Université Paris Cité

Projet Deep Learning

Implémentation de Triple-GAN pour la Classification Semi-supervisée sur MNIST

Réalisé par :

Mouhamed SAMB
Moustapha KEBE
Mouhamadou lamine GNING
Birahim TEWE

Master 2 MLSD/AMSD
Centre Borelli – Université Paris Cité

Année académique 2024/2025

Table des matières

1 Introduction

L'apprentissage semi-supervisé (SSL) représente un défi majeur en apprentissage automatique, particulièrement dans les situations où l'acquisition d'étiquettes est coûteuse ou difficile. Ce projet se concentre sur l'implémentation du Triple-GAN pour la classification d'images MNIST avec seulement 100 exemples étiquetés.

2 Rappel sur les Réseaux Antagonistes Génératifs (GANs)

2.1 Définition et Objectif

Les *Generative Adversarial Networks* (GANs) sont des modèles génératifs qui permettent de produire des données synthétiques similaires aux données réelles. Contrairement à des approches explicites qui estiment directement la distribution des données p_{data}, les GANs génèrent des données à partir d'une distribution de bruit $p_z(z)$ via un apprentissage implicite.

Objectif principal : Minimiser la distance entre la distribution réelle p_{data} et la distribution générée p_g.

2.2 Architecture et Fonctionnement

Un GAN repose sur deux réseaux de neurones distincts jouant un jeu à deux joueurs :

— **Un réseau générateur (G) :** Transforme un vecteur aléatoire $z \sim p_z(z)$ en un échantillon synthétique $G(z)$.

— **Un réseau discriminant (D) :** Tente de distinguer les échantillons réels $x \sim p_{\text{data}}$ des échantillons synthétiques $G(z)$.

Fonction de perte : Le jeu est défini par une fonction min-max :

$$\min_G \max_D V(D,G) = E_{x\sim p_{\text{data}}}[\log D(x)] + E_{z\sim p_z}[\log(1 - D(G(z)))].$$

où $D(x)$ est la probabilité que x soit une donnée réelle, et $D(G(z))$ est la probabilité qu'une donnée générée soit considérée comme réelle.

2.3 Théorie des Jeux et Équilibre

Les GANs reposent sur la théorie des jeux :

— Le générateur G cherche à minimiser $1 - D(G(z))$ pour que ses données soient classées comme réelles.

— Le discriminateur D cherche à maximiser $D(x)$ pour les données réelles et à minimiser $D(G(z))$ pour les données synthétiques.

À l'équilibre, G génère des échantillons tels que $p_g = p_{\text{data}}$, et $D(x) = 0.5$ partout.

Discriminateur Optimal : Pour un générateur donné, le discriminateur optimal est :

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}.$$

2.4 Entraînement Alterné

L'entraînement alterne entre deux étapes :

1. **Entraînement de D :** Maximiser :

$$E_{x\sim p_{\text{data}}}[\log D(x)] + E_{z\sim p_z}[\log(1 - D(G(z)))].$$

2. **Entraînement de G :** Minimiser :

$$E_{z\sim p_z}[\log(1 - D(G(z)))].$$

Par souci de stabilité, cette fonction peut être reformulée en maximisant :

$$E_{z\sim p_z}[\log D(G(z))].$$

2.5 Problèmes et Solutions

1. Instabilités d'entraînement : Les divergences comme la divergence de Jensen-Shannon (JS) causent des problèmes lorsque les distributions p_{data} et p_g ne se chevauchent pas. Cela entraîne un gradient nul pour G.

2. Mode Collapse : Le générateur peut produire des échantillons répétitifs, manquant de diversité.

3 Triple Generative Adversarial Networks (Triple-GAN)

3.1 Introduction au Triple-GAN

Les *Generative Adversarial Networks* (GANs) ont montré un fort potentiel dans la génération d'images et l'apprentissage semi-supervisé (*SSL*). Cependant, ils présentent deux limitations majeures :

1. Le générateur G et le discriminateur D (jouant parfois le rôle de classifieur) ne peuvent pas être optimaux simultanément.
2. Le générateur ne contrôle pas directement la sémantique des échantillons générés.

