# TP 4.2 - VAE conditionnel et PixelCNN

In [ ]:
```
nom='jai'
prenom='ilyass'
```

In [1]:
```
# Import des bibliothèques utiles
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
```

In [2]:
```
from torchvision.transforms import ToTensor, ToPILImage

from torch.utils.data import DataLoader
from torchvision.datasets import MNIST


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

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [WinError 10061] Aucune connexion n'a pu être établie car l'ordina
teur cible l'a expressément refusée>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyt
e.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyt
e.gz to ./data/MNIST\MNIST\raw\train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:04<00:00, 2305868.28it/s]
Extracting ./data/MNIST\MNIST\raw\train-images-idx3-ubyte.gz to ./data/MNIST\MNIS
T\raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [WinError 10061] Aucune connexion n'a pu être établie car l'ordina
teur cible l'a expressément refusée>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyt
e.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyt
e.gz to ./data/MNIST\MNIST\raw\train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 365344.14it/s]
```

```
Extracting ./data/MNIST\MNIST\raw\train-labels-idx1-ubyte.gz to ./data/MNIST\MNIS
T\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [WinError 10061] Aucune connexion n’a pu être établie car l’ordina
teur cible l’a expressément refusée>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.
gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.
gz to ./data/MNIST\MNIST\raw\t10k-images-idx3-ubyte.gz
100%|██████████| 1648877/1648877 [00:01<00:00, 1527044.84it/s]
Extracting ./data/MNIST\MNIST\raw\t10k-images-idx3-ubyte.gz to ./data/MNIST\MNIST
\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [WinError 10061] Aucune connexion n’a pu être établie car l’ordina
teur cible l’a expressément refusée>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.
gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.
gz to ./data/MNIST\MNIST\raw\t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 4944336.56it/s]
Extracting ./data/MNIST\MNIST\raw\t10k-labels-idx1-ubyte.gz to ./data/MNIST\MNIST
\raw
```

In [8]:

```python
num_classes = 10
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [23]:

```python
class Encoder(nn.Module):
    def __init__(self, latent_dim, num_classes=10):
        super(Encoder, self).__init__()

        # Feature extractor (same as in the non-conditional VAE)
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=4, stride=2, p
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2,
            nn.ReLU()
        )

        # After two Conv2d(…, stride=2), for a 28×28 input, the feature map size
        # => Flattened dimension = 64 * 7 * 7 = 3136
        # We also concatenate the conditioning vector c (length = num_classes)
        in_features = 64 * 7 * 7 + num_classes

        self.fc_mu = nn.Linear(in_features, latent_dim)
        self.fc_logvar = nn.Linear(in_features, latent_dim)

    def forward(self, x, c):
        """
        x: input images of shape (B, 1, 28, 28)
        c: one-hot labels of shape (B, num_classes)
        """
        # Pass through the convolutional encoder
```

```
        x = self.encoder(x)                  # (B, 64, 7, 7)
        x = x.view(x.size(0), -1)            # (B, 64*7*7) -> (B, 3136)

        # Concatenate conditioning
        # c has shape (B, 10) for MNIST => total (B, 3136 + 10)
        x = torch.cat([x, c], dim=1)

        # Compute mu and logvar
        x_mu = self.fc_mu(x)                 # (B, latent_dim)
        x_logvar = self.fc_logvar(x)         # (B, latent_dim)

        return x_mu, x_logvar
```

In [24]:
```
class Decoder(nn.Module):
    def __init__(self, latent_dim, num_classes=10):
        super(Decoder, self).__init__()

        # We again add the conditioning vector c to the latent code z
        # So the input features to the first linear layer is (latent_dim + num_c
        in_features = latent_dim + num_classes

        self.decoder_fc = nn.Linear(in_features, 64 * 7 * 7)

