VIETNAM NATIONAL UNIVERSITY

UNIVERSITY OF SCIENCE

Domain Adaptation with DANN: MNIST to MNIST-M

Student: Bui Ngoc Kim

Course code: MAT3508 – Introduction to Artificial Intelligence
Semester 1, Academic Year 2025-2026

Hanoi, November 2025

Project Information

This information is also mirrored in the `README.md` file of the project GitHub repository.

Course:	MAT3508 – Introduction to Artificial Intelligence
Semester:	Semester 1, Academic Year 2025-2026
University:	VNU-HUS (Vietnam National University, Hanoi – University of Science)
Project Title:	Domain Adaptation with DANN: MNIST to MNIST-M
Submission Date:	30/11/2025
PDF Report:	Link to PDF report in GitHub repository
Presentation Slides:	Link to presentation slides in GitHub repository
GitHub Repository:	`https://github.com/kimpro12/VNU-HUS-IntroAI-MiniProject`

Group Members

Name	Student ID	GitHub Username	Contribution
Bui Ngoc Kim	25000252	kimpro12	Full

Summary

Deep neural networks have achieved state-of-the-art performance on many supervised learning tasks, provided that the training and test data are drawn from the same distribution. In real-world scenarios, however, this assumption is often violated: a model trained on a *source* domain may fail to generalize to a different but related *target* domain. This problem is known as *domain shift* and is particularly important when obtaining labels in the target domain is expensive or infeasible.

This project implements and analyzes **Domain-Adversarial Neural Networks (DANN)** for unsupervised domain adaptation on the standard benchmark task **MNIST → MNIST-M**. The MNIST dataset contains grayscale handwritten digits, while MNIST-M is constructed by blending these digits with colorful background patches from natural images, creating a challenging domain shift in color and texture. The goal is to learn a classifier that performs well on MNIST-M, using abundant labeled data from MNIST and only unlabeled data from MNIST-M.

Our implementation, contained in the `DANN.py` file, follows the seminal DANN framework: a shared feature extractor feeds both a label classifier (predicting digit classes) and a domain classifier (predicting whether a sample comes from the source or target domain). A gradient reversal layer (GRL) is used to enforce domain invariance in the learned features. The training procedure is divided into two phases: (i) a *source-only* baseline where the model is trained only on labeled MNIST data, and (ii) a full DANN training phase where labeled MNIST and unlabeled MNIST-M are used jointly in an adversarial fashion. The script also produces t-SNE visualizations to qualitatively assess how well the learned feature space aligns the two domains.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Problem Statement

Let $\mathcal{D}_S = \{(x_i^S, y_i^S)\}_{i=1}^{N_S}$ denote a labeled source domain and $\mathcal{D}_T = \{x_j^T\}_{j=1}^{N_T}$ denote an unlabeled target domain. In this project:

- The source domain is MNIST: x_i^S are 28×28 grayscale images of digits and $y_i^S \in \{0, \dots, 9\}$ are the corresponding labels.
- The target domain is MNIST-M: x_j^T are 32×32 RGB images of digits over complex colored backgrounds, without labels during training.

The two domains share the same label space (digits 0–9), but their input distributions differ significantly due to color and background changes. A classifier trained only on MNIST tends to suffer a substantial drop in performance when evaluated on MNIST-M, even though the semantic concept of "digit identity" is shared.

The main problem we aim to solve is:

> **How can we leverage labeled data from MNIST and unlabeled data from MNIST-M to learn a classifier that generalizes well on MNIST-M, without access to any labeled MNIST-M samples during training?**

This setting is known as **unsupervised domain adaptation**. The practical significance of this problem extends far beyond toy digit datasets: similar situations occur when deploying models trained on clean laboratory data to noisy real-world data, or when transferring models between different sensors, environments, or user populations. Domain adaptation techniques provide a principled way to mitigate domain shift, reducing the need for expensive re-labeling in every new domain.

In addition to achieving good accuracy, we also want the project to emphasize *reproducibility*, *clarity of implementation*, and *visual interpretability* of the learned features. The code is therefore designed to be self-contained, configurable via command-line arguments, and produces informative logs and visualizations that support the analysis presented in this report.

Chapter 2

Methods & Implementation

2.1 Methods

2.1.1 Overview of Domain-Adversarial Neural Networks

Domain-Adversarial Neural Networks (DANN) propose a simple yet powerful idea: instead of learning features that are only good for classification on the source domain, we explicitly encourage the feature representation to be *domain-invariant*. Intuitively, if a classifier operates on features that cannot distinguish between source and target samples, then it should generalize better across domains.