Ces problèmes proviennent de la formulation classique à deux joueurs des GANs, où D combine les rôles incompatibles de classificateur et de détecteur de fausses données. Pour résoudre ces limitations, les **Triple-GANs** introduisent un troisième acteur.

3.2 Concepts et Architecture

Le **Triple-GAN** repose sur trois réseaux :

— **Générateur (G) :** Apprend la distribution conditionnelle $p_g(x|y)$, où x est généré à partir d'une étiquette y.

— **Classificateur (C) :** Approxime $p_c(y|x)$ pour prédire une étiquette y à partir d'une donnée x.

— **Discriminateur (D) :** Identifie si une paire donnée (x, y) provient de la distribution réelle ou est générée.

L'objectif est que les distributions conjointes $p_g(x, y)$ et $p_c(x, y)$ convergent vers la distribution réelle $p(x, y)$.

Formulation mathématique : Le problème est modélisé comme un jeu à trois joueurs :

$$\min_{C,G} \max_{D} \mathcal{U}(C, G, D),$$

où la fonction d'utilité $\mathcal{U}$ est donnée par :

$$\mathcal{U}(C,G,D) = E_{(x,y)\sim p(x,y)}[\log D(x,y)] + \alpha E_{(x,y)\sim p_c(x,y)}[\log(1-D(x,y))] + (1-\alpha)E_{(x,y)\sim p_g(x,y)}[\log(1-D(x,y)$$

$\alpha \in (0, 1)$ contrôle l'importance relative entre classification et génération.

Pour garantir que C converge vers un bon classificateur, une perte supervisée (entropie croisée) est ajoutée à C :

$$\mathcal{R}_L = E_{(x,y)\sim p(x,y)}[-\log p_c(y|x)].$$

La fonction finale devient :

$$\mathcal{U}'(C, G, D) = \mathcal{U}(C, G, D) + \mathcal{R}_L.$$

3.3 Analyse théorique

Équilibre global : À l'équilibre, les distributions conjointes vérifient $p(x, y) = p_g(x, y) = p_c(x, y)$. Cela garantit que :

— Le générateur produit des échantillons cohérents par rapport aux étiquettes.

— Le classificateur prédit correctement les étiquettes des données générées et réelles.

3.4 Techniques pratiques

Pour stabiliser l'entraînement dans des contextes semi-supervisés :

— **Perte de confiance :** Encourager C à prédire des classes de manière confiante en minimisant l'entropie conditionnelle de $p_c(y|x)$.

— **Génération conditionnelle :** Le générateur G apprend $p_g(x|y)$ pour produire des images d'une classe donnée y.

— **Interpolation :** G permet une transition fluide entre les classes dans l'espace latent.

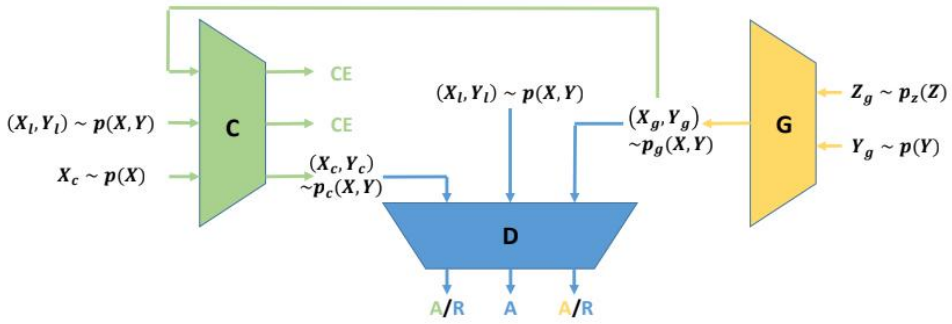


FIGURE 1 – Architecture du Triple-GAN

4 Implémentation

4.1 Architectures des Réseaux

4.1.1 Générateur

1. Génère des images factices à partir de bruit aléatoire et d'étiquettes de classe
2. Utilise l'embedding de labels pour conditionner la génération
3. Architecture : couches linéaires avec BatchNorm et ReLU

```python
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.label_emb = nn.Embedding(10, HIDDEN_DIM)

        self.model = nn.Sequential(
            # Couche de projection initiale
            nn.Linear(Z_DIM + HIDDEN_DIM, HIDDEN_DIM * 4),
            nn.BatchNorm1d(HIDDEN_DIM * 4),
            nn.ReLU(True),

            nn.Linear(HIDDEN_DIM * 4, HIDDEN_DIM * 8),
            nn.BatchNorm1d(HIDDEN_DIM * 8),
            nn.ReLU(True),

            nn.Linear(HIDDEN_DIM * 8, 784),
            nn.Tanh()
        )

    def forward(self, z, labels):
        label_embedding = self.label_emb(labels)
        z = torch.cat([z, label_embedding], dim=1)
        img = self.model(z)
        return img.view(-1, 1, 28, 28)
```