        # Transposed Convolutional layers to go back to (1, 28, 28)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=4, s
            nn.ReLU(),
            nn.ConvTranspose2d(in_channels=32, out_channels=1, kernel_size=4, st
            nn.Sigmoid()  # final pixel values in [0, 1]
        )

    def forward(self, z, c):
        """
        z: latent code of shape (B, latent_dim)
        c: one-hot labels of shape (B, num_classes)
        """
        # Concatenate latent code z with conditioning
        z = torch.cat([z, c], dim=1)           # (B, latent_dim + 10)

        # Pass through the linear layer
        x = self.decoder_fc(z)                 # (B, 64 * 7 * 7)

        # Reshape into (B, 64, 7, 7) for transposed convolutions
        x = x.view(x.size(0), 64, 7, 7)

        # Decode to reconstruct the original image
        hat_x = self.decoder(x)                # (B, 1, 28, 28)

        return hat_x
```

In [25]:
```
class VariationalAutoencoder(nn.Module):
    def __init__(self, latent_dim, num_classes=10):
        super(VariationalAutoencoder, self).__init__()
        self.encoder = Encoder(latent_dim, num_classes)
        self.decoder = Decoder(latent_dim, num_classes)

    def latent_sample(self, mu, logvar):
        # Reparameterization trick: z = mu + eps * sigma
        if self.training:
```

```
            std = (0.5 * logvar).exp()        # standard deviation
            eps = torch.randn_like(std)       # random ~ N(0,1)
            return mu + eps * std
        else:
            # at inference, you might just use mu
            return mu

    def forward(self, x, c):
        """
        x: (B, 1, 28, 28)
        c: (B, num_classes)
        """
        # Encode
        mu, logvar = self.encoder(x, c)
        # Sample latent code z
        z = self.latent_sample(mu, logvar)
        # Decode
        hat_x = self.decoder(z, c)
        return hat_x, mu, logvar
```

In [26]:
```
from tqdm.notebook import trange, tqdm

def vae_loss(hat_x, x, mu, logvar):
    reconstruction_loss = F.binary_cross_entropy(hat_x.view(-1, 28*28), x.view(-
    kl_divergence = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return reconstruction_loss + kl_divergence

def train_vae(net, train_dataset, epochs=10, learning_rate=1e-3, batch_size=32,
    # Création du DataLoader pour charger les données
    train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=
    # Définition de l'algorithme d'optimisation (Adam, variante de la SGD)
    optimizer = torch.optim.Adam(params=net.parameters(), lr=learning_rate, weig
    # Choix de la fonction de coût
    criterion = vae_loss
    # Passe le modèle en mode "apprentissage"
    net = net.to(device)
    net = net.train()

    t = trange(1, epochs + 1, desc="Entraînement du modèle")
    for epoch in t:
        avg_loss = 0.
        # Parcours du dataset pour une epoch
        for images, labels in tqdm(train_dataloader):
            images = images.to(device)
            # Encodage one-hot des labels
            labels = F.one_hot(labels, num_classes=10).to(device)

            # Calcul de la reconstruction
            reconstructions, latent_mu, latent_logvar = net(images, labels)
            # Calcul de l'erreur
            loss = criterion(reconstructions, images, latent_mu, latent_logvar)

            # Rétropropagation du gradient
            optimizer.zero_grad()
            loss.backward()
            # Descente de gradient (une itération)
            optimizer.step()
            avg_loss += loss.item()

        avg_loss /= len(train_dataloader)
```

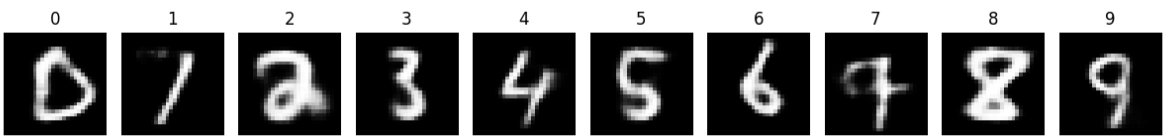
```python
        t.set_description(f"Epoch {epoch}: loss = {avg_loss:.3f}")
    return net.to("cpu").eval()
```

In [27]:
```python
latent_dim=10
num_classes=10
vae = VariationalAutoencoder(latent_dim=10, num_classes=10)
vae = train_vae(vae, train_dataset)
```