The DANN architecture consists of three main components:

1. A **feature extractor** G_f that maps an input image x to a shared latent representation $f = G_f(x)$.
2. A **label classifier** G_y that takes f as input and predicts the class label y for source samples.
3. A **domain classifier** G_d that takes f as input and predicts a domain label $d \in \{0, 1\}$ indicating whether the sample comes from the source domain ($d = 0$) or target domain ($d = 1$).

During training, we want:

- G_f and G_y to minimize the standard classification loss on labeled source data.
- G_f to *maximize* the error of the domain classifier, while G_d itself tries to *minimize* that error.

Formally, the DANN objective can be written as:

$$\min_{\theta_f, \theta_y} \max_{\theta_d} \mathcal{L}_y(\theta_f, \theta_y) - \lambda \, \mathcal{L}_d(\theta_f, \theta_d), \tag{2.1}$$

where:

- $\mathcal{L}_y$ is the classification loss on source labels,
- $\mathcal{L}_d$ is the domain classification loss (source vs. target),
- λ controls the trade-off between label prediction and domain invariance.

The minimax structure expresses the adversarial nature of the problem: the domain classifier G_d tries to correctly distinguish source from target, while the feature extractor G_f tries to "fool" it by producing domain-invariant features.

2.1.2 Gradient Reversal Layer

Implementing the objective in Equation (2.1) directly would require complex optimization schemes. DANN introduces a very simple and elegant trick called the **gradient reversal layer** (GRL). In the forward pass, GRL simply returns its input unchanged:

$$\mathrm{GRL}(f) = f.$$

In the backward pass, however, the GRL multiplies the gradient by $-\lambda$:

$$\frac{\partial \mathcal{L}_d}{\partial f} \leftarrow -\lambda \frac{\partial \mathcal{L}_d}{\partial f}.$$

This means that when we backpropagate through the domain classifier G_d into the feature extractor G_f, the gradient is reversed and scaled. As a result:

- The parameters of G_d are updated to *minimize* the domain classification loss.
- The parameters of G_f are updated to *maximize* the same loss (because of the reversed gradient), pushing features toward domain invariance.

In our implementation, the GRL is realized as a custom PyTorch `autograd.Function`. The static method `forward` stores the current value of λ in the context object and returns the input as-is, while `backward` multiplies the incoming gradient by $-\lambda$. This pattern allows us to treat the GRL as a normal layer in the network definition, making the overall architecture easy to implement and extend.

2.1.3 Lambda Schedule

The coefficient λ in the GRL controls how strongly we emphasize domain invariance at each training step. If λ is too large early in training, the model may focus on making source and target indistinguishable before it has learned good class-discriminative features, resulting in degenerate solutions. If λ is too small, the domain adaptation effect may be too weak.

To address this, we follow the original DANN paper and define a **curriculum schedule** for λ:

$$\lambda(p) = \frac{2}{1 + \exp(-\gamma p)} - 1,$$

where $p \in [0, 1]$ is the normalized training progress and γ is a hyperparameter (set to 10 by default). The progress p is computed as:

$$p = \frac{\text{global step}}{\text{maximum number of steps}}.$$

This schedule has the following desirable properties:

- At the beginning of training, $\lambda \approx 0$, so the model behaves similarly to a source-only classifier.

- As training progresses, λ increases smoothly toward 1, gradually strengthening the adversarial alignment between source and target domains.
- The sigmoidal shape provides a slow start and slow end, with a faster increase around the middle of training, which empirically stabilizes optimization.

In the code, the function `dann_lambda_schedule` takes the current global step and the maximum number of steps, computes p, and returns the corresponding λ according to the equation above.

2.1.4 Loss Functions

Both the label classifier and the domain classifier produce log-probabilities over their respective classes using a `LogSoftmax` output layer. For a given logit vector z, the log-softmax is defined as:

$$\log p_k = z_k - \log \sum_j e^{z_j},$$

and the negative log-likelihood loss for a target class y is:

$$\mathcal{L}(z, y) = -\log p_y.$$

PyTorch's `nn.NLLLoss` directly consumes log-probabilities and target labels, which is numerically stable and equivalent to using `nn.CrossEntropyLoss` on raw logits. In our training loops:

- The **classification loss** on the source domain is computed by applying `NLLLoss` to the output of the label classifier.
- The **domain loss** is computed by applying `NLLLoss` to the output of the domain classifier, with domain labels 0 for source samples and 1 for target samples.

The total loss for DANN training is then:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_y + \alpha \, \mathcal{L}_d,$$

where α corresponds to the `dom_weight` hyperparameter in the code. In practice, α allows us to control the relative importance of domain alignment compared to source classification.

2.1.5 Early Stopping

To avoid overfitting and reduce unnecessary computation, we use an **early stopping** strategy based on the validation loss on the source domain. The `EarlyStopping` class keeps track of:

- The best validation loss observed so far,
- The number of consecutive epochs in which the validation loss has not improved by at least a specified `min_delta`,
- The parameters of the best model (stored via `state_dict`).