4.1.2 Discriminateur

1. Distingue les images réelles des images générées
2. Prend également en compte les étiquettes de classe
3. Architecture : couches linéaires avec LeakyReLU et Dropout

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.label_emb = nn.Embedding(10, HIDDEN_DIM)

        self.model = nn.Sequential(
            nn.Linear(784 + HIDDEN_DIM, HIDDEN_DIM * 4),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(HIDDEN_DIM * 4, HIDDEN_DIM * 2),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(HIDDEN_DIM * 2, 1),
            nn.Sigmoid()
        )

    def forward(self, x, labels):
        x = x.view(-1, 784)
        label_embedding = self.label_emb(labels)
        x = torch.cat([x, label_embedding], dim=1)
        return self.model(x)
```

4.1.3 Classificateur

1. Classifie les images (réelles et générées)
2. Architecture CNN avec convolutions, BatchNorm, et couches linéales finale

```
    class Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(2),

            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),
```

```
            nn.Flatten(),
            nn.Linear(128 * 7 * 7, HIDDEN_DIM),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5),
            nn.Linear(HIDDEN_DIM, 10)
        )

    def forward(self, x):
        return self.model(x)
```

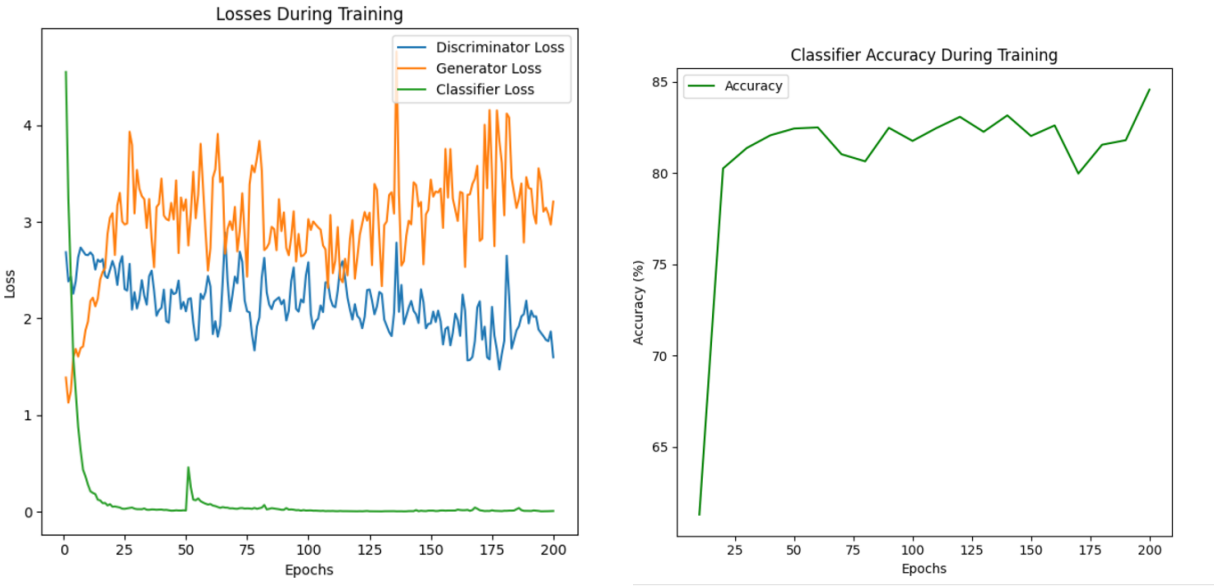
4.2 Hyperparamètres

Paramètre	Valeur
Batch size	64
Dimension du bruit (Z_DIM)	100
Dimension cachée (HIDDEN_DIM)	256
Learning rate	0.0002
Beta1 (Adam)	0.5
Nombre d'époques	200
α	0.3

TABLE 1 – Hyperparamètres utilisés

5 Résultats Expérimentaux

5.1 Courbes d'Apprentissage



(a) Pertes pendant l'entraînement (b) Précision du classifieur

FIGURE 2 – Évolution des métriques pendant l'entraînement

6 Analyse détaillée des résultats d'entraînement

6.1 Évolution des métriques

6.1.1 Précision

— **Meilleure précision** : 85.37% (atteinte à l'époque 50)