```
Entraînement du modèle:   0%|          | 0/10 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
```

In [28]:
```python
def generate_and_plot_digits(vae, num_classes=10, latent_dim=10, device='cpu'):
    vae.eval()
    vae.to(device)

    fig, axes = plt.subplots(1, num_classes, figsize=(12, 2))

    with torch.no_grad():
        for digit in range(num_classes):
            # 1) On crée un code latent z tiré de N(0, I)
            z = torch.randn(1, latent_dim).to(device)

            # 2) On crée le vecteur de conditionnement (one-hot) pour la classe
            c = torch.zeros(1, num_classes).to(device)
            c[0, digit] = 1.0

            # 3) On reconstruit l'image à partir du décodeur
            generated = vae.decoder(z, c)  # (B=1, 1, 28, 28)

            # 4) On récupère l'image pour l'afficher
            gen_image = generated.squeeze().cpu().numpy()  # shape (28, 28)

            axes[digit].imshow(gen_image, cmap='gray')
            axes[digit].axis('off')
            axes[digit].set_title(str(digit))

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

In [29]:
```python
generate_and_plot_digits(vae, num_classes=10, latent_dim=10, device=device)
```

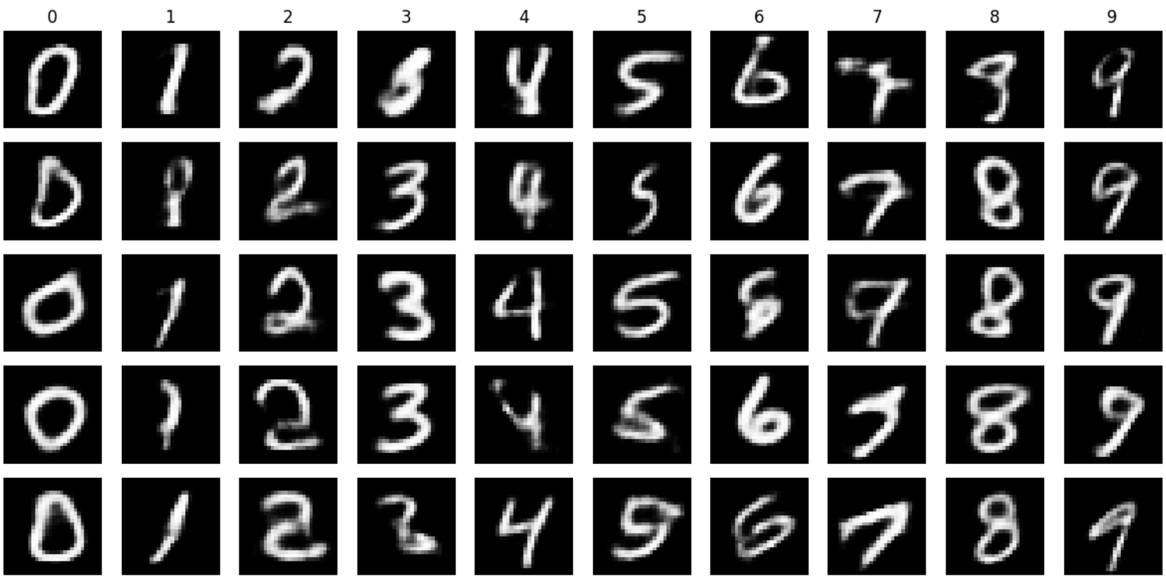



Les résultats montrent que, globalement, le VAE conditionnel a bien appris à générer des formes ressemblant aux dix chiffres de MNIST.

