Sorbonne Université

Rapport projet Machine Learning

Tan Khiem HUYNH

2022

Table des matières

1 Premier modèle linéaire **1**

2 L'encapsulage et non-linéarité **2**

3 Multi-classification **4**

4 Auto-Encoder **8**

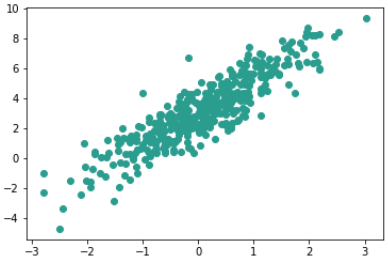
5 Convolution **11**

A Cahier des charges **12**

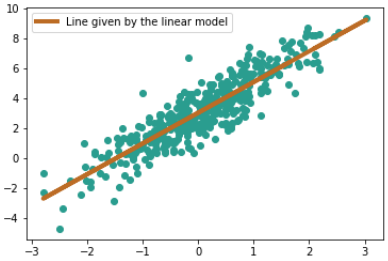
B Manuel utilisateur **13**
B.1 Installation 13
B.2 Utilisation 13
B.3 Outils 14

Premier modèle linéaire

Le premier modèle pour commencer c'est un modèle linéaire simple pour réaliser une régression linéaire. Les données sont générées à partir de la fonction $y = 2x + 3$ en ajoutant des bruits Gaussien



(a) Visualisation des données



(b) La ligne appris par le modèle

On peut contaster l'effet de la **MSELoss** où la ligne se trouve bien au milieu des points

L'encapsulage et non-linéarité

La partie prochaine dédiée à développer un encapsulage qui enchaîne plusieur module en un réseaux neurone complété. Les activations sont considérées également comme une module spéciale qui n'a pas des paramètres.

On va étudier dans cette partie un problème de classification en utilisant un réseaux neurone simple avec une couche cachée et **TanH** comme activation. La loss utilisée est **BinaryCrossEntropy**, avec l'activation **Sigmoid** appliquée directement sur les sortie du réseaux.

Les données étudier sont comme suit :

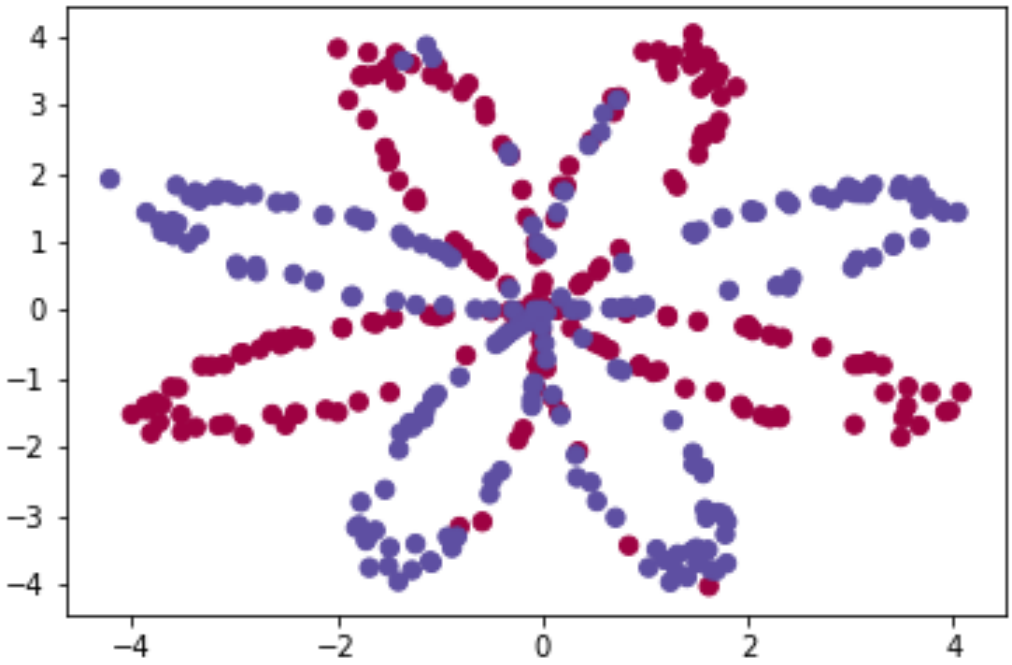


Figure 2.1 – Planar dataset

En suite, on va faire varier le nombre de neurone de la couche cachée et étudier son effet. On obtient donc les résultats comme suivant :

