SIMCLR
Données MNIST

Tristan MARGATE, Ahmed OSMAN, Bourahima COULIBALY

Abstract

Ce rapport se présente comme un benchmark de modèles de deep learning sur les données MNIST, très largement utilisée aujourd'hui pour servir de données type pour comparer les performances des modèles. La difficulté présente ici est que nous utiliserons uniquement 100 données labélisées pour l'entraînement, nous obligeant à trouver des méthodes permettant d'utiliser les données non labélisées afin d'améliorer les performances de nos modèles. En premier lieu sera mis en place une baseline, c'est à dire des modèles d'apprentissage supervisé uniquement sur les 100 données, puis nous présenterons une méthode nommée SIMCLR, qui nous vient de [4], permettant de mettre en place un pré-entraînement du modèle sur les données non labélisées, puis un entraînement supervisé sur les données labélisées. Nous obtenons un gain de presque 7% d'accuracy avec SIMCLR, permettant notre modèle d'atteindre un peu plus de 91% d'accuracy.

Contents

1 Introduction

L'IA, ou encore l'intelligence artificielle, est maintenant un sujet très répandu dans notre génération, notamment grâce aux recherches dans le domaine et à la divulgation des connaissances au grand public en open source. C'est grâce à cette culture de partage que nous sommes en possession d'un article sur le modèle SimCLR (Contrastive Learning for Unsupervised Representation Learning), développé par des chercheurs de Google en 2020, qui est le support de notre projet de classification avec le deep learning. Pour apprécier ce modèle, nous allons le construire ainsi que d'autres modèles, notamment une Baseline, pour mener une étude de performance et un benchmark à la fin.

Pour mener cette étude, nous utiliserons le jeu de données MNIST, qui est un ensemble de données de chiffres manuscrits labellisés composé de 60 000 images d'entraînement et de 10 000 images de test en noir et blanc de taille 28x28 pixels. Il représente des chiffres de 0 à 9 et est idéal pour une étude de classification et de comparaison de classifieurs.

Figure 1: Données MNIST

Tout en se basant sur l'article, notre étude se fera de la manière suivante:

- La construction d'une Baseline (notre référence de performance)
- construction du modèle SimCLR en tenant compte de nos données, en particulier la taille des inputs 28 x 28
- faire un premier entraînement avec les hyperparamètres de l'article et par la suite tourner notre modèle par rapport au performances.
- faire un benchmark entre notre modèle et la Baseline.
- mettre en place une application web pour la prédiction avec les deux modèles.

2 Baseline

Afin de réaliser nos prédictions comme mentionné dans l'introduction, nous allons dans un premier temps développer des modèles de façons intuitives et ainsi mesurer les performances de ceux-ci pour nous en servir de référence, nous appellerons cela dans la suite la **Baseline**.

On précise tout d'abord comment nous allons importer nos données, ainsi que la façon dont nous allons créer nos loaders afin de charger les données. Comment on peut le voir ici : [5], On procède à une simple transformation de nos données de la façon suivante :

- transforms.ToTensor: Cette transformation convertit l'image en un tenseur PyTorch.
- transforms.Normalize: Cette transformation normalise les valeurs des pixels de l'image en soustrayant la moyenne et en divisant par l'écart type. Cela peut aider à améliorer la convergence de l'entraînement et à réduire les effets des variations d'éclairage. Les valeurs de pixel mean et pixel std sont les moyennes et les écart-types de chaque canal de couleur de l'ensemble de données.

afin de les transformer en tenseur et on les normalise de la façon ci-montré afin que nos données soient utilisables par nos modèles par la suite. Nous créons ensuite nos loaders de la façon suivante (voir 5) :

- Train_unlabeled_loader : loader pour les 59 900 données du train set non labélisées.
- Train_labeled_loader : loader pour les 100 du train données labélisées.
- test_loader : loader pour les 10 000 données du test labélisées

On précise que pour le loader des données labélisées, une stratification a été mise en place afin que 10 images de chaque chiffres soit présent, afin de ne pas se retrouver avec un déséquilibre pour l'entraînement de nos modèles.

Maintenant que nous avons défini la façon de crée nos loader, nous allons décrire la façon dont nous allons entraîner nos modèles, pour cela nous avons besoin :

- Du modèle à entraîner
- Du nombre d'epochs
- De l'optimiser
- Du critère à minimiser
- et enfin nous lui donnerons quelques paramètres en plus

Après avoir définis nos paramètres d'entrés du modèle, on peut faire notre entraînement comme défini dans l'annexe (voir 5). On peut noté que nous avons rajouté un paramétrage permettant de diminuer le learning rate au fil des epochs si nécessaire, ainsi qu'un earlystopping, afin de stopper l'entraînement de notre modèle si la loss sur la loader de validation se met à croître de façon répété pendant plusieurs epochs. On précise que le batch size pour nos données d'entraînement sera égale à 50 tout au long de ce rapport (suite à de nombreux essais avec des batch sizes différents, 50 paraît être une bonne valeur pour entraîner nos modèles)

Maintenant que nos principales fonctions sont définis, nous pouvons à présent créer notre réseaux de convolution (CNN) afin de l'entraîner et regarder ses performances.

2.1 Réseaux de convolution

Dans cette sous-partie, nous allons mettre en place un réseau de convolution un minimum complexe (dans le sens qu'il contient un nombre assez grands de paramètres) afin de l'entraîner sur nos données, ce choix est en fait une conséquences d'une multitude d'essais afin de trouver un réseau de convolution ayant des bonnes performances, afin d'avoir une baseline solide (ce CNN nous vient de [5])

Il s'agit de la succession de trois réseaux suivi d'un layer dense, chaque réseau est définit de la façon suivante :

