TP - AutoEncoder

Maéva Bachelard - Margot Laleu

CPU et Dataset

L'entraînement d'une IA demande une grande puissance de calcul, c'est pourquoi nous ne pouvons pas coder ce TP depuis notre machine personnelle et que nous devons le faire en ligne sur GoogleColab.
La première chose à faire est donc de vérifier la disponibilité d'un CPU, comme on peut le voir ci-contre.

```
[2]  print('torch.cuda.is_available()', torch.cuda.is_available())

     DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
     print('device is :', DEVICE)

     torch.cuda.is_available() True
     device is : cuda
```

Le second élément crucial pour entraîner une IA est la dataset. C'est-à-dire l'ensemble des données sur lesquelles l'IA va baser son entraînement. Pour éviter le sur-apprentissage (over-fitting), il nous faut scinder notre dataset en 3 parties.
Une partie "training set" qui sert à l'entraînement, c'est-à-dire qu'à chaque passage de l'une de ces données dans le réseau de neurones, ce dernier va voir ses poids modifiés pour en optimiser la sortie. Le "validation set" sert à faire passer ses données alternativement avec celles du training set, mais le réseau de neurone n'est jamais modifié suite au passage de ces données. Cela sert à observer l'évolution de l'apprentissage.
Enfin, le "testing set" est passé dans le réseau de neurones à la fin de l'entraînement, afin de constater la qualité de l'apprentissage.

Ci-contre, on télécharge donc le "training set" et le "testing set". Ils contiennent respectivement 60 000 et 10 000 données.

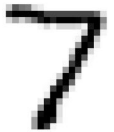
```
[15] # TODO : Load the test dataset. Inspire yourself from the mnist_train dataset loading

     mnist_test = MNIST('', train=False, download=True, transform=transforms.ToTensor())
```

```
[6]  # TODO: Load MNIST Train Dataset from TorchVision

     train = True # bool (False/True)

     dataset = MNIST('', train=train, download=True, transform=transforms.ToTensor())
```

Eléments de la dataset

Pour que la division en 3 datasets fonctionne, il faut que chaque dataset comporte le même type d'image mais pas les mêmes images.

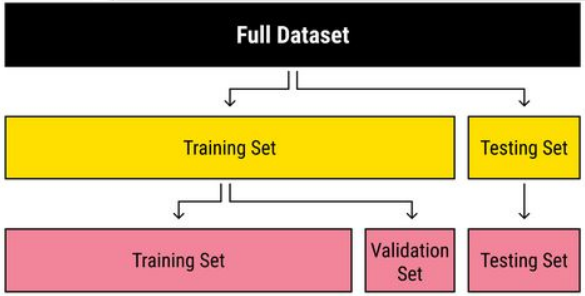
Ci-contre on a donc affiché un élément du "training set", et un élément du "testing set" dont l'indice est le même. Autrement dit, on vérifie ainsi que les images des 2 datasets sont du même type et qu'il s'agit bien d'images différentes.



La dernière étape est donc d'obtenir le "validation set". Contrairement aux 2 autres on ne le télécharge pas, mais on l'obtient en scindant en 2 parties le "training set" précédent, comme on le voit sur le schéma ci-dessous.

```
# Questions : what does 55000 and 5000 mean ? Hint: look at the Dataset length and determine the split value
mnist_train, mnist_val = random_split(dataset, [55000, 5000])
print(len(mnist_val), len(mnist_train))
```

```
5000 55000
```



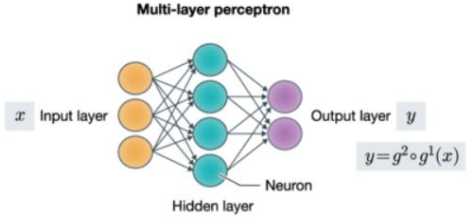
Finalement, notre "training set" contient 55000 échantillons (données) et notre "validation set" en contient 5000.
Donc on retire 8,3 % de notre "training set" afin de constituer notre "validation set".

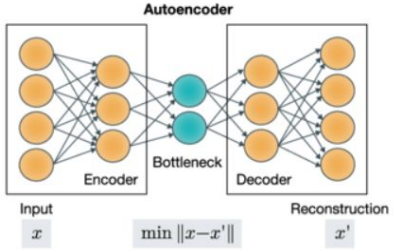
Dataloaders et batches

Ainsi créée, notre dataset envoie les données une par une. Mais en DeepLearning il est préférable d'en envoyer plusieurs à la fois à notre réseau de neurones.
Pour cela nous allons avoir besoin de former et récupérer des lots de données, nommés batch, depuis notre dataset, c'est ce que fait un dataloader. Et on fait cela sur nos trois jeux de données. Ici, chacun de nos batch contient 128 données.

```
train_loader = DataLoader(mnist_train, batch_size=128, drop_last = True)
val_loader = DataLoader(mnist_val, batch_size=128, drop_last = True)
test_loader = DataLoader(mnist_test, batch_size=128, drop_last = True)
```

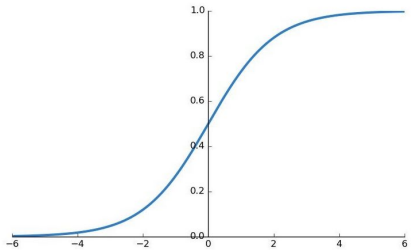
Multi Layers Perceptron (MLP) Model of AutoEncoder





On remarque donc que le model d'AutoEncoder que nous devions créer utilise donc 2 MLP. L'un servant à encoder et compresser l'image, et le second dont l'entrée est la sortie du premier, qui sert à décoder l'image.

Dans la définition du modèle ci-contre, on définit la taille de l'entrée et de la sortie (dimension de l'image mais vectorisée). Puis on définit les couches d'entrée et de sortie, comme étant des "Dense Layer", c'est-à-dire des couches dont tous les neurones sont reliés à chacun des neurones de la couche suivante (respectivement précédente pour la sortie). Le calcul de ces couches est donc linéaires excepté l'application de la fonction d'activation, Sigmoïd ici, dont la courbe est ci-dessous.