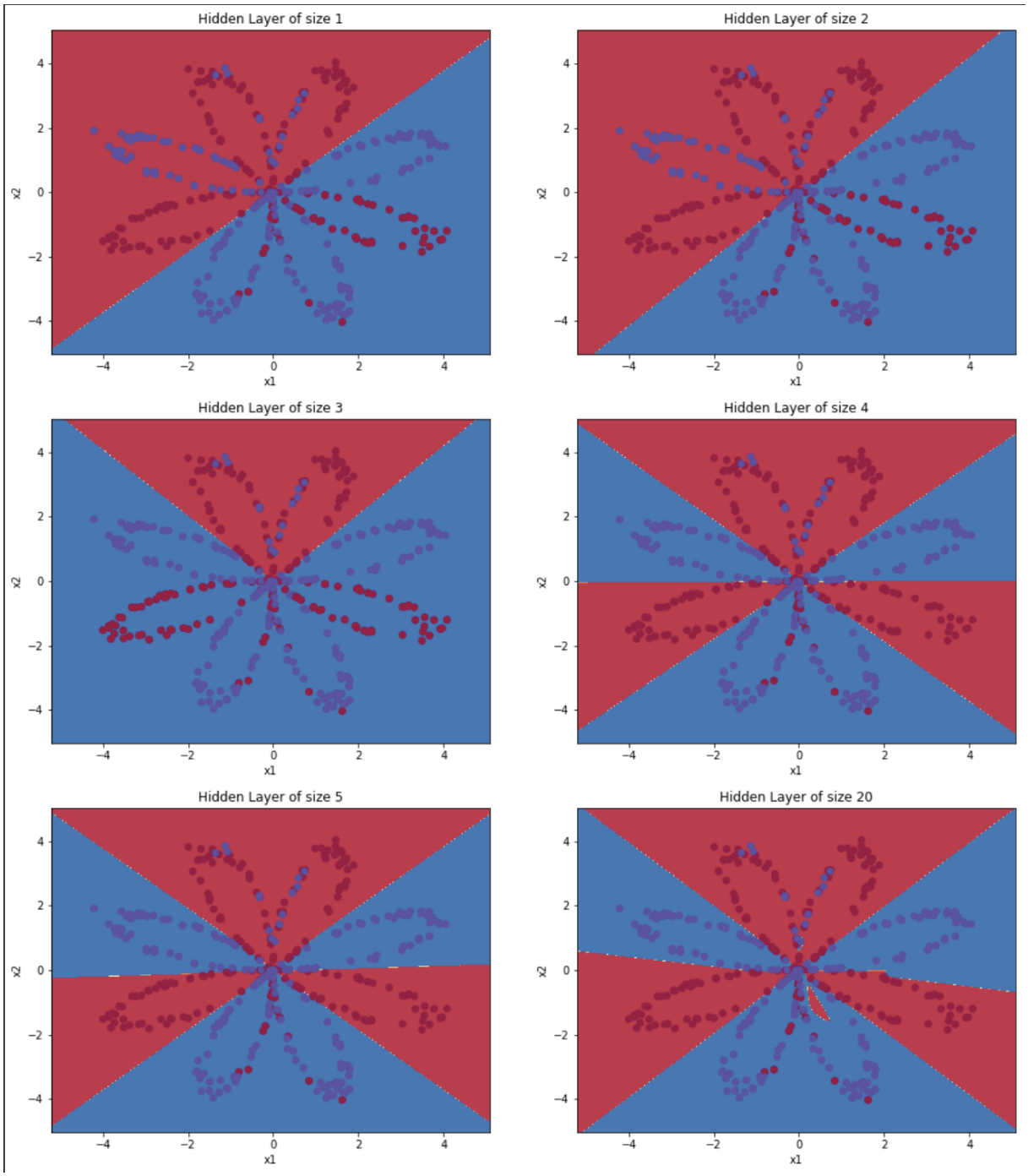


FIGURE 2.2 – Les frontières de décisions de différentes tailles de couche cachée

L'interprétation :

