Guide to Semi-Supervised Learning

Minor in AI - IIT ROPAR

7th April, 2025

Diagnosing Disease with Limited Expert Annotations

Imagine a hospital radiology department in a bustling urban city. Every day, hundreds of MRI scans arrive—scans of the brain, spine, and other critical organs. These images hold life-changing insights—evidence of tumors, lesions, strokes—but understanding them correctly requires the expertise of seasoned radiologists.

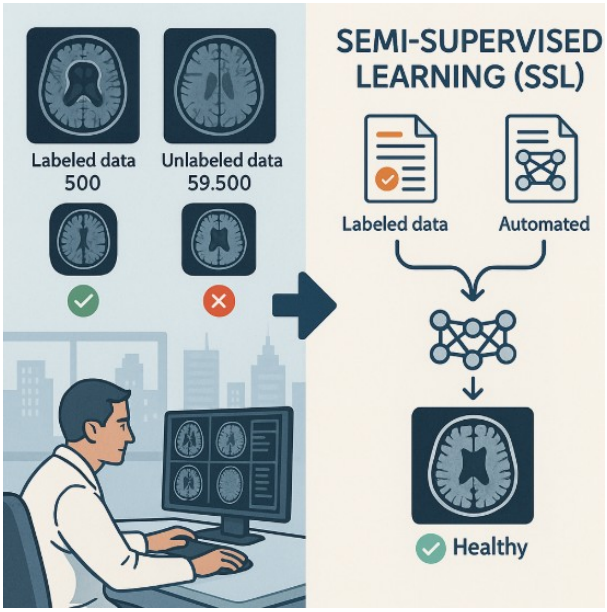
Unfortunately, there are only a few radiologists available, and their time is stretched thin across surgery planning, urgent diagnosis, and patient consultations. Out of the 60,000 scans collected over a few months, only about 500 could be meticulously labeled by the experts, identifying whether each scan indicates a healthy or abnormal condition.

Despite having so little labeled data, the hospital's data science team wants to build a deep learning model that can automatically screen incoming scans. Training a supervised model with just 500 labeled examples will almost certainly result in underfitting. However, discarding the remaining 59,500 unlabeled scans—filled with potentially rich patterns—seems like a waste.

This is where **Semi-Supervised Learning (SSL)** comes in.

SSL methods bridge the gap between limited expert-labeled data and abundant unlabeled data. By combining the strengths of both supervised and unsupervised learning, SSL enables the model to learn not just from what has been labeled, but also from the structure and distribution of the vast pool of unlabeled data.

As we proceed through this notebook, we will explore how SSL algorithms—like ladder networks, the Pi model, self-training, co-training, and VAE-based classifiers—can extract knowledge from unlabeled samples and build robust, accurate models even in resource-constrained settings. Just like in our hospital scenario, SSL has the power to make high-quality learning accessible, efficient, and scalable in real-world challenges.



Semi-Supervised Learning (SSL)

Semi-supervised learning (SSL) is a machine learning paradigm that seeks to improve model performance by leveraging both a small amount of labeled data and a large pool of unlabeled data. This is particularly useful in real-world scenarios where obtaining labels is expensive, time-consuming, or requires domain expertise, such as in medical image analysis, legal document classification, or speech recognition. A common example is the classification of MRI scans: medical professionals may label a small subset of images due to the expertise required, while the remaining scans, although unannotated, can still be valuable in training a robust model using SSL techniques.

Core Assumptions in SSL

SSL methods generally rely on a few fundamental assumptions about the structure of the data to effectively leverage unlabeled information:

Smoothness Assumption: This implies that if two points in a high-dimensional space are close to each other, then their corresponding labels should also be similar. This assumption underpins most neighborhood-based algorithms in SSL.

Cluster Assumption: According to this, the data naturally forms clusters, and points that are within the same cluster are likely to belong to the same class. This is used to propagate labels through the data graph in methods like label propagation.

Manifold Assumption: Real-world high-dimensional data often lies on a much lower-dimensional manifold. If the manifold structure can be identified, the model can generalize well even from a small number of labeled examples.