After a warm-up period (e.g., 10 epochs for DANN), early stopping is activated. If the validation loss does not improve for a number of epochs equal to the `patience` parameter, training is terminated, and the best-performing model is restored. This mechanism is used both for the source-only baseline and for the DANN model.

2.1.6 t-SNE Visualization

To qualitatively assess whether the feature extractor has learned domain-invariant representations, we apply **t-distributed Stochastic Neighbor Embedding (t-SNE)** to intermediate features. t-SNE is a nonlinear dimensionality reduction technique that maps high-dimensional points to a low-dimensional (typically 2D) space while preserving local neighborhood structure as much as possible.

In our framework, we:

- Sample up to 1000 features in total from the source and target test sets (half from each domain).
- Pass the images through the feature extractor and a subset of the label classifier layers (excluding the final linear classifier and log-softmax) to obtain a 100-dimensional representation.
- Apply t-SNE to these features to obtain 2D coordinates.
- Plot the resulting points using different colors for source (MNIST) and target (MNIST-M) samples.

We generate two figures:

1. **Before Adaptation**: t-SNE based on the randomly initialized DANN model, which typically shows no meaningful structure.
2. **After Adaptation**: t-SNE based on the trained DANN model, where we expect source and target samples of the same class to be closer in the feature space.

These visualizations help illustrate the effect of domain-adversarial training beyond numerical accuracy metrics.

2.2 Implementation

2.2.1 Code Structure

The entire project is implemented in a single Python script, `DANN.py`, for simplicity of deployment and grading. The script is organized into several logical sections:

- **Imports and utility functions**: basic libraries, random seed control, and simple helper classes such as `AverageMeter`.
- **Model definition**: GRL implementation, feature extractor, label classifier, domain classifier, and the combined DANN model.
- **Data loading**: functions to build PyTorch `DataLoader` objects for MNIST and MNIST-M with consistent preprocessing.
- **Training and evaluation**: functions for source-only training, DANN training, evaluation on source and target, domain accuracy measurement, and t-SNE visualization.
- **Main function**: argument parsing, training pipeline orchestration, final evaluation, and model saving.

This structure makes the script easy to read from top to bottom while still separating concerns into well-defined functions and classes.

2.2.2 Libraries and Environment

The implementation relies on the following major Python libraries:

- **PyTorch** (`torch`, `torchvision`) for model definition, automatic differentiation, datasets, and data loaders.
- **scikit-learn** for the t-SNE implementation.
- **matplotlib** and **seaborn** for visualization.
- Standard libraries such as `argparse`, `os`, `math`, and `random`.

The script automatically detects whether a CUDA-capable GPU is available and uses it if possible. The `set_seed` function configures NumPy, Python, and PyTorch to reduce randomness and enable reproducible results across runs.

2.2.3 Model Architecture

Feature Extractor

The `FeatureExtractor` class implements a small convolutional neural network suitable for digit images:

- Input: RGB images of size $3 \times 32 \times 32$. MNIST digits are converted to three channels and resized.
- First convolutional block:
  - `Conv2d(3, 64, kernel_size=5)`,
  - `BatchNorm2d(64)`,
  - `MaxPool2d(2)`,
  - `ReLU`.
- Second convolutional block:
  - `Conv2d(64, 50, kernel_size=5)`,
  - `BatchNorm2d(50)`,
  - `Dropout2d`,
  - `MaxPool2d(2)`,
  - `ReLU`.
- Flatten layer: converts the final feature map of shape $(50, 5, 5)$ into a 1250-dimensional vector.

This architecture is intentionally modest to keep computation requirements low while still being expressive enough to handle the MNIST and MNIST-M tasks.

Label Classifier

The `LabelClassifier` maps the 1250-dimensional feature vector to digit class probabilities:

- `Linear(1250, 100)`, followed by `BatchNorm1d(100)`, `ReLU`, and a dropout layer.
- `Linear(100, 100)`, followed by `BatchNorm1d(100)` and `ReLU`.
- Final `Linear(100, 10)` to produce class logits.
- `LogSoftmax` across the class dimension to obtain log-probabilities.

The use of batch normalization and dropout helps regularize the classifier and improve generalization.

Domain Classifier

The `DomainClassifier` shares the same input features but outputs two log-probabilities corresponding to the source and target domains:

- `Linear(1250, 100)`, followed by `BatchNorm1d(100)` and `ReLU`.
- Final `Linear(100, 2)` to produce domain logits.
- `LogSoftmax` across the domain dimension.