— Les modèles plus grands (avec plus d'unités cachées) sont capables de mieux s'adapter à l'ensemble d'apprentissage, jusqu'à ce que finalement les modèles les plus grands deviennent overfit sur les données.

— La meilleure taille de couche cachée semble être d'environ $n_h = 5$. En effet, une valeur ici semble bien s'adapter aux données sans entraîner également un overfit notable.

Multi-classification

Dans cette partie on étudie la multi-classification sur 2 ensembles de données : **USPS** et **MNIST**. On va faire varier les activations du réseaux neurone utilisé entre **TanH**, **ReLU** et **Sigmoid** et essayer de trouver la plus performante. L'architecture du réseaux neurone utilisé pour **USPS** est :

```
    m = Sequentiel(
        Linear(256, 128),
        activation,
        Linear(128, 64),
        activation,
        Linear(64, 32),
        activation,
        Linear(32, 10)
    )
```

Listing 3.1 – Multi-classif USPS

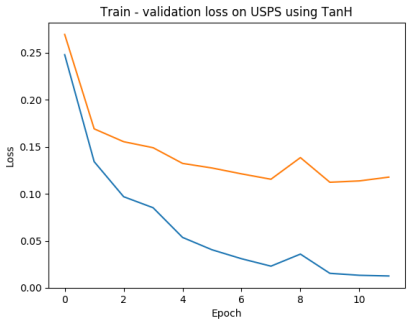
L'architecture utilisé pour MNIST sont assez similaire :

```
    m = Sequentiel(
        Linear(784, 392),
        activation,
        Linear(392, 196),
        activation,
        Linear(196, 98),
        activation,
        Linear(98, 49),
        activation,
        Linear(49, 10)
    )
```

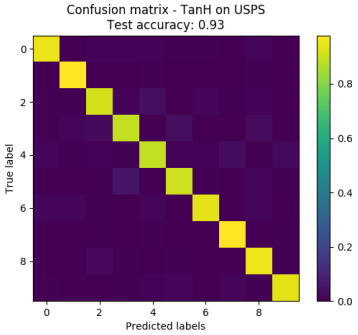
Listing 3.2 – Multi-classif MNIST

La loss utilisé est la **CrossEntropy** combiné avec **SoftMax** comme dans l'énoncé. En plus, on a implémenté une version Stochastic Gradient Descent **SGD** et un méchanism de "early stopping" pour arrêter l'entraînement au bon moment et éviter d'overfit. Pour toutes les expérimentations dans cette partie, on a utilisé un batch de taille 32, learning rate $\alpha = 0.1$

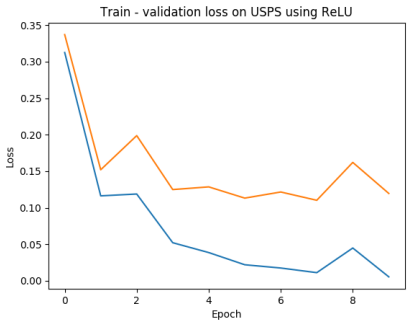
Sur les données **USPS**, on a obtient les résultats suivant avec les différentes activations :



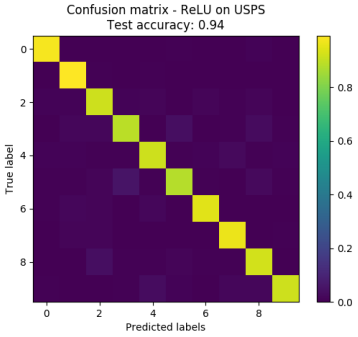
(a) Train-validation loss sur USPS multi-classif avec TanH