Notre réseau ainsi créé ne contient donc pas 4 couches comme on le voit dans le schéma ci-dessus, mais seulement 2 couches et un latent space correspondant à l'image intermédiaire compressée.

```python
class AutoEncoder_MLP(nn.Module):
  def __init__(self, input_size, compressed_space_size):
    """
    The model is an Input Layer, a Hidden Layer and an Output layer
    """
    super().__init__()
    # TODO : Init the class attributes thanks to the arguments of the init methods
    self.input_size = input_size
    self.output_size = input_size
    self.compressed_space_size = compressed_space_size
    # TODO : Correct 3 mistakes from the model
    self.input = nn.Sequential(nn.Linear(self.input_size, self.compressed_space_size),
                               nn.Sigmoid()) # Hint: mistake this line
    self.output = nn.Sequential(nn.Linear(self.compressed_space_size, self.output_size),
                               nn.Sigmoid())
    # end TODO: in total, there are 6 mistakes in all these 5 above lines

  def forward(self,x):
    """
    The input x is forwarded through the neural net.
    """
    # TODO
    compressed_image = self.input(x)
    decompressed_image = self.output(compressed_image)
    return decompressed_image
```

Fonction de perte et Optimizer

Comme expliqué dans la préparation, le but original de la fonction de perte est de nous indiquer à quel point nos prédictions sont éloignées des véritables étiquettes. Or ici on ne veut pas que notre IA reconnaisse les chiffres sur les images de la dataset mais qu'elle les compresse puis les décompresse.
Le but de cette fonction ici sera donc de déterminer à quel point notre reconstruction est fiable par rapport à l'image originale, fournie en entrée. Autrement dit à quelle point elles sont ressemblantes. Pour cela on peut calculer la distance mathématique entre ces 2 vecteurs que constituent l'entrée et la sortie du modèle.

Pour cela, on utilise la fonction de perte MSELoss() de la bibliothèque torch, qui effectue le calcule suivant avec Y l'image d'entrée et Y^ l'image de sortie.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE = mean squared error
n = number of data points
Y_i = observed values
$\hat{Y}_i$ = predicted values

C'est le résultat de cette fonction de perte que l'on va chercher à optimiser via un otpimizer qui applique une descente de gradient afin de recalculer les poids du modèle.
L'optimizer que nous allons utiliser est l'Adam optimizer. Et comme toute descente de gradient le pas est un élément important et va déterminer la vitesse et la qualité de convergence.
Ici on utilise un pas constant que l'on va appeler learning rate.

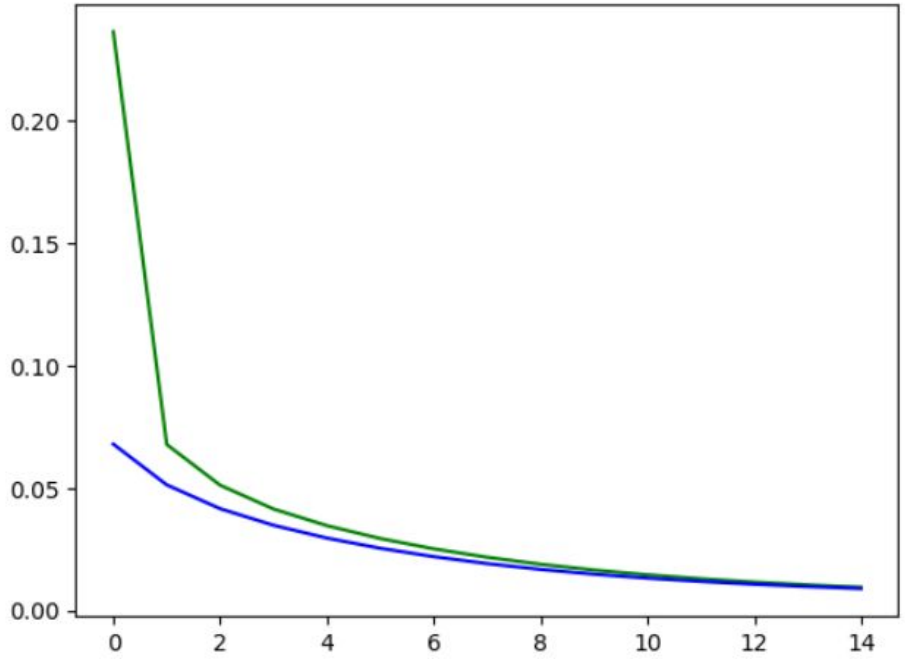
Training

Tout d'abord on crée un nombre d'epoch, c'est-à-dire le nombre de fois que notre modèle va s'entraîner sur notre dataset (mnist_train).
Ensuite, on veut envoyer au modèle des lots, que l'on appelle batch, de 128 images à la fois. Or, comme on travail avec un modèle MLP, chaque neurone de la couche d'entrée prend la valeur d'un pixel d'une image, c'est pourquoi, chacune des 128 images ne doit plus être une matrice pour être traiter par ce genre de modèle, mais doit être sous la forme d'un vecteur. Donc on reformate les images du batch sélectionné.

On passe donc du format : `[128, 1, 28, 28]`, au format : `[128, 784]`

Il ne reste plus qu'à appliquer notre modèle à notre batch ainsi préparé. Ensuite on remet les images du batch au format matriciel. On applique ensuite notre fonction de perte suivie de l'optimizer précédemment choisi (Les poids de notre modèle sont donc modifiés) et on stocke les valeurs de la fonction de perte dans un tableau.
On fait tourner cette boucle pour chaque batch. Et on fait tourner une seconde boucle sur des batch prélevés dans le set de validation (mnist_val) et on fait les mêmes actions à l'exception de l'application de l'optimizer car on ne veut pas modifier le modèle suite au passage de cette dataset, mais seulement observer l'apprentissage, puisqu'on stocke également le résultat de la fonction de perte observé.

Observation de l’évaluation des fonctions de perte, du set d'entraînement en vert et du set de validation en bleu.

