


Université Paris Cité

UFR des Sciences fondamentales et biomédicale

Apprentissage Profond, Deep Learning

Spécialité:

Master 2 Machine learning pour la Science des Données

Projet:

Prédiction des données MNIST avec uniquement 100 labels

Réalisé par:

SAHI Kenza

HAMEL Amir

2023-2024

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Projet deep learning

1.1 Introduction

L'apprentissage semi-supervisé et auto-supervisé visent à résoudre le défi consistant à exploiter une vaste quantité de données non étiquetées pour effectuer une classification dans des situations où les données étiquetées sont généralement limitées.

1.2 Problématique et objectif du projet

L'objectif de ce projet est de construire un modèle basé sur les réseaux de neurones qui va prédire l'étiquette(classe) de l'image qu'il reçoit en entrée en utilisant uniquement 100 images étiquetées (de 0 à 9) dans la phase d'entraînement.

1.3 Description de l'ensemble de données MNIST

Dans notre projet, nous utilisons un ensemble des images des chiffres manuscrites qui s'appelle MNIST, c'est un jeu de données très utilisé en apprentissage automatique, ce dernier contient une base d'apprentissage de 60000 images et une base de test d'environ 10000 images. Elles sont classées sur 10 classes de 0 à 9. Ces images sont de taille 28* 28.

Nous avons divisé notre ensemble de données d'apprentissage en 100 images étiquetées pour la tâche de classification et 59900 images non étiquetées. Pour les données d'apprentissage, nous avons sélectionné 10 images aléatoirement pour chaque étiquette afin de garantir un équilibre au sein des 100 images étiquetées.

Chapitre 2

Méthode basée sur la rotation d'images

2.1 Contexte

Au cours des dernières années, les réseaux de neurones convolutifs profonds ont transformé le domaine de la vision par ordinateur grâce à leur capacité incomparable à apprendre des caractéristiques sémantiques d'image de haut niveau. Cependant, pour apprendre ces caractéristiques avec succès, ils nécessitent généralement des quantités massives de données annotées (étiquetées) manuellement ce qui est à la fois coûteux et impraticable à grande échelle.

En raison de cela, il y a récemment un intérêt croissant pour apprendre des représentations de haut niveau basées sur ConvNets de manière non supervisée, en évitant l'annotation manuelle (humaine) des données visuelles. Parmi elles, un paradigme important est celui de l'apprentissage auto-supervisé (self supervised), qui définit une tâche prétexte sans annotation, en utilisant uniquement les informations visuelles présentes sur les images ou les vidéos, afin de fournir un signal de supervision substitut pour l'apprentissage des caractéristiques sémantiques.

2.2 Description de la méthode

Dans cette section nous proposons une méthode de self-supervised learning qui permet d'apprendre des caractéristiques d'image de l'ensemble non étiquetées de MNIST (59900 non étiquetées) en entraînant un modèle à reconnaître la transformation géométrique appliquée à l'image qu'il reçoit en entrée, ce qui permet d'apprendre des caractéristiques sémantiques d'image qui peuvent être utiles à la tâche de classification des images du MNIST, puis nous comparons les performances des résultats obtenus avec celles du modèle de base supervisé (baseline) qui doit être entraîné sur seulement les 100 images étiquetées.

Nous présentons les deux étapes de la transformation géométrique [1] :

— Nous définissons un ensemble de K transformations géométriques discrètes

$$G = \{g(\cdot|y)\}k_{y=1}$$

, où $g(\cdot|y)$ est l'opérateur qui applique à l'image X la transformation géométrique avec l'étiquette y, produisant ainsi l'image transformée $X_y = g(X|y)$. Le modèle ConvNet $F(\cdot)$ prend en entrée une image Xy (où l'étiquette y est inconnue du modèle $F(\cdot)$) et produit en sortie une distribution de probabilité sur toutes les transformations géométriques possibles .

$$F_y(X_{y^*}|\theta) \quad \text{pour } K_y = 1,$$

où $F_y(X_{y^*}|\theta)$, est la probabilité prédite pour la transformation géométrique avec l'étiquette y, et θ, représente les paramètres adaptables du modèle F(.).

— Etant donné un ensemble de N images d'entraînement D = $\{\mathrm{X}_i\}_{i=0}^{N}$
l'objectif d'entraînement self-supervisé est que le modèle ConvNet doit apprendre à résoudre cette fonction.

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \text{loss}(X_i, \theta)$$

Loss est une fonction de perte définie comme suite :

$$\text{loss}(X_i, \theta) = -\frac{1}{K} \sum_{y=1}^{K} \log\left(F_y(g(X_i|y)|\theta)\right)$$

2.3 Choix de la transformation géométrique

Nous avons défini la transformation géométrique la rotation d'images par des multiples de 90 degrés, c'est-à-dire, les rotations d'image en 2D par 0, 90, 180 et 270 degrés. L'opération de rotation est noté `Rot`(X, ϕ) prend une image en entrée et renvoie en sortie quatre images pivotées selon des rotations de 0, 90, 180 et 270 degrés respectivement.

2.3.1 Pourquoi la rotation ?

L'avantage le plus important de l'utilisation de la rotation d'image est qu'elles peuvent être mises en œuvre par des opérations de retournement (flip) et de transposition (transpose). Ces opérations n'engendrent pas d'artefacts visuels de bas niveau facilement détectables qui pourraient conduire le ConvNet à apprendre des caractéristiques triviales sans valeur pratique pour les tâches de perception visuelle [1].

2.3.2 Pourquoi des rotations de degrés multiples de 90 ?

Les rotations par des multiples de 90 degrés préservent souvent l'intégrité structurelle et la symétrie des objets dans l'image et cela facilite l'apprentissage par le modèle de caractéristiques significatives et invariantes car les objets conservent leur forme et leur orientation globales [1].

2.4 Le processus de rotation des images

[1],Afin de mettre en œuvre les rotations d'images de 90, 180 et 270 degrés (le cas de 0 degré correspond à l'image elle-même), nous utilisons des opérations de retournement (flip) et de transposition. Plus précisément,

- Pour une rotation de 90 degrés, nous transposons d'abord l'image, puis nous la retournons verticalement (retournement tête en bas).
- Pour une rotation de 180 degrés, nous retournons d'abord l'image verticalement, puis horizontalement (retournement gauche-droite).
- Pour une rotation de 270 degrés, nous retournons d'abord l'image verticalement, puis nous la transposons.

Nous effectuons l'opération de rotation spécifiée sur chaque image non labélisée du train et du test, à l'exclusion des 100 images étiquetées, et affectons une étiquette à l'image indiquant le type de rotation défini précédemment. Dans ce cas, nous avons obtenu 239600 images transformées en 4 degrés pour les données de train et 40000 images transformées pour le test set.

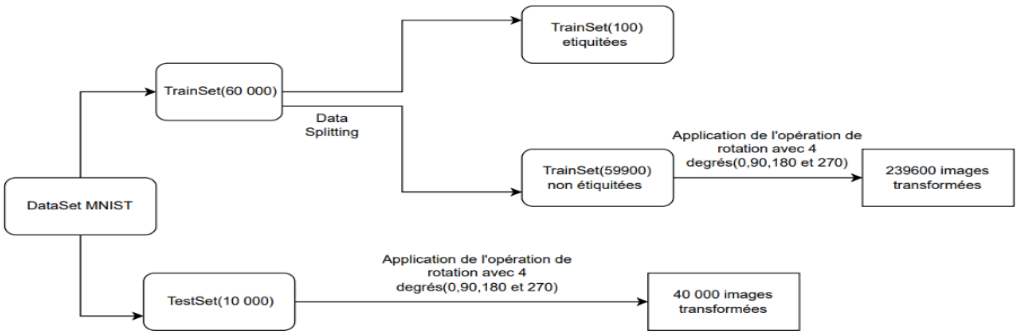

Figure 2.1 : *Processus de rotation des images*

Petite visualisation des images transformées :

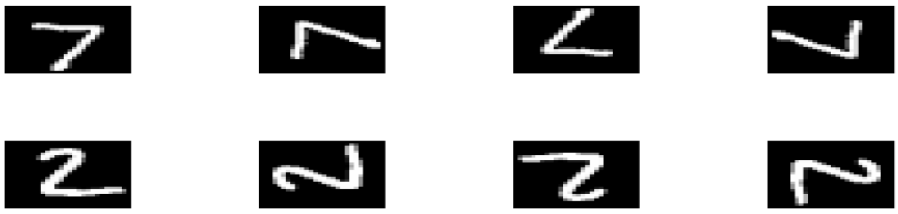

FIGURE 2.2 : *Exemples des images transformées*

2.5 Le modèle RotNet (self-supervised)

Le modèle RotNset est le modèle qui s'occupe de la prédiction du degré de rotation d'une image donnée en entrée, le but de ce modèle est d'apprend à reconnaitre et extraire les caractéristiques d'une image .

La figure suivante montre l'application des rotations de 0, 90, 180 et 270 degrés, nous entraînons un modèle ConvNet F(.) à reconnaître la rotation appliquée à l'image qu'il reçoit en entrée. Fy(Xy) représente la probabilité de la transformation de rotation y prédite par le modèle F(.) lorsqu'il reçoit en entrée une image transformée par la rotation y.

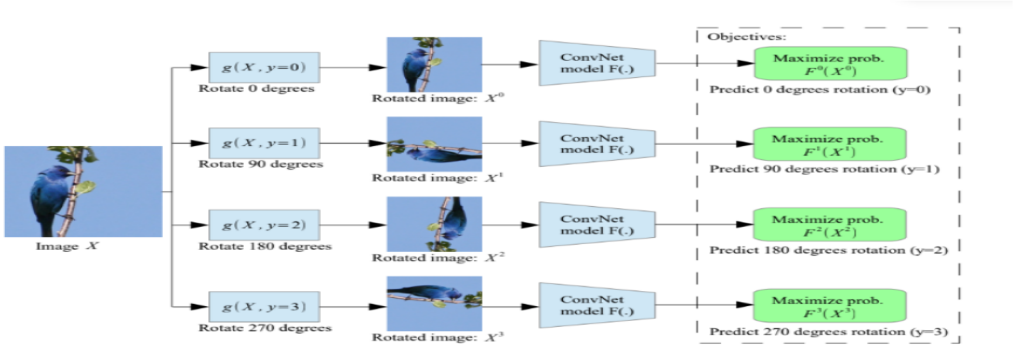

FIGURE 2.3 : *Prédiction du degré de rotation effectuée par RotNet*

2.5.1 L'architecture du modèle RotNet

Le modèle RotNet est composé de trois blocs , chacun contient :

- 3 couches convolutifs (Conv2D)
- 3 batch normalization layers
- 3 activation layers
- pooling layer

et à la fin nous avons la couche de sortie avec la fonction d'activation SoftMax.