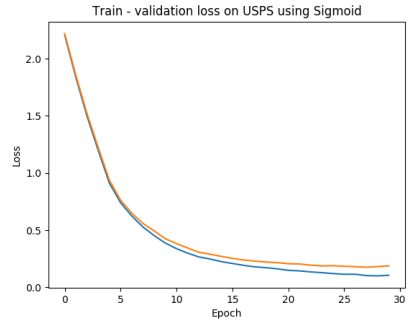
(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe



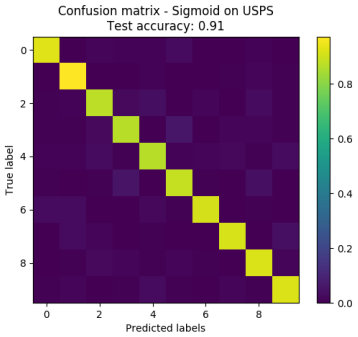
(a) Train-validation loss sur USPS multi-classif avec ReLU



(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe

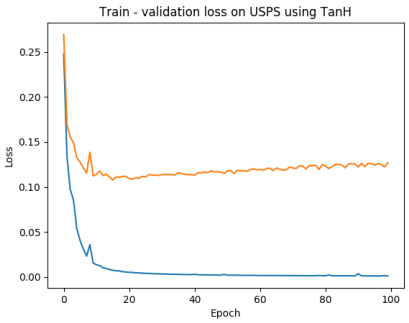


(a) Train-validation loss sur USPS multi-classif avec Sigmoid

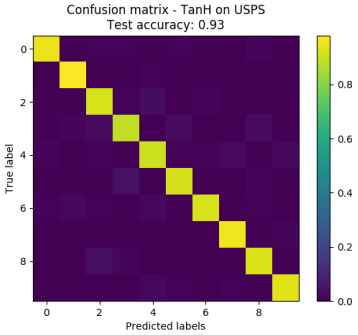


(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe

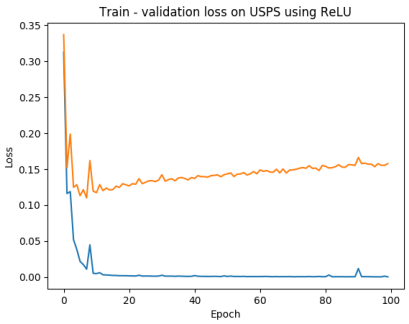
On peut constater que avec **Sigmoid**, les courbes de loss semblent plus "smooth" mais les résultats sur l'ensemble de test sont moins biens que les deux autres. Le méchanism de "early stopping" est aussi assez important, comme on peut voir les résultats sans "early stopping" comme suivant :



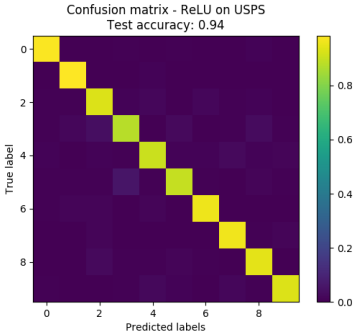
(a) Train-validation loss sur USPS multi-classif avec TanH



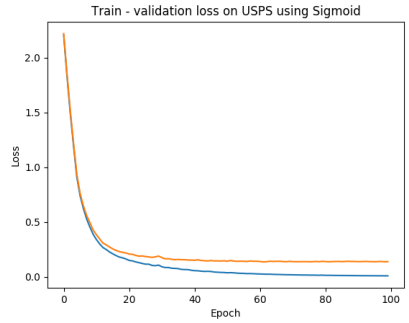
(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe



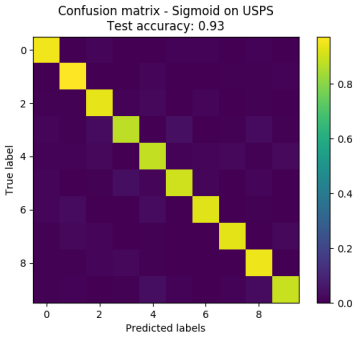
(a) Train-validation loss sur USPS multi-classif avec ReLU