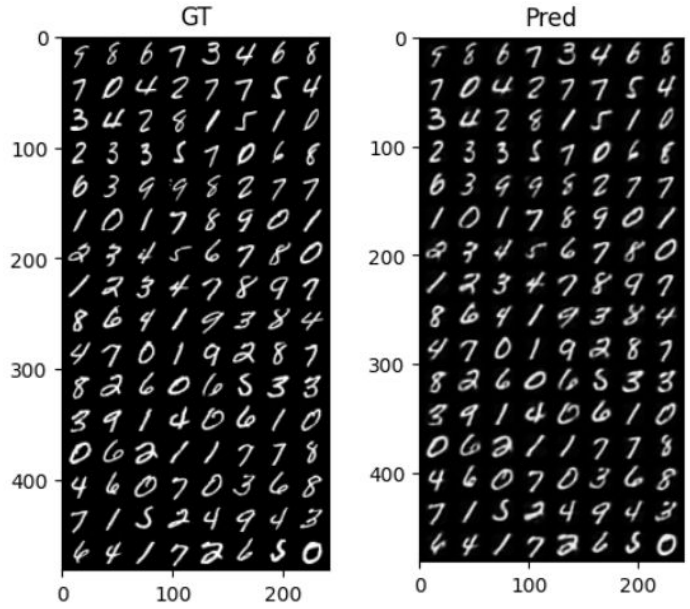


On remarque que les courbes diminuent très vite et semblent tendre vers 0.
L’apprentissage est rapide.

Testing

Il s'agit maintenant de tester notre modèle avec une dataset que le modèle n'a encore jamais vu. On fait une boucle similaire à celle pour le set de validation.

```
# TODO : form your testing loop. Is it different than the validation loop?
with torch.no_grad():
  losses = 0
  for i, data in enumerate(test_loader, 0):
    image, label = data[0].view(data[0].shape[0],-1).to(device), data[1]
    outputs = net(image.view(128,-1).to(device))
    outputs = outputs.view([128, 1, 28, 28], -1);
    loss = criterion(outputs ,image.to(device).view([128, 1, 28, 28])) # use criterion()
    losses += loss.item() # accumulate 'loss' into 'losses'

# Plot the last batch
imshow(torchvision.utils.make_grid(outputs.detach().cpu()),'Pred')
imshow(torchvision.utils.make_grid(data[0]),'GT')

# TODO : Print the difference in decompression and write it somewhere
print('The difference between the Real Images and the Decompressed Images is: ', losses)
```



Afin d'observer l'effet de notre modèle, on affiche un batch entier, avant (GT) et après (Pred) passage dans le modèle. Et on calcul ensuite la fonction de perte de ce batch.

On observe alors que l'image reconstruite est globalement plus floue que l'originale.
Le résultat de la fonction de perte est 0.605. Ce qui n'est pas très élevé, on en déduit donc que la reconstruction est plutôt correcte.

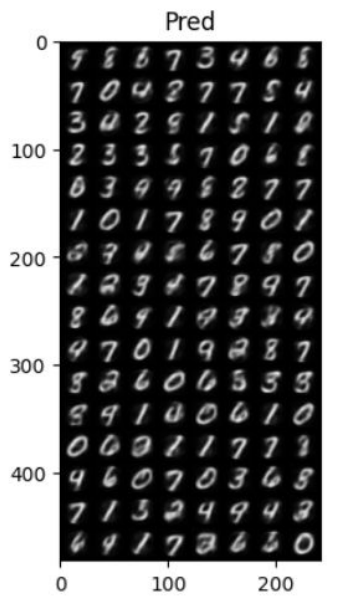
Taille de l'image compressée

Le code présenté sur la slide suivante synthétise et rassemble les étapes précédentes pour les rendre plus lisibles ensemble.
L'objectif va être de tester différentes tailles de l'image compressée et observer la qualité de reconstruction de l'image dans les différents cas. Nous avions déjà testé pour une taille de 512. (l'image compressée est sous forme de vecteur)

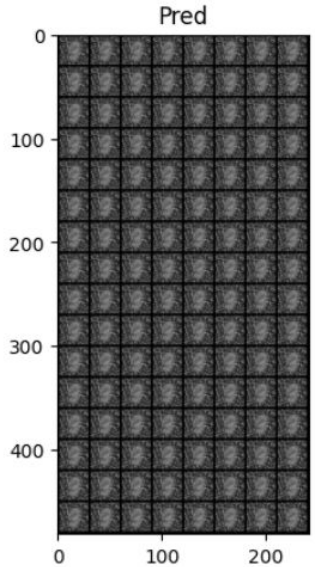
Pour une taille de 128, on observe ci-contre que l'image reconstruite est encore plus floue que pour une taille de 512.

Pour une taille de 512, on avait une erreur de reconstruction de 0.605.
Pour une taille de 128, on a une erreur de reconstruction valant 2.16.
Pour une taille de 1, on a une erreur de reconstruction valant 9.47 et les chiffres ne sont plus du tout reconnaissables.

On remarque logiquement que plus l'image est compressée et donc moins on stocke de données à son sujet, plus l'erreur à la reconstruction est élevée et plus l'image reconstruite est floue.



taille 128



taille 1

Code de l'entraînement du modèle MLP

```
model = AutoEncoder_MLP(28*28,512)

# TODO : Train it. Can we copy paste previous things ?
# Globalement oui, les boucles d'entraînement et de validation reste les mêmes.
#device = 'cuda' if torch.cuda.is_available() else 'cpu'
losses = 0
net = model.to(device)
criterion = nn.MSELoss()
learning_rate = 0.0001
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
num_epochs = 15
```