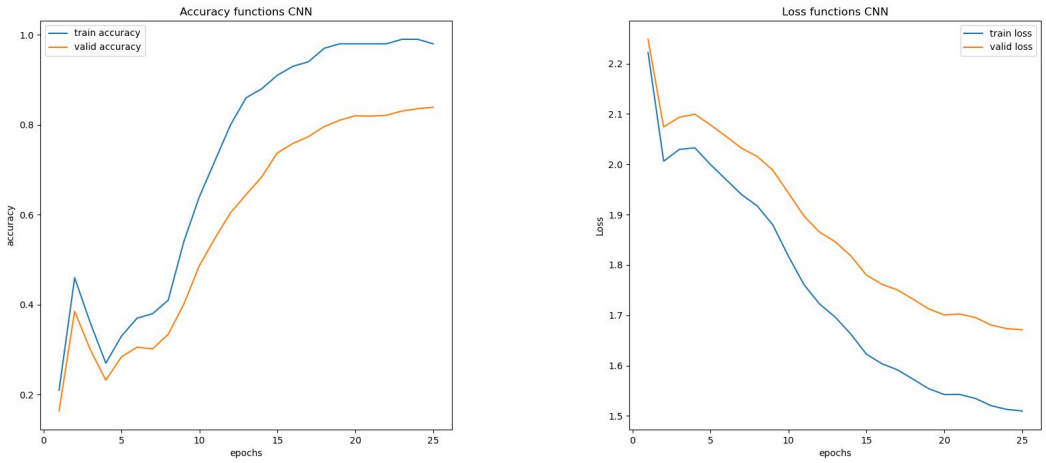
- Un Conv2d (réseaux de convolution)
- Une batch normalisation
- Une fonction Relu
- Un Max Pooling
- Un dropout

On peut voir l'architecture directement dans l'annexe (5).

Pour entraîner notre modèle, nous allons prendre la paramétrisation suivante : le nombre d'epochs sera égale à 25, notre optimizer sera un Adam avec un learning rate égale à 0.001, notre critère sera la fonction CrossEntropy telle que définit dans le module **Torch.nn**. Notre loader de training sera le loader des données labélisées avec les 100 données labélisées, et notre loader de validation sera le test loader avec les 10 000 données labélisées à tester. Nous mettrons en place un scheduler pour le learning rate avec une patience égale à 3 et un factor de 0.5, se traduisant donc par une division du learning rate par 2 au cas ou la loss sur le training augmenterait sur 3 epochs consécutifs. Un earlystopping sera également mis en place sur la loss de training avec une patience égale à 8, stoppant l'entraînement si la loss sur le train augmente pendant 8 epochs d'affilié. On précise bien ici que notre scheduler et

notre earlystopping est réglé sur la validation du training et non de la validation, car les données de validation ne sont pas censés être connus, nous devons uniquement prendre en compte les 100 données labélisées.

Pour un entraînement, on obtient les courbes d'accuracy et de loss suivantes (2) :



(a) Accuracy CNN

(b) Loss CNN

Figure 2: Loss CNN

On peut constater que notre modèle performe très bien, notre accuracy avoisine les 85% et notre loss diminue. Afin d'avoir une bonne observation, nous allons répéter l'opération plusieurs fois et prendre les valeurs pour l'accuracy que prend notre modèle après 25 epochs, ainsi nous pourrons en avoir une moyenne et un écart-type, permettant d'assurer une bonne vision de notre modèle. Nous répéterons l'opération uniquement 5 fois pour des soucis de temps d'entraînement, on obtient les performances suivantes 2.1

Model	Accuracy
CNN	0.8462 ± 0.0042

Table 2.1: Accuracy CNN

2.2 Transfert learning

Une deuxième idée de modèle à intégré pour obtenir des performances dans notre baseline serait d'utiliser du transfert learning, via un modèle déjà entraîné, afin de profiter de leur précédant entraînement coûteux sur divers données pour prédire nos données. Nous avons donc envisagé d'utiliser le célèbre modèle Resnet 50, que l'on peut retrouver ici [[6]]. Resnet étant principalement construit pour des images à 3 channels (étant des images colorés), il sera nécessaire de modifier le premier réseau de convolution, et le dernier layer dense afin de ramener sur seulement 10 labels au lieu de 1000 comme effectué par défaut. Cela se fait par le code suivant (voir 5)

En utilisant les mêmes hyperparamètres que pour notre CNN précédemment établis, dans les mêmes conditions d'entraînement, nous obtenons les performances suivantes 3:

Après plusieurs essais en changeant la valeur des hyperparamètres, les résultats demeurent très bas, seulement 30% d'accuracy produit par le transfert learning sur Resnet 50. Cela peut s'expliquer par le fait que Resnet 50 est entraîné pour des images en 3 channels, et que modifier la première convolution entraîne certainement des pertes de performances énormes.

Notre baseline reposera donc sur le résultat qu'a donné notre réseau de convolution ci-dessus, l'objectif maintenant est d'utiliser l'article [4] pour concevoir une méthode permettant d'utiliser les

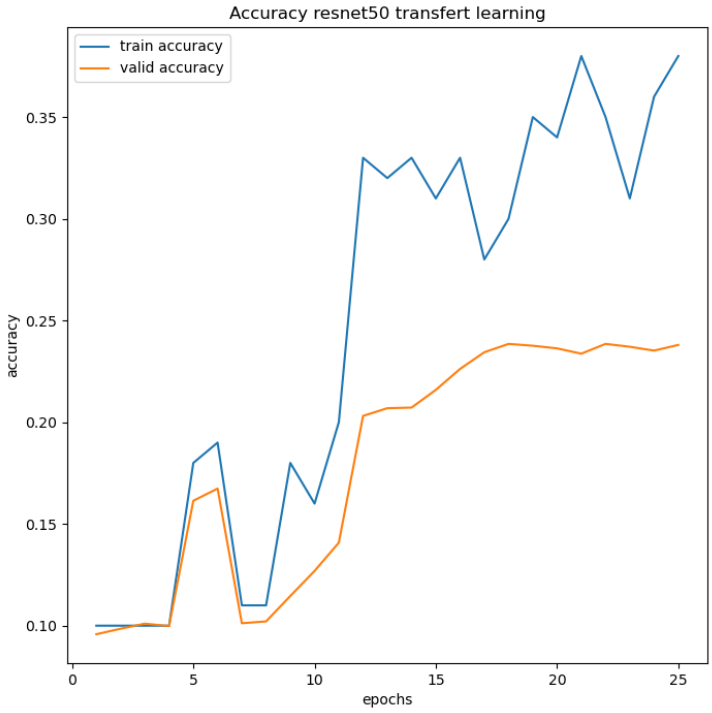


Figure 3: Accuracy resnet 50 transfert learning

données non labélisées, que nous n'avons pas utilisé jusqu'à présent, afin de pratiquer une méthode de pré-entraînement en amont de notre entraînement, dans le but d'obtenir des meilleurs performances.