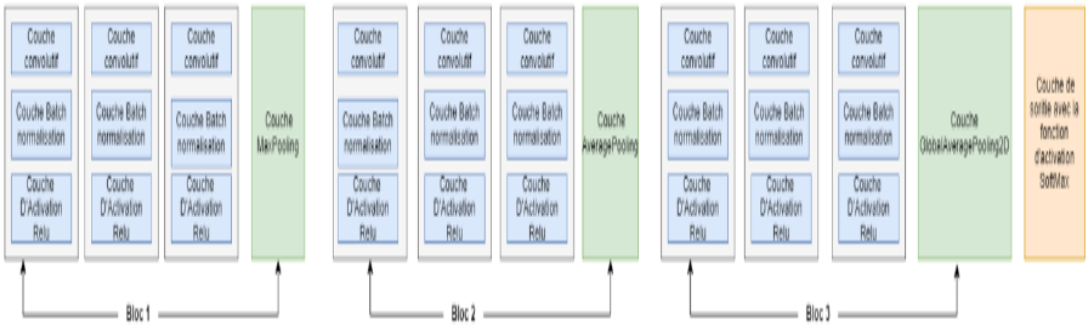

FIGURE 2.4 : *L'architecture du modèle RotNet*

Pour l'entraîner sur la tâche de prédiction de rotation (tâche prétexte),nous entrainons le modèle sur train set et validation set.

Nous adoptons la descente de gradient stochastique (SGD) avec une taille de lot (batch size) de 128, un momentum de 0.9, et un taux d'apprentissage (lr) de 0.1. L'entraînement est réalisé sur un total de 20 époques.

Au cours de nos essais initiaux, nous avons observé une amélioration significative en alimentant simultanément le réseau avec les quatres copies rotatives d'une image pendant l'entraînement, plutôt que de choisir aléatoirement une seule transformation de rotation à chaque itération. Ainsi, à chaque lot d'entraînement(batch), le réseau est exposé à quatre fois plus d'images que la taille du lot.

2.5.2 Résultats de RotNet

Nous représentons graphiquement l'évolution de la fonction de perte et de l'accuracy du modèle RotNet entraîné.

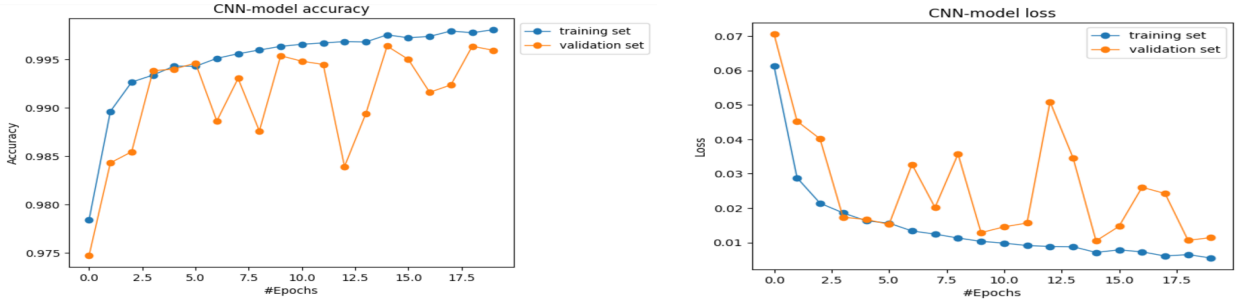

FIGURE 2.5 : *Evaluation de l'accuracy et de la fonction loss sur le train et validation set*

En analysant la progression de l'accuracy entre l'ensemble d'entraînement et l'ensemble de validation, nous notons une convergence vers presque 100 % après 18 époques.

Il est à noter que la fonction de perte converge également après 18 époques pour les deux ensembles, à savoir l'ensemble d'entraînement et l'ensemble de validation. Cette observation est positive, suggérant que notre modèle ne souffre pas de surajustement(overfitting).
Pour les données de test, nous avons obtenu un score d'accuracy de 99%, ce qui indique que le modèle est généralisé. Les exemples de test suivants illustrent sa performance et sa robustesse.

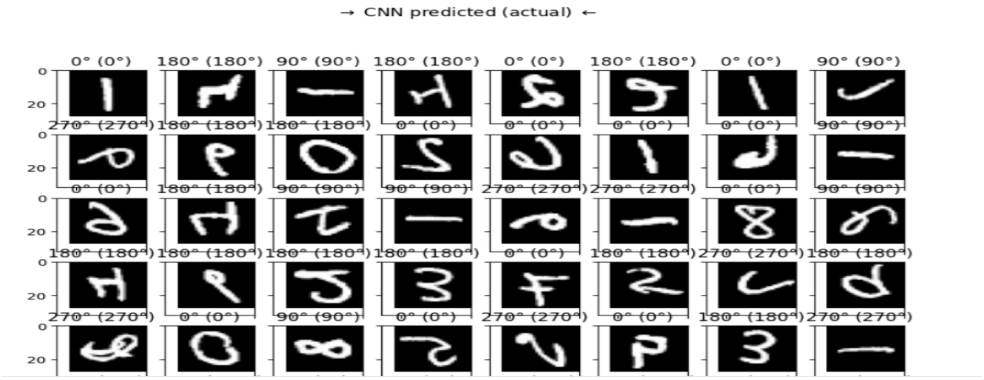

Figure 2.6 : *Comparaison des prédictions issues de RotNet avec les vrais étiquettes de degrés de rotation*

En comparant les classes prédites avec les classes réelles sur les données de test , nous constatons que le RotNet a correctement classé presque toutes les images.

2.6 Modèle basé sur RotNet

2.6.1 Adaptation de RotNet à la tâche de classification

Après avoir entraîné notre modèle RotNet sur la tâche de reconnaissance de rotation avec des données non étiquetées, nous avons gelé les deux premiers blocs du modèle (les paramètres associées) et modifié le dernier bloc en y ajoutant les couches d'un modèle supervisé (une couche de sortie adapté aux nombre de classe de MNIST). De cette façon, nous avons pu entraîner le modèle sur les 100 images étiquetées de l'ensemble d'images MNIST.

Voici les étapes suivies :

— Les deux premiers blocs sont gelés

— Les poids restent modifiables dans le dernier (troisième) bloc

— Les 20 premières couches correspondent aux deux premiers blocs de notre modèle de base (RotNet) , (pour chaque bloc, 3 couches (couche Con2D, couche de normalisation par lots, couches d'activation) + une couche de Pooling).

— Chaque bloc contient 10 couches.

Pour modifier le dernier bloc , nous avons choisi la sortie de la quatrième couche en partant de la fin, à partir du dernier bloc de RotNet. Cette couche correspond à la dernière couche avant les couches de classification, et ses sorties représentent les caractéristiques de l'image.

Par la suite, un Global Average Pooling est appliqué pour extraire les poids des couches figées.

Enfin, une couche Dense avec 10 unités et une fonction d'activation softmax est ajoutée pour prédire les dix classes (0, 1, ..., 9) de MNIST.

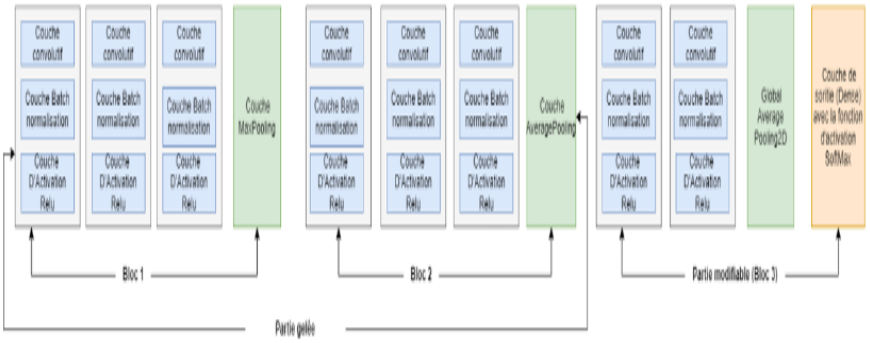

Figure 2.7 : *L'architecture du modèle final*

Après la construction du modèle final, nous l'avons entrainé à la tâche de prédiction des classes des images de MNIST en utilisant les 100 images étiquetées , nous adoptons la descente de gradient stochastique (SGD) avec une taille de lot (batch size) de 10, un momentum de 0,9 et un taux d'apprentissage (lr) de 0,1. L'entraînement est réalisé sur un total de 30 époques.

2.6.2 Résultats obtenus

Nous représentons graphiquement l'évolution de la fonction de perte et de l'accuracy du modèle final entraîné.

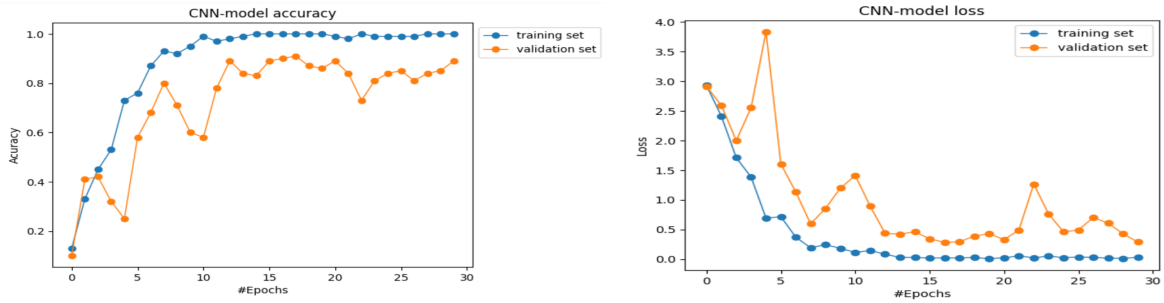

FIGURE 2.8 : *Evaluation de l'accuracy et de la fonction loss du modèle final sur le train et validation set*

En analysant la progression de l'accuracy de l'ensemble d'entraînement et l'ensemble de validation, nous notons une convergence vers presque 100 % après 25 époques. Il est à noter que la fonction de perte converge également après 15 époques.

Nous représentons l'évolution de l'accuracy du modèle final sur les données de test.

