TP 3: Implémentation de module, Gestion de données, Checkpointing et GPU

Ismaël Bonneau & Issam Benamara

Le vrai code, comme les imports, le chargement des données se trouve dans le fichier *tp3.py*. Ce rapport sert à présenter l'avancée de notre travail et notre compréhension du sujet.

Dataset & Dataloader

Nous avons exploré les possibilités des dataloaders et datasets.

Un ***Dataset*** en pytorch: classe encapsulant des données labelisées (ou non) X et Y.

Un ***DataLoader***: interface de manipulation d'un dataset fournissant un itérateur dessus et de nombreuses options utiles. Parmi elles, on peut noter:

- ***batch_size***, qui permet de parcourir le dataloader comme un itérable sur le dataset en nous renvoyant des batches x, y de la taille spécifiée. Très utile.
- ***shuffle***, qui dit au dataloader de re mélanger dans un ordre random les exemples du dataset à chaque itération dessus (= chaque *epoch* de l'apprentissage)
- ***sampler et batch_sampler*** qui permettent de spécifier la méthode de sampling des exemples.
- ***collate_fn*** permet de spécifier la façon dont on veut assembler les données (par défaut, renvoie un tuple (exemple(s), label(s).

Voici un exemple de dataset pour MNIST qui renvoie un couple (Tenseur 1D, label) pour chaque image

In [ ]:

```
class Dataset_MNIST(Dataset):
    def __init__(self, x, y):
        super(Dataset_MNIST, self).__init__()
        self.labels = torch.from_numpy(y)
        self.data = torch.from_numpy(x).float()
    def __getitem__(self, index):
        return self.data[index], self.labels[index]
    def __len__(self):
        return len(self.labels)
```

Le dataloader associé avec une taille de batch de 40 sera créé ainsi:

In [ ]:

```
train_dataset = Dataset_MNIST(X_train, y_train)
trainloader = DataLoader(dataset=train_dataset, batch_size=40, shuffle=True)
```

Implémentation d'un tied auto-encoder

Un auto encoder permet d'encoder des objets dans un espace de dimension plus petite, pour en créer une représentation de plus haut niveau. On guide la construction de cet espace en cherchant à reconstruire les objets de départ d'après l'espace latent de façon la plus précise possible.

Dans le cas d'un tied auto-encoder, les paramètres servant à l'encodage sont les mêmes que ceux servant au décodage. Cela présente plusieurs avantages:

- Cela fait moins de paramètres à apprendre.
- Il s'agit d'une façon de régulariser.
- C'est au final très proche d'une PCA.

In [ ]:

```
class TiedAutoEncoder(nn.Module):
    """ tied AutoEncoder: encoder and decoder share weights"""
    def __init__(self, input_dim, latent):
        super(TiedAutoEncoder, self).__init__()
        self.W = torch.nn.Parameter(torch.randn(input_dim, latent), requires_gra
d=True)
        self.b1 = torch.nn.Parameter(torch.randn(1), requires_grad=True)# bias f
or encoder
        self.b2 = torch.nn.Parameter(torch.randn(1), requires_grad=True)# bias f
or decoder

    def encode(self, x):
        return nn.functional.relu(x @ self.W + self.b1)

    def decode(self, x):
        return torch.sigmoid(x @ self.W.t() + self.b2)

    def forward(self, x):
        return self.decode(self.encode(x))
```

On pourra comparer avec un auto encoder classique:

In [ ]:

```
class AutoEncoder(nn.Module):
    """ classical & simple AutoEncoder """
    def __init__(self, input_dim, latent):
        super(AutoEncoder, self).__init__()
        self.encoder = torch.nn.Linear(input_dim, latent)
        self.decoder = torch.nn.Linear(latent, input_dim)

    def encode(self, x):
        return nn.functional.relu(self.encoder(x))

    def decode(self, x):
        return torch.sigmoid(self.decoder(x))

    def forward(self, x):
        return self.decode(self.encode(x))
```

GPU & checkpoints

In [ ]:

```
class State:
    def __init__(self, model, optim):
    self.model = model
    self.optim = optim
    self.epoch, self.iteration = 0, 0
```

Campagne d'expériences:

