Variational Autoencoders for SSLs

Minor in AI - IIT ROPAR

8th April, 2025

Dr. Ada and the Case of the Missing Labels

In a hospital lab tucked behind stacks of research papers and humming MRI machines, Dr. Ada — a machine learning researcher with a deep love for puzzles — faced a familiar challenge.

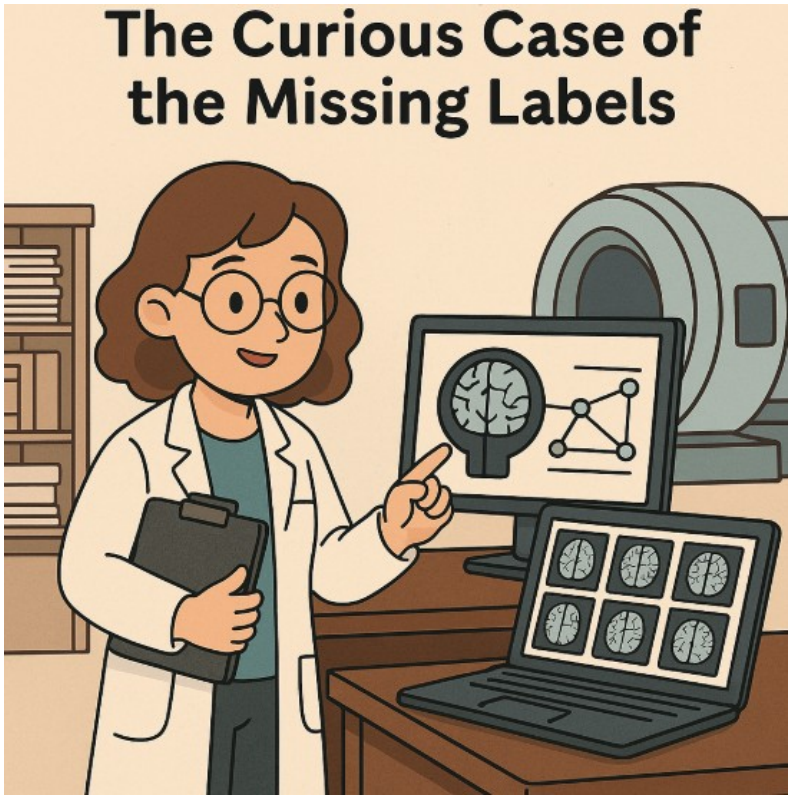
She had been given a task: train a model to detect brain tumors from MRI scans. A hundred labeled images sat on her drive, each one carefully annotated by medical experts. But beside them lay over ten thousand unlabeled scans — untouched, unexplored, and full of hidden patterns.

Labeling all of them would take months. Each annotation required time, expertise, and funding the lab didn't have. But ignoring them felt like throwing away treasure.

As she explored solutions, she came across an idea that struck a balance: Semi-Supervised Learning. It didn't demand every image to be labeled. Instead, it learned from both the few labeled examples and the many unlabeled ones, using structure and probability to fill in the gaps.

Dr. Ada designed a model that used the labeled scans as guides, and the unlabeled ones as students. With each iteration, the model refined its guesses, slowly improving its ability to recognize tumors — even in data it had never truly "seen."

She didn't need to label every image. She just had to let the model learn from what it had — both the known and the unknown.



Approaches in Semi-Supervised Learning

Several methods have been developed to bridge the gap between the labeled and unlabeled worlds. Let's look at some foundational strategies.

Ladder Networks

Ladder networks are based on autoencoders, which aim to compress data into a compact representation and then reconstruct it. In a ladder network, noise is intentionally added to the input, which forces the model to learn how to denoise and reconstruct the data properly.

What makes ladder networks special is that the reconstruction isn't just done at the final layer — it's also done at intermediate layers. These reconstructions are compared and aligned, helping the model learn structure throughout its architecture.