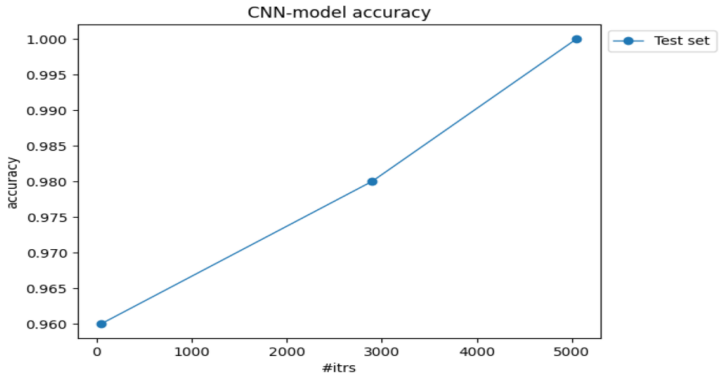

FIGURE 2.9 : *Evaluation de l'accuracy du modèle final sur le test set*

Pour les données de test, nous remarquons dans le graphe que l'accuracy converge, et plus précisément, nous avons obtenu un score de 88%, ce qui indique que le modèle est généralisé.

Les exemples de test suivants illustrent sa performance et sa robustesse.

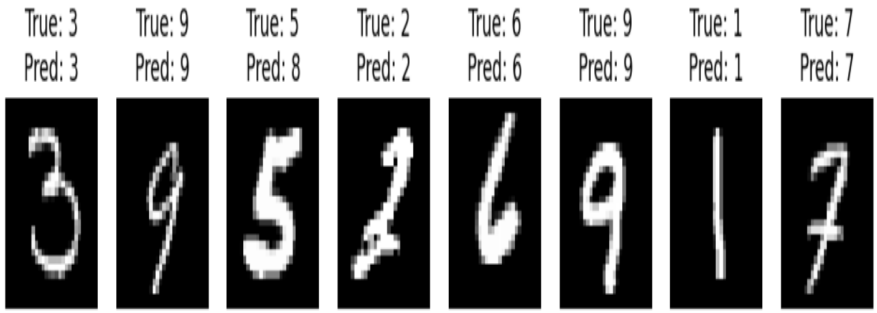

Figure 2.10 : *Comparaison des prédictions issues du modèle final avec les vrais étiquettes de degrés de rotation*

2.7 Modèle Baseline (modèle supervisé) :

2.7.1 L'architecture du modèle baseline

Dans ce cas, nous construisons un modèle qui partage la même architecture que notre modèle final basé sur RotNet, en particulier la composante de classification. Néanmoins, nous n'utilisons aucune transformation géométrique dans le but d'extraire des caractéristiques d'image. L'objectif est de comparer ce modèle avec notre modèle final.

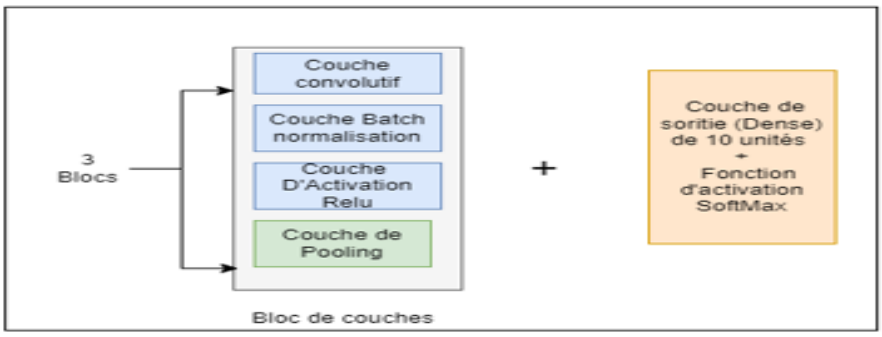

Figure 2.11 : *L'architecture du modèle Baseline*

Nous adoptons la descente de gradient stochastique (SGD) avec une taille de lot (batch size) de 50, un momentum de 0.9 et un taux d'apprentissage (lr) de 0.01. L'entraînement est réalisé sur un total de 10 époques.

Nous avons utilisé la fonction de perte et l'accuracy pour évaluer le modèle lors de son entraînement sur les 100 images étiquetées .

2.7.2 Résultats obtenus

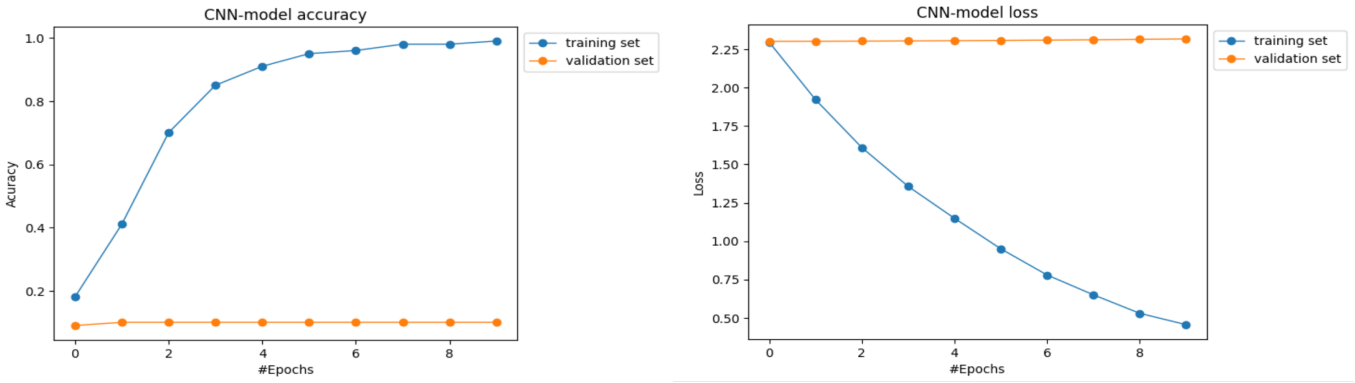

FIGURE 2.12 : *Evaluation de l'accuracy et la fonction de perte du modèle Baseline sur le train et validation set*

Nous notons que la fonction de perte diminue et converge vers zéro après 8 époques, et l'accuracy converge vers 1. Cependant, lors de l'évaluation du modèle de base sur les données de validation, nous avons obtenu un score d'accuracy trés faible, ce qui indique un surajustement (overfitting).

Nous représentons graphiquement l'évolution de l'accuracy du modèle de base sur les données de test.

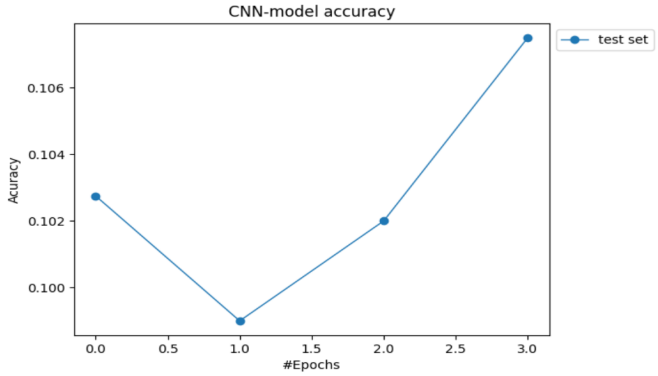

FIGURE 2.13 : *Evaluation de l'accuracy du modèle Baseline sur le test set*

En analysant ce graphique, il est apparent que l'accuracy sur les données de test atteint 10%, suggérant ainsi que le modèle est très faible et a du mal à généraliser. Les exemples de test suivants confirment ce que nous avons constaté précédemment.

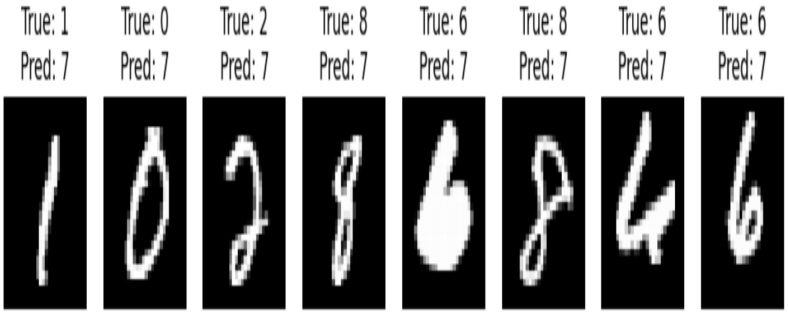

Figure 2.14 : *Comparaison des prédictions issues du modèle baseline avec les vrais étiquettes de degrés de rotation*

2.8 Comparaison des performances entre les deux modèles :

Dans le tableau ci-dessous, nous exposons les divers paramètres utilisés durant la phase d'entraînement des deux modèles.

	Modèle final	**Modèle baseline**
Architecture	3 Blocs de couches	3 Blocs de couches
Train data, Test data	(59900 + 100 étiquitées, 10 000)	(100, 10 000)
Optimiser(momentum,learning rate)	SGD(0.9, 0.1)	SGD(0.9, 0.01)
Batch size	10	50
Epochs	30	10
Accuracy pour le train	100%	99%
Accuracy pour la validation	85%	10%
Accuracy pour le test	88%	10.28%
Loss pour le train	0.011	0.45

Table 2.1 : *Comparaison des paramètres des deux modèles*

En examinant ce tableau, il est notable que l'accuarcy du modèle final basé sur la méthode self-supervised assuré par le modèle RotNet, atteint un score bien plus élevé que celle du modèle de base. Cela suggère que le modèle final est généralisé et robuste.

2.9 Conclusion

Dans cette section, nous avons développé une méthode d'apprentissage des caractéristiques qui construit un modèle basé sur les CNN qui permet de reconnaître la rotation appliquée aux images en entrée. Cette approche a permis à notre modèle RotNet d'acquérir des caractéristiques des images utilisées dans la classification de l'ensemble MNIST.

En ajoutant des couches supervisées (pour la prédiction) à la partie gelée du RotNet, nous avons abouti à un modèle final que nous avons entraîné sur les 100 images étiquetées du MNIST. Ce dernier a considérablement amélioré les résultats en termes d'accuracy et de fonction de perte, démontrant ainsi sa capacité à généraliser, contrairement au modèle de base qui présentait une accuracy très faible.