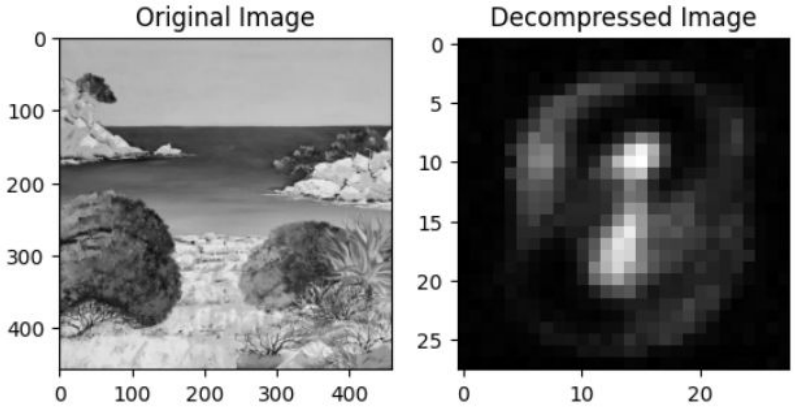
```
for epoch in range(num_epochs) :
    running_loss_t,running_loss_v = 0.0, 0.0
    # Train Loop
    for i, data in enumerate(train_loader, 0):
      image, label = data[0], data[1]
      image, label = image.view(image.size(0),-1).to(device), data[1]
      optimizer.zero_grad()
      outputs = net(image)
      outputs = outputs.reshape([128, 1, 28, 28], -1).to(device)
      loss = criterion(outputs ,image.to(device).view([128, 1, 28, 28]))
      loss.backward()
      optimizer.step()
      running_loss_t += loss.item()
      if i == 50:
        running_loss_t= running_loss_t/50
        print('training loss is :',running_loss_t)
```

```
    # Validation Loop
    with torch.no_grad():
      for i, data in enumerate(val_loader, 0):
        image, label = data[0].view(data[0].shape[0],-1).to(device), data[1]
        outputs = net(image.view(128,-1).to(device))
        outputs = outputs.view([128, 1, 28, 28], -1);
        loss = criterion(outputs ,image.to(device).view([128, 1, 28, 28]))
        running_loss_v += loss.item()
        if i == 50:
            running_loss_v= running_loss_v/50
            print('validation loss is :',running_loss_v)
            loss_val.append(running_loss_v) # show validation loss
```

```
# TODO : Test the Trained Model. Can we copy paste previous stuff ?
# La boucle de test reste également la même.

with torch.no_grad():
  running_loss = []
  for i, data in enumerate(test_loader, 0):
    image, label = data[0].view(data[0].shape[0],-1).to(device), data[1]
    outputs = net(image.view(128,-1).to(device))
    outputs = outputs.view([128, 1, 28, 28], -1);
    loss = criterion(outputs ,image.to(device).view([128, 1, 28, 28])) # use criterion()
    losses += loss.item() # accumulate 'loss' into 'losses'
```

Un autre set de test

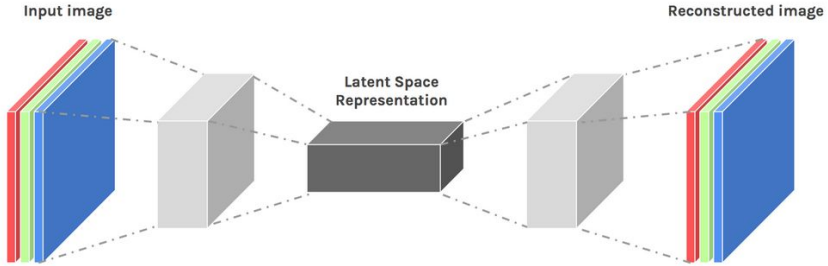
L'objectif est maintenant de tester notre modèle entraîné, sur d'autre type d'image que des chiffres manuscrits.
En effet, nous avons vu que notre modèle fonctionne plutôt bien pour la reconstruction de ce type d'image, mais qu'en est-il pour des images d'un autre type ?
Tout d'abord, pour que notre modèle puisse ressortir un résultat il faut lui donner en entrée une image en noir et blanc et réduite aux dimensions 28*28.



Pour une image de paysage, donc beaucoup plus complexe qu'un simple chiffre manuscrit, l'image décompressée ne ressemble en rien à l'originale.
Cela vient sûrement du fait que le réseau de neurones que nous avons créé reste assez petit. Pour qu'il soit plus efficace et puisse reconstruire des images plus complexes il faudrait augmenter le nombre de couches cachées.
Une autre source possible de ce résultat insatisfaisant vient peut-être du fait que notre modèle ne s'est entraîné que sur des chiffres manuscrits et sur aucun autre type d'image et aurait donc adapté la méthode d'encodage et décodage à ce type d'images.

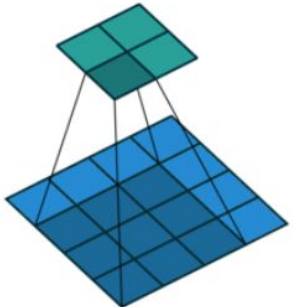
Convolutional layers Model of AutoEncoder

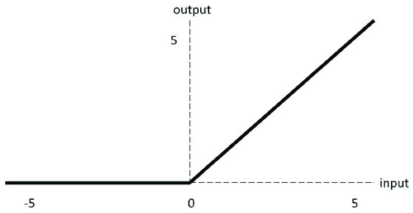
Nous voulons maintenant réaliser la même chose mais avec un réseau de neurones à convolution. Et tout comme pour le modèle MLP, on utilise un réseau de neurones à convolution pour l'encodage et un second pour la décompression. L'entrée du second étant la sortie du premier.



Cela signifie que nous allons d'abord devoir réduire la dimension de l'image originale, à l'aide de kernels (matrices filtre dont les valeurs sont les poids de notre modèle), cette première étape s'effectue à l'aide de couches appelée convdown.

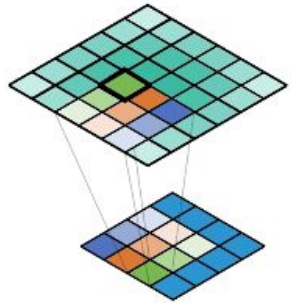
Notre modèle convdown contient deux étapes : la conv2D layers représenté ci-dessous, puis on applique une fonction d'activation non linéaire, la fonction Relu.





Pour le décodage, nous voulons en quelque sorte augmenter la taille de l'image fournie. Pour cela il existe des couches nommées ConvTranpose2D, qui exécutent une convolution de transposition, c'est-à-dire qui apprennent à suréchantillonner les images.
C'est ce que va faire notre seconde étape appelée convup.

Convup contient deux étapes, l'étape de ConvTranspose2D suivie d'une fonction d'activation. Or ici, il est possible de choisir entre la fonction d'activation Sigmoïd ou Relu en argument de convup. Si rien n'est spécifié en argument, ce sera l'activation Sigmoïd.