(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe



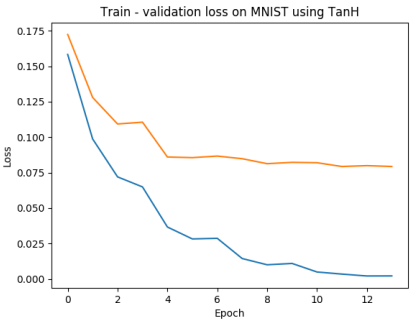
(a) Train-validation loss sur USPS multi-classif avec Sigmoid



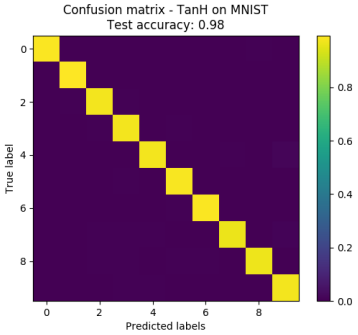
(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe

Si on continue l'entraînement même si notre modèle a convergé, on peut voir que la loss sur l'ensemble de validation commence à ré-augmenter. C'est aussi un bon habitude pour ne pas perdre trop de temps

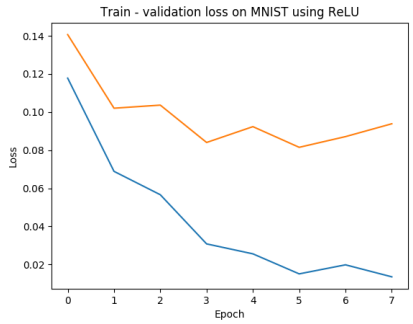
En suit, les résultats obtenus sur **MNIST**. Une étape de normalisation est nécessaire avec **MNIST**, en divisant toutes les valeurs par 255 pour les amener dans $[0, 1]$.



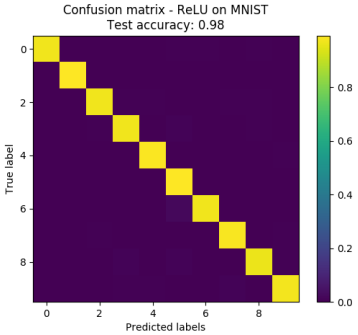
(a) Train-validation loss sur USPS multi-classif avec TanH



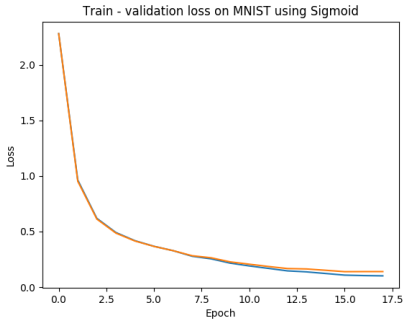
(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe



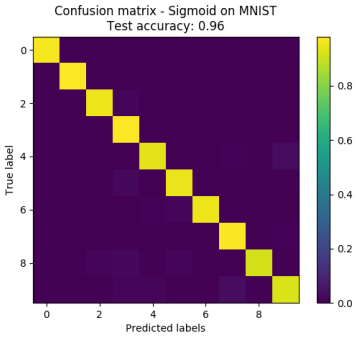
(a) Train-validation loss sur USPS multi-classif avec ReLU



(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe



(a) Train-validation loss sur USPS multi-classif avec Sigmoid

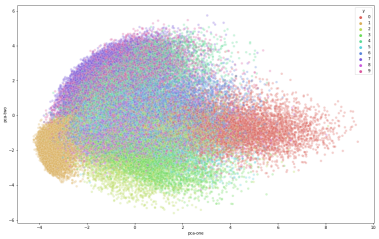


(b) Confusion matrix sur l'ensemble de test, normalisé par chaque classe

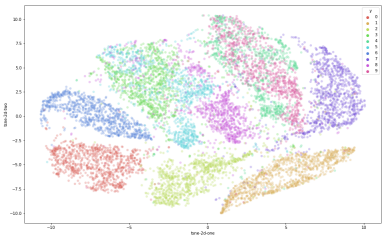
Auto-Encoder

Dans cette partie, on va entraîner un Auto-Encoder sur MNIST. Comme la dernière partie, on va faire varier l'activation entre **ReLU** et **TanH**. Après l'entraînement, la représentation latente construite par la partie encodage du réseaux neurone peut être utilisée pour la classification.