2.10 Code

```
#Loading the Dataset
(X_train_Original, Y_train_Original), (X_test_Original,
↪  Y_test_Original) = mnist.load_data()
# Counting the number of every class
EachClassSize = Counter(Y_train_Original.reshape(-1))

# Taking  100 Data with labels
nber = 100 # number of labeled data
# take only the key of the dictionary object
labels = [key for (key,value) in EachClassSize.items()]
idx100 = []
for i in range(len(labels)):
  idx = labels[i]==Y_train_Original
  index = np.arange(0,len(Y_train_Original))[idx]
  for j in range(10):
    idx100.append(index[j])


y_train100 = Y_train_Original[idx100]
x_train100 = X_train_Original[idx100]
x_train59900 = np.delete(X_train_Original,idx100,axis=0)

#The rotation operator
def Rot(X, d):
  if d == 0: # 0 degrees rotation
```

```
        return X
    elif d == 90: # 90 degrees rotation
        return np.flipud(np.transpose(X))
    elif d == 180: # 180 degrees rotation
        return np.fliplr(np.flipud(X))
    elif d == 270: # 270 degrees rotation / or -90
        return np.transpose(np.flipud(X))
    else:
        raise ValueError('rotation should be 0, 90, 180, or 270
        ↪  degrees')

#Applying of the operator in the whole dataset
#The rotation operation on training data.
n = x_train59900.shape[0]
RotX = np.zeros([4*n,28,28])# data obtained by rotation
Roty = np.zeros(4*n)# our pseudo labels
i=0
j=0
while i < 4*n:
    if j < n:
        RotX[i] = Rot(x_train59900[j],  0)
        RotX[i+1] = Rot(x_train59900[j],  90)
        RotX[i+2] = Rot(x_train59900[j],  180)
        RotX[i+3] = Rot(x_train59900[j],  270)
        Roty[i:i+4] = np.array([0,1,2,3])
        j = j + 1
    i = i + 4

#The rotation operation on test data.
n_test = X_test_Original.shape[0]
RotX_test = np.zeros([4*n_test,28,28])# data obtained by rotation
Roty_test = np.zeros(4*n_test)# our pseudo labels
i=0
j = 0
while i < 4*n_test:
    if j < n:
        RotX_test[i] = Rot(X_test_Original[j],  0)
        RotX_test[i+1] = Rot(X_test_Original[j],  90)
        RotX_test[i+2] = Rot(X_test_Original[j],  180)
        RotX_test[i+3] = Rot(X_test_Original[j],  270)
        Roty_test[i:i+4] = np.array([0,1,2,3])
        j = j + 1
    i = i+ 4
```

```
67
68  #The function that displays the result of the rotation operation
69  def plot(samples):
70      fig = plt.figure(figsize=(15, 15))
71      gs = gridspec.GridSpec(4, 4)
72      gs.update(wspace=1, hspace=1)
73
74      for i, sample in enumerate(samples):
75          ax = plt.subplot(gs[i])
76          plt.axis('off')
77          ax.set_xticklabels([])
78          ax.set_yticklabels([])
79          ax.set_aspect('equal')
80          plt.imshow(sample.reshape(28, 28), cmap='Greys_r')
81      return
82
83
84  #Display the result of the rotation applied to the train data.
85  plot(RotX[0:8])
86
87  #Display the result of the rotation applied to the test data.
88  plot(RotX_test[0:8])
89
90  # Data Preprocessing
91
92  # Splitting data
93  RotX_train, RotX_val, Roty_train, Roty_val = train_test_split(
94                                      RotX, Roty,
95                                      test_size = 0.2,
                                        ↪  random_state=12,
96                                      shuffle = True, stratify = Roty)
97
98  # Convert class vectors to binary class matrices. (one-hot
    ↪  encoding)
99  num_classes = np.int64(Roty.max()) +1
100
101 #Normalize the data (images) as the model expects images of type
    ↪  float32.
102 RotX_train = RotX_train.astype('float32')
103 RotX_train /= 255
104 RotX_train = RotX_train.reshape(-1, 28,28,1)
105
106 RotX_test = RotX_test.astype('float32')
```

```
RotX_test /= 255
RotX_test = RotX_test.reshape(-1, 28,28,1)

RotX_val = RotX_val.astype('float32')
RotX_val /= 255
RotX_val = RotX_val.reshape(-1, 28,28,1)

#Transform y into categorical data for all three sets (train,
↪  validation, and test).

Roty_train_before = Roty_train
Roty_test_before = Roty_test
Roty_val_before = Roty_val

Roty = to_categorical(Roty, num_classes)
Roty_train = to_categorical(Roty_train, num_classes)
Roty_val = to_categorical(Roty_val, num_classes)
Roty_test = to_categorical(Roty_test, num_classes)

#we will build the CNN model for training to recognize The
↪  rotations applied to the images; it is referred to as RotNet
↪  here.

clear_session()
#Rotnet Architecture
RotNet = Sequential()

# Convolution

#Block 1

RotNet.add(Conv2D(filters = 192, kernel_size= 5, use_bias=False,
↪  padding='same', strides=(1, 1),input_shape=(28, 28,1)))
RotNet.add(BatchNormalization())
RotNet.add(Activation('relu'))

RotNet.add(Conv2D(filters = 160, kernel_size= 1, use_bias=False,
↪  padding='same', strides=(1, 1)))
RotNet.add(BatchNormalization())
RotNet.add(Activation('relu'))

RotNet.add(Conv2D(filters = 96, kernel_size= 1, use_bias=False,
↪  padding='same', strides=(1, 1)))
```

```
144 RotNet.add(BatchNormalization())
145 RotNet.add(Activation('relu'))
146
147 RotNet.add(MaxPooling2D(pool_size = 3, padding='same',
    ↪  strides=(2,2)))
148
149
150 # Block 2
151
152 RotNet.add(Conv2D(filters = 192, kernel_size= 5, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
153 RotNet.add(BatchNormalization())
154 RotNet.add(Activation('relu'))
155
156 RotNet.add(Conv2D(filters = 192, kernel_size= 1, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
157 RotNet.add(BatchNormalization())
158 RotNet.add(Activation('relu'))
159
160 RotNet.add(Conv2D(filters = 192, kernel_size= 1, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
161 RotNet.add(BatchNormalization())
162 RotNet.add(Activation('relu'))
163
164 RotNet.add(AveragePooling2D(pool_size = 3, padding='same',
    ↪  strides=(2,2)))
165
166
167 # Block 3
168
169 RotNet.add(Conv2D(filters = 192, kernel_size= 3, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
170 RotNet.add(BatchNormalization())
171 RotNet.add(Activation('relu'))
172
173 RotNet.add(Conv2D(filters = 192, kernel_size= 1, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
174 RotNet.add(BatchNormalization())
175 RotNet.add(Activation('relu'))
176
177 RotNet.add(Conv2D(filters = 192, kernel_size= 1, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
178 RotNet.add(BatchNormalization())
```

```
RotNet.add(Activation('relu'))



RotNet.add(GlobalAveragePooling2D())


RotNet.add(Dense(num_classes))
RotNet.add(Activation('softmax'))

#Show the architecture of the RotNet model.
RotNet.summary()

#Define the optimizer
opt = SGD(learning_rate=0.1, momentum=0.9,  nesterov=True)

#Define the optimizer, the loss function and the metrics
RotNet.compile(loss='categorical_crossentropy', optimizer = opt,
 ↪  metrics=["accuracy"])

# Démarrage de l'entrainement du réseau
hist = RotNet.fit(RotX_train, Roty_train,
            batch_size=128,
            epochs=20,
            shuffle=True,                        # verbosité
            verbose = 1,
            validation_data=(RotX_val, Roty_val))



#Plot the accuracy for the training and validation sets.
plt.plot(hist.history['accuracy'], label='training
 ↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
plt.plot(hist.history['val_accuracy'], label='validation
 ↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
plt.title("CNN-model accuracy")
plt.xlabel('#Epochs')
plt.ylabel('Accuracy')
plt.legend(bbox_to_anchor=( 1., 1.))

#Plot the loss function for the training and validation sets
plt.plot(hist.history['loss'], label='training set',marker='o',
 ↪  linestyle='solid',linewidth=1, markersize=6)
```

```
plt.plot(hist.history['val_loss'], label='validation set',marker='o', linestyle='solid',linewidth=1, markersize=6)
plt.title("CNN-model loss")
plt.xlabel('#Epochs')
plt.ylabel('Loss')
plt.legend(bbox_to_anchor=( 1., 1.))



#Test on the validation data
#Apply the model to RotX_val to obtain the predicted Y.
y_pred = RotNet.predict(RotX_val)
y_pred = np.argmax(y_pred, axis=1)

#Reshape the correct Y according to the shape of the predicted Y.
y_true = Roty_val_before.reshape(y_pred.shape)

#
Compute the accuracy score by comparing predicted y with the correct y.
accuracy_score(y_true=y_true,y_pred=y_pred)

#visualize the predictions of the rotations Function
def visualize_prediction_RotNet(X,Y):
    # Sample test data
    LABELS = {
    0 : "0°",
    1 : "90°",
    2 : "180°",
    3 : "270°",
   }
  ix = np.random.randint(0, 10000, size=36)
  ex_im = X[ix]
  ex_lb = Y[ix]

  # Predict
  out = RotNet.predict(ex_im) #RotNet
  classes = np.argmax(out, axis=1)  # softmax output -> class

  # Plot
  fig, axes = plt.subplots(6, 6, figsize=(10, 10), sharey=True, sharex=True);
```

```
  fig.suptitle(r'$\rightarrow$ CNN predicted (actual)
  ↪ $\leftarrow$')

  k = 0
  for i in range(6):
      for j in range(6):
          # Switch Axes
          ax = axes[i, j]

          # Show image
          ax.imshow(ex_im[k].reshape(28, 28), cmap='Greys_r');

          # Determine labels
          actual_lab = LABELS[np.argmax(ex_lb[k])]
          pred_lab = LABELS[classes[k]]

          # Format title
          title = "{} ({})".format(pred_lab, actual_lab)
          title_color = 'black'

          # Mark image if wrong prediction
          if actual_lab != pred_lab:
              ax.plot(np.array([0, 32]), np.array([0, 32]), 'r-')
              ax.plot(np.array([0, 32]), np.array([32, 0]), 'r-')
              title_color = 'red'

          # Set title
          ax.set_title(title, color=title_color);

          # Set limits
          ax.set_xlim(32, 0)
          ax.set_ylim(32, 0)

          k += 1


visualize_prediction_RotNet(RotX_val,Roty_val)

#Test on the test data
#Apply the model to RotX_test to obtain the predicted Y.
y_pred = RotNet.predict(RotX_test)
y_pred = np.argmax(y_pred, axis=1)