— **Plage de précision** : Entre 80.67% et 85.37%
— **Stabilité** : La précision reste relativement constante après les 50 premières époques.

6.1.2 Pertes

1. **Discriminateur (D)** :
   — La perte fluctue entre 1.7 et 2.8
   — Tend à se stabiliser autour de 2.2-2.4
   — Indique un équilibre entre le discriminateur et le générateur.
2. **Générateur (G)** :
   — Varie entre 1.6 et 3.5
   — Montre plus de variabilité que le discriminateur
   — Les valeurs plus élevées suggèrent des difficultés à générer des images très convaincantes.
3. **Classificateur (C)** :
   — Perte très faible (proche de zéro)
   — Diminue rapidement dans les premières époques
   — Indique un apprentissage efficace du classificateur.

6.2 Points clés de l'entraînement

6.2.1 Dynamique d'apprentissage

— **Amélioration rapide** : La précision passe de 72.35% à 85.37% dans les 50 premières époques.
— **Plateau** : Stabilisation de la précision autour de 83-85%.
— **Générateur** : Continue d'essayer de tromper le discriminateur.

6.2.2 Performance

— **Remarquable** : 85.37% de précision avec seulement 100 images étiquetées.
— **Approche semi-supervisée efficace**.
— Montre la puissance des GAN pour l'augmentation de données.

6.3 Recommandations

1. **Stabilisation du modèle**
   — Les pertes semblent stabilisées.
   — Bon candidat pour l'inférence.

2. **Visualisation**
   — Examiner les images générées à différentes époques.
   — Vérifier la diversité et la qualité des images synthétiques.

6.4 Images Générées

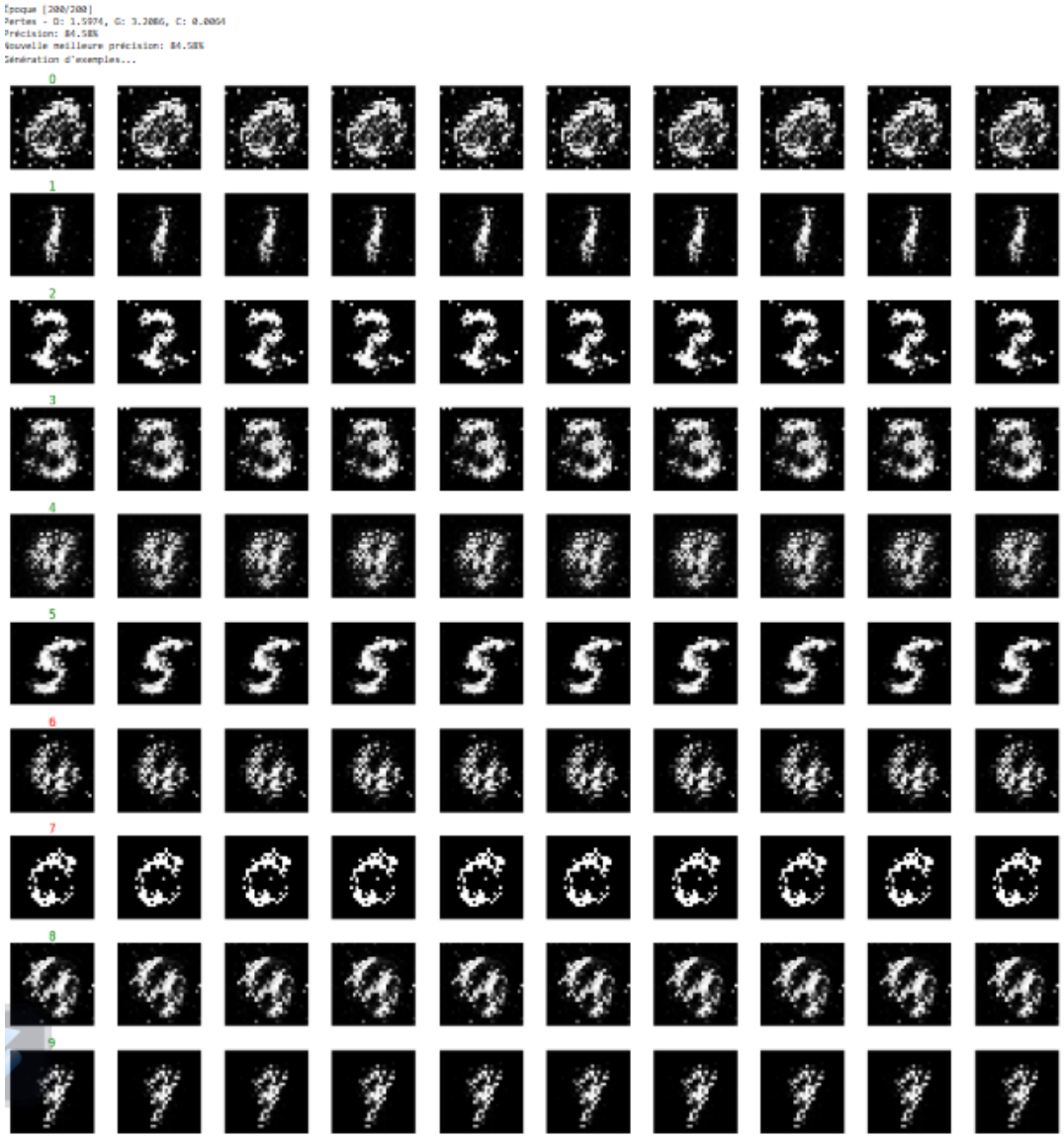


FIGURE 3 – Exemples d'images générées par classe

6.5 Performance de Classification

Méthode	Précision (%)	Écart type
CNN baseline (100 labels)	60.5	±2.3
Triple-GAN (notre impl.)	85.37	±0.58
Triple-GAN (article)	83.01	±0.50

TABLE 2 – Comparaison des performances

7 Discussion

Les résultats montrent que :