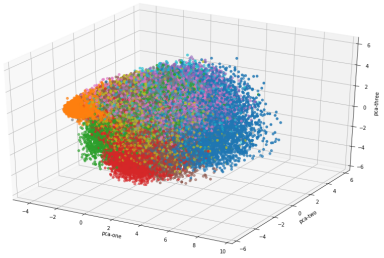
D'abord, on visualise les données en 2D par **PCA** et **t-SNE**



(a) Visualisation de MNIST en 2D par PCA



(b) Visualisation de MNIST en 2D par t-SNE



(c) Visualisation de MNIST en 3D par PCA

L'architecture utilisée est comme suite :

```
    m = Sequentiel(
        Linear(784, 392),
        activation,
        Linear(392, 196),
        activation,
        Linear(196, 98),
        activation,
        Linear(98, 49),
        activation,
        Linear(49, 10),
        activation,
        Linear(10, 49),
        activation,
        Linear(49, 98),
        activation,
```

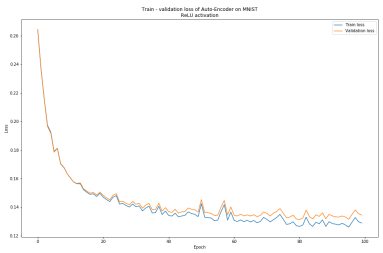
```
16          Linear(98, 196),
17          activation,
18          Linear(196, 392),
19          activation,
20          Linear(392, 884),
21          Sigmoid()
22      )
```

Listing 4.1 – Auto-Encoder MNIST

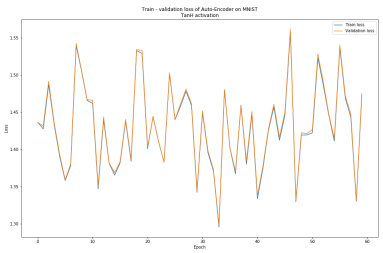
avec activation soit **ReLU**, soit **TanH**.

La loss utilisée est **BinaryCrossEntropy**, l'optimizer est toujours **SGD** et aussi le mécanisme de "early stopping". La taille de batch est 32 et learning rate $\alpha = 0.1$

Les courbes d'entraînement sont suivant :



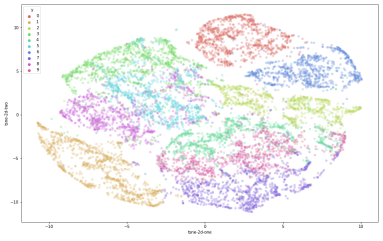
(a) Auto-Encoder train-validation loss on MNIST avec ReLU



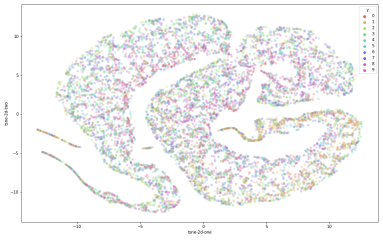
(b) Auto-Encoder train-validation loss on MNIST avec ReLU

Les courbes obtenues avec **ReLU** sont clairement plus "smooth" et stable. Pour expliquer, **ReLU** n'est pas affecté par la disparition du gradient (gradient vanishing) comme **TanH**. De plus, le réseau de neurones que nous utilisons pour l'auto-encodeur est un réseau de neurones assez profond et complexe, ce phénomène devient donc plus évident et affecte encore plus les performances de TanH. Pour cette raison, aujourd'hui, avec les grands réseaux de neurones profonds (CNN, RNN), **ReLU** est souvent préféré.