3 SimCLR

SimCLR est un framework de l'état de l'art pour l'apprentissage contrastif auto-supervisé qui est concentré sur les représentations visuelles significatives à partir de larges ensembles de données. L'architecture du modèle SimCLR se compose de trois éléments principaux : **réseau d'encodage**, une **pipeline d'augmentation des données** et une fonction de **perte contrastive**. Le réseau d'encodage est chargé d'extraire des caractéristiques de haut niveau des images d'entrée, qui sont ensuite utilisées pour apprendre des représentations significatives. La pipeline d'augmentation des données est conçu pour générer diverses vues de la même image, ce qui encourage le modèle à apprendre des représentations robustes non sensible aux changements de point de vue et de conditions d'éclairage. Enfin, la fonction de perte contrastive est utilisée pour entraîner le modèle en comparant des paires d'images transformées, où les paires positives sont des images provenant de la même image originale et les paires négatives proviennent d'images différentes. Cette fonction de perte encourage le modèle à apprendre des représentations qui sont similaires pour les paires positives et dissemblables pour les paires négatives. En combinant ces trois composantes, SimCLR a obtenu des résultats impressionnant (proche de l'apprentissage supervisé) dans la classification.

3.1 Data augmentation

La data augmentation est un technique ou un ensemble de technique d'augmentation de données utile pour augmenter la diversité des données d'entraînement et améliorer la capacité du modèle à généraliser vers de nouvelles images. Donc on peut aisément dire que la data augmentation une partie important dans la généralisation d'un modèle mais pour ce qui concerne notre étude et le SimCLR, la data augmentation est l'une des parties les plus important car si pas d'augmentation, la similarité

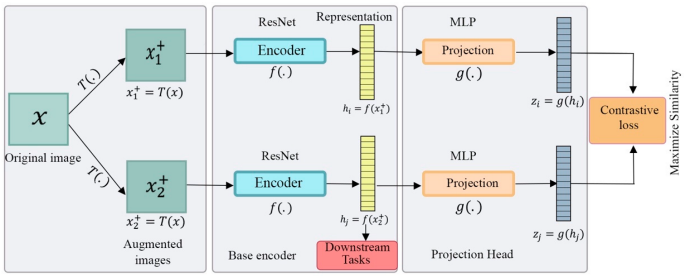


Figure 4: Architecture du modèle SimCLR

sera toujours égale pour près que égale a 1 et inversement si la data augmentation n'est pas adapter aux données et pas de bonne qualité, le modèle aura du mal a trouvé les similarité dont dans les deux cas modèle le sera de mauvaise qualité. [1]

Pour ce qui concerne les transformations de nos données de pre-apprentissage, ils utilises les transformation suivantes:

- transforms.RandomHorizontalFlip: Cette transformation effectue une rotation aléatoire de l'image à l'horizontale avec une probabilité de p.
- transforms.RandomResizedCrop: Cette transformation recadre l'image de manière aléatoire pour obtenir une taille de size x size pixels et ajuste l'échelle du recadrage entre scale[0]% et scale[1]% de la taille de l'image originale.
- transforms.RandomRotation: Cette transformation effectue une rotation aléatoire de l'image dans un intervalle de degrés spécifié. Dans ce cas, les images sont tournées de manière aléatoire dans un intervalle d'angles.
- transforms.RandomPerspective: Cette transformation applique une transformation de perspective aléatoire à l'image, ce qui simule une transformation de la caméra. Le paramètre distortion scale contrôle l'intensité de la distorsion.
- transforms.GaussianBlur: Cette transformation applique un flou gaussien à l'image, ce qui peut aider à réduire le bruit et à améliorer la robustesse de l'algorithme aux variations d'éclairage. Les paramètres kernel size et sigma contrôlent la taille du noyau de convolution et l'intensité du flou. Ces transformations visuellement donne :

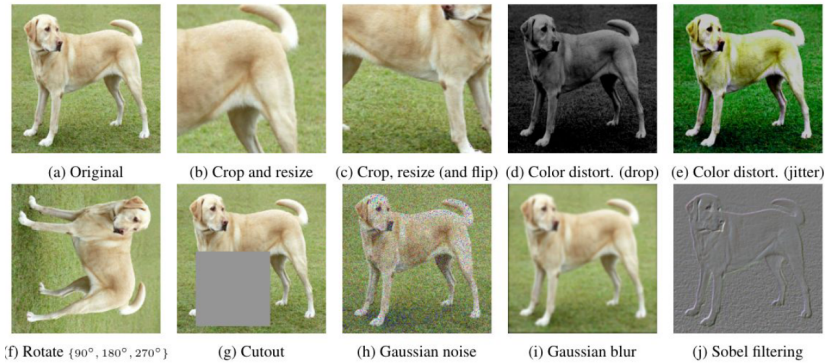


Figure 5: Rotation de 6 ou 9

nous avons essayés d'une part de faire exactement les même que dans l'article mais après nous décidés supprimer certaines (en relation avec la couleur) ou contraindre (Rotation, Elastic, cutout et resize) certaines transformations car elles n'étaient pas adaptées a nos image par exemple:

- pour une rotation supérieur 90° ou inférieur -90°, le 6 et 9 sont indiscernables (contraint en -45 et 45)

- les transformation sur la couleur de l'image pas besoin que nos images sont en blanc et noir.
- Cutout et resize utilisé séparément peuvent faire disparaître l'information sur l'image. Par exemple le 9 et 8 découper peut donné un 0 ou encore le 7 découper peut donné un 1.

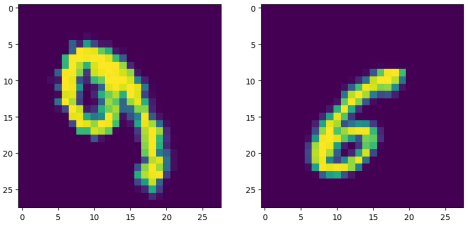


Figure 6: Rotation de 6 ou 9

Finalement pour la data augmentation, nous utiliseront les transformation suivantes mais avec les contraintes appropriés (Voir code 5) Les transformations seront de la forme : Un exemple de