```

```
298 #Reshape the correct Y according to the shape of the predicted Y.
299 y_true = Roty_test_before.reshape(y_pred.shape)
300
301 #Compute the accuracy score by comparing predicted y with the
    ↪  correct y.
302 accuracy_score(y_true=y_true,y_pred=y_pred)
303
304 #We visualize the predictions of the rotations on the
305
306 visualize_prediction_RotNet(RotX_test,Roty_test)
307
308 #Freezing certain part from RotNet model
309 model_frozen = RotNet
310 #summury
311 model_frozen.summary()
312
313 #Layers freezing
314 for layer in model_frozen.layers[:20]:
315     layer.trainable=False
316
317 for layer in model_frozen.layers[20:]:
318     layer.trainable=True
319
320 #Retrieve class numbers from the training labels (y_train).
321 num_classes2 = np.int64(y_train100.max()) +1
322
323 #The architecture of the frozen model
324 model_frozen.summary()
325
326 X= model_frozen.layers[-4].output
327 InputMod = tf.keras.Input(shape=(28, 28, 1))
328 globalAv = GlobalAveragePooling2D()(X)
329 predictions = Dense(num_classes2, activation="softmax")(globalAv)
330 model_final = tf.keras.Model(model_frozen.input,predictions)
331
332
333
334 # spliting data on test and validation set
335 X_test, X_val, Y_test, Y_val = train_test_split(
336                                    X_test_Original,
                                    ↪  Y_test_Original,
337                                    test_size = 0.01,
                                    ↪  random_state=12,
```

```
                                    shuffle = True)

#Optimizer of the final model
opt1= SGD(learning_rate=0.1, momentum=0.9,  nesterov=True)

#model_final.compile(loss='categorical_crossentropy',
↪  optimizer=opt1, metrics=["accuracy"])

x_train100_2 = x_train100.astype('float32')
x_train100_2 /= 255
x_train100_2 = x_train100_2.reshape(-1, 28,28,1)
y_train100_before = y_train100

X_val2 = X_val.astype('float32')
X_val2 /= 255
X_val2 = X_val2.reshape(-1, 28,28,1)
y_val_before = Y_val

#We convert y_train100 and Y_val  into categorical data
y_train100_2 = to_categorical(y_train100, num_classes2)
Y_val2 = to_categorical(Y_val, num_classes2)

#We train the ultimate model using the 100 labeled data points
hist2 = model_final.fit(x_train100_2, y_train100_2,
            batch_size = 10,
            epochs=30,
            shuffle=True,                        # verbosité
            verbose = 1,
             validation_data=(X_val2, Y_val2)
            )


# Plot the accuracy
plt.plot(hist2.history['accuracy'], label='training
↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
plt.plot(hist2.history['val_accuracy'], label='validation
↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
plt.title("CNN-model accuracy")
plt.xlabel('#Epochs')
plt.ylabel('Acuracy')
plt.legend(bbox_to_anchor=( 1., 1.))

# Plot the loss
```

```
plt.plot(hist2.history['loss'], label='training set',marker='o',
 ↪  linestyle='solid',linewidth=1, markersize=6)
plt.plot(hist2.history['val_loss'], label='validation
 ↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
plt.title("CNN-model loss")
plt.xlabel('#Epochs')
plt.ylabel('Loss')
plt.legend(bbox_to_anchor=( 1., 1.))



#Test the final model with train100 data (labelled data)
y_true = y_train100_before.reshape(y_pred.shape)
accuracy_score(y_true=y_true,y_pred=y_pred)

X_test = X_test_Original.astype('float32')
X_test /= 255
X_test = X_test.reshape(-1, 28,28,1)

y_pred3 = model_final.predict(X_test)
y_pred3 = np.argmax(y_pred3, axis=1)

y_true3 = Y_test_Original.reshape(y_pred3.shape)

#Convert the test labels to categorical.
y_test = to_categorical(Y_test_Original, num_classes2)

accuracy_score(y_true=y_true3,y_pred=y_pred3)

#Accuracy for the test data.

accuracy=[]
epochs=[]
j=0
last_accuracy=0
#Convert the y test  to categorical.
y_test = to_categorical(Y_test_Original, num_classes2)

for i in range(50,len(y_test)+1,50):
    y_pred3 = model_final.predict(X_test[j:i])

    y_pred3 = np.argmax(y_pred3, axis=1)
    y_pred3= to_categorical(y_pred3, 10)
```

```
419
420     y_true3 = y_test[j:i].reshape(y_pred3.shape)
421     current_accuracy=accuracy_score(y_true=y_true3,y_pred=y_pred3)
422     if (i % 10)==0:
423       if current_accuracy >  last_accuracy :
424          accuracy.append(current_accuracy)
425          epochs.append(i)
426          last_accuracy=current_accuracy
427     j=i
428
429
430
431 #PLot the curve of accuracy
432 plt.plot(epochs,accuracy, label='Test
    ↪  set',marker='o',linewidth=1, markersize=6)
433 plt.title("CNN-model accuracy")
434 plt.xlabel('#itrs')
435 plt.ylabel('accuracy')
436 plt.legend(bbox_to_anchor=( 1., 1.))
437
438
439 def show_data_label_prediction(X, y_true, model, num_examples=8):
440     # Make predictions on the data
441     y_pred = model.predict(X)
442     y_pred_classes = np.argmax(y_pred, axis=1)
443
444     # Randomly select examples
445     indices = np.random.choice(len(X), num_examples)
446
447     # Plot the examples
448     plt.figure(figsize=(12, 6))
449     for i, index in enumerate(indices, 1):
450         plt.subplot(1, num_examples, i)
451         plt.imshow(X[index].reshape(28, 28), cmap='gray')
452         plt.title(f'True: {np.argmax(y_true[index])}\nPred:
            ↪  {y_pred_classes[index]}')
453         plt.axis('off')
454
455     plt.show()
456
457
458 #Results obtained after applying the final model
459
```

```
460 #to the test data
461 show_data_label_prediction(X_test, y_test, model_final)
462
463
464 #Baseline model (supervised model)
465
466 clear_session()
467
468 baseline = Sequential()
469
470 # Convolution
471
472 #Block 1
473
474 baseline.add(Conv2D(filters = 192, kernel_size= 5,
    ↪  use_bias=False, padding='same', strides=(1,
    ↪  1),input_shape=(28, 28,1)))
475 baseline.add(BatchNormalization())
476 baseline.add(Activation('relu'))
477
478 baseline.add(Conv2D(filters = 160, kernel_size= 1,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
479 baseline.add(BatchNormalization())
480 baseline.add(Activation('relu'))
481
482 baseline.add(Conv2D(filters = 96, kernel_size= 1, use_bias=False,
    ↪  padding='same', strides=(1, 1)))
483 baseline.add(BatchNormalization())
484 baseline.add(Activation('relu'))
485
486 baseline.add(MaxPooling2D(pool_size = 3, padding='same',
    ↪  strides=(2,2)))
487
488 # Block 2
489
490 baseline.add(Conv2D(filters = 192, kernel_size= 5,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
491 baseline.add(BatchNormalization())
492 baseline.add(Activation('relu'))
493
494 baseline.add(Conv2D(filters = 192, kernel_size= 1,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
495 baseline.add(BatchNormalization())
```

```
496 baseline.add(Activation('relu'))
497
498 baseline.add(Conv2D(filters = 192, kernel_size= 1,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
499 baseline.add(BatchNormalization())
500 baseline.add(Activation('relu'))
501
502 baseline.add(AveragePooling2D(pool_size = 3, padding='same',
    ↪  strides=(2,2)))
503
504 # Block 3
505
506 baseline.add(Conv2D(filters = 192, kernel_size= 3,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
507 baseline.add(BatchNormalization())
508 baseline.add(Activation('relu'))
509
510 baseline.add(Conv2D(filters = 192, kernel_size= 1,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
511 baseline.add(BatchNormalization())
512 baseline.add(Activation('relu'))
513
514 baseline.add(Conv2D(filters = 192, kernel_size= 1,
    ↪  use_bias=False, padding='same', strides=(1, 1)))
515 baseline.add(BatchNormalization())
516 baseline.add(Activation('relu'))
517
518 baseline.add(GlobalAveragePooling2D())
519
520
521 baseline.add(Dense(num_classes2))
522 baseline.add(Activation('softmax'))
523
524 #Show the architecture of the Baseline model.
525 baseline.summary()
526
527 #Training the Baseline model
528
529 #Define the optimizer
530 opt2 = SGD(learning_rate=0.01, momentum=0.9,  nesterov=True)
531 baseline.compile(loss='categorical_crossentropy', optimizer=opt2,
    ↪  metrics=["accuracy"])
532
```

```
533 # train model
534 hist3 = baseline.fit(x_train100_2, y_train100_2,
535             batch_size=50,
536             epochs=10,
537             shuffle=True,                          # verbosité
538             verbose = 1,
539             validation_data=(X_val2, Y_val2)
540             )
541
542 #Plot the learning curves and loss function for the training and
    ↪  validation set
543
544 # Plot the accuracy
545 plt.plot(hist3.history['accuracy'], label='training
    ↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
546 plt.plot(hist3.history['val_accuracy'], label='validation
    ↪  set',marker='o', linestyle='solid',linewidth=1, markersize=6)
547 plt.title("CNN-model accuracy")
548 plt.xlabel('#Epochs')
549 plt.ylabel('Acuracy')
550 plt.legend(bbox_to_anchor=( 1., 1.))
551
552 #Plot the Loss function
553 plt.plot(hist3.history['loss'], label='training set',marker='o',
    ↪  linestyle='solid',linewidth=1, markersize=6)
554 plt.title("CNN-model loss")
555 plt.xlabel('#Epochs')
556 plt.ylabel('Loss')
557 plt.legend(bbox_to_anchor=( 1., 1.))
558
559
560 #Test the model Baseline on the Test set
561
562 #Apply the model to Baseline to obtain the predicted Y.
563
564 y_pred5 = baseline.predict(X_test)
565 y_pred5 = np.argmax(y_pred5, axis=1)
566
567 #Reshape the correct Y according to the shape of the predicted Y.
568 y_true5 = Y_test_Original.reshape(y_pred5.shape)
569
570 #Compute the accuracy score by comparing predicted y with the
    ↪  correct y.
```

```
accuracy_score(y_true=y_true5,y_pred=y_pred5)


accuracy=[]
epochs=[]
j=0
for i in range(4000,len(y_test)+1,2000):
   y_pred3 =baseline.predict(X_test[j:i])

   y_pred3 = np.argmax(y_pred3, axis=1)
   y_pred3= to_categorical(y_pred3, 10)

   y_true3 = y_test[j:i].reshape(y_pred3.shape)
   if (i % 200==0):
    current_accuracy=accuracy_score(y_true=y_true3,y_pred
    =y_pred3)
    accuracy.append(current_accuracy)
    epochs.append(i)
   j=i

# Plot accuracy for test set
plt.plot(accuracy, label='test set',marker='o',
 ↪  linestyle='solid',linewidth=1, markersize=6)
plt.title("CNN-model accuracy")
plt.xlabel('#Epochs')
plt.ylabel('Acuracy')
plt.legend(bbox_to_anchor=( 1., 1.))

Results obtained after applying the baseline model to the test
 ↪  data
show_data_label_prediction(X_test, y_test, baseline)
```

Chapitre 3

Méthode VAT (Virtual Adversarial Training)

3.1 Contexte

Dans le contexte des problèmes de classification, le phénomène de sur-apprentissage se manifeste souvent par une disparité significative entre les erreurs de prédiction sur l'ensemble d'entraînement et celles sur l'ensemble de test. La régularisation, généralement impliquant une augmentation de la fonction de coût, constitue une méthode couramment utilisée pour atténuer cette disparité. L'approche de régularisation par perturbation adversarielle confère au modèle une robustesse face aux perturbations présentes dans les données.
Dans cette section, nous nous intéressons particulièrement à la méthode semi-supervisée basée sur la régularisation appelée VAT (Virtual Adversarial Training).

