Normalized Coordinate
0.0
0.2
0.4
0.6
0.8
1.0


CNN: GT vs Pred (Center Coordinates)
xc (Center X)
yc (Center Y)
Ideal
Predicted Normalized Coordinate
Ground Truth Normalized Coordinate
0.0
0.2
0.4
0.6
0.8
1.0


MLP Qualitative Localization (IoU is between GT and Pred)


CNN Qualitative Localization (IoU is between GT and Pred)
Pred (IoU:0.82)
Pred (IoU:0.66)
Pred (IoU:0.95)
Pred (IoU:0.88)
Pred (IoU:0.96)
Pred (IoU:0.56)
Pred (IoU:0.88)
Pred (IoU:0.90)
Pred (IoU:0.67)
Pred (IoU:0.84)
Pred (IoU:0.86)