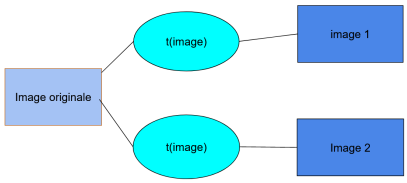
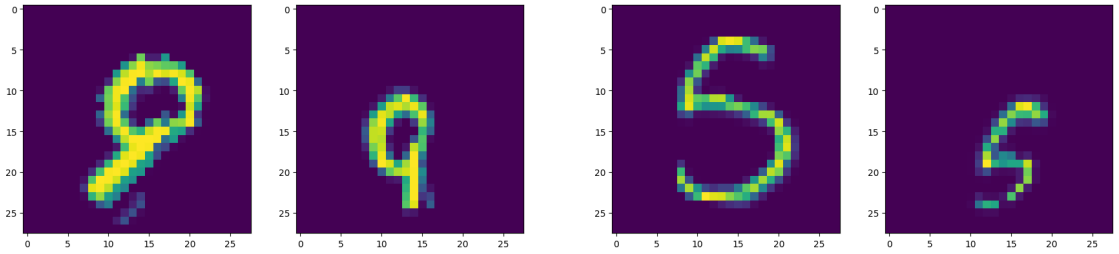


Figure 7: Architecture de la transformation

transformation d'image mnist est le suivant:



(a) 2 transformations de 9

(b) 2 transformations de 5

Figure 8: Image de deux couples de transformations

3.2 Encodeur

L'encodeur ResNet50 est un réseau de neurones convolutionnel profond qui a été largement utilisé dans diverses tâches de vision par ordinateur. Plus précisément, ResNet50 a été utilisé comme architecture d'encodage dans le modèle SimCLR, qui est un cadre d'apprentissage non supervisé pour l'apprentissage des représentations visuelles. L'encodeur est chargé d'extraire des caractéristiques de haut niveau des images d'entrée, qui sont ensuite utilisées pour entraîner une fonction de perte contrastive (Contrastive Loss) afin d'apprendre des représentations significatives des données. ResNet50 a démontré des performances impressionnantes sur plusieurs ensembles de données de référence, tels que ImageNet, et est connu pour sa capacité à gérer des tâches de reconnaissance visuelle à grande échelle. Avec l'aide de l'encodeur ResNet50, SimCLR a obtenu des résultats de pointe dans diverses tâches de classification d'images et de segmentation sémantique.

Au lieu d'utiliser un Encoder avec une architecture complexe comme le ResNet50 (utilisé dans l'article), nous avons utilisé un réseau de convolution de 3 layers, avec des fonctions d'activations ReLu, un dropout de 0.2 et un maxpooling avec un kernel size et stride de taille 2 pour chaque layer.

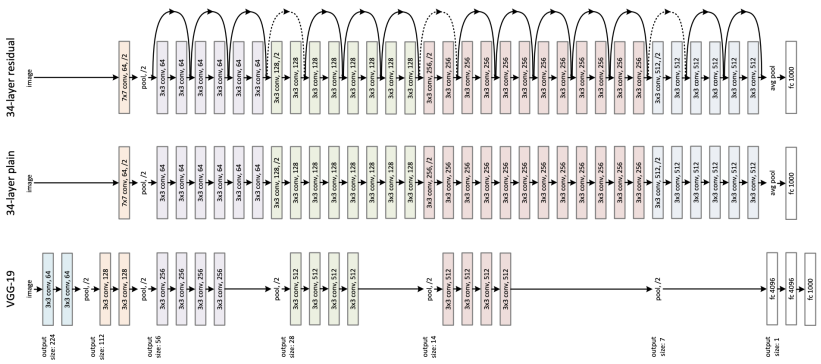


Figure 9: ResNet architecture

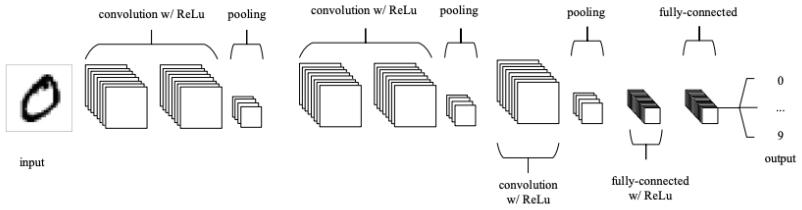


Figure 10: Exemple d'un réseau de convolution

3.3 Projection Head - MLP

La tête de projection (MLP) est un composant crucial du modèle SimCLR. Comme son nom l'indique, la tête de projection est chargée de projeter les caractéristiques à haute dimension extraites par l'encodeur dans un espace à plus faible dimension (de 2048 à 128 dimensions), qui sera la sortie du modèle.

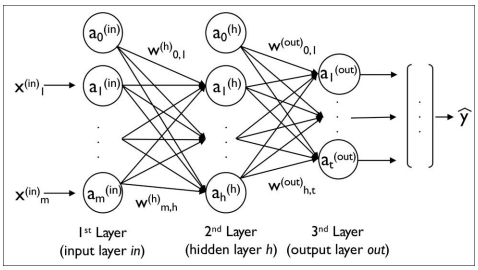


Figure 11: Exemple d'un réseau MLP

La tête de projection est généralement mise en œuvre sous la forme d'un perceptron multicouche (MLP) **avec une couche cachée** qui applique une transformation linéaire suivie d'une fonction d'activation non linéaire **(ReLu)** aux caractéristiques d'entrée.

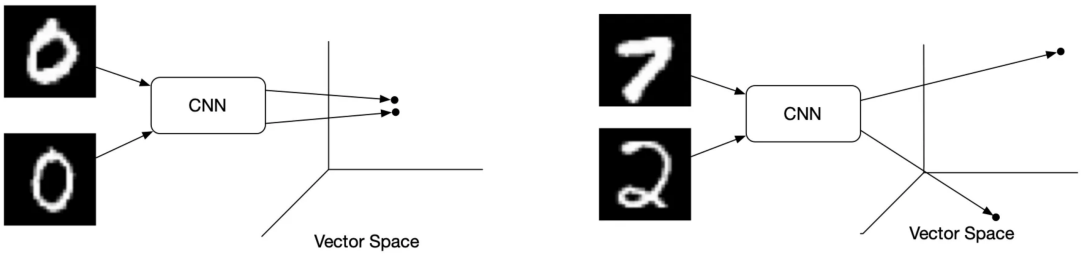
Ce processus permet non seulement de réduire la dimensions, mais aussi d'apprendre une représentation plus compacte et plus significative des données visuelles qui peut être utilisée pour des tâches ultérieures comme la classification. Le choix de l'architecture MLP pour la tête de projection est essentiel pour le succès du modèle SimCLR, car il affecte la qualité des représentations apprises et la capacité de généralisation du modèle.