- Certains chiffres sont encore un peu flous ou mal formés, par exemple le « 1 » ressemble davantage à un « 7 ».
- Le chiffre « 0 » présente une forme relativement correcte, bien qu'il ait l'air un peu aplati.
- Les autres chiffres (3, 5, 6, 8, 9) sont reconnaissables, mais pourraient encore gagner en netteté.

on peux meme générer plusieurs images par classe en bouclant plusieurs fois pour chaque digit

In [30]:
```
def generate_grid(vae, num_samples=5, num_classes=10, latent_dim=10, device='cpu
    vae.eval()
    vae.to(device)

    fig, axes = plt.subplots(num_samples, num_classes, figsize=(12, 6))

    with torch.no_grad():
        for row in range(num_samples):
            for digit in range(num_classes):
                z = torch.randn(1, latent_dim).to(device)

                c = torch.zeros(1, num_classes).to(device)
                c[0, digit] = 1.0

                generated = vae.decoder(z, c)
                gen_image = generated.squeeze().cpu().numpy()

                ax = axes[row, digit]
                ax.imshow(gen_image, cmap='gray')
                ax.axis('off')
                if row == 0:
                    ax.set_title(str(digit))

    plt.tight_layout()
    plt.show()

# Exemple d'appel
generate_grid(vae, num_samples=5, num_classes=10, latent_dim=10, device=device)
```



- on remarque quelques chiffres plus flous ou un peu ambigus (par exemple certains « 1 » tirent vers un « 7 », ou certains « 2 » et « 7 » sont un peu déformés), ce qui indique que le modèle a encore un certain degré d'incertitude.

# PIXELCNN

Le masque de type A empêche la couche de voir l'information du pixel qu'elle est en train de prédire. Autrement dit, il censure le centre du noyau de convolution, de manière à ce que lorsque l'on prédit le pixel ( $i$ , $j$ ) , le modèle ne prenne pas en compte le pixel ( $i$ , $j$ ) lui-même dans ses calculs.

C'est nécessaire pour la première couche de PixelCNN afin de garantir le caractère autorégressif: à l'etat initial, le réseau ne doit s'appuyer que sur les pixels passés (ceux qui précèdent ( $i$ , $j$ )

In [37]:
```
class MaskedCNN(nn.Conv2d):

        def __init__(self, mask_type, *args, **kwargs):
                self.mask_type = mask_type
                assert mask_type in ['A', 'B'], "Unknown Mask Type"
                super(MaskedCNN, self).__init__(*args, **kwargs)
                self.register_buffer('mask', self.weight.data.clone())

                _, depth, height, width = self.weight.size()
                self.mask.fill_(1)
                if mask_type =='A':

                        self.mask[:, :, height // 2, width // 2:] = 0
                        self.mask[:, :, height // 2+1:] = 0
                else:

                        self.mask[:, :, height // 2, width // 2+1:] = 0
                        self.mask[:, :, height // 2+1:] = 0


        def forward(self, x):
                self.weight.data*=self.mask
                return super(MaskedCNN, self).forward(x)
```

In [38]:
```
class PixelCNN(nn.Module):

  def __init__(self, classes = 256, kernel = 7, channels=64):
    super(PixelCNN, self).__init__()

    self.kernel = kernel
    self.channels = channels
    self.classes = classes

    self.masked_conv_A = MaskedCNN('A', 1, channels, kernel_size=kernel, stride=
    self.batchnorm_A = nn.BatchNorm2d(channels)
    self.relu_A = nn.ReLU()

    self.masked_convs_B = nn.ModuleList([
```

```
        nn.Sequential(
            MaskedCNN('B', channels, channels, kernel_size=kernel, stride=1, pad
            nn.BatchNorm2d(channels),
            nn.ReLU()
        ) for _ in range(7)
    ])

    self.final_conv = nn.Sequential(
        nn.Conv2d(channels, self.classes, kernel_size=1),
        nn.BatchNorm2d(self.classes),
        nn.ReLU()
    )

  def forward(self, x):

    x = self.relu_A(self.batchnorm_A(self.masked_conv_A(x)))
    for masked_conv_B in self.masked_convs_B:
        x = masked_conv_B(x)
    return self.final_conv(x)
```