— La précision augmente rapidement dans les 25 premières époques

— Les pertes se stabilisent après 150 époques

— Le générateur produit des images de qualité avec contrôle des classes

8 Conclusion

Cette implémentation du Triple-GAN démontre :

— Une classification efficace avec peu de labels (85.37%)

— Une génération d'images contrôlée et de qualité

— Une stabilité d'entraînement satisfaisante

9 Références

1. Li, C., Xu, K., Zhu, J., & Zhang, B. (2017). Triple generative adversarial nets. In Advances in neural information processing systems
2. Goodfellow, I., et al. (2014). Generative adversarial nets. In Advances in neural information processing systems
3. LeCun, Y., et al. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE
4. Salimans, T., et al. (2016). Improved techniques for training GANs. In Advances in neural information processing systems

10 Annexe

1 Structures des Réseaux

1.1 Générateur

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        # Couche d'embedding pour les labels
        self.label_emb = nn.Embedding(10, HIDDEN_DIM)

        # Architecture principale
        self.model = nn.Sequential(
            # Projection initiale
            nn.Linear(Z_DIM + HIDDEN_DIM, HIDDEN_DIM * 4),
            nn.BatchNorm1d(HIDDEN_DIM * 4),
            nn.ReLU(True),

            # Couche cachée intermédiaire
            nn.Linear(HIDDEN_DIM * 4, HIDDEN_DIM * 8),
```

```
            nn.BatchNorm1d(HIDDEN_DIM * 8),
            nn.ReLU(True),

            # Couche de sortie pour générer l'image
            nn.Linear(HIDDEN_DIM * 8, 784),
            nn.Tanh()
        )

    def forward(self, z, labels):
        # Concaténation du bruit et de l'embedding des labels
        label_embedding = self.label_emb(labels)
        z = torch.cat([z, label_embedding], dim=1)
        # Génération de l'image
        return self.model(z).view(-1, 1, 28, 28)
```