Ladder Network

The ladder network is a deep learning architecture designed to incorporate both supervised and unsupervised learning signals. It does so by combining a standard feedforward network (encoder) with a decoder that reconstructs intermediate representations at each layer. The key idea is not just to get the final prediction right but to ensure that the network learns meaningful features at all layers.

The encoder first takes the input data and adds Gaussian noise to simulate perturbations. As the data passes through the network, it generates a sequence of noisy intermediate outputs (activations). The decoder then attempts to reconstruct the original (clean) intermediate representations using these noisy ones.

The training objective includes two types of loss: a supervised loss, typically a cross-entropy loss computed using the model's output and the true labels for the small labeled dataset, and a reconstruction loss, which is a mean squared error computed between the clean and reconstructed intermediate representations. The total loss is a weighted sum of these two components.

This architecture is especially powerful for image classification tasks where features extracted at various layers carry rich hierarchical representations.

Pi Model

The Pi model, short for π-model, is based on the principle of consistency regularization. The model encourages similar outputs (predictions) for the same input under different perturbations. This enforces that the model is robust to slight changes or noise in the input.

In practice, two different noise samples are added to the same input image, resulting in two different versions of it. These perturbed inputs are passed through the same model, and the output predictions are compared. Since the true label for the input is unknown (unlabeled data), we cannot use standard supervised loss. Instead, we use a mean squared error (MSE) loss between the softmax outputs of the two predictions.

If the input is labeled, standard cross-entropy is computed using the ground truth. The final loss function is a combination of the supervised loss and the unsupervised MSE loss. This forces the model to be confident and consistent in its predictions across noisy versions of the same input.

Self-Training (Pseudo-Labeling)

Self-training is one of the simplest yet effective semi-supervised learning strategies. The central idea is to iteratively improve the model by using its own high-confidence predictions as ground truth labels for unlabeled data.

Initially, a classifier is trained using the small set of labeled data. This classifier is then used to predict labels for the unlabeled data. From these predictions, only those with a confidence score above a specified threshold (e.g., 95%) are selected. These high-confidence predictions are treated as pseudo-labels, and the corresponding data points are added to the labeled dataset. The model is then retrained using this expanded dataset.

This process is repeated over several iterations. Over time, the model becomes more confident and starts labeling more data correctly. However, care must be taken as poor pseudo-labels early on can mislead the training process. It is crucial to use a well-calibrated confidence threshold to mitigate noise in pseudo-labels.

Co-Training

Co-training is another classical SSL approach, especially useful when data can be represented in multiple independent feature spaces or "views." The key idea is that two classifiers are trained separately on different views of the same data, and they iteratively help each other by labeling examples for one another.

For example, consider image data. If the image can be split horizontally into two halves, the left half can be used to train one model and the right half to train another. These two models are trained using the small labeled dataset. Then, each model is used to label the unlabeled data, but only those predictions that are made with high confidence are retained. These pseudo-labeled samples are added to the training set of the opposite model.

This mutual teaching continues iteratively. The assumption here is that the views are conditionally independent given the label and that both views are sufficient to predict the label. Co-training works well when each view alone is a strong predictor.

Dropout as Regularization

Dropout is a widely used regularization technique to prevent overfitting in deep neural networks. The idea is simple: during training, randomly set a fraction of the neural network's activations to zero. This prevents the network from becoming overly reliant on specific neurons and promotes redundancy in feature learning.

By using dropout, a neural network effectively trains an ensemble of subnetworks. During inference, all units are used, and their outputs are scaled accordingly to account for the missing units during training.

In SSL settings, dropout can help models better generalize from a small labeled dataset by preventing co-adaptation of features and making the model robust to missing information.

Applications to NLP

While SSL is most often used in vision-related tasks, it can also be adapted for use in natural language processing (NLP). However, since many NLP tasks are already structured in a way that facilitates self-supervision (e.g., masked language modeling in BERT), the need for traditional SSL is somewhat reduced.