Notre modèle d'AutoEncoder empilera plusieurs couches convdown pour la compression et plusieurs couches convup pour la décompression.

Code de notre modèle

Notre classe Encoder contient l'empilement des couches convDown (une seule pour l'instant), et notre classe Decoder contient l'empilement des couches ConvUp (une seule pour l'instant). Ci-dessous le code de Decoder qui est très similaire au code de Encoder. Enfin, notre classe AutoEncoder organise la succession d'exécution de l'encodage via la classe Encoder puis le décodage via la classe Decoder.

```python
class Decoder(nn.Module):
    """
    The Decoder stacks multiple ConvUp to upsample and reconstruct from the input
    another feature map
    For the moment, we just keep one ConvDown layer
    NEEDS TO BE CORRECTED

    """
    def __init__(self,input_channel, output_channel, kernel_size = 3, output = True):
        super().__init__()
        self.input_channel = input_channel
        self.output_channel = output_channel
        self.kernel_size = kernel_size
        self.output = output
        self.model = nn.Sequential( ConvUp(self.input_channel,self.output_channel,self.kernel_size, output))

    def forward(self,x):
        # TODO :  Send the data through the model and return the output
        outputt = self.model(x)
        return outputt
```

Attention ! Les input_channel et output_channel constituent "l' épaisseur de l'image". Par exemple, une image en couleur RGB possède 3 channels et une image en noir et blanc 1 channel.
Ainsi, en donnant le nombre de channels de notre latent space alors on choisit le nombre de kernel de même dimension utilisé par conv2D sur notre image initiale.
En effet, pour l'instant on appelle une seule fois conv2D car on appelle une seule fois convDown, mais conv2D utilise déjà autant de matrices que l'épaisseur du channel que l'on souhaite en sortie.

```python
class AutoEncoder_Conv(nn.Module):

    def __init__(self,input_size,latent_size, output= True):
        super().__init__()
        self.input_size = input_size
        self.latent_size = latent_size
        self.output = output
        self.model_encoder = nn.Sequential(Encoder(self.input_size, self.latent_size, kernel_size=3))
        self.model_decoder = nn.Sequential(Decoder(self.latent_size, self.input_size, kernel_size=3, output=self.output))

    def forward(self,x):
        outputt = self.model_decoder(self.model_encoder(x))
        return  outputt
```

Code de l'entraînement du modèle Convolutionnel

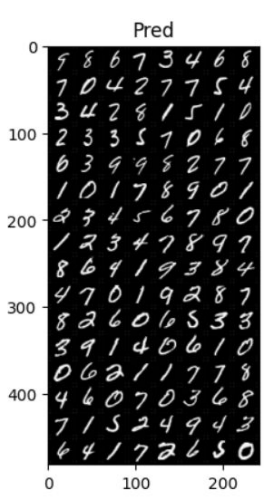
```python
model = AutoEncoder_Conv(input_size=1, latent_size=128, output=True)

# Nous avons considéré que la latent_size est l'épaisseur de l'image compressée.

# TODO : Reload your HyperParameters

losses = 0
net = model.to(device)
criterion = nn.MSELoss()
learning_rate = 0.0001
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
num_epochs = 15

# TODO : Rewrite your Training and Validation Loop
for epoch in range(num_epochs) :
    running_loss_t,running_loss_v = 0.0, 0.0
    # Train Loop
    for i, data in enumerate(train_loader, 0):
      image, label = data[0].to(device), data[1]
      optimizer.zero_grad()
      outputs = net(image).to(device)
      loss = criterion(outputs, image)
      loss.backward()
      optimizer.step()
      running_loss_t += loss.item()
      if i == 50 :
        running_loss_t = running_loss_t/50
        print('training loss is : ', running_loss_t)
```

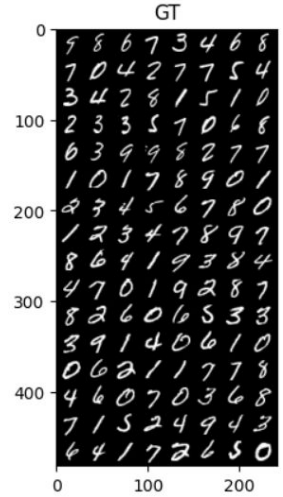
```python
    # Validation Loop
    with torch.no_grad():
      for i, data in enumerate(val_loader, 0):
        image, label = data[0].to(device), data[1]
        outputs = net(image).to(device)
        loss = criterion( outputs, image.to(device))
        running_loss_v += loss.item()
        if i == 50 :
          running_loss_v = running_loss_v/50
          print('validation loss is : ', running_loss_v)
          loss_val.append(running_loss_v)


# Testing Loop
with torch.no_grad():
  running_loss = []
  for i, data in enumerate(test_loader, 0):
    image, label = data[0].to(device), data[1]
    outputs = net(image).to(device)
    loss = criterion(outputs, image.to(device))
    losses += loss.item()


# TODO :  Plot the last batch and the Reconstruction Errors
imshow(torchvision.utils.make_grid(outputs.detach().cpu()), 'Pred')
imshow(torchvision.utils.make_grid(data[0]), 'GT')

plt.plot(loss_train, color='green')
plt.plot(loss_val, color='blue')
plt.show()
```

On remarque que l'on ne reformate plus les images sous forme de vecteurs, puisque le modèle convolutionnel travail directement sur des matrices.

Résultats de l'Auto Encodeur Convolutionnel

On remarque déjà que pour une latent-size de 512 (l'épaisseur de la couche centrale), on a une image reconstruite aussi nette que l'originale et beaucoup plus nette que celle reconstruite en MLP avec une latent-size de 512 (nombre de bits dans le vecteur).
De plus, le taux d'erreur entre l'image originale et l'image reconstruite était de 0.605 pour une latent-size de 512 en MLP et est ici de seulement 0.024 pour une latent-size de 521 en convolutionnel. Pour des latent-size de 128 et 16, on obtient des images toujours aussi nettes et respectivement des taux d'erreur de 0.042 et 0.17. Ce qui est très faible même pour de petites latent-size.
Pour une latent-size de 1 les chiffres sont lisibles mais grisés et le taux d'erreur est de 3.53. Ce qui semble logique puisque cela revient à n'appliquer qu'un filtre (=kernel) et donc ne relever qu'une caractéristique de l'image.