3.2 Description de la méthode

[2], Le VAT (Virtual Adversarial Training) est une méthode de régularisation qui marche pour l'apprentissage supervisé et l'apprentissage semi-supervisé. Elle est basée sur la perte virtuelle d'adversaires qui est une nouvelle mesure de l'uniformité locale de la distribution conditionnelle de l'étiquette donnée en entrée. L'approche VAT est une technique d'entraînement permettant de régulariser un modèle profond en utilisant des données non labélisées. Le principe de cette technique consiste à optimiser un bruit adapté, qui est une fois ajouté aux images non labélisées, il les rende plus difficiles à classer pour le modèle, car elles sont plus éloignées du centre de distribution de la classe.
Le bruit virtuellement adversaire est calculé pour chaque image, par rétropropagation de la fonction non supervisée des distributions des scores entre l'image

bruitée et l'image sans bruit.

— Soit $x \in \mathbb{R}^{N \times I}$ le jeu de données d'entrée et Y l'espace des labels. Dans notre cas, $Y = \{0, \ldots, 9\}$.
— N est le nombre de lignes de x (le nombre d'images) et chaque image $x_i \in x$ est de dimension I.
— On note $p(y|x, \theta)$ la distribution de y conditionnellement à x paramétrée par θ.
— On note aussi $D_l = \{(x_l, y_l)\}$ le jeu de données labellisé et $D_{ul} = \{x_{ul}\}$ le jeu de données non labellisé. Les nombres d'images dans D_l et D_{ul} sont respectivement notés par N_l et N_{ul}.

Nous entraînons donc le modèle avec D_l et D_{ul} en suivant les étapes principales suivantes :

1. On transforme x en ajoutant une petite perturbation r pour obtenir un nouveau jeu de données x + r. La perturbation doit être dans la direction adversaire et telle que la sortie (les prédictions des labels) issue du jeu de données perturbé est différente de celle du jeu de données non perturbé.

2. La fonction de coût de la perturbation est ensuite calculée comme la divergence de Kullback-Leibler entre les 2 sorties, et elle doit être maximale. vu qu'on veut éloigner le plus possible les prédictions telles que décrites dans l'étape 1.
La divergence de Kullback-Leibler est définie par :

$$\mathrm{KL}(r, x, \theta) = K(q(y|x), p(y|x + \mathrm{radv}, \theta))$$

où $\mathrm{radv} = \arg\max_r \{\mathrm{KL}(r, x, \theta); \|\mathbf{r}\|_2 \leq \epsilon\}$

La variable *radv* est appelée la direction adversaire, et $\epsilon > 0$ est un hyperparamètre que l'on choisit pour contrôler la perturbation que l'on veut petite.

Nous n'avons aucune information sur $q(y|x)$ pour les données non labellisées. Cette méthode remplace $q(y|x)$ par une estimation $p(y|x, \theta)$. Les labels de x_{ul} sont donc générés d'une manière virtuelle, d'où le nom de la méthode.

L'uniformité locale de la distribution mesure l'uniformité de la sortie du modèle par rapport aux données d'entrée. Nous voulons rendre le modèle assez robuste (presque insensible aux petites perturbations appliquées aux données d'entrée). En d'autres termes, nous ne voulons pas de grands changements dans la sortie du modèle par rapport aux petits changements (petites perturbations) dans le jeu d'entrée.Avec ceci, nous obtenons comme fonction de coût de la perturbation :

$$\mathrm{LDS}(x, \theta) = K(p(y|x, \theta), p(y|x + radv, \theta))$$

3. Le terme de perturbation est donné par

$$R_{\text{vadv}}(D_l, D_{\text{ul}}, \theta) = \frac{1}{N_l + N_{\text{ul}}} \sum_{x \in D_l, D_{\text{ul}}} \text{LDS}(x, \theta)$$

4. La fonction de coût complète est donnée par $l(D_l, \theta) + \alpha R_{\text{vadv}}(D_l, D_{\text{ul}})$, où $l(D_l, \theta)$ est la log-vraisemblance négative du jeu de données labellisé et α est notre deuxième hyperparamètre important, le coefficient de régularisation. Il contrôle l'équilibre entre $l(D_l, \theta)$ et $R_{\text{vadv}}(D_l, D_{\text{ul}})$.

5. Cette fonction de coût est ensuite minimisé.

Nous entraînons ensuite le modèle avec une architecture choisie et obtenons différents résultats. Le VAT est semblable à la méthode de Adversarial Training, mais se distingue de cette méthode dans le sens où elle détermine la direction adversaire avec les données non labéllisées, d'où son utilisation pour l'apprentissage semi supervisé. Cette méthode a plusieurs avantages par rapport aux autres méthodes tels que :

- Il peut être appliquée à l'apprentissage supervisé et non supervisé.
- Il y a au plus deux hyperparamètres à faire varier.
- Faible coût en terme de calcul. Le calcul du gradient de LDS peut être fait en trois retropropagations
- La performance de la méthode ne change pas sous la reparamétrisation du modèle .

3.3 Modèle baseline

Un modèle de référence a été formé exclusivement avec les 100 données labellisées en utilisant une architecture identique à celle du modèle VAT. Ce processus exige un nombre d'époques considérablement plus élevé que le premier modèle pour démontrer des améliorations. La fonction de coût est simplement calculée sur les 100 données labellisées, sans recourir à un terme de régularisation.

3.3.1 Architecture du modèle

- Conv2D avec la fonction d'activation Relu ,5 filtres et kernel size de 5 .
- Max Pooling avec un pool size de 2.
- Une couche dense avec une fonction d'activation Relu et 7210 unités .
- Une couche dense avec une fonction d'activation Softmax et 110 unités .
- Minimisation de la fonction de perte de categorical cross entropy .

— Nombre d'epochs 1000 et optimiser :adam

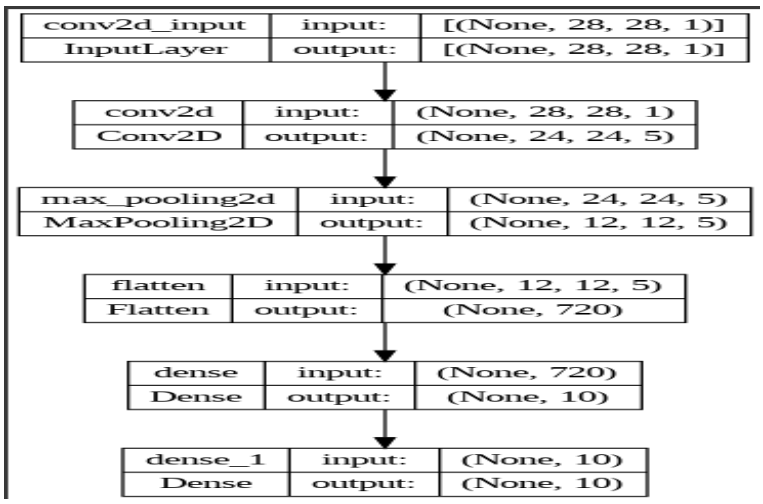

Figure 3.1 : *L'architecture du modèle baseline*

3.3.2 Résultats obtenus

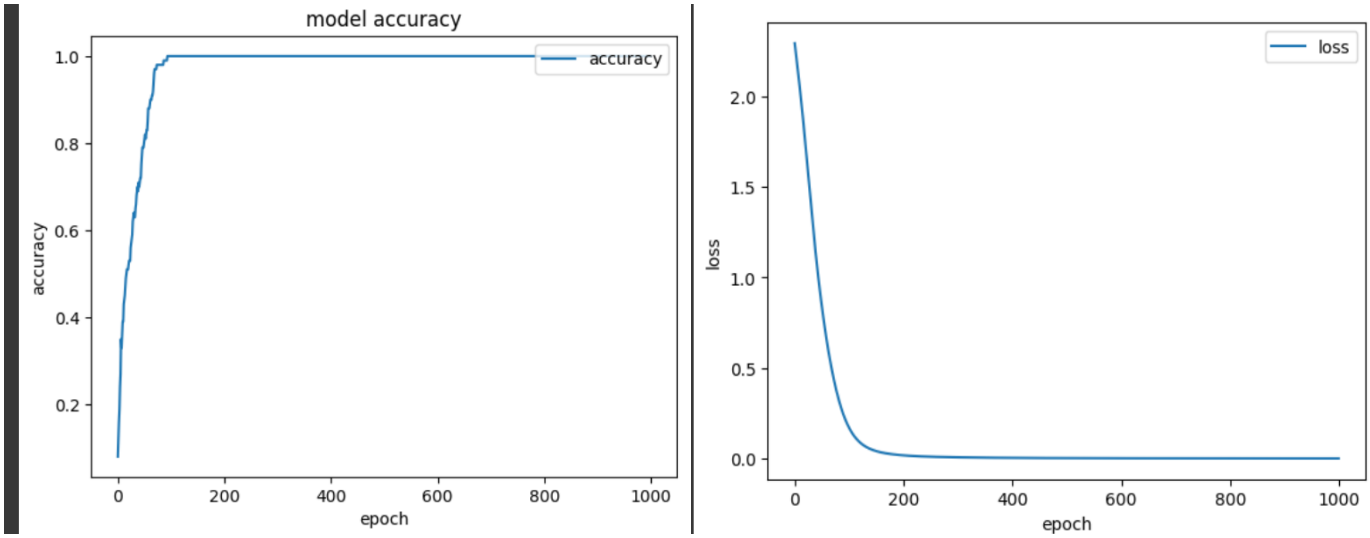

Figure 3.2 : *L'évaluation de l'accuracy et de la focntion loss du modèle baseline sur le train set*

Dans l'apprentissage supervisé (modèle baseline), on observe que l'accuracy du modèle sur le train set augmente en fonction du nombre d'epoch et devient relativement stable à partir de l'époch 1000 environ. Elle atteint une valeur environ de 100%. Cependant ,l'accuarcy obtenue sur le test set est de 67% ce qui confirme

que le modéle entrainé sur les 100 images labelisée n'as pas été généralisé et souffre du sur-apprentissage.