3.4 Pré-entraînement

Pour créer notre modèle de Pré-entraînement, on relie notre Encoder suivi de notre Projection Head dans une seule classe (voir [7]). (voir code) 5

3.5 Contrastive Loss

La perte contrastive prend la sortie du réseau pour un exemple positif et calcule sa distance par rapport à un exemple de la même classe et la compare à la distance par rapport aux exemples négatifs. En d'autres termes, la perte est faible si les échantillons positifs sont encodés dans des représentations similaires (plus proches) et si les exemples négatifs sont encodés dans des représentations différentes (plus éloignées). 12



(a) exemple positif (positif pairs)

(b) exemple négatif (negative pairs)

Figure 12: source : [3], exemples positifs et négatifs dans un espace vectoriel

Pour ce faire, on prend les distances en cosinus des vecteurs et on traite les distances résultantes comme des probabilités de prédiction à partir d'un réseau de catégorisation typique. L'idée principale est de pouvoir traiter la distance de l'exemple positif et les distances des exemples négatifs comme des probabilités de sortie et utiliser la perte d'entropie croisée. (voir code) 5

$$\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$$

$$\ell_{i,j} = -\log \frac{\exp\left(\text{sim}\left(\boldsymbol{z}_i, \boldsymbol{z}_j\right)/\tau\right)}{\sum_{k=1}^{2N} \mathbb{1}_{\{k \neq i\}} \exp\left(\text{sim}\left(\boldsymbol{z}_i, \boldsymbol{z}_k\right)/\tau\right)}$$

avec la métrique de similarité : $\text{sim}(u, v) = \frac{u^T v}{\|u\|\|v\|}$ (cosine similarity) qui nous sert a comparer les représentations produites par la tête de projection (projection head)

Ici, z_i et z_j sont les vecteurs de sortie obtenus à partir de la tête de projection et τ désigne le paramètre de température. La perte est appelée perte d'entropie croisée normalisée à l'échelle de la température *(NT-Xent Loss : Normalized Temperature-scaled Cross Entropy Loss)*

La perte contrastive ressemble étrangement à la fonction softmax. C'est en effet le cas, avec l'ajout d'un vecteur de similarité et d'un facteur de normalisation de la température. La fonction de similarité est simplement la distance cosinus dont nous avons parlé précédemment. L'autre différence est que les valeurs du dénominateur sont la distance en cosinus entre l'exemple positif et les échantillons négatifs. Ce n'est pas très différent de CrossEntropyLoss. L'intuition ici est que nous voulons que nos vecteurs similaires soient aussi proches de 1 que possible, puisque $-log(1) = 0$, c'est la perte optimale. Nous voulons que les exemples négatifs soient proches de 0 , car toute valeur non nulle réduira la valeur des vecteurs similaires.

3.6 Hyper paramètres

- Nombre d'époch s : 100
- Temperature τ : 0.7
- Batch Size : 64 , 256 , 1024
- Learning Rate : 0.001

D'après l'article, on sait que le modèle apprend mieux avec des batch size plus large, mais on a pas vraiment remarquer une amélioration significative sur les données MNIST.

3.7 Optimizer

On utilise Adam Optimizer avec un :

- learning rate de 0.001
- betas = (0.9, 0.999) *"betas" est un paramètre qui contrôle les taux d'actualisation des estimations de la première et de la deuxième moment, et qui en pratique, peut influencer la vitesse et la qualité de la convergence de l'algorithme d'optimisation.*
- eps = 1e-08 *"eps" est un petit nombre positif qui est utilisé pour éviter la division par zéro lors de la normalisation des gradients au carré*
- weight decay = 1e-3 (décroissance de poids) *"weight_decay" est un paramètre utilisé pour régulariser le modèle de réseau de neurones et éviter le surapprentissage.*

On utilise aussi un "scheduler" (StepLR avec un step_size = 20) qui est une technique qui ajuste le taux d'apprentissage en fonction de l'avancement de l'entraînement du modèle. Il peut être utilisé pour réduire le taux d'apprentissage au fur et à mesure que le modèle se rapproche de la convergence, ce qui permet de stabiliser l'entraînement et d'éviter le sur-ajustement.

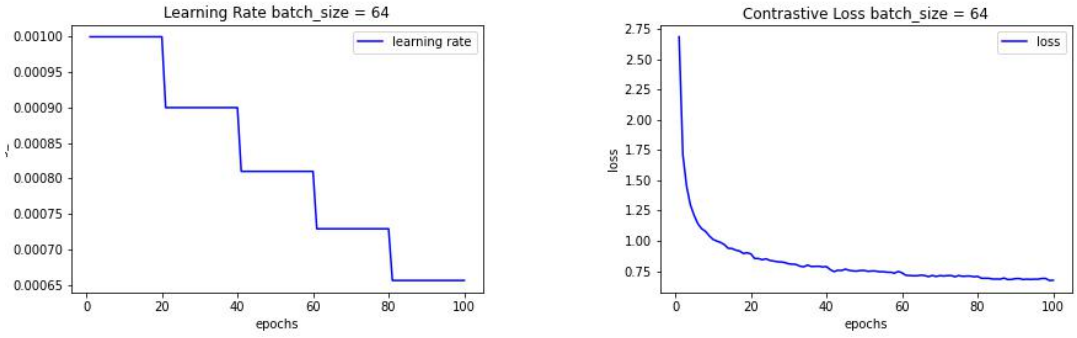
3.8 Linear Classifier

Après avoir terminer la phase de pré-entraînement, on obtiens les poids du modèle de pré-entraînement, ensuite on enlève notre Projection Head (on garde que l'Encoder) et on rajoute un classifieur Linéaire, pour qu'on puisse faire notre classification. (voir code)

4 Benchmark

Dans cette partie, nous allons nous intéresser à utiliser la méthode SIMCLR pour évaluer ses performances et surpasser les performances de notre CNN précédemment établi. Nous allons, conformément à la description de la méthode vu précédemment, mettre en place un pré-entraînement sur les données non labélisées avec la contrastive loss afin d'entraîner notre encoder suivi de notre projection head (un layer dense). Pour ce faire, nous nous sommes inspiré fortement du code présent dans [5] afin de réaliser ce pré entraînement, nous pouvons voir notre annexe notre fonction de réalisant celle-ci (voir 5)