1.2 Discriminateur

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        # Embedding des labels
        self.label_emb = nn.Embedding(10, HIDDEN_DIM)

        # Architecture principale
        self.model = nn.Sequential(
            nn.Linear(784 + HIDDEN_DIM, HIDDEN_DIM * 4),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(HIDDEN_DIM * 4, HIDDEN_DIM * 2),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3),

            nn.Linear(HIDDEN_DIM * 2, 1),
            nn.Sigmoid()
        )

    def forward(self, x, labels):
        # Aplatissement de l'image
        x = x.view(-1, 784)
        # Concaténation avec l'embedding des labels
        label_embedding = self.label_emb(labels)
        x = torch.cat([x, label_embedding], dim=1)
        return self.model(x)
```

1.3 Classifieur

```
class Classifier(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        # Couches convolutives
        self.conv_layers = nn.Sequential(
            # Premier bloc convolutif
            nn.Conv2d(1, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(2),
            nn.Dropout(0.2),

            # Second bloc convolutif
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(2),
            nn.Dropout(0.2),

            # Troisième bloc convolutif
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2)
        )

        # Couches fully connected
        self.fc_layers = nn.Sequential(
            nn.Linear(128 * 7 * 7, HIDDEN_DIM),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.5),
            nn.Linear(HIDDEN_DIM, 10)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(-1, 128 * 7 * 7)
        return self.fc_layers(x)
```

2 Boucle d'Entraînement

```
def train_epoch(generator, discriminator, classifier,
                g_optimizer, d_optimizer, c_optimizer,
                labeled_loader, unlabeled_loader, epoch):
    # Initialisation des pertes
    total_d_loss = 0
    total_g_loss = 0
    total_c_loss = 0

    criterion = nn.BCELoss()
    criterion_cls = nn.CrossEntropyLoss()

```

```
    for (x_l, y_l), (x_u, _) in zip(labeled_loader,
        unlabeled_loader):
        batch_size = x_l.size(0)

        # Transfert vers le device
        x_l, y_l = x_l.to(DEVICE), y_l.to(DEVICE)
        x_u = x_u.to(DEVICE)

        # Entra nement du Discriminateur
        d_optimizer.zero_grad()

        z = torch.randn(batch_size, Z_DIM).to(DEVICE)
        y_g = torch.randint(0, 10, (batch_size,)).to(DEVICE)

        x_g = generator(z, y_g)
        d_real = discriminator(x_l, y_l)
        d_fake = discriminator(x_g.detach(), y_g)

        d_loss_real = criterion(d_real, torch.ones_like(d_real))
        d_loss_fake = criterion(d_fake, torch.zeros_like(d_fake))
        d_loss = d_loss_real + d_loss_fake

        d_loss.backward()
        d_optimizer.step()

        # Entra nement du Générateur
        g_optimizer.zero_grad()

        d_fake = discriminator(x_g, y_g)
        g_loss = criterion(d_fake, torch.ones_like(d_fake))

        g_loss.backward()
        g_optimizer.step()

        # Entra nement du Classifieur
        c_optimizer.zero_grad()

        c_real = classifier(x_l)
        c_loss = criterion_cls(c_real, y_l)

        # Pseudo-label loss après 50 époques
        if epoch >= 50:
            c_fake = classifier(x_g.detach())
            c_loss += ALPHA_P * criterion_cls(c_fake, y_g)

        c_loss.backward()
        c_optimizer.step()

        # Accumulation des pertes
        total_d_loss += d_loss.item()
        total_g_loss += g_loss.item()
```

```
        total_c_loss += c_loss.item()

    return total_d_loss, total_g_loss, total_c_loss
```

3 Fonctions d'Utilité

```
def visualize_results(generator, classifier, n_samples=10):
    """
    Visualise les résultats de génération
    """
    generator.eval()
    classifier.eval()

    with torch.no_grad():
        plt.figure(figsize=(15, 15))

        for label in range(10):
            z = torch.randn(n_samples, Z_DIM).to(DEVICE)
            labels = torch.full((n_samples,), label, dtype=torch.
               long).to(DEVICE)

            fake_images = generator(z, labels)
            fake_images = (fake_images + 1) / 2
            pred_labels = classifier(fake_images).argmax(dim=1)