Pi Model (Π-Model)

The Π-model introduces a clever trick: it feeds two slightly different noisy versions of the same input into the model and expects the predictions to match. The idea is that small perturbations shouldn't change what the model thinks — a concept known as *consistency regularization*.

This approach makes the model robust and encourages it to produce stable outputs even when the input isn't perfect.

Pseudo-Labeling Techniques

This category includes several techniques that pretend to label the unlabeled data using the model's own predictions.

- **Self-Training**: Train a model on the small labeled dataset, then use it to predict labels for the unlabeled data. If the model is confident in its prediction, that data point is added to the training set with the predicted label. This process is repeated, expanding the labeled dataset over time.
- **Co-Training**: This assumes that the features can be split into two different "views." Two separate models are trained on each view and label data for one another, based on high-confidence predictions.
- **Co-Teaching**: Two networks are trained in parallel. Each model selects samples with the lowest loss (least noisy) and shares them with the other model for training. This mutual filtering helps with label noise.

Enter Variational Autoencoders (VAEs)

While traditional autoencoders learn to compress and reconstruct data, **Variational Autoencoders (VAEs)** bring a twist — they model the latent representation as a probability distribution.

The main idea is not just to reconstruct data, but to learn the *generative process* behind the data. This allows the model to generate new, similar samples by sampling from the learned distribution.

Intuition Behind VAEs

Imagine an encoder that compresses an input image (say, of a dog) into a summary z. The decoder then reconstructs the image from this summary. But unlike a regular autoencoder, VAEs treat z as a random variable — a point sampled from a normal distribution parameterized by a mean μ and variance σ^2.

This means the same image might map to slightly different z vectors each time, and the decoder can learn to reconstruct slightly different outputs from them. Over time, the model learns to generate diverse and realistic outputs.

VAE Loss Function

To train a VAE, the objective combines two components:

1. **Reconstruction Loss:** Measures how close the reconstructed output $\hat{x}$ is to the original input x. Common choices include mean squared error (MSE) or binary cross-entropy.
2. **KL Divergence:** Denoted $\mathrm{KL}(q(z|x) \parallel p(z))$, this term forces the learned latent distribution $q(z|x)$ to be close to a prior distribution, usually $p(z) = \mathcal{N}(0, I)$.

The total loss is:

$$\mathcal{L} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - \mathrm{KL}(q(z|x) \parallel p(z))$$

Why Use a Distribution?

By modeling z as a distribution rather than a fixed vector, the decoder can sample an infinite number of latent codes and generate infinite variants of the data.

For example, once a VAE is trained on handwritten digits (MNIST), you can sample a $z \sim \mathcal{N}(0, 1)$ and decode it to produce a new digit image — even if it was not part of the original training data.

Diffusion Models: A Powerful Evolution

VAEs work in a single jump: $x \to z \to \hat{x}$. But this one-shot approach can struggle to capture complex details in high-quality data like realistic images.

Diffusion models solve this by taking a multistep approach. During training, noise is gradually added to the input data across many steps until the image becomes pure noise. Then, the model learns to reverse this noise — step-by-step — to reconstruct the original image.

This "denoising" process leads to much sharper, more detailed, and realistic outputs. Tools like **DALL·E 3** and **Stable Diffusion** are based on this concept.

Why Are They Called Denoising Models?

Because the model learns to recover a clean image from noisy inputs. The training phase involves adding noise, and the generation phase is the reverse: removing that noise gradually to form coherent outputs.

VAEs in Semi-Supervised Learning (SSL)

To apply VAEs in an SSL setting, we extend the VAE by adding a **classifier** to the architecture. Now, in addition to reconstructing the input, the model also predicts a class label y.

Architecture Overview

- The input x is passed through the encoder, yielding a latent variable z.
- z is used in two branches:
  - A decoder reconstructs x as $\hat{x}$.
  - A classifier predicts the label $\hat{y}$.