Visualiser l'espace latente construites par **t-SNE** :



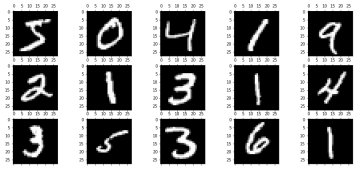
(a) Visualisation de l'espace latente construite en 2D par t-SNE - ReLU



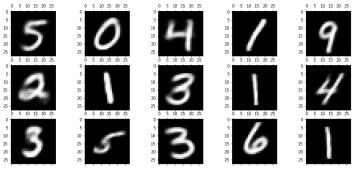
(b) Visualisation de l'espace latente construite en 2D par t-SNE - TanH

On voit que l'espace latent construite par l'auto-encoder utilisant **ReLU** est plus claire et séparable. Donc très probablement on peut se servir cette nouvelle présentation pour faire de la classification en espérant que les résultats obtenus sont prometteux.

Mais tout d'abord, regardons les images réconstruites par notre auto-encoder :



(a) Images MNIST origins



(b) Images réconstruites

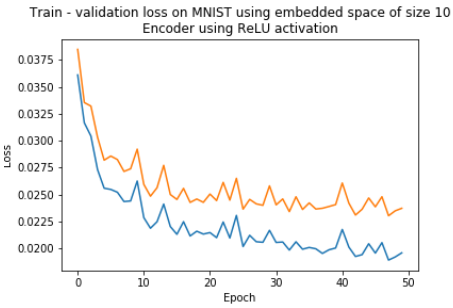
Les deux sont assez similaires.

Ensuite, on va utiliser la nouvelle représentation obtenue avec notre auto-encoder pour faire de la classification. L'architecture utilisée :

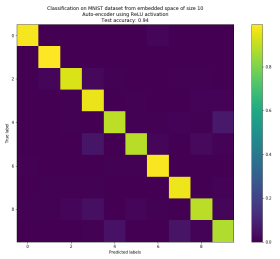
```
1       m = Sequentiel(
2           Linear(10, 20),
3           ReLU(),
4           Linear(20, 40),
5           ReLU(),
6           Linear(40, 10)
7       )
```

Listing 4.2 – Classification from embedded representation MNIST

Avec **CrossEntropy** loss, **SGD** et "early stopping", le résultat obtenu :



(a) Train-validation loss



(b) Confusion matrix

On est à 94% précision sur l'ensemble de test.

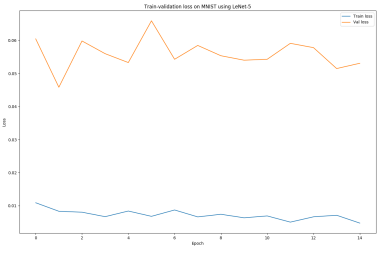
Convolution

Dans cette partie, en inspirant de l'architecture **LeNet-5**, on va faire de la classification sur MNIST avec les couches convolutionnelle 2D. L'architecture utilisée est comme suite :

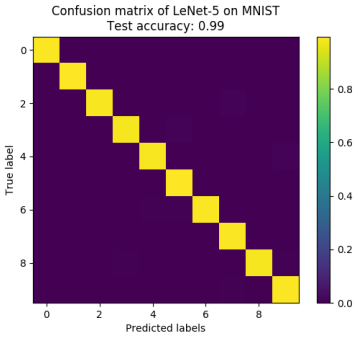
```
    m = Sequentiel(
        Conv2D(kernel_size=5, n_chanel_in=1, n_kernel=6, stride=1, pad=2),
        ReLU(),
        MaxPool(kernel_size=2, stride=2),
        Conv2D(kernel_size=5,n_chanel_in=6,n_kernel=16,stride=1,pad=0),
        ReLU(),
        MaxPool(kernel_size=2, stride=2),
        Flatten(),
        Linear(400, 120),
        ReLU(),
        Linear(120, 10)
    )
```

Listing 5.1 – Classification USPS avec CNN

Avec **SGD**, "early stopping", et **Numba** pour accélérer les boucles Python, les résultats obtenus sont très bons :



(a) Train-validation loss



(b) Confusion matrix

Chapitre A

Cahier des charges

Lors de la première réunion avec notre encadrant, Olivier Sigaud, nous nous sommes fixé les objectifs suivants :

— comprendre l'algorithme SVPG ;
— moderniser son implémentation avec SaLinA ;
— développer des outils de visualisation ;
— reproduire les résultats de l'article original ;
— mettre en avant les cas d'utilisation pertinents.