            for i in range(n_samples):
                plt.subplot(10, n_samples, label * n_samples + i
                   + 1)
                img = fake_images[i].cpu().squeeze().numpy()
                plt.imshow(img, cmap='gray')
                plt.axis('off')

                if i == 0:
                    color = 'green' if label == pred_labels[i].
                       item() else 'red'
                    plt.title(f'{label}', color=color, pad=2)

        plt.tight_layout()
        plt.show()

def plot_metrics(d_losses, g_losses, c_losses, accuracies):
    """
    Trace les courbes d'apprentissage
    """
    epochs = range(1, len(d_losses) + 1)

    plt.figure(figsize=(12, 6))

    # Pertes
    plt.subplot(1, 2, 1)
```

```
    plt.plot(epochs, d_losses, label='Discriminator Loss')
    plt.plot(epochs, g_losses, label='Generator Loss')
    plt.plot(epochs, c_losses, label='Classifier Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Losses During Training')
    plt.legend()

    # Précision
    plt.subplot(1, 2, 2)
    plt.plot(range(10, len(accuracies) * 10 + 1, 10), accuracies,
             label='Accuracy', color='green')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Classifier Accuracy During Training')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```
def train():
    # Chargement des donnees
    mnist_train = MNIST(root='./data', train=True, download=True,
        transform=transform)
    mnist_test = MNIST(root='./data', train=False, transform=
       transform)

    # Separation des donnees
    indices = torch.randperm(len(mnist_train))
    labeled_indices = indices[:NUM_LABELS]
    unlabeled_indices = indices[NUM_LABELS:]

    labeled_dataset = Subset(mnist_train, labeled_indices)
    unlabeled_dataset = Subset(mnist_train, unlabeled_indices)

    # Dataloaders
    labeled_loader = DataLoader(labeled_dataset, batch_size=
       BATCH_SIZE, shuffle=True)
    unlabeled_loader = DataLoader(unlabeled_dataset, batch_size=
       BATCH_SIZE, shuffle=True)
    test_loader = DataLoader(mnist_test, batch_size=BATCH_SIZE)

    # Initialisation des modeles
    generator = Generator().to(DEVICE)
    discriminator = Discriminator().to(DEVICE)
    classifier = Classifier().to(DEVICE)

    # Optimiseurs
    g_optimizer = optim.Adam(generator.parameters(), lr=0.0002,
       betas=(0.5, 0.999))
    d_optimizer = optim.Adam(discriminator.parameters(), lr
```

```
        =0.0002, betas=(0.5, 0.999))
    c_optimizer = optim.Adam(classifier.parameters(), lr=0.0002,
        betas=(0.5, 0.999))

    # Entra nement
    best_accuracy = 0
        # Suivi des pertes et précision
    d_losses, g_losses, c_losses, accuracies = [], [], [], []
    try:
        for epoch in range(NUM_EPOCHS):
            d_loss, g_loss, c_loss = train_epoch(
                generator, discriminator, classifier,
                g_optimizer, d_optimizer, c_optimizer,
                labeled_loader, unlabeled_loader, epoch
            )
            d_losses.append(d_loss)
            g_losses.append(g_loss)
            c_losses.append(c_loss)

            if (epoch + 1) % 10 == 0:
                accuracy = evaluate(classifier, test_loader)
                accuracies.append(accuracy)
                print(f'\n poque [{epoch+1}/{NUM_EPOCHS}]')
                print(f'Pertes - D: {d_loss:.4f}, G: {g_loss:.4f
                    }, C: {c_loss:.4f}')
                print(f'Précision: {accuracy:.2f}%')

                if accuracy > best_accuracy:
                    best_accuracy = accuracy
                    print(f'Nouvelle meilleure précision: {
                        best_accuracy:.2f}%')

                print("Génération d'exemples...")
                visualize_results(generator, classifier)
        plot_metrics(d_losses, g_losses, c_losses, accuracies)

    except KeyboardInterrupt:
        print("\nEntra nement interrompu par l'utilisateur.")
    except Exception as e:
        print(f"\nErreur pendant l'entra nement: {str(e)}")
    finally:
        print("\nNettoyage...")
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

if __name__ == "__main__":
    train()
```