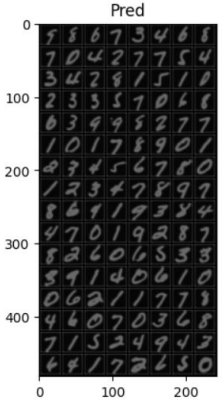




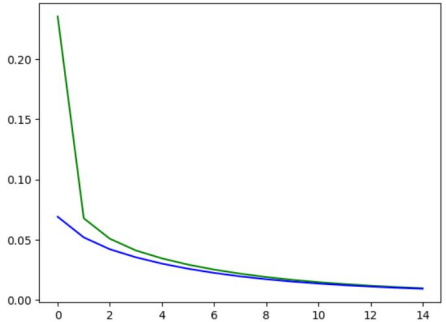
latent-size 128 originale

Concernant la rapidité de l'entraînement de ce petit modèle convolutionnel, il ne semble pas plus long que notre modèle MLP. On avait un temps d'exécution du code d'entraînement et de test du modèle MLP de 127 secondes, contre 159 secondes pour le modèle convolutionnel.
De plus, on remarque que la rapidité avec laquelle les courbes des fonctions de perte du set d'entraînement (en vert) et du set de validation (en bleu) décroissent est la même dans les deux cas.

On en conclut que le modèle convolutionnel est beaucoup plus efficace et adapté que le modèle MLP pour travailler sur des images.



latent-size 1



Un modèle plus profond

Jusqu'alors, nous avions un modèle en trois couches :

```
input layer => hidden layer => output layer
```

Mais cela n'est pas vraiment très profond (cf Deep Learning), donc on veut ajouter des couches à notre modèle, comme l'indique le schéma suivant :

```
input layer => hidden layer => hidden layer => hidden layer => output layer
```

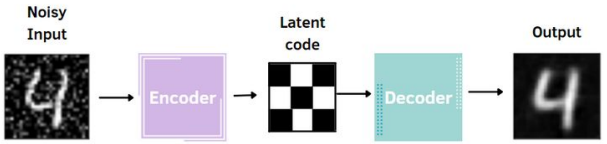
C'est-à-dire, ajouter une couche cachée de ConvDown dans la compression et une couche cachée de ConvUp dans la décompression. On modifie donc nos classes Encoder et Decoder en conséquence.

```python
class Encoder(nn.Module):
    """
    Conv Encoder Class
    """
    def __init__(self,input_channel, output_channel, kernel_size = 3):
        super().__init__()
        self.input_channel = input_channel
        self.output_channel = output_channel
        self.kernel_size = kernel_size
        self.model = nn.Sequential(ConvDown(self.input_channel, 128, self.kernel_size),
                                   ConvDown(128, self.output_channel, self.kernel_size))

    def forward(self,x):
        # TODO :  Send the data through the model and return the output
        output = self.model(x)
        return output
```

```python
class Decoder(nn.Module):
    """
    Conv Decoder Class
    Be careful with the output attribute
    """
    def __init__(self,input_channel, output_channel, kernel_size = 3, output = True):
        super().__init__()
        self.input_channel = input_channel
        self.output_channel = output_channel
        self.kernel_size = kernel_size
        self.output = output
        self.model= nn.Sequential(ConvUp(self.input_channel, 128, self.kernel_size),
                                  ConvUp(128, self.output_channel, self.kernel_size))

    def forward(self,x):
        # TODO :  Send the data through the model and return the output
        output = self.model(x)
        return output
```

On a donc choisit une épaisseur entre les deux couches cachées de l'encoder et du decoder de 128.
On entraîne maintenant notre nouveau modèle exactement comme précédemment, avec une latent-size de 128 puis une latent-size de 64. On obtient respectivement un taux d'erreur de 0.004 et de 0.037. Ce qui est encore plus faible que précédemment, on peut donc dire que l'ajout de couches de convolution améliore grandement le résultat final.
On note qu'on a maintenant un temps d'exécution plus long que précédemment, de 250 à 278 secondes.

Débruiter une image via l'Auto Encoder



Nous allons maintenant voir que notre AutoEncoder est aussi capable de débruiter une image. Pour cela, nous allons commencer par ajouter du bruit à l'une des images de notre dataset via la fonction de la librairie Pytorch randn_like qui renvoie une matrice, de même dimension que l'input fournie en argument, remplie de valeurs aléatoires entre 0 et 1 suivant une loi normale centrée réduite.
On crée notre propre fonction add_noise qui permet en plus de moduler l'amplitude de ce bruit.

```
def add_noise(inputs, noise_factor):
     noise = torch.randn_like(inputs)
     return(noise*noise_factor + inputs)


# TODO : Pick an image from the test set and add noise to it
test_image = data_test[0] # Pick an image from test dataset
test_image_with_noise = add_noise(data_test[0], 0.3) # Add noise
show(test_image)
show(test_image_with_noise)

# TODO : Send the Image through your model and plot the original image and the infered image
denoised_image = model(test_image_with_noise.to(device))
fig,axarr = plt.subplots(1, 2)
axarr[0].imshow(test_image.squeeze(0).squeeze(0).squeeze(0))
axarr[1].imshow(denoised_image.detach().cpu().squeeze(0).squeeze(0).squeeze(0))
```