3.4 Le modèle VAT

3.4.1 L'architecture du modèle VAT

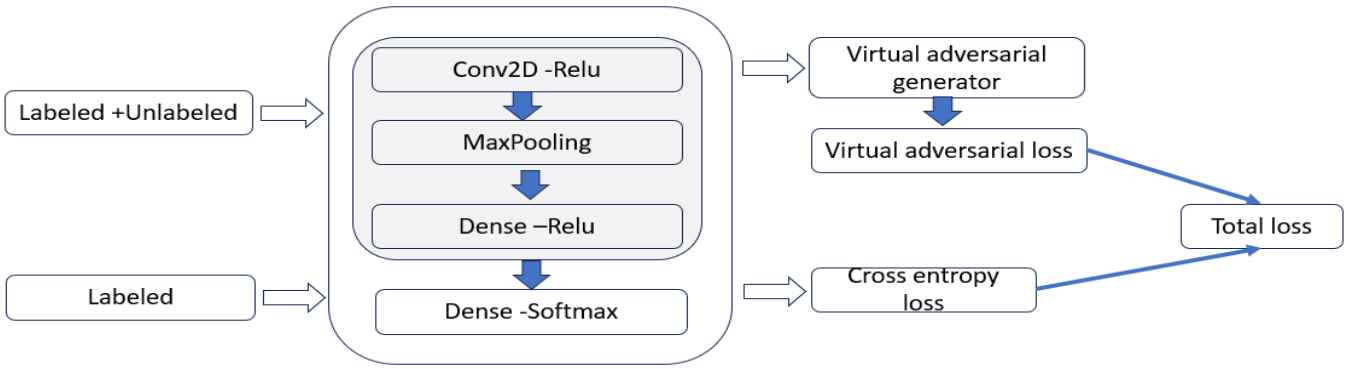

FIGURE 3.3 : *L'architecture du modèle VAT*

3.4.2 Résultats obtenus

Pour l'apprentissage semi-supervisé avec le modèle VAT, on voit que l'accuracy du modèle sur le test set augmente en fonction du nombre d'épochs jusqu'à atteindre une valeur de 88%. On rappelle que la meilleure valeur d'accuracy est obtenue avec les hyperparamètres pour $\psi = e(-6), (\alpha = 2$ et lr $=0.0001$. $F(\cdot)$

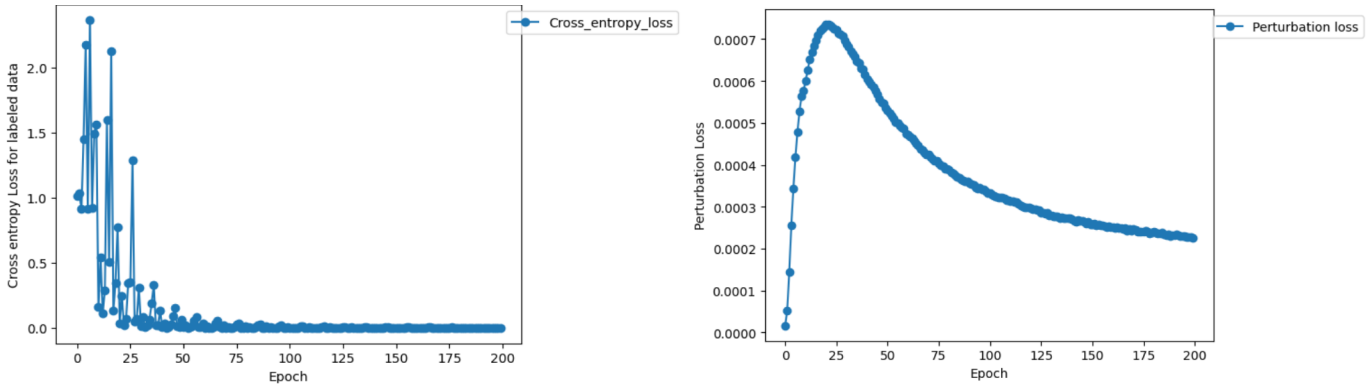

FIGURE 3.4 : *Le Cross entropy et perturbation loss du modèle VAT*

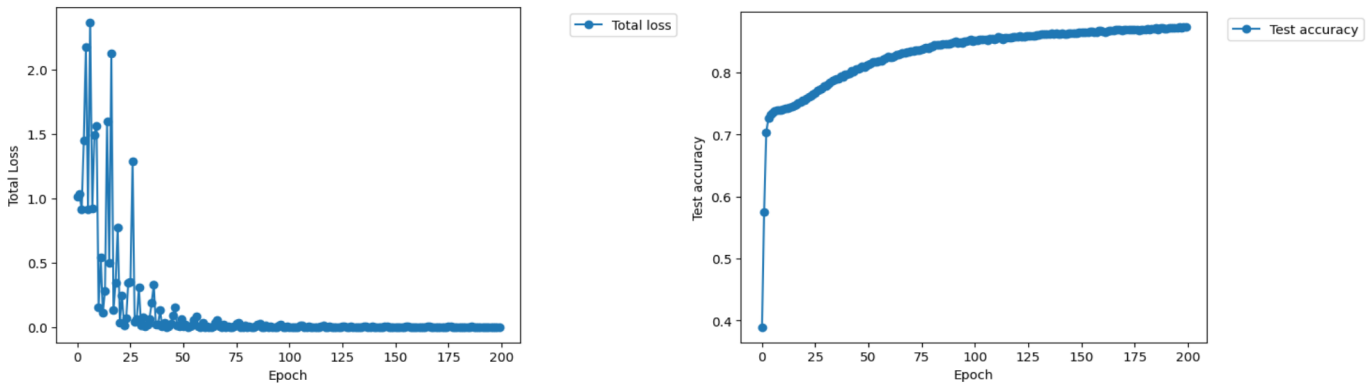

FIGURE 3.5 : *L'accuracy et le total loss du modèle VAT*

D'aprés l'analyse des courbes ci-dessus les résultats obtenus avec modèle VAT sur les données MNIST sont largement meilleurs. Nous constatons clairement la valeur ajoutée de cette démarche donc on peut dire que les résultats sont concluants en apprentissage semi-supervisé sur les données MNIST.

3.5 Conclusion

En conclusion,le modèle baseline, bien que présentant une accuracy de 100% sur l'ensemble d'entraînement, souffre de sur-apprentissage, comme en témoigne sa faible précision de 67% sur l'ensemble de test. En revanche, le modèle semi-supervisé avec Virtual Adversarial Training (VAT) montre des performances nettement meilleures, atteignant une accuracy de 88% sur le test set. Cette amélioration substantielle, associée à l'utilisation de 60,000 données non labellisées pour l'entraînement, démontre la valeur ajoutée de l'approche semi-supervisée, en particulier dans le contexte où seulement 100 données sont labellisées. Ainsi, les résultats obtenus avec le modèle VAT sur les données MNIST sont concluants et mettent en évidence l'efficacité de cette approche dans un scénario de faible supervision.