Manuel utilisateur

B.1 Installation

Pour installer notre projet il suffit de cloner le dépôt github :

```
git clone https://github.com/Anidwyd/pandroide-svpg.git
pip install -e ./pandroide-svpg/
```

Nous recommandons de plus d'installer les librairies suivantes : my_gym, rllab, pybox2D, et le fork d'Olivier Sigaud de salina.

B.2 Utilisation

Des exemples sont disponibles dans le répertoire `tests`. Dans la pratique, on utilise une configuration qui permet de définir les hyper-paramètres des algorithmes, la structure des réseaux de neurones des agents, l'environnement, l'optimizer...

```
config = OmegaConf.create({
    "logger": {
        "classname": "salina.logger.TFLogger",
        "log_dir": "./tmp/",
        "verbose": False,
    },
    "algorithm": {
        "n_particles": 16,
        "seed": 4,
        "n_envs": 8,
        "n_steps": 100,
        "eval_interval": 4,
        "n_evals": 1,
        "clipped": True,
        "max_epochs": 625,
        "discount_factor": 0.95,
        "gae_coef": 0.8,
        "policy_coef": 0.1,
        "entropy_coef": 0.001,
        "critic_coef": 1.0,
        "architecture": {"hidden_size": [64, 64]},
    },
    "gym_env": {
        "classname": "svpg.agents.env.make_gym_env",
        "env_name": "CartPoleContinuous-v1",
    },
    "optimizer": {"classname": "torch.optim.Adam", "lr": 0.01},
})  # Il est aussi possible d'utiliser hydra
```

Listing B.1 – Exemple de configuration

On peut ensuite créer des algorithmes défini dans le package `svpg.algos`. Pour SVPG, le paramètre `is_annealed` permet d'activer l'option d'annealing ou non. Pour lancer un algorithme, on utilise sa fonction `run`.

```
from svpg.algos import A2C, SVPG

a2c = A2C(config) # A2C-Independent
a2c.run()
SVPG(A2C(config), is_annealed=False).run()  # A2C-SVPG
SVPG(A2C(config), is_annealed=True).run()  # A2C-SVPG_annealed
```

Listing B.2 – Exemple d'utilisation des algos

B.3 Outils

Le package `utils` offre plusieurs utilitaires permettant de sauvegarder et charger un algorithme, ou encore d'évaluer un agent.

```
from svpg.utils.utils import save_algo, load_algo
directory = "./runs/A2C"

# Sauvegarde / chargement de A2C
a2c.save_algo(directory)
action_agents, critic_agents, rewards, timesteps = load_algo(directory)

# Evaluation des politiques
eval_rewards = eval_agents_from_dir(directory, n_eval=100, seed=432)
```

Listing B.3 – Exemple d'utilisation des utilitaires

La visualisation des critics et des politiques à été directement été incluse dans le fork d'Olivier Sigaud de salina.

```
from svpg.agents.env immport make_gym_env
env = make_gym_env("CartPoleContinuous-v1")

# Visualisation des critics / politiques
plot_cartpole_critic(critic_agents[0], env, "critic0", directory)
plot_cartpole_policy(action_agents[0], env, "policy0", directory)

# Visualisation des histogrammes
rewards = eval_agents_from_dir(directory)
plot_histograms(rewards, "CartPoleContinuous-v1", save_fig=True)

# Calcul des espaces réduits avec t-SNE
embedded_spaces, rewards = get_embedded_spaces(directory, algo_name="A2C",
    n_eval=100)
# Visualisation des densités de visites des états
plot_state_visitation(directory, a2c_spaces, np.array(a2c_rewards),
    "A2C-Independent", cmap="Reds")
```

Listing B.4 – Exemple d'utilisation des fonctions de visualisation