Still, SSL can be employed in NLP by using word embeddings or sentence embeddings as input representations and then applying noise (e.g., word dropout, synonym replacement, etc.) to simulate perturbations. The idea is similar to SSL in vision: force the model to remain consistent in its predictions regardless of small input changes. Moreover, reconstruction losses can be defined at the embedding level instead of pixel level, and consistency losses can be based on cosine similarity or MSE between semantic vectors.

Coding Portion

Step 1: Data Preparation

```
# Transform: Normalize MNIST images
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

This code creates a transformation pipeline. ‘ToTensor()‘ converts the images into PyTorch tensors, and ‘Normalize((0.1307,), (0.3081,))‘ normalizes them using the mean and standard deviation of the MNIST dataset.

```
# Load full MNIST dataset
train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)
```

This loads the training and test datasets from torchvision. The training dataset downloads MNIST and applies the transform.

```
# Create labeled subset (e.g., 100 examples evenly distributed across classes)
num_labels = 100
labels = np.array(train_dataset.targets)
labeled_idx = []

for i in range(10):
    idx = np.where(labels == i)[0][:num_labels // 10]
    labeled_idx.extend(idx)
```

Here, we pick 100 labeled examples, 10 per digit (0-9), for a balanced labeled dataset. We use numpy to extract index positions per digit.

```
unlabeled_idx = list(set(range(len(train_dataset))) - set(labeled_idx))
```

This determines the indices of all the unlabeled data, i.e., the remaining data after the labeled samples.

```
labeled_dataset = Subset(train_dataset, labeled_idx)
unlabeled_dataset = Subset(train_dataset, unlabeled_idx)
```

Create subsets using PyTorch’s ‘Subset‘ to separate labeled and unlabeled data.

```
labeled_loader = DataLoader(labeled_dataset, batch_size=64, shuffle=True)
unlabeled_loader = DataLoader(unlabeled_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)
```

Set up data loaders for training and testing. The labeled loader uses a smaller batch size due to fewer examples.

Gaussian Noise Layer

```
class GaussianNoise(nn.Module):
    def __init__(self, stddev):
        super().__init__()
        self.stddev = stddev
    def forward(self, x):
        if self.training:
            noise = torch.randn_like(x) * self.stddev
            return x + noise
        else:
            return x
```

This class defines a custom module that adds Gaussian noise to the input during training to simulate corruption. ‘torch.randn$_{l}ike(x)$‘$generatesnoisewiththesameshapeas$‘$x$‘.

Encoder and Decoder

```
class Encoder(nn.Module):
    def __init__(self, noise_std):
        super().__init__()
        self.noise = GaussianNoise(noise_std)
        self.fc1 = nn.Linear(784, 1000)
        self.fc2 = nn.Linear(1000, 500)
        self.fc3 = nn.Linear(500, 250)
        self.fc4 = nn.Linear(250, 250)
        self.fc5 = nn.Linear(250, 10)
```

The encoder is a feedforward neural network with 5 fully connected layers. It starts with input of size 784 (flattened MNIST image) and gradually reduces dimensionality to 10.

```
    def forward(self, x):
        z = []
        x = x.view(-1, 784)
        x = self.noise(x)
        z1 = self.fc1(x)
        z.append(z1)
        z2 = self.fc2(F.relu(z1))
        z.append(z2)
        z3 = self.fc3(F.relu(z2))
        z.append(z3)
        z4 = self.fc4(F.relu(z3))
        z.append(z4)
        z5 = self.fc5(F.relu(z4))
        z.append(z5)
        return z
```

The forward pass adds noise to the input, then passes it through each layer with ReLU activations. It stores each intermediate representation (‘z1‘ to ‘z5‘).

```
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(10, 250)
        self.fc2 = nn.Linear(250, 250)
        self.fc3 = nn.Linear(250, 500)
        self.fc4 = nn.Linear(500, 1000)
```

The decoder mirrors the encoder in reverse. Its goal is to reconstruct the internal representations from the topmost layer.