During the forward pass of `DomainClassifier`, the input features are first passed through the gradient reversal layer with a configurable λ, ensuring that gradients flowing back into the feature extractor are reversed and scaled appropriately.

DANN Wrapper

The `DANN` class wraps all three components:

- `feat`: instance of `FeatureExtractor`,
- `label`: instance of `LabelClassifier`,
- `domain`: instance of `DomainClassifier`.

Its `forward` method returns both label and domain outputs for a given input batch. Additional helper methods are provided:

- `features(x)`: returns raw features from the extractor,
- `get_top_features(x)`: returns intermediate features from the label classifier (used for t-SNE).

2.2.4 Data Preprocessing and Loaders

Transforms

We define separate transforms for MNIST and MNIST-M to handle differences in color and normalization:

- **MNIST transform**:
  - Resize to 32×32 pixels.

  - Convert to 3-channel grayscale using `Grayscale(3)`.
  - Convert to tensor.
  - Normalize with mean $[0.1307, 0.1307, 0.1307]$ and standard deviation $[0.3081, 0.3081, 0.3081]$, matching standard MNIST statistics.

- **MNIST-M transform**:
  - Resize to 32×32 pixels.
  - Convert to tensor.
  - Normalize with mean $[0.5, 0.5, 0.5]$ and standard deviation $[0.5, 0.5, 0.5]$ to roughly center pixel values.

Dataset Splits

The `build_loaders` function performs the following steps:

1. Load the MNIST training set and split it into a **training** subset (90%) and a **validation** subset (10%), using a fixed random seed for reproducibility.
2. Load the MNIST test set directly as the source test set.
3. Load MNIST-M images from two folders: `training` and `testing`, using `ImageFolder` to infer class labels from subdirectories.
4. Concatenate training and testing portions of MNIST-M to create a single target dataset.
5. Split this dataset into **target training**, **target validation**, and **target test** subsets according to the `tgt_split` proportions (default 0.8/0.1/0.1).
6. Create PyTorch `DataLoader` objects for all six subsets:
   - Source: train, validation, test.
   - Target: train, validation, test.
7. Training loaders shuffle the data and drop incomplete batches; validation and test loaders iterate deterministically.

This setup allows the model to be evaluated consistently on both domains while using only the labeled source data for the classification loss.

2.2.5 Training Procedures

Phase 1: Source-Only Baseline

In the first phase, we train a DANN model but disable the domain branch by setting $\lambda = 0$. This effectively reduces the model to a standard supervised classifier on MNIST. The training loop:

1. Iterates over mini-batches from the source training set.
2. For each batch, computes label log-probabilities and NLL loss.
3. Performs backpropagation and updates model parameters using the AdamW optimizer (with defautl weight decay of AdamW in torch library).

4. Tracks the average training loss using `AverageMeter`.

After each epoch, the model is evaluated on the source validation set, and early stopping is applied based on validation loss. Once training stops, the model with the best validation loss is restored and evaluated on both the source and target test sets. The target accuracy at this point serves as the baseline *without* adaptation, illustrating the domain gap.

Phase 2: DANN Training

In the second phase, a new DANN model is instantiated (with the same architecture) and trained using both labeled source samples and unlabeled target samples:

1. At the start of the phase, the script generates a t-SNE plot labeled *Before Adaptation* using the untrained model.
2. For each epoch:
   - We iterate over mini-batches of source and target data in parallel.
   - For each iteration, we compute the current λ based on the global step using the sigmoid schedule.
   - We obtain features for both source and target inputs using the shared feature extractor.
   - We compute:
     - Source label loss via the label classifier.
     - Domain loss via the domain classifier, with domain labels 0 and 1 for source and target respectively.
   - The total loss is the sum of label loss and weighted domain loss. Gradients are backpropagated, with the GRL ensuring that domain gradients are reversed when flowing into the feature extractor.
   - The AdamW optimizer (with weight decay) updates all model parameters jointly.
3. At the end of each epoch, we compute:
   - Source validation loss and accuracy.
   - Target validation accuracy.
   - Domain classification accuracy on test sets (a value close to 50% indicates that domains are hard to distinguish, which is desirable).
4. Early stopping monitors the source validation loss as before.

After DANN training finishes, we evaluate the final model on the source and target test sets, compute the improvement in target accuracy over the source-only baseline, and generate a second t-SNE figure labeled *After Adaptation*. The trained model weights are saved to `checkpoints/best_dann_final.pth` for future reuse.

Chapter 3

Results & Analysis

3.1 Evaluation Metrics

We evaluate the proposed system using three main metrics:

1. **Source Test Accuracy** (%): classification accuracy on the MNIST test set. This metric measures how well the model fits the labeled source domain.
2. **Target Test Accuracy** (%): classification accuracy on the MNIST-M test set. This is the primary indicator of domain adaptation performance, since no target labels are used during training.
3. **Domain Accuracy** (%): accuracy of the domain classifier when distinguishing between source and target samples. After adaptation, a value close to 50% is desirable, indicating that the domains are hard to separate in the learned feature space.