Training Strategy

For labeled data: Use all three components of loss:

1. Reconstruction loss ($x \to z \to \hat{x}$)
2. KL divergence ($z \sim \mathcal{N}(0, 1)$)
3. Classification loss ($z \to \hat{y}$)

For unlabeled data:

- Predict $\hat{y}$ from z.
- If confident, use $\hat{y}$ as a pseudo-label and compute classification loss.

Example: MNIST Digits

Assume you have:

- 500 labeled digit images (0–9)
- 50,000 unlabeled digit images

Train the VAE + classifier with labeled data. For unlabeled data, let the model assign a pseudo-label if it's confident. This helps the classifier generalize better and allows the decoder to generate new digits by sampling new z vectors.

Applications of VAE + SSL

- **Medical Imaging:** Train with a small labeled dataset and a large pool of unlabeled MRI or X-ray scans. Also useful for generating synthetic scans for augmentation.
- **Autonomous Driving:** Use unlabeled video frames to learn useful representations for pedestrians, traffic signs, etc.
- **Signature and Handwriting Analysis:** VAEs can learn from a few signature samples and generate realistic variations for verification systems.
- **Face Generation and Deepfakes:** Conditional VAEs can generate faces by age, gender, or expression, based on given labels.

Conditional VAEs: Controlled Generation

Regular VAEs sample z and generate data randomly. **Conditional VAEs (CVAEs)** extend this by also feeding in a label y into the decoder. This way, you can guide the generation process.

Example

To generate a digit "4":

- Sample $z \sim \mathcal{N}(0, 1)$
- Provide $y = 4$
- Pass $[z, y]$ into the decoder
- Output is a synthetic image of digit 4

This is particularly useful in *text-to-image generation*, *controlled synthesis*, and *attribute-guided sampling.*

Coding Portion

1. Imports and Device Setup

```
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
```

These are standard imports used for neural network modeling, optimization, data loading, and transformations.

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

This sets the computation device (GPU if available) and seeds the random number generator for reproducibility.

2. Data Preparation and Splitting

```
transform = transforms.Compose([
    transforms.ToTensor()
])
```

Transforms MNIST images into tensors normalized in $[0, 1]$.

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

Downloads and prepares the MNIST dataset.

Split into Labeled and Unlabeled

```
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)
```

Selects 10 labeled samples per class and creates subsets for labeled and unlabeled data.

```
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)
```

DataLoaders are created with appropriate batch sizes.

3. Model Architecture

One-Hot Encoding

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

Converts class labels into one-hot vectors.

Encoder Network

```
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)
```

The encoder maps image + label to a hidden vector, then outputs μ and $\log \sigma^2$.

```
    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)
```

Concatenates the image vector and the one-hot label, then processes through hidden layers.

Decoder Network

```
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))
```

Takes latent vector z and label y and reconstructs image through two layers.

Classifier

```
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 basic feedforward classifier using log-softmax.

4. Loss Functions

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

KL divergence between learned posterior and standard normal.

```
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)
```

Binary cross-entropy reconstruction loss for each pixel.

5. Full VAE-SSL Model

```
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
```

Combines encoder, decoder, and classifier.

Forward Pass for Labeled Data

```
    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
```

Standard VAE reparameterization trick and decoding.

Forward Pass for Unlabeled 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()
```

Computes expected ELBO under all class hypotheses weighted by predicted probabilities.

6. Training and Evaluation

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

Model initialization and optimizer setup.

Training for One Epoch

```
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)

        # Labeled VAE loss
        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()

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

        # Unlabeled VAE loss
```

```
        unlabeled_loss = model.forward_unlabeled(x_u)

        # Total loss
        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
```

Trains on both labeled and unlabeled data. Combines:

- Labeled VAE loss (reconstruction + KL)
- Supervised classification loss
- Unlabeled VAE loss (expected ELBO)

Evaluation

```
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)
```

Computes test accuracy of the classifier head.