```
    def forward(self, z_corr):
        d1 = self.fc1(z_corr[-1])
        d2 = self.fc2(F.relu(d1))
        d3 = self.fc3(F.relu(d2))
        d4 = self.fc4(F.relu(d3))
        return [d4, d3, d2, d1]
```

Takes in the final noisy representation (‘z5‘) and tries to reconstruct the previous layers in reverse order.

Loss Functions and Training

```
def supervised_loss(output, target):
    return F.cross_entropy(output, target)
```

The standard cross-entropy loss for classification of labeled data.

```
def reconstruction_loss(z_clean, z_recon):
    loss = 0
    for zc, zr in zip(z_clean[:4], z_recon):
        loss += F.mse_loss(zr, zc.detach())
    return loss
```

This unsupervised loss is based on Mean Squared Error between clean and reconstructed intermediate layers (except the final one).

Training One Epoch

```
def train_epoch(encoder, decoder, optimizer, labeled_loader, unlabeled_loader, alpha):
    encoder.train()
    decoder.train()
    for (x_l, y_l), (x_u, _) in zip(labeled_loader, unlabeled_loader):
        x_l, y_l = x_l.to(device), y_l.to(device)
        x_u = x_u.to(device)
```

This function trains both the encoder and decoder for one epoch. It fetches batches from both labeled and unlabeled loaders.

```
        z_corr_l = encoder(x_l)
        z_clean_l = encoder(x_l)
        z_corr_u = encoder(x_u)
        z_clean_u = encoder(x_u)
```

It performs both noisy (corrupted) and clean passes through the encoder for both labeled and unlabeled data.

```
        output = z_corr_l[-1]
        loss_sup = supervised_loss(output, y_l)
```

It uses the final noisy output to calculate supervised classification loss.

```
        recon_l = decoder(z_corr_l)
        recon_u = decoder(z_corr_u)
        loss_unsup = reconstruction_loss(z_clean_l, recon_l) + reconstruction_loss(
            z_clean_u, recon_u)
```

Decodes the noisy activations and compares them to the clean versions using reconstruction loss for both labeled and unlabeled inputs.

```
        loss = loss_sup + alpha * loss_unsup
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Combines both losses into the total loss, then runs the backward pass and updates model parameters.

Evaluation

```
def evaluate(encoder, loader):
    encoder.eval()
    correct = 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            output = encoder(x)[-1]
            pred = output.argmax(dim=1)
            correct += (pred == y).sum().item()
    return correct / len(loader.dataset)
```

This function evaluates classification accuracy on a data loader. It disables gradient calculation for efficiency.

Training Loop

```
encoder = Encoder(noise_std=0.3).to(device)
decoder = Decoder().to(device)
optimizer = optim.Adam(list(encoder.parameters()) + list(decoder.parameters()), lr=1e-3)

epochs = 30
alpha = 0.5

for epoch in range(1, epochs + 1):
    train_epoch(encoder, decoder, optimizer, labeled_loader, unlabeled_loader, alpha)
```

```
    test_acc = evaluate(encoder, test_loader)
    labeled_acc = evaluate(encoder, labeled_loader)
    unlabeled_acc = evaluate(encoder, DataLoader(unlabeled_dataset, batch_size=256))
    print(f"Epoch {epoch:02d} | Test Acc: {test_acc:.4f} | Labeled Acc: {labeled_acc:.4f} | Unlabeled Pseudo Acc: {unlabeled_acc:.4f}")
```

Finally, this block trains the model over 30 epochs and prints test, labeled, and pseudo-labeled accuracies.

VAE-based Semi-Supervised Learning (SSL)

This section of the code implements a VAE (Variational Autoencoder)-based semi-supervised learning (SSL) approach for the MNIST dataset using PyTorch. The key idea is to use both labeled and unlabeled data to learn a latent representation that not only helps reconstruct input data but also classifies digits accurately. The entire workflow is broken down into steps for clarity.