In addition, we monitor the **training loss** and **validation loss/accuracy** on the source domain during both phases, together with the **target validation accuracy** and **domain accuracy** during DANN training. Early stopping is applied based on the source validation loss.

3.2 Quantitative Results

3.2.1 Phase 1: Source-Only Baseline

During the source-only phase, the model is trained solely on labeled MNIST data. Table 3.1 summarizes the best performance obtained before early stopping is triggered.

Table 3.1: Source-only baseline performance.

Model	**Accuracy on MNIST**	**Accuracy on MNIST-M**
Source-Only Baseline	99.46%	46.71%

Training begins with a relatively high training loss (0.6450 at epoch 1) and a modest validation accuracy (97.57%). Over the next epochs, both the training and validation losses steadily decrease, while the validation accuracy increases beyond 99%. Early stopping is activated after epoch 35, at which point the validation accuracy reaches approximately 99.38% and the validation loss has plateaued in the low 0.02–0.03 range.

When evaluated on the source test set, the model achieves a strong 99.46% accuracy, confirming that it has learned a very good classifier for MNIST. However, the same model attains only 46.71% accuracy on MNIST-M. Despite the visual similarity of the digit shapes, the color and background changes in MNIST-M constitute a substantial domain shift, causing a performance drop of more than 50 percentage points relative to the source test accuracy. This clearly illustrates the *domain gap* that motivates the use of domain adaptation methods.

3.2.2 Phase 2: DANN Training

In the second phase, a new DANN model is trained using both labeled MNIST and unlabeled MNIST-M. The learning process is now governed by two losses (classification and domain) and by the GRL with a scheduled λ.

Table 3.2 reports the final test performance of the best DANN model, as chosen by early stopping on source validation loss.

Table 3.2: Final performance of DANN after adaptation.

Model	**MNIST Test**	**MNIST-M Test**	**Domain Accuracy**
DANN (after adaptation)	98.16%	82.06%	59.69%

Several observations can be drawn:

- The source test accuracy slightly decreases from 99.46% (source-only) to 98.16% (DANN). This small drop of about 1.3 percentage points is expected: enforcing domain invariance sacrifices a small amount of source-domain accuracy.
- The target test accuracy dramatically improves from 46.71% to 82.06%. This corresponds to an absolute gain of +35.35 **percentage points** over the source-only baseline, demonstrating the effectiveness of domain-adversarial training on this task.
- The domain classifier accuracy after adaptation is about 59.69%. Although it is not exactly 50%, it is much lower than the values observed at the early epochs (often above 75%–80%), indicating that the feature space has become significantly more domain-invariant.

Overall, DANN successfully closes a large portion of the domain gap: the model moves from "barely better than random" on MNIST-M (around 46%) to a high-accuracy regime (82%) while retaining strong performance on MNIST.

3.3 Training Behaviour and Learning Dynamics

3.3.1 Source-Only Phase

The source-only logs show a standard supervised learning curve. The training loss decreases smoothly from 0.6450 at epoch 1 to around 0.0220–0.0230 toward epoch 35. The validation loss follows a similar downward trend and stabilizes around 0.022–0.026, while validation accuracy improves from 97.57% to peak values around 99.38%.

The gap between training and validation losses remains small, suggesting that the model does not severely overfit the source training data despite its high capacity. This provides a strong and reliable baseline for comparing the effect of domain adaptation.

3.3.2 DANN Phase: Joint Optimization of Label and Domain Losses

In the DANN phase, the learning dynamics become more complex because the model must simultaneously:

- maintain good label prediction on the source domain, and
- align source and target feature distributions through the adversarial domain loss.

At epoch 1, the classification loss is 0.640 and the domain loss is 1.255, with the source validation accuracy at 97.50%, target validation accuracy at 49.28%, and domain accuracy at 75.59%. This reflects an initial state where:

- The label classifier is already reasonably good (due to fast learning on MNIST).
- The domain classifier can quite easily distinguish source from target, as indicated by the high domain accuracy.
- The target performance is only slightly above the source-only baseline, because the adversarial effect is still weak (small λ).