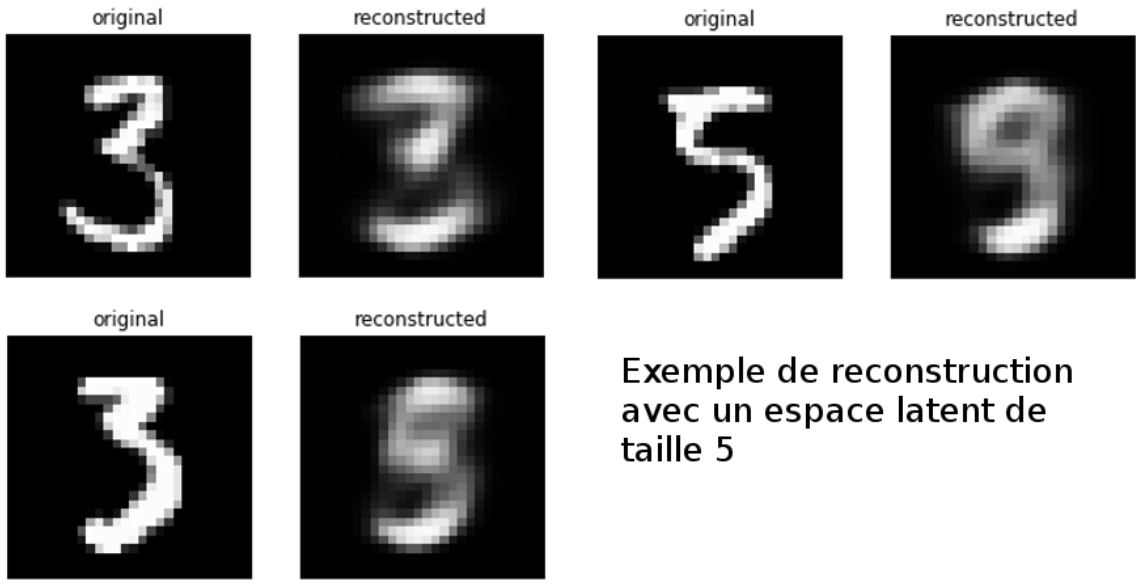
- Evaluer l'influence de la taille de l'espace latent dans la qualité de la reconstruction
- Comparer Tied Autoencoder et Autoencoder classique

Pour cette campagne d'expériences, on prendra les données MNIST, qui sont des images d'écriture manuscrite de chiffres entre 0 et 9. Ces images sont en taille 28x28 pixels

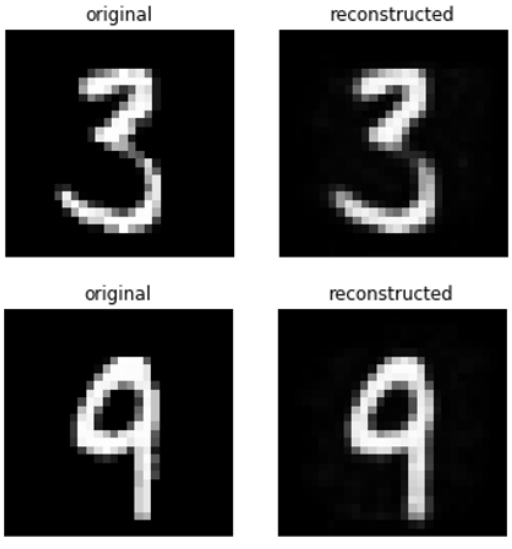
modèle	dimension 10	dimension 50	dimension 100	dimension 300
Tied AutoEncoder	0.0304	0.0094	0.0029	0.0026
AutoEncoder	0.0291	0.0079	0.0043	0.004

Comme attendu, plus l'espace latent est grand, mieux on encode l'information et mieux on arrive à reconstruire l'image de départ. En revanche, un espace latent trop grand favorisera le surapprentissage.

> On remarque bien que le tied auto encoder fait mieux en test que l'auto encoder en grande dimension: il agit comme mode de régularisation. Cependant, en petite dimension, l'auto encoder est meilleur en moyenne.