Step 1: Data Preparation

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
import numpy as np
from torchvision import datasets, transforms

# Setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(0)

dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)

def split_labeled_unlabeled(dataset, num_labels=100):
    targets = np.array(dataset.targets)
    labeled_idx = []
    for i in range(10):
        idx = np.where(targets == i)[0][:num_labels // 10]
        labeled_idx.extend(idx)
    unlabeled_idx = list(set(range(len(dataset))) - set(labeled_idx))
    return Subset(dataset, labeled_idx), Subset(dataset, unlabeled_idx)

test_dataset = datasets.MNIST('./data', train=False, transform=transform)
labeled_dataset, unlabeled_dataset = split_labeled_unlabeled(train_dataset)
labeled_loader = DataLoader(labeled_dataset, batch_size=64, shuffle=True)
unlabeled_loader = DataLoader(unlabeled_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)
```

We begin by importing necessary libraries. The MNIST dataset is loaded and split into a small labeled set (100 examples, 10 per class) and a large unlabeled set. These subsets are wrapped in PyTorch DataLoaders.

Step 2: VAE Components and Classifier

```
def one_hot(y, num_classes=10):
    return F.one_hot(y, num_classes=num_classes).float()
```

Defines a helper function to convert class labels into one-hot encoded vectors.

```
class Encoder(nn.Module):
    def __init__(self, latent_dim=20, n_classes=10):
        super().__init__()
        self.fc1 = nn.Linear(784 + n_classes, 400)
        self.fc_mu = nn.Linear(400, latent_dim)
        self.fc_logvar = nn.Linear(400, latent_dim)

    def forward(self, x, y):
        xy = torch.cat([x, y], dim=1)
        h = F.relu(self.fc1(xy))
        return self.fc_mu(h), self.fc_logvar(h)
```

The encoder takes both input image and class label (in one-hot form) and outputs the mean and log-variance for the latent space. The label conditioning enables class-specific latent representations.

```
class Decoder(nn.Module):
    def __init__(self, latent_dim=20, n_classes=10):
        super().__init__()
        self.fc1 = nn.Linear(latent_dim + n_classes, 400)
        self.fc2 = nn.Linear(400, 784)

    def forward(self, z, y):
        zy = torch.cat([z, y], dim=1)
        h = F.relu(self.fc1(zy))
        return torch.sigmoid(self.fc2(h))
```

The decoder reconstructs the input from latent vector ‘z‘ and the one-hot label ‘y‘.

```
class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(784, 10)

    def forward(self, x):
        return F.log_softmax(self.fc(x), dim=1)
```

A simple classifier maps input images to class probabilities using a single linear layer followed by log-softmax.

Step 3: VAE-SSL Model

```
def kl_divergence(mu, logvar):
    return -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1)

def reconstruction_loss(x_recon, x_true):
    x_recon = torch.clamp(x_recon, 1e-6, 1 - 1e-6)
    x_true = torch.clamp(x_true, 0, 1)
    assert x_recon.shape == x_true.shape
    return F.binary_cross_entropy(x_recon, x_true, reduction='none').sum(dim=1)
```

These functions compute VAE loss components: KL divergence and reconstruction loss. Clamping is used for numerical stability.

```
class VAESSL(nn.Module):
    def __init__(self, latent_dim=20, n_classes=10):
        super().__init__()
        self.encoder = Encoder(latent_dim, n_classes)
        self.decoder = Decoder(latent_dim, n_classes)
        self.classifier = Classifier()
        self.n_classes = n_classes
        self.latent_dim = latent_dim
```

Initializes the complete semi-supervised VAE with encoder, decoder, and classifier.

```
    def forward_labeled(self, x, y):
        mu, logvar = self.encoder(x, y)
        std = torch.exp(0.5 * logvar)
        z = mu + std * torch.randn_like(std)
        x_recon = self.decoder(z, y)
        return x_recon, mu, logvar
```

Generates latent vector ‘z‘ using the reparameterization trick and reconstructs the input for labeled data.