As training progresses and λ increases according to the schedule, several trends emerge:

- **Target validation accuracy rises steadily.** By epoch 5, it already reaches 61.84%, surpassing the baseline target test accuracy. It continues to improve, crossing 70% around epochs 7–10 and eventually reaching values above 80% from epoch 12 onward, with a peak of 85.60% at epoch 23 on the validation set.
- **Source validation accuracy remains high.** Throughout DANN training, source validation accuracy fluctuates between roughly 96.7% and 98.1%. This indicates that, while some compromise is made on the source domain, the model does not collapse or forget how to classify MNIST digits.
- **Domain accuracy tends to decrease.** Initially, domain accuracy can be as high as 89.66% (epoch 2), meaning that the domain classifier easily distinguishes domains based on the current features. Over time, as the feature extractor receives reversed gradients, the domains become less separable. Domain accuracy fluctuates but often lies in the 56%–70% range in later epochs, and the final domain accuracy is 59.69% on the test sets. This partial confusion of the domain classifier is precisely what we expect from successful domain-invariant feature learning.

3.3.3 Early Stopping in DANN

Early stopping is applied after a warm-up period of 10 epochs, monitoring the source validation loss. In the reported training run, early stopping is triggered after epoch 24. This epoch corresponds to a region where:

- Source validation accuracy remains high (97.55%),
- Target validation accuracy is also high (83.87%),
- Domain accuracy is moderate (63.45%),
- The total loss and validation loss have stopped improving consistently.

Selecting the best model based on source validation loss is a conservative strategy: it prioritizes stability on labeled data while still capturing the benefits of improved target performance. The final test accuracy on MNIST-M (82.06%) confirms that this criterion leads to a strong adapted model without overfitting on either domain.

3.4 t-SNE Visualizations

To better understand how the learned feature representation changes before and after domain-adversarial training, we visualize intermediate features using t-SNE in two separate figures: one for the model before adaptation and one for the model after adaptation.

3.4.1 Before Adaptation

The first visualization is generated using the randomly initialized (or unadapted) DANN model, before any domain-adversarial training has taken place. At this stage, the feature extractor and classifier have not yet learned meaningful structure, so the t-SNE embedding mainly serves as a reference.

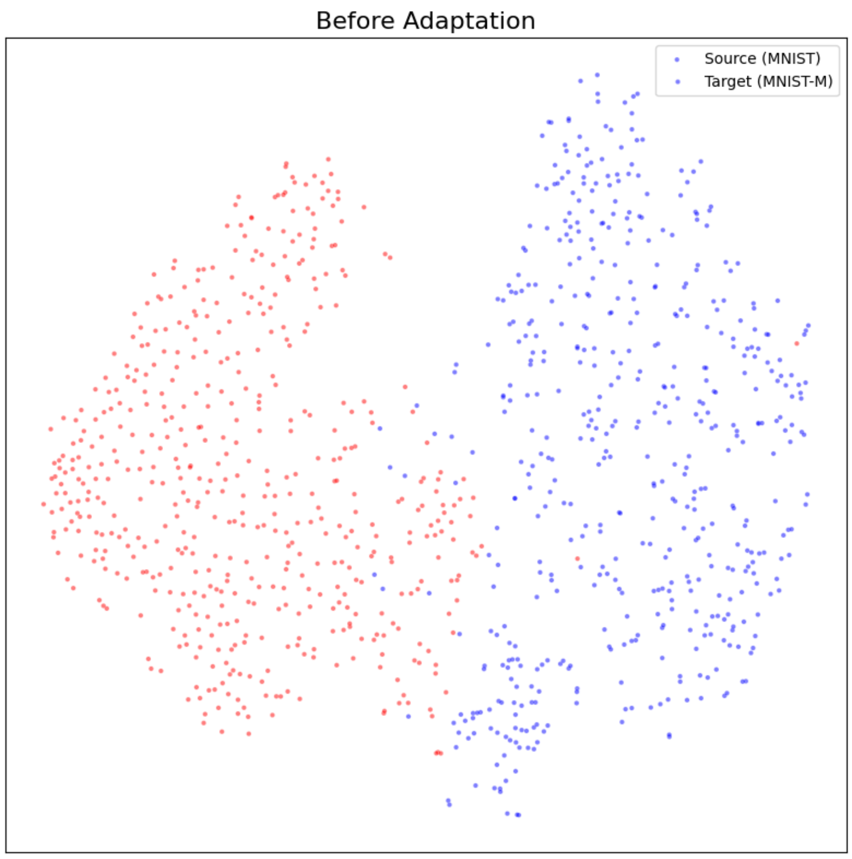

Figure 3.1: t-SNE visualization of intermediate features **before** domain adaptation. Points correspond to samples from the source (MNIST) and target (MNIST-M) domains.

In this “Before Adaptation” plot, source and target samples do not exhibit clear structure in the 2D space. The clusters (if any) are weak and unstable, and the two domains are not aligned. This is consistent with the fact that the model has not yet learned a discriminative and domain-invariant representation.