3.6 Code

```
tf.keras.backend.set_floatx('float64') #Change numerical type to
↪  'float64'
#Load the data
from keras.datasets import mnist
(x_train_original, y_train_original), (x_test, y_test) =
↪  mnist.load_data()
num_classes = y_train_original.max() +1 #number of labels
```

```
y_train = to_categorical(y_train_original, num_classes)
y_test = to_categorical(y_test, num_classes)
#Normalize the data
x_train = x_train_original/255
x_test = x_test/255
#Train-test split
from sklearn.model_selection import train_test_split
x_labeled, x_unlabeled, y_labeled, y_unlabeled =
↪  train_test_split(x_train, y_train,
test_size=59900, random_state=42, shuffle= True,stratify=y_train)

print("x_train.shape", x_train.shape)
print("y_train.shape", y_train.shape)
print("x_test.shape", x_test.shape)
print("y_test.shape", y_test.shape)
print("x_labeled.shape", x_labeled.shape)
print("x_unlabeled.shape", x_unlabeled.shape)

n_filters = 5
kernel_size = 5
pool_size = 2
n_classes = 10

Baseline_model = Sequential([
    Conv2D(n_filters,
                  kernel_size,
                  activation='relu',
                  input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=pool_size),
    Flatten(),
    Dense(10, activation="relu"),
    Dense(n_classes, activation='softmax')
])

Baseline_model.compile(optimizer="adam",
↪  loss="categorical_crossentropy",
              metrics=["accuracy"])
history = Baseline_model.fit(x_labeled, y_labeled,
↪  batch_size=128,
         epochs=1000,
         verbose=2)

plt.plot(history.history['accuracy'])
```

```
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['accuracy'], loc='upper right')
plt.show()

plt.plot(history.history['loss'])
plt.legend(['loss'], loc='upper right')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()

# Function for calculating the KL divergence of 2 distributions from output logit of the neural network
def KL_divergence(p_logit, q_logit, value_min = 1e-30):
  #Input: p_logit, q_logit : logit values
  # value_min: to prevent log(probability) = -inf when the probability is too small
  #Output: KL divergence of p_logit and q_logit
      p = tf.nn.softmax(p_logit)
      q = tf.nn.softmax(q_logit)
      return tf.reduce_sum(p*(tf.math.log(p + value_min) - tf.math.log(q + value_min)), axis=1)



# Function to obtain a unit vector
def normalization(d):
    #Input: d: a vector
    #Output: an unit vector having same direction as d
      d = d/(1e-15 + tf.reduce_max(tf.abs(d), range(1, len(d.get_shape())), keepdims=True))
      norm_d = tf.norm(d+1e-15,axis=[1,2])
      return (d+1e-15) / tf.reshape(norm_d, [d.shape[0],1,1])



#Function for generating the data with virtual adversarial pertubation
def generator(x, psi, epsilon):
    #Input: x: original input data
    # psi: small number
    # epsilon: norm of the pertubation term
```

```
84      #Output: add the virtual adversarial pertubation back into
        ↪ the original input
85        r = psi * normalization(tf.random.normal(shape=tf.shape(x),
          ↪ dtype=tf.float64))
86        with tf.GradientTape() as tape:
87          tape.watch(r)
88          p_logit = VAT_model(x)
89          p_logit_r = VAT_model(x + r)
90          kl = tf.reduce_mean(KL_divergence(p_logit , p_logit_r))
91        grad = tape.gradient(kl, r)
92        return x + epsilon * normalization(grad)
93
94
95  #Function for calculating the virtual adversarial pertubation
    ↪ loss
96  def pertubation_loss(p_logit, p_logit_r):
97      #Input: p_logit: logit value of train data
98      # p_logit_r: logit value of train data with virtual
        ↪ adversarial pertubation
99      #Output: R_vadv of p_logit and p_logit_r
100       return tf.reduce_mean(KL_divergence(p_logit, p_logit_r))
101
102 #Function for calculating the cross entropy loss of the labeled
    ↪ data
103 def cross_entropy_loss(y_labeled, logit_labeled):
104     #Input: y_labled: true value of output
105     # logit_labeled: predicted logit value for labeled data
106     #Output: Categorical Cross-Entropy Loss
107       return
108       tf.keras.losses.CategoricalCrossentropy(from_logits=True)
109       .call(y_labeled, logit_labeled)
110
111
112 #Calculate the total loss of VAT
113 def vat_loss(logit_p, logit_p_r, y_labeled, logit_labeled, alpha
    ↪ = 1):
114   #vat_loss = alpha * pertubation_loss + cross_entropy_loss
115   return pertubation_loss(logit_p, logit_p_r) * alpha +
      ↪ cross_entropy_loss(y_labeled, logit_labeled)
116
117 n_filters = 5
118 kernel_size = 5
119 pool_size = 2
```

```
n_classes = 10

VAT_model = Sequential([
    Conv2D(n_filters,
                  kernel_size,
                  activation='relu',
                  input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=pool_size),
    Flatten(),
    Dense(10, activation="relu"),
    Dense(n_classes, activation='softmax')

])

#Implement the network and setting hyperparameters for trainning
x_train = tf.constant(x_train, dtype = tf.float64)
x_labeled = tf.constant(x_labeled, dtype = tf.float64)
EPOCHS = 200 #number of epochs
BATCH_SIZE = 256
BUFFER_SIZE = 60000
DISPLAY_STEP = 1
epsilon = 1e-1
psi = 1e-6
alpha = 2.0
lnr = tf.keras.optimizers.schedules.PolynomialDecay(
initial_learning_rate = 0.0001, decay_steps=100000,
 ↪  end_learning_rate=0.0000001, power=1.0,
cycle=False, name=None) #learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=lnr)
 ↪  #optimizer

#Implement training loop
@tf.function
def train_step(x_train, x_labeled, y_labeled):
  x_gen = generator(x_train, psi, epsilon) #generate virtual
   ↪  perturbation
  with tf.GradientTape() as tape:
    tape.watch(VAT_model.trainable_weights)
    logit_p = VAT_model(x_train)
    logit_p_r = VAT_model(x_gen)
    logit_labeled = VAT_model(x_labeled)
    loss = vat_loss(logit_p, logit_p_r, y_labeled, logit_labeled,
     ↪  alpha)
```

```
159     grad = tape.gradient(loss, VAT_model.trainable_weights)
160     #minimize total loss by updating weight parameters of the
        ↪  neural VAT_model
161     optimizer.apply_gradients(zip(grad,
        ↪  VAT_model.trainable_weights))
162
163
164 #Create training batch and shuffle
165 train_dataset =
    ↪  tf.data.Dataset.from_tensor_slices(x_train).shuffle(BUFFER_SIZE)
166 .batch(BATCH_SIZE)
167 Total_loss = []
168 Pertubation_loss = []
169 Cross_entropy_loss = []
170 Test_accuracy = []
171 Epoch = []
172 for epoch in tqdm(np.arange(0,EPOCHS)):
173     for image_batch in train_dataset:
174         train_step(image_batch, x_labeled, y_labeled)
175
176
177     if (epoch % DISPLAY_STEP == 0 ) or epoch == 0 or
        ↪  (epoch==EPOCHS-1):
178         #Calculate loss functions
179         x_gen = generator(x_train, psi, epsilon)
180         logit_p = VAT_model(x_train)
181         logit_p_r = VAT_model(x_gen)
182         logit_labeled = VAT_model(x_labeled)
183         per_loss = pertubation_loss(logit_p, logit_p_r)
184         cross_loss = cross_entropy_loss(y_labeled, logit_labeled)
185         loss = vat_loss(logit_p, logit_p_r, y_labeled,
            ↪  logit_labeled, alpha)
186         #Prediction on test set
187         logit_test = VAT_model(x_test)
188         test_acc = sum(np.argmax(logit_test,
            ↪  1)-np.argmax(y_test,1)==0) / 10000
189         #Save perfomance
190         Epoch = np.append(Epoch, epoch)
191         Total_loss = np.append(Total_loss, loss.numpy())
192         Pertubation_loss = np.append(Pertubation_loss,
            ↪  per_loss.numpy())
193         Cross_entropy_loss = np.append(Cross_entropy_loss,
            ↪  cross_loss.numpy())
```

```
        Test_accuracy = np.append(Test_accuracy, test_acc)
        print(test_acc) #Print result for each epoch


plt.plot(Epoch, Test_accuracy, marker='o', label='Test accuracy')
plt.xlabel('Epoch')
plt.ylabel('Test accuracy')
plt.legend(bbox_to_anchor=( 1.35, 1.))
plt.show()

plt.plot(Epoch,Cross_entropy_loss[0:2000:10], marker='o',
 ↪  label='Cross_entropy_loss')
plt.xlabel('Epoch')
plt.ylabel('Cross entropy Loss for labeled data')
plt.legend(bbox_to_anchor=( 1.35, 1.))
plt.show()

plt.plot(Epoch,Pertubation_loss, marker='o', label='Perturbation
 ↪  loss')
plt.xlabel('Epoch')
plt.ylabel('Perturbation Loss')
plt.legend(bbox_to_anchor=( 1.35, 1.))
plt.show()

plt.plot(Epoch, Total_loss[0:2000:10], marker='o', label='Total
 ↪  loss')
plt.xlabel('Epoch')
plt.ylabel('Total Loss')
plt.legend(bbox_to_anchor=( 1.35, 1.))
plt.show()

```

Chapitre 4

Méthode de Unsupervised Data Augmentation (UDA)

4.1 Description de la méthode

Dans le domaine de la vision, des augmentations simples telles que le rognage et le retournement sont appliquées aux exemples étiquetés. Pour minimiser la divergence entre l'entraînement supervisé et la prédiction sur les exemples non étiquetés, nous appliquons les mêmes augmentations simples aux exemples non étiquetés pour améliorer le modèle.

Dans notre projet, nous avons utilisé la méthode semi-supervisée d'augmentation RandAugment sur les images non étiquetées afin de les rendre plus similaires aux images étiquetées pour la tache de classification. Cette approche sera détaillée par la suite.

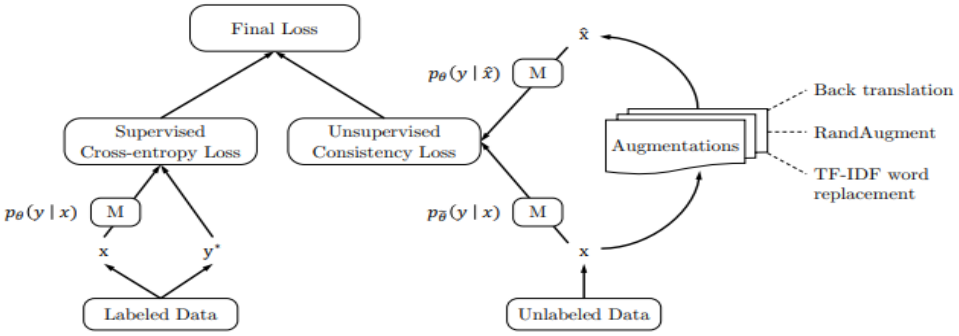

Figure 4.1 : *Résumé sur la méthode d'augmentation non supervisée des données*

- [3], Étant donné une entrée x, calculer la distribution de sortie $p_\theta(y|x)$ étant

donné x et une version bruitée $p_\theta(y|x,\varepsilon)$ en injectant un petit bruit ε. Le bruit peut être appliqué à x ou aux états cachés.
• Minimisez une métrique de divergence entre les deux distributions $D(p_\theta(y|x)$ et $D(p_\theta(y|x,\varepsilon)$.
Cette procédure contraint le modèle à être insensible au bruit ϵ et donc plus régulier par rapport aux variations dans l'espace d'entrée (ou caché). D'un autre point de vue, la minimisation de la perte de cohérence propage progressivement les informations d'étiquette des exemples étiquetés aux exemples non étiquetés.

Dans ce travail, nous nous intéressons à un contexte particulier où le bruit est injecté dans l'entrée x, c'est-à-dire, $\hat{x} = q(x,\epsilon)$. Lorsque le modèle M est entraîné conjointement avec des exemples étiquetés (100 images), nous utilisons un facteur de pondération pour équilibrer l'entropie croisée supervisée et la perte d'entraînement par cohérence non supervisée, comme illustré dans la Figure 1. Formellement, l'objectif complet peut être écrit comme suit :

$$\min_\theta J(\theta) = \mathbb{E}_{x_1\sim p_L(x)}\left[-\log p_\theta(f^*(x_1)|x_1)\right] + \lambda \mathbb{E}_{x_2\sim p_U(x)} \mathbb{E}_{\hat{x}\sim q(\hat{x}|x_2)} \mathrm{CE}\left(p_{\theta'}(y|x_2), p_\theta(y|\hat{x})\right)$$

où CE désigne l'entropie croisée, $q(\hat{x}|x)$ est une transformation d'augmentation de données et θ' est une copie fixe des paramètres actuels θ, indiquant que le gradient n'est pas propagé à travers θ'. Nous fixons à 1 pour la plupart de nos expériences. En pratique, à chaque itération, nous calculons la perte supervisée sur un mini-lot (batch) d'exemples étiquetés et calculons la perte de cohérence sur un mini-lot (batch) de données non étiquetées. Les deux pertes sont ensuite additionnées pour obtenir la perte finale. Nous utilisons une taille de lot plus grande pour la perte de cohérence.

4.2 RandAugment

Nous utilisons une méthode d'augmentation de données appelée RandAugment basée sur l'échantillonnement uniforme à partir d'un ensemble de transformations d'augmentation prédifini. En d'autres termes, RandAugment est simple et ne nécessite pas de données étiquetées, car il n'est pas nécessaire de rechercher des politiques optimales [3].

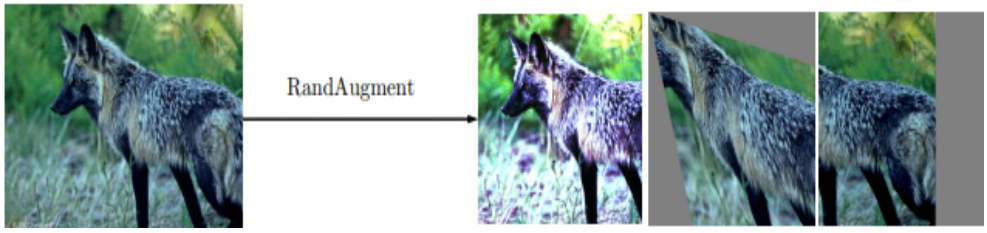

FIGURE 4.2 : *Le processus d'augmentation d'une image avec RandAugment*

4.3 Le modèle basé sur UDA

4.3.1 Architecture du modèle basé sur UDA :

Le modèle UDA est composé des couches suivantes :

- Une couche d'entrée Flatten de 784 unités qui prend en entrée une image augmentée.
- 2 couches Dense avec la fonction d'activation Relu.
- 2 couches Dropout.
- Une couche de sortie Dense avec 10 unités.