Nous allons réaliser nos pré-entraînement telle que définit avec les hyperparamètres, l'optimizer et la loss choisis précédemment. On rajoute un scheduler afin de diminuer très légèrement notre learning rate de 0.00005 tous les 20 epochs. Ainsi, avec un batch size de 64, on obtient les graphiques suivant 13 :



(a) learning rate batch size = 64

(b) contrastive loss batch size = 64)

Figure 13: pré entraînement : batch size = 64

De même, pour un batch size plus élevé (1024), on obtient les mêmes graphiques, on remarque simplement que la loss est plus élevé, cela étant normal par construction de la contrastive loss (voir 16)

Une fois cette étape réalise, il nous faut donc freezer les poids de l'encoder remplacer la projection head par soit

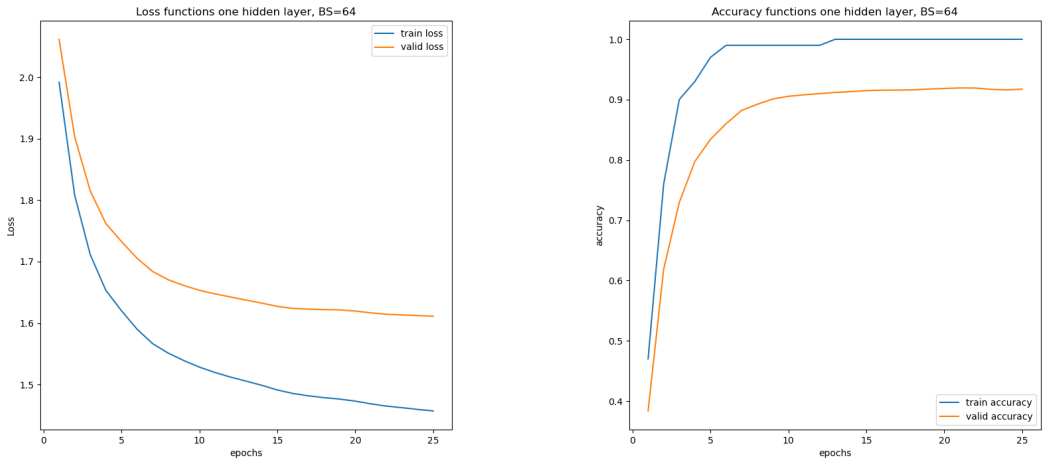
- un layer dense
- un MLP

qui projettera alors sur nos 10 neurones, représentant nos 10 labels possible, puis à l'entraîner sur les 100 données labélisées afin d'obtenir nos performances.

4.1 Un layer dense

Concentrons nous ici sur les performances dans le cas ou l'on remplace la projection head par un simple layer dense. Alors dans ce cas, en effectuant un training de la même manière que pour nos modèles de Baseline, avec les mêmes hyperparamètres.

On obtient alors, pour là encore un batch size de 64, les graphiques suivant 14



(a) Loss batch size = 64

(b) Accuracy batch size = 64)

Figure 14: Un layer dense : batch size = 64

On constate alors une nette amélioration des performances par rapport à notre baseline, le pré entraînement semble très efficace pour augmenter l'accuracy. Pour un batch size de 1024 pour le pré entraînement, la conclusion est la même, on peut voir les graphiques ici 17

Après 5 répétitions du training, on peut obtenir une bonne approximation des performances de notre modèle, on obtient le tableau 4.1

Model	batch size = 64	batch size = 256	batch size = 1024
(SIMCLR) 1 layer dense	0.9194 ± 0.0030	0.9129 ± 0.0004	0.9112 ± 0.0024

Table 4.1: Accuracy 1 layer dense

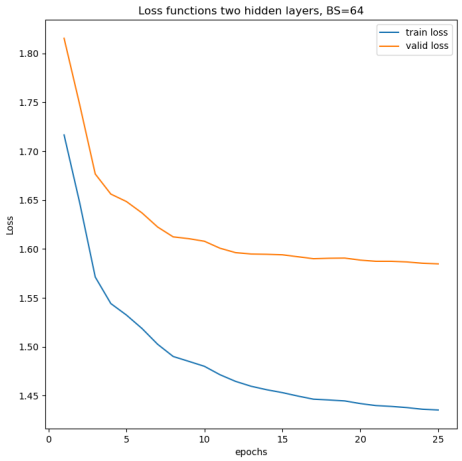
On remarque alors que l'augmentation de la taille du batch sur le pré entraînement n'a en réalité aucun effet sur l'accuracy lors de l'entraînement, il n'a pas l'air intéressant d'augmenter la taille de celle-ci.

4.2 MLP

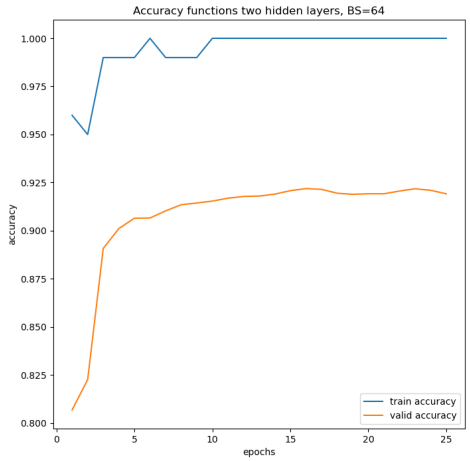
Ici, on s'intéresse au fait de mettre plusieurs layers dense en sortie de l'encoder dans le but d'augmenter notre score lors de l'entraînement. Il est possible, par exemple, de mettre deux layers dense en sortie d'encoder

- Un layer allant d'un espace de 2048 à 512
- Un deuxième layer allant d'un espace de 512 à 10

Ainsi, notre modèle pourrait s'avérer plus performant. Là encore, avec un encoder pré entraîné sur un batch size de 64, on obtient les performances suivantes :



(a) Loss batch size = 64



(b) Accuracy batch size = 64)

Figure 15: Deux layers dense : batch size = 64

Quand là aussi on expérimente 5 fois avec les trois tailles de batch sizes choisis, on peut obtenir une approximation de l'accuracy porté par le modèle. On peut résumer toutes les performances dans ce tableau suivant 4.2

Model	batch size = 64	batch size = 256	batch size = 1024
(SIMCLR) 1 layer dense	0.9194 ± 0.0030	0.9129 ± 0.0004	0.9112 ± 0.0024
(SIMCLR) 2 layers dense	0.9193 ± 0.0022	0.9180 ± 0.0026	0.9130 ± 0.0020

Table 4.2: Comparaisons accuracy

On constate alors deux choses

- La taille du batch pour le pré entraînement n'a aucun impact (voir à l'air de faire diminuer l'accuracy)
- Augmenter le nombre de layers dense n'a aucun impact non plus sur l'accuracy

On obtient donc des accuracy avoisinant les 92%, contre environ 85% pour notre CNN. Cette nouvelle méthode permet donc bien d'augmenter les performances de nos modèles dans le cadre de la prédiction des chiffres dans les données MNIST.