```
    def forward_unlabeled(self, x):
        log_probs = self.classifier(x)
        probs = log_probs.exp()
        total_loss = 0
        for i in range(self.n_classes):
            y_i = torch.eye(self.n_classes)[i].repeat(x.size(0), 1).to(device)
            mu, logvar = self.encoder(x, y_i)
            std = torch.exp(0.5 * logvar)
            z = mu + std * torch.randn_like(std)
```

```
            x_recon = self.decoder(z, y_i)
            recon = reconstruction_loss(x_recon, x)
            kl = kl_divergence(mu, logvar)
            elbo = recon + kl - log_probs[:, i]
            total_loss += probs[:, i] * elbo
        return total_loss.mean()
```

For unlabeled data, the model evaluates the expected reconstruction and KL divergence losses over all possible class labels (weighted by predicted probabilities).

Step 4: Training and Evaluation

```
model = VAESSL().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

Creates the model and optimizer.

```
def train_epoch(model, labeled_loader, unlabeled_loader, alpha=0.1, beta=1.0):
    model.train()
    total_labeled_loss = 0
    total_unlabeled_loss = 0

    for (x_l, y_l), (x_u, _) in zip(labeled_loader, unlabeled_loader):
        x_l = x_l.view(-1, 784).to(device)
        x_u = x_u.view(-1, 784).to(device)
        y_l = y_l.to(device)
        y_l_1h = one_hot(y_l).to(device)

        x_recon, mu, logvar = model.forward_labeled(x_l, y_l_1h)
        recon_loss = reconstruction_loss(x_recon, x_l)
        kl_loss = kl_divergence(mu, logvar)
        labeled_vae_loss = (recon_loss + kl_loss).mean()

        class_pred = model.classifier(x_l)
        class_loss = F.nll_loss(class_pred, y_l)

        unlabeled_loss = model.forward_unlabeled(x_u)

        loss = labeled_vae_loss + alpha * unlabeled_loss + beta * class_loss

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_labeled_loss += (labeled_vae_loss + beta * class_loss).item()
        total_unlabeled_loss += unlabeled_loss.item()

    return total_labeled_loss, total_unlabeled_loss
```

This function trains the model for one epoch. It computes labeled VAE loss, supervised classification loss, and unsupervised loss, then backpropagates the total loss.

```
def evaluate_classifier(model, loader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for x, y in loader:
            x = x.view(-1, 784).to(device)
            y = y.to(device)
            pred = model.classifier(x).argmax(1)
            correct += (pred == y).sum().item()
    return correct / len(loader.dataset)
```

Evaluates classifier accuracy.

Step 5: Training Loop

```
for epoch in range(1, 21):
    l_loss, u_loss = train_epoch(model, labeled_loader, unlabeled_loader, alpha=0.1)
    acc = evaluate_classifier(model, test_loader)
```

```
    print(f"Epoch {epoch:02d} | Test Accuracy: {acc:.4f} | Labeled Loss: {l_loss:.2f} |
        Unlabeled Loss: {u_loss:.2f}")
```

Trains the model for 20 epochs and prints out test accuracy, labeled loss, and unlabeled loss.

Key Takeaways

- **Semi-Supervised Learning (SSL)** bridges the gap between limited labeled data and abundant unlabeled data, making it especially valuable in domains like healthcare where annotations are expensive.
- The ladder network combines supervised and unsupervised losses by adding noise to inputs and reconstructing intermediate representations, ensuring the model learns robust features across all layers.
- The π-model introduces the concept of consistency regularization, where the model is trained to make consistent predictions for the same input under different noise conditions.
- **Self-training** uses the model's own high-confidence predictions as pseudo-labels to iteratively expand the labeled dataset and improve generalization.
- **Co-training** leverages multiple independent views of the data, where two models teach each other by labeling examples for their counterpart, resulting in mutual improvement.
- Dropout serves as an essential regularization method to prevent overfitting, especially when labeled data is scarce.
- The **VAE-SSL model** integrates a variational autoencoder (VAE) with classification, utilizing class-conditional latent variables for improved reconstruction and classification performance on both labeled and unlabeled data.
- Loss components in VAE-SSL include:
  - Reconstruction loss (binary cross-entropy) to measure how well inputs are reconstructed.
  - KL-divergence to regularize the latent space.
  - Classification loss (cross-entropy) for supervised learning on labeled data.
- The combined training objective allows the model to benefit from both labeled and unlabeled examples, improving performance with minimal labeled supervision.
- Practical implementation requires careful balancing of supervised and unsupervised losses using hyperparameters like α and β.