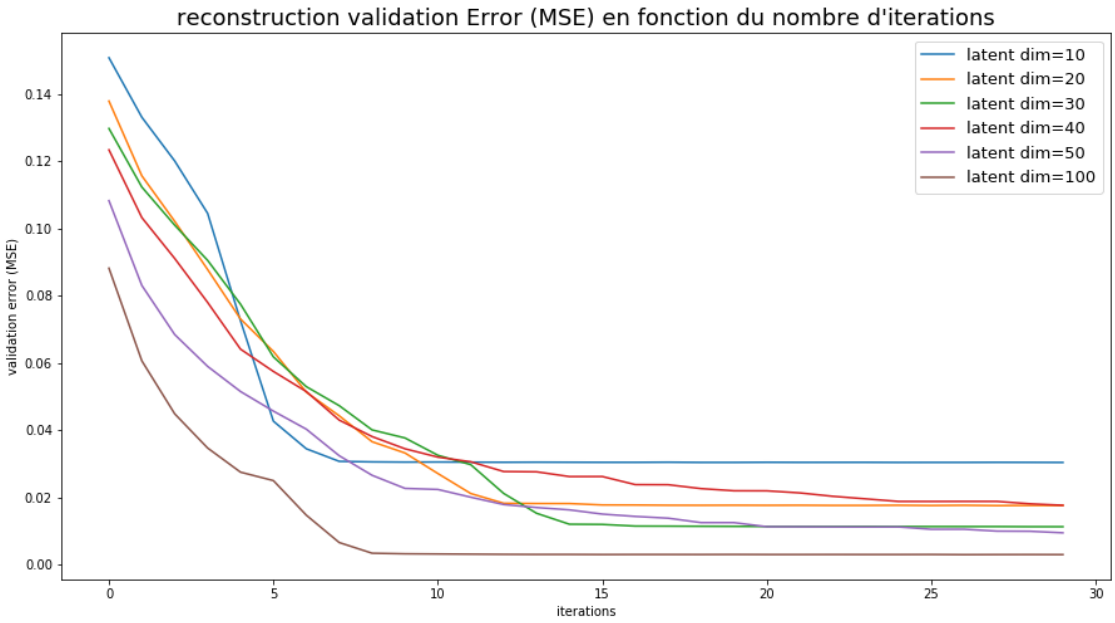


Exemple de reconstruction avec un espace latent de taille 5

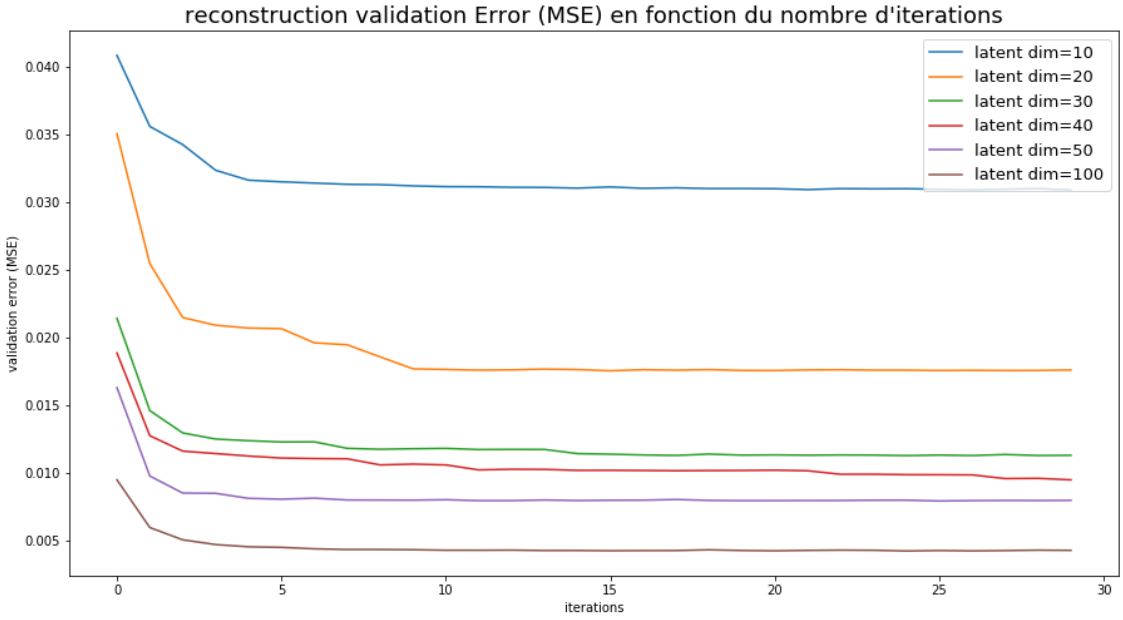


Exemple de reconstruction avec un espace latent de taille 100

Pour le tied autoencoder:



Pour l'autoencoder classique:

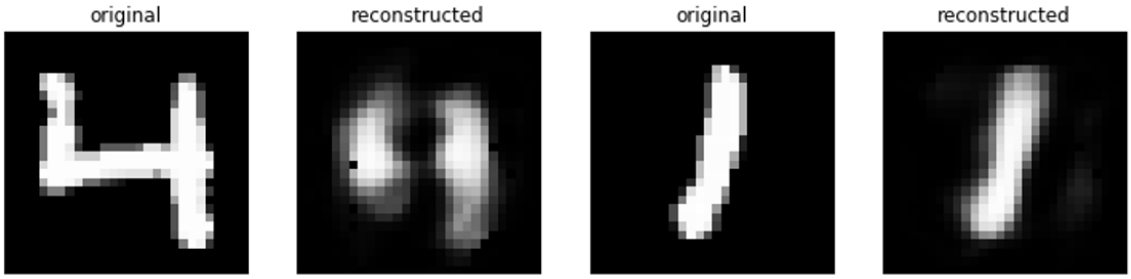


On converge beaucoup plus rapidement avec un auto encoder classique.

Pire reconstruction vs. Meilleure reconstruction
(latent dim = 10)

Pire

Meilleure



Pire reconstruction vs. Meilleure reconstruction
(latent dim = 100)

Pire

Meilleure