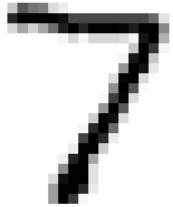


image originale

image bruité

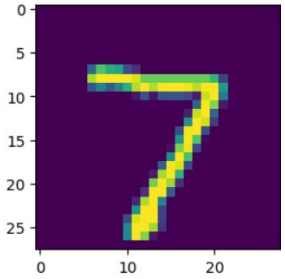
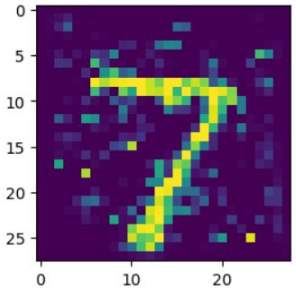
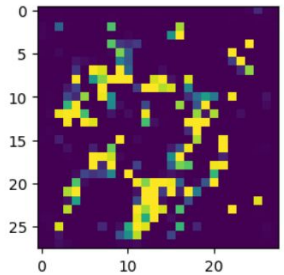


image originale

Lorsque nous avons envoyé notre image ainsi bruité à notre modèle, sa réponse est ci-contre. (Les couleurs sont dues à l'affichage, il ne faut pas en tenir compte.)
On observe alors bien qu'il renvoie l'image originale mais fortement débruité.
Cela peut s'expliquer par le fait que le modèle s'est entraîné uniquement à renvoyer des images de ce type (des chiffres manuscrits), alors même si celle qu'on lui fournit en entrée est un peu bruité, il fournira l'image la plus proche qu'il s'est entraîné à fournir, donc celle d'un chiffre.
En revanche, pour un bruit plus important (facteur multiplicateur du bruit = 0.8 contre 0.3 auparavant), cela ne fonctionne plus.





images décompressées

bruit : amplitude 0.3 bruit : amplitude 0.8

Images en couleur : Dataset

Nous allons maintenant recréer des modèles équivalent aux précédent mais permettant non plus de traiter des images en noir et blanc mais des images en couleur, format RGB.
Pour cela, il nous faut tout d'abord charger une nouvelle dataset contenant des images en couleur et scinder cette dataset en un train_set, validation_set et un test_set comme précédemment. Les images de cette dataset ont pour dimensions : [3, 32, 32].

```
transform = transforms.Compose([transforms.ToTensor()])
dataset_train = CIFAR10(root='./data', train=True,download=True, transform=transform)
color_test =  CIFAR10(root='./data', train=False,download=True, transform=transform)
color_train, color_val = random_split(dataset_train, [45000, 5000])
```

Ci-dessous, 2 exemples d'images de notre nouvelle dataset.

Puis on crée les batch à l'intérieur de ces 3 sets :

```
train_loader = DataLoader(color_train, batch_size=128, drop_last = True)
val_loader = DataLoader(color_val, batch_size=128, drop_last = True)
test_loader = DataLoader(color_test, batch_size=128, drop_last = True)
```

Images en couleur : Modèle MLP

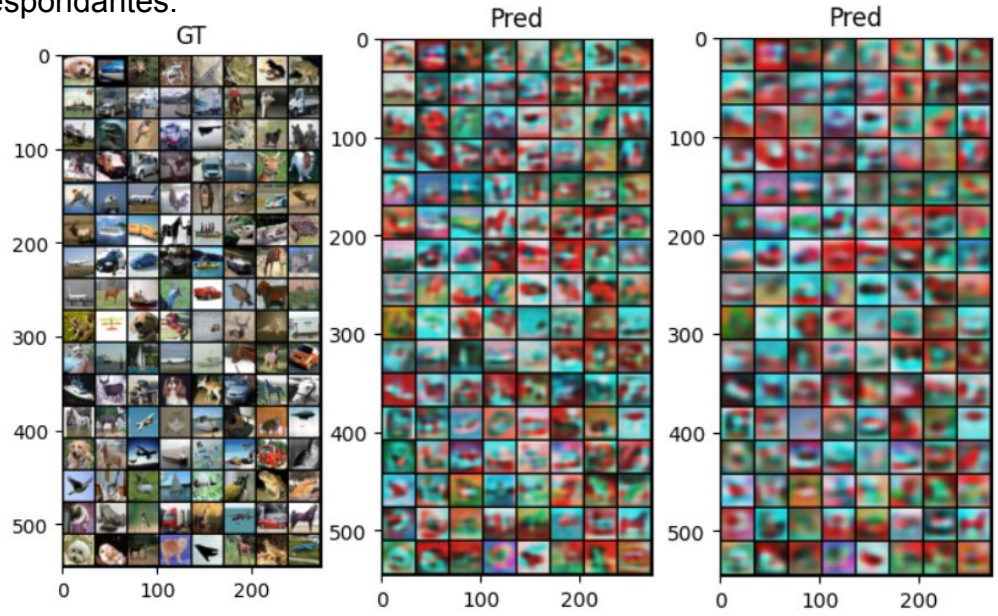
Nous allons maintenant re-créer un modèle MLP de 3 couches. Concernant l'Auto Encoder, il reste identique à celui du précédent modèle MLP. Il s'applique donc toujours à une matrice (2 dimensions et non pas 3 dimensions). Ainsi, notre modèle MLP prend en entrée une des 3 couches, R, G ou B, sous la forme d'un vecteur.

Ce qui change est donc qu'il va falloir extraire les matrices R, G et B de l'image, les passer séparément dans notre modèle, puis rassembler les 3 résultats retournés afin de reconstruire une image RGB. Le code est présenté sur la slide suivante et ci-dessous, un échantillon d'images originales et les images décodées correspondantes.

On remarque que le décodage ne correspond pas du tout aux images originales et ne semble pas fonctionner. En effet, on reconnaît la forme globale des images mais les couleurs n'ont pas été correctement compressées et décompressées.

On se demande dans un premier temps, si la plus grande complexité de ces images ne nécessite pas d'augmenter le nombre de couches cachées. Dans notre 1er test, il y a une seule couche cachée et nous avons utilisé un unique modèle dont les mêmes poids sont modifiés pour les matrices R, G et B. On se demande donc si ces poids ne sont pas propres à chacune des 3 matrices et nécessitent donc un modèle par matrice.

Nous avons donc écrit un second code où nous utilisons 8 couches cachées et où on utilise un modèle par matrice R, G et B. Mais le résultat semble même plus floue que pour la première tentative.



Peut-être que la complexité supérieure de ce type d'image n'est tout simplement pas décodable par un MLP. Ou peut-être que le MLP n'est pas fait pour gérer plusieurs couches à superposer. Il semble plutôt pensé pour traiter des objets plan puisqu'on doit fournir un vecteur en entrée.