In [39]:

```
from tqdm.notebook import trange, tqdm

def train(net, train_dataset, epochs=5, learning_rate=1e-3, batch_size=32, devic
    # Création du DataLoader pour charger les données
    train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=
    # Définition de l'algorithme d'optimisation (Adam, variante de la SGD)
    optimizer = torch.optim.Adam(params=net.parameters(), lr=learning_rate)
    # Choix de la fonction de coût (entropie croisée)
    criterion = nn.CrossEntropyLoss()
    # Passe le modèle en mode "apprentissage"
    net = net.to(device)
    net = net.train()

    t = trange(1, epochs + 1, desc="Entraînement du modèle")
    for epoch in t:
        avg_loss = 0.
        # Parcours du dataset pour une epoch
        for images, _ in tqdm(train_dataloader):
            # les labels sont ignorés pour l'apprentissage de l'auto-encodeur

            images = images.to(device)
            # Conversion en 256 classes
            target = (images[:,0]*255).long().to(device)

            # Calcul de la reconstruction
            reconstructions = net(images)
            # Calcul de l'erreur
            loss = F.cross_entropy(reconstructions, target)

            # Rétropropagation du gradient
            optimizer.zero_grad()
            loss.backward()
            # Descente de gradient (une itération)
            optimizer.step()
            avg_loss += loss.item()
            t.set_description(f"Epoch {epoch}: loss = {loss.item():.3f}")

        avg_loss /= len(train_dataloader)
```

```
        t.set_description(f"Epoch {epoch}: loss = {avg_loss:.3f}")
    return net.eval()
```

In [40]:
```
net = PixelCNN()
net = train(net, train_dataset)
```

```
Entraînement du modèle:   0%|          | 0/5 [00:00<?, ?it/s]
  0%|          | 0/1875 [00:00<?, ?it/s]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[40], line 2
      1 net = PixelCNN()
----> 2 net = train(net, train_dataset)

Cell In[39], line 32, in train(net, train_dataset, epochs, learning_rate, batch_s
ize, device)
     30 # Rétropropagation du gradient
     31 optimizer.zero_grad()
---> 32 loss.backward()
     33 # Descente de gradient (une itération)
     34 optimizer.step()

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\torch\_tensor.py:521, in Tenso
r.backward(self, gradient, retain_graph, create_graph, inputs)
    511 if has_torch_function_unary(self):
    512     return handle_torch_function(
    513         Tensor.backward,
    514         (self,),
   (...)
    519         inputs=inputs,
    520     )
--> 521 torch.autograd.backward(
    522     self, gradient, retain_graph, create_graph, inputs=inputs
    523 )

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\torch\autograd\__init__.py:28
9, in backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables,
inputs)
    284     retain_graph = create_graph
    286 # The reason we repeat the same comment below is that
    287 # some Python versions print out the first line of a multi-line function
    288 # calls in the traceback and some print out the last line
--> 289 _engine_run_backward(
    290     tensors,
    291     grad_tensors_,
    292     retain_graph,
    293     create_graph,
    294     inputs,
    295     allow_unreachable=True,
    296     accumulate_grad=True,
    297 )

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\torch\autograd\graph.py:769, i
n _engine_run_backward(t_outputs, *args, **kwargs)
    767     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    768 try:
--> 769     return Variable._execution_engine.run_backward(  # Calls into the C++
engine to run the backward pass
    770         t_outputs, *args, **kwargs
    771     )  # Calls into the C++ engine to run the backward pass
    772 finally:
    773     if attach_logging_hooks:

KeyboardInterrupt:
```

usage limits in Colab !!