5 Annexe

Obtenir les données :

```
def get_data():
    transform = transforms.Compose(
        [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
    )

```

```
    train_data = datasets.MNIST(
        root="./data", train=True, download=True, transform=transform
    )

    test_data = datasets.MNIST(
        root="./data", train=False, download=True, transform=transform
    )
    return train_data, test_data
```

Obtenir les loaders :

```
def get_loader(train_data, test_data, batch_size=32):
    random_state = 2023
    unlabeled_indices, labeled_indices = train_test_split(
        list(range(len(train_data.targets))),
        test_size=100,
        stratify=train_data.targets,
        random_state=random_state,
    )
    unlabeledset = torch.utils.data.Subset(train_data, unlabeled_indices)
    labeledset = torch.utils.data.Subset(train_data, labeled_indices)
    train_unlabeled_loader = torch.utils.data.DataLoader(
        unlabeledset, batch_size=batch_size, shuffle=True, num_workers=0
    )
    train_labeled_loader = torch.utils.data.DataLoader(
        labeledset, batch_size=batch_size, shuffle=False, num_workers=0
    )
    test_loader = torch.utils.data.DataLoader(
        test_data, batch_size=batch_size, shuffle=False, num_workers=0
    )
    return train_unlabeled_loader, train_labeled_loader, test_loader
```

Data augmentation code

```
class TwoCropTransform:
    """Create two crops of the same image"""

    def __init__(self, transform):
        self.transform = transform

    def __call__(self, x):
        return [self.transform(x), self.transform(x)]
```

```
contrastive_transform = transforms.Compose([
                                    transforms.RandomHorizontalFlip(),
                                    transforms.RandomResizedCrop(size=28, scale=(0.2,
                                    ↪  1.)),
                                    transforms.RandomRotation(degrees=(-45, 45)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (0.5,)),
                                    ])
```

Entraînement du modèle

```
    def train(self, model, epochs, optimizer, criterion, output_fn, RGB = False,
    ↪  patience_LR = 3, patience_earlystop = 8):
        scheduler = ReduceLROnPlateau(optimizer, factor=0.5, patience=patience_LR)  #to
        ↪  vary the learning rate
```

```
        early_stopping = EarlyStopping(patience=patience_earlystop, delta=0)   #early
        ↪  stopping

        for epoch in tqdm(range(epochs)):
            # Training
            model.train()
            running_loss = 0.0
            for idx, batch in enumerate(self.train_loader):
                # get the inputs; batch is a list of [inputs, labels]
                inputs, labels = batch
                if RGB:
                    inputs = torch.cat([inputs,inputs,inputs], dim=1)
                inputs = inputs.to(self.device)  # train on GPU
                labels = labels.to(self.device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward + backward + optimize
                out = model(x=inputs)
                loss = criterion(out, labels)
                loss.backward()
                optimizer.step()
```

CNN :

```
class CNN(torch.nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        # L1 (?, 28, 28, 1) -> (?, 28, 28, 32) -> (?, 14, 14, 32)
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(32),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Dropout(p=0.2),
        )
        # L2 (?, 14, 14, 32) -> (?, 14, 14, 64) -> (?, 7, 7, 64)
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(64),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Dropout(p=0.2),
        )
        # L3 (?, 7, 7, 64) -> (?, 7, 7, 128) -> (?, 4, 4, 128)
        self.layer3 = torch.nn.Sequential(
            torch.nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(128),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=1),
            torch.nn.Dropout(p=0.2),
        )
        self.layer4 = nn.Linear(2048,10)  #for 10 digits

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = x.view(x.size(0), -1)  # Flatten them for FC
```

```
        x = self.layer4(x)
        return x
```

Transfert learning Resnet :

```
for name, param in resnet.named_parameters():
    param.requires_grad = False    #freeze the weights
first_conv = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
↪  bias=False)
last_layer = nn.Sequential(
    nn.Linear(in_features=2048, out_features=512),
    nn.Linear(in_features=512, out_features=10)
)
resnet.conv1 = first_conv  #change first convolutionnal network
resnet.fc = last_layer  #change last layer
```

SimCLR pré-entraînement :

```
import torch.nn as nn
import torch.nn.functional as F


class Simclr(nn.Module):
    """backbone + projection head"""

    def __init__(self, model, head="mlp", feat_dim=128):
        super(Simclr, self).__init__()

        self.dim_in = model._to_linear
        self.encoder = model

        if head == "linear":
            self.head = nn.Linear(self.dim_in, feat_dim)
        elif head == "mlp":
            self.head = nn.Sequential(
                nn.Linear(self.dim_in, self.dim_in),
                nn.ReLU(inplace=True),
                nn.Linear(self.dim_in, feat_dim),
            )
        else:
            raise NotImplementedError("Head not supported: {}".format(head))

    def forward(self, x):
        feat = self.encoder(x)
        feat = F.normalize(self.head(feat), dim=1)
        return feat
```

Contrastive Loss :

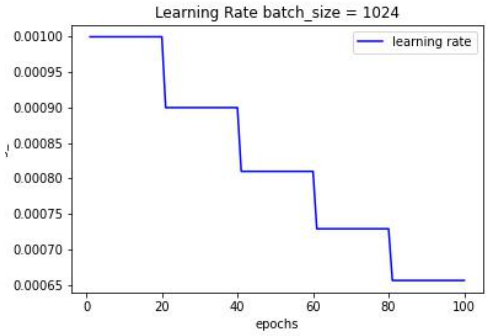
```
import torch
import torch.nn as nn


class Contrastive_loss(nn.Module):
    def __init__(self, temperature=0.07, contrast_mode="all", base_temperature=0.07):
        super(Contrastive_loss, self).__init__()
        self.temperature = temperature
        self.contrast_mode = contrast_mode
        self.base_temperature = base_temperature