Code de l'entraînement du modèle MLP couleur

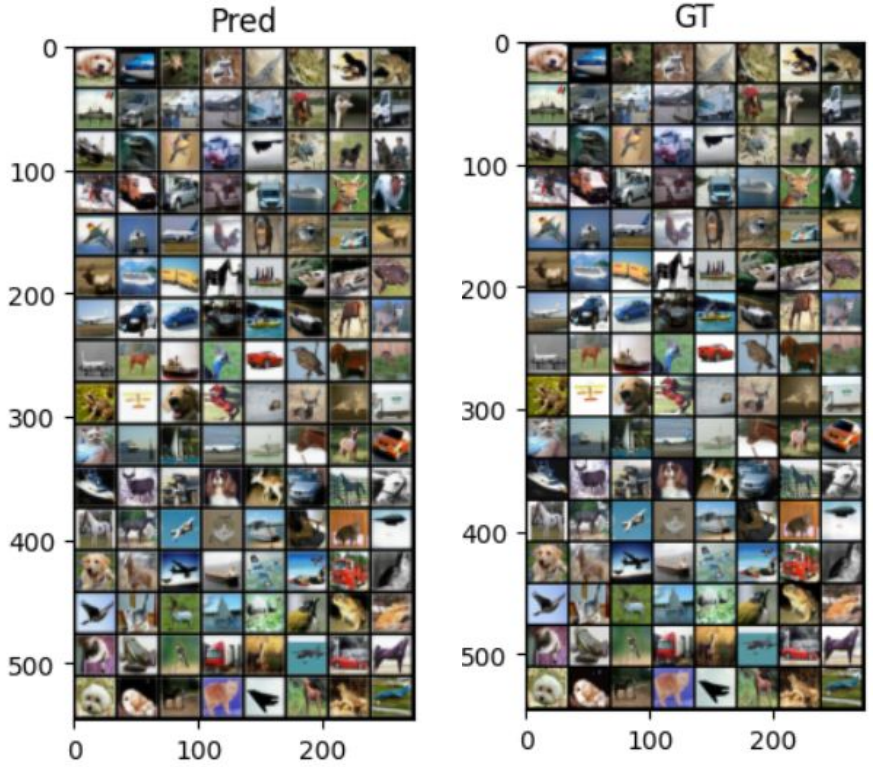
```
losses = 0
model = AutoEncoder_MLP(32*32,512).to(device)
criterion = nn.MSELoss()
learning_rate = 0.0001
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
num_epochs = 15

for epoch in range(num_epochs):
    running_loss_t, running_loss_v = 0.0, 0.0
    # Train Loop
    for i, data in enumerate(train_loader, 0):
        image, label = data[0], data[1]
        image, label = image.to(device), label.to(device)
        r, g, b = image[:, 0].unsqueeze(1), image[:, 1].unsqueeze(1), image[:, 2].unsqueeze(1)
        optimizer.zero_grad()
        outputs_r = model(r.view(-1, 32*32))
        outputs_g = model(g.view(-1, 32*32))
        outputs_b = model(b.view(-1, 32*32))
        outputs = torch.cat((outputs_r.view(-1, 1, 32, 32), outputs_g.view(-1, 1, 32, 32), outputs_b.view(-1, 1, 32, 32)), dim=1)
        loss = criterion(outputs, image)
        loss.backward()
        optimizer.step()
        running_loss_t += loss.item()
        if i == 50:
            running_loss_t = running_loss_t / 50
            print('Training loss is:', running_loss_t)
```

```
    # Validation Loop
    with torch.no_grad():
        for i, data in enumerate(val_loader, 0):
            image, label = data[0], data[1]
            image, label = image.to(device), label.to(device)
            r, g, b = image[:, 0].unsqueeze(1), image[:, 1].unsqueeze(1), image[:, 2].unsqueeze(1)
            outputs_r = model(r.view(-1, 32*32))
            outputs_g = model(g.view(-1, 32*32))
            outputs_b = model(b.view(-1, 32*32))
            outputs = torch.cat((outputs_r.view(-1, 1, 32, 32), outputs_g.view(-1, 1, 32, 32), outputs_b.view(-1, 1, 32, 32)), dim=1)
            loss = criterion(outputs, image)
            running_loss_v += loss.item()
            if i == 50:
                running_loss_v = running_loss_v / 50
                print('validation loss is :', running_loss_v)
                loss_val.append(running_loss_v)
```

```
with torch.no_grad():
  running_loss = []
  for i, data in enumerate(test_loader, 0):
    image, label = data[0], data[1]
    image, label = image.to(device), label.to(device)
    r, g, b = image[:, 0].unsqueeze(1), image[:, 1].unsqueeze(1), image[:, 2].unsqueeze(1)
    utputs_r = model(r.view(-1, 32*32))
    outputs_g = model(g.view(-1, 32*32))
    outputs_b = model(b.view(-1, 32*32))
    outputs = torch.cat((outputs_r.view(-1, 1, 32, 32), outputs_g.view(-1, 1, 32, 32), outputs_b.view(-1, 1, 32, 32)), dim=1)
    loss = criterion(outputs, image)
    running_loss_v += loss.item()


imshow(torchvision.utils.make_grid(outputs.detach().cpu()), 'Pred')
imshow(torchvision.utils.make_grid(data[0]), 'GT')

print('The difference between the Real Images and the Decompressed Images is: ', losses)
```

Images en couleur : Modèle Convolutionnel

Nous allons maintenant re-créer un modèle Convolutionnel de 3 couches. Concernant l'Auto Encoder, il reste identique à celui du précédent modèle convolutionnel de 3 couches. Ce code s'applique indifféremment à une image noir et blanc ou RGB, puisqu'il suffit de stipuler le nombre de channels en entrée de l'Auto Encoder (3 pour une image RGB).
Il n'y a donc rien qui change dans notre code d'entraînement de notre modèle, hormis la input-size, renseignant le nombre de channels présentés par l'image en entrée, qui passe de 1 à 3.
On obtient alors le résultat ci-dessous.



Les images semblent très bien re-construitent. Ce que confirme le taux d'erreur qui est de 0.21. Cela est moins bon que pour une image en noir et blanc (pour un même modèle) mais reste très satisfaisant.

On peut donc en conclure que le modèle à convolution semble bien plus adapté que le modèle MLP dans le traitement d'image, peut importe leur format de couleur des images.