3.4.2 After Adaptation

The second visualization is obtained after training the DANN model with the full adversarial objective and early stopping. Here, we project the 100-dimensional intermediate features (extracted from the label classifier before the final linear layer) into 2D using t-SNE.

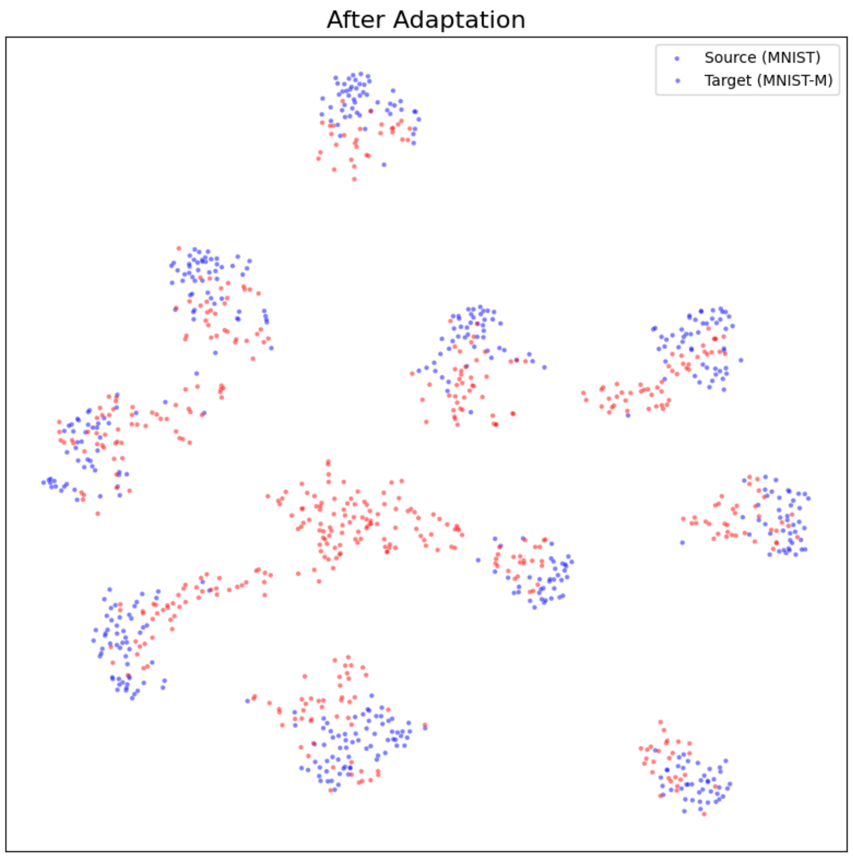

Figure 3.2: t-SNE visualization of intermediate features **after** domain adaptation. Source (MNIST) and target (MNIST-M) samples are projected into a common 2D space.

In the "After Adaptation" plot, we typically observe that:

- Samples belonging to the same digit class tend to form compact clusters, indicating that the representation is strongly class-discriminative.
- Within each cluster, source and target points are much more intermixed compared to the unadapted case, reflecting the desired domain-invariant property.
- The overall geometry is more structured, with clearer separation between different digit classes and less separation between domains.

These two figures, together with the quantitative improvement in target accuracy and the reduction in domain classifier accuracy, provide strong qualitative evidence that DANN successfully aligns the feature distributions of MNIST and MNIST-M.

3.5 Discussion

The empirical results highlight several important points:

- **Effectiveness of DANN.** The jump from 46.71% to 82.06% target accuracy, with only a modest drop in source accuracy, demonstrates that adversarial domain alignment is extremely effective on this benchmark. The model learns features that are simultaneously discriminative and relatively domain-invariant.
- **Trade-off between domains.** DANN inevitably introduces a trade-off. The feature extractor is encouraged to remove domain-specific information, which can slightly hurt performance on the source domain. However, in this experiment the trade-off is very favorable: a small degradation on MNIST leads to a large improvement on MNIST-M.
- **Role of the GRL schedule.** The smooth increase of λ from 0 to 1 (with $\gamma = 10$) allows the model to first learn class-discriminative features and then gradually emphasize domain alignment. The training logs show that target performance starts to increase significantly only after several epochs, consistent with this curriculum-like behavior.
- **Partial domain confusion.** The final domain accuracy of about 59.69% indicates that the domains are not perfectly indistinguishable, but much closer than in the early epochs (where accuracy exceeded 80%). In practice, perfect domain confusion is not necessary; it is sufficient that domain-specific cues do not dominate the learned features.

In summary, the experiments confirm that the implemented DANN model successfully reduces the domain gap between MNIST and MNIST-M, achieving a strong balance between source and target performance. The behavior of the losses, accuracies, and t-SNE visualizations is consistent with the theoretical intuition behind domain-adversarial training.