7. 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 test accuracy, labeled and unlabeled losses.
The output should look like this:

```
100% ████████████ 9.91M/9.91M [00:00<00:00, 13.0MB/s]
100% ████████████ 28.9k/28.9k [00:00<00:00, 350kB/s]
100% ████████████ 1.65M/1.65M [00:00<00:00, 3.17MB/s]
100% ████████████ 4.54k/4.54k [00:00<00:00, 8.92MB/s]
Epoch 01 | Test Accuracy: 0.1742 | Labeled Loss: 1078.33 | Unlabeled Loss: 1080.25
Epoch 02 | Test Accuracy: 0.2821 | Labeled Loss: 986.18 | Unlabeled Loss: 991.26
Epoch 03 | Test Accuracy: 0.3730 | Labeled Loss: 884.88 | Unlabeled Loss: 891.55
Epoch 04 | Test Accuracy: 0.4287 | Labeled Loss: 767.73 | Unlabeled Loss: 773.04
Epoch 05 | Test Accuracy: 0.4709 | Labeled Loss: 645.22 | Unlabeled Loss: 661.49
Epoch 06 | Test Accuracy: 0.5054 | Labeled Loss: 598.14 | Unlabeled Loss: 608.19
Epoch 07 | Test Accuracy: 0.5233 | Labeled Loss: 558.58 | Unlabeled Loss: 585.65
Epoch 08 | Test Accuracy: 0.5420 | Labeled Loss: 512.50 | Unlabeled Loss: 538.79
Epoch 09 | Test Accuracy: 0.5645 | Labeled Loss: 482.95 | Unlabeled Loss: 512.54
Epoch 10 | Test Accuracy: 0.5852 | Labeled Loss: 466.08 | Unlabeled Loss: 495.61
Epoch 11 | Test Accuracy: 0.6055 | Labeled Loss: 462.86 | Unlabeled Loss: 485.65
Epoch 12 | Test Accuracy: 0.6221 | Labeled Loss: 458.06 | Unlabeled Loss: 478.30
Epoch 13 | Test Accuracy: 0.6367 | Labeled Loss: 437.29 | Unlabeled Loss: 471.51
Epoch 14 | Test Accuracy: 0.6488 | Labeled Loss: 433.57 | Unlabeled Loss: 468.42
Epoch 15 | Test Accuracy: 0.6564 | Labeled Loss: 431.34 | Unlabeled Loss: 454.62
Epoch 16 | Test Accuracy: 0.6605 | Labeled Loss: 423.91 | Unlabeled Loss: 467.30
Epoch 17 | Test Accuracy: 0.6651 | Labeled Loss: 419.01 | Unlabeled Loss: 457.65
Epoch 18 | Test Accuracy: 0.6674 | Labeled Loss: 412.33 | Unlabeled Loss: 454.28
Epoch 19 | Test Accuracy: 0.6711 | Labeled Loss: 410.05 | Unlabeled Loss: 456.14
Epoch 20 | Test Accuracy: 0.6739 | Labeled Loss: 409.20 | Unlabeled Loss: 437.79
```

Key Takeaways

- **Semi-Supervised Learning (SSL)** bridges the gap between fully labeled and unlabeled data, enabling efficient learning with minimal annotation effort.
- **VAEs (Variational Autoencoders)** combine reconstruction and generative modeling by learning probabilistic latent representations.
- In SSL, VAEs can be extended with a classifier, allowing the model to both reconstruct inputs and predict labels, even with limited supervision.
- **Unlabeled data** is leveraged by marginalizing over all possible labels, weighted by the model's predicted class probabilities.
- The **loss function** combines three parts: reconstruction loss, KL divergence, and classification loss — balancing generative and discriminative learning.
- **Conditional VAEs (CVAEs)** enhance control over generated outputs by incorporating label information directly into the decoder.
- Applications span multiple domains, including medical imaging, autonomous driving, and handwriting analysis — wherever labeled data is scarce but unlabeled data is abundant.
- Overall, SSL with VAE-based models offers a scalable and flexible framework to utilize unlabeled data effectively without sacrificing performance.