```

```
    def forward(self, features):
        device = torch.device("cuda") if features.is_cuda else torch.device("cpu")

        if len(features.shape) < 3:
            raise ValueError(
                "`features` needs to be [bsz, n_views, ...],"
                "at least 3 dimensions are required"
            )
        if len(features.shape) > 3:
            features = features.view(features.shape[0], features.shape[1], -1)

        batch_size = features.shape[0]
        mask = torch.eye(batch_size, dtype=torch.float32).to(device)

        contrast_count = features.shape[1]
        contrast_feature = torch.cat(torch.unbind(features, dim=1), dim=0)
        if self.contrast_mode == "one":
            anchor_feature = features[:, 0]
            anchor_count = 1
        elif self.contrast_mode == "all":
            anchor_feature = contrast_feature
            anchor_count = contrast_count
        else:
            raise ValueError("Unknown mode: {}".format(self.contrast_mode))

        # compute logits
        anchor_dot_contrast = torch.div(
            torch.matmul(anchor_feature, contrast_feature.T), self.temperature
        )
        # for numerical stability
        logits_max, _ = torch.max(anchor_dot_contrast, dim=1, keepdim=True)
        logits = anchor_dot_contrast - logits_max.detach()

        # tile mask
        mask = mask.repeat(anchor_count, contrast_count)
        # mask-out self-contrast cases
        logits_mask = torch.scatter(
            torch.ones_like(mask),
            1,
            torch.arange(batch_size * anchor_count).view(-1, 1).to(device),
            0,
        )
        mask = mask * logits_mask

        # compute log_prob
        exp_logits = torch.exp(logits) * logits_mask
        log_prob = logits - torch.log(exp_logits.sum(1, keepdim=True))

        # compute mean of log-likelihood over positive
        mean_log_prob_pos = (mask * log_prob).sum(1) / mask.sum(1)

        # loss
        loss = -(self.temperature / self.base_temperature) * mean_log_prob_pos
        loss = loss.view(anchor_count, batch_size).mean()

        return loss
```

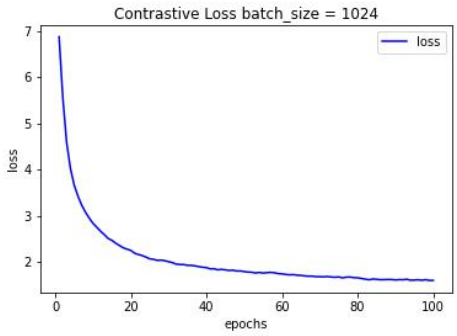
https://towardsdatascience.com/understanding-contrastive-learning-d5b19fd96607 [2]

Pré-entraînement

```python
def pretraining(epoch, model, contrastive_loader, optimizer, criterion):
    "Contrastive pre-training over an epoch"
    metric_monitor = MetricMonitor()
    model.train()
    for batch_idx, (data, labels) in enumerate(train_unlabeled_loader):
    ↪  #unlabeled_loader
        data = torch.cat([data[0], data[1]], dim=0)
        if torch.cuda.is_available():
            data, labels = data.cuda(), labels.cuda()
        data, labels = torch.autograd.Variable(data, False), torch.autograd.Variable(
            labels
        )
        bsz = labels.shape[0]
        features = model(data)      #in the 128 dimensions space
        f1, f2 = torch.split(features, [bsz, bsz], dim=0)
        features = torch.cat([f1.unsqueeze(1), f2.unsqueeze(1)], dim=1)
        loss = criterion(features)   #compute the loss
        metric_monitor.update("Loss", loss.item())
        metric_monitor.update("Learning Rate", optimizer.param_groups[0]["lr"])
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(
        "[Epoch: {epoch:03d}] Contrastive Pre-train | {metric_monitor}".format(
            epoch=epoch, metric_monitor=metric_monitor
        )
    )
```

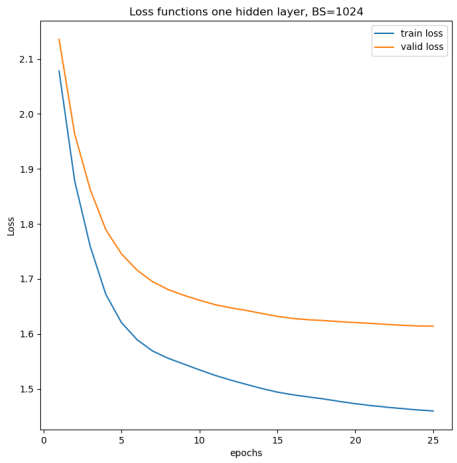


(a) learning rate batch size = 1024

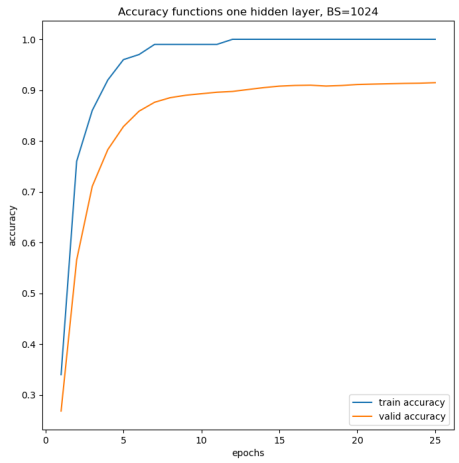


(b) contrastive loss batch size = 1024)

Figure 16: pré entrainement : batch size = 1024



(a) Loss batch size = 1024



(b) Accuracy batch size = 1024)

Figure 17: Un layer dense : batch size = 1024

References

[1] https://icml.cc/media/icml-2020/slides/6762.pdf.

[2] https://medium.com/@nainaakash012/simclr-contrastive-learning-of-visual-representations-52ecf1ac11fa.

[3] https://towardsdatascience.com/contrastive-loss-explaned-159f2d4a87ec.

[4] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[5] https://github.com/giakou4. Classification of the mnist dataset using various deep learning techniques.

[6] Brett Koonce and Brett Koonce. Resnet 50. *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*, pages 63–72, 2021.

[7] Chi Ian Tang, Ignacio Perez-Pozuelo, Dimitris Spathis, and Cecilia Mascolo. Exploring contrastive learning in human activity recognition for healthcare. *arXiv preprint arXiv:2011.11542*, 2020.