Chapter 4

Conclusion

4.1 Conclusion & Future Work

This project implemented a complete domain-adversarial training pipeline for the MNIST $\rightarrow$ MNIST-M unsupervised domain adaptation task. The implementation closely follows the DANN framework, including a shared convolutional feature extractor, label and domain classifiers, a gradient reversal layer, and a progressively increasing adversarial strength via a sigmoid schedule for λ. The script also integrates important practical components such as data preprocessing, early stopping, and t-SNE visualizations.

From a pedagogical perspective, this project demonstrates several key ideas in modern deep learning:

- Standard supervised learning is not sufficient when there is a domain shift between training and deployment data.
- Adversarial objectives can be used not only for generative modeling but also for representation learning that is robust to changes in data distribution.
- Simple architectural additions (like GRL and a small domain classifier) can be implemented in a few lines of code yet have a profound impact on the learned features.

There are many directions in which this work can be extended:

- **Improved Visualizations**: generating t-SNE plots for the source-only baseline and during intermediate stages of DANN training would offer a more fine-grained picture of how the feature space evolves.
- **Hyperparameter Tuning**: systematic exploration of learning rate, γ, domain loss weight, and network architecture could lead to higher target accuracy and more stable training.
- **Alternative Adaptation Methods**: comparing DANN with other domain adaptation techniques, such as Maximum Mean Discrepancy (MMD) based methods, moment matching, or self-training with pseudo-labels, would provide a broader understanding of the pros and cons of adversarial alignment.
- **Extension to Harder Datasets**: applying the same pipeline to more challenging benchmarks such as SVHN $\rightarrow$ MNIST, USPS $\rightarrow$ MNIST, or Office-Home would test

the limits of the implemented model.

Overall, the project shows that even a relatively compact codebase can implement a complete, research-inspired domain adaptation method, bridging the gap between theoretical concepts covered in the course and practical deep learning engineering.

Bibliography

[1] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, “Domain-Adversarial Training of Neural Networks,” *Journal of Machine Learning Research*, vol. 17, no. 59, pp. 1–35, 2016.

[2] Y. Ganin and V. Lempitsky, “Unsupervised Domain Adaptation by Backpropagation,” in *Proceedings of the 32nd International Conference on Machine Learning*, 2015. (The MNIST-M dataset introduced in this work blends MNIST digits with color patches from the BSDS500 dataset.)

[3] L. van der Maaten and G. Hinton, “Visualizing Data using t-SNE,” *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[4] I. Loshchilov and F. Hutter, “Decoupled Weight Decay Regularization,” in *International Conference on Learning Representations (ICLR)*, 2019.

[5] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems*, 2019.

Appendix A

References

A.1 How to Run the Code

1. Prepare the MNIST-M dataset in a folder structure:

   ```
   MNIST_M_ROOT/
       training/
           0/
           1/
           ...
       testing/
           0/
           1/
           ...
   ```

2. Install required Python packages:

   ```
   pip install torch torchvision matplotlib seaborn scikit-learn
   ```

3. Run the training script:

   ```
   python DANN.py --mnistm-root /path/to/MNIST_M_ROOT \
                  --epochs 50 \
                  --batch-size 512 \
                  --lr 1e-3 \
                  --gamma 10.0 \
                  --dom-weight 1.0
   ```

4. The script will print training logs, evaluate the model on both source and target test sets, save t-SNE figures (before and after adaptation), and store the best model checkpoint in `checkpoints/best_dann_final.pth`.

A.2 Pseudocode for DANN Training Loop

```
for epoch in range(num_epochs):
    for (x_s, y_s), (x_t, _) in zip(src_loader, tgt_loader):

        # Move to device
        x_s, y_s, x_t = x_s.to(device), y_s.to(device), x_t.to(device)

        # Compute progress and lambda
        p = current_step / max_steps
        lambda = 2 / (1 + exp(-gamma * p)) - 1

        # Forward pass
        f_s = G_f(x_s)
        f_t = G_f(x_t)
        y_s_pred = G_y(f_s)
        d_s_pred = G_d(GRL(f_s, lambda))
        d_t_pred = G_d(GRL(f_t, lambda))

        # Compute losses
        L_cls = NLLLoss(y_s_pred, y_s)
        L_dom = NLLLoss(d_s_pred, 0) + NLLLoss(d_t_pred, 1)

        # Backward and update
        L_total = L_cls + alpha * L_dom
        optimizer.zero_grad()
        L_total.backward()
        optimizer.step()
```

This pseudocode summarizes the core idea implemented in the `train_dann_one_epoch` function and highlights the interaction between the feature extractor, label classifier, domain classifier, and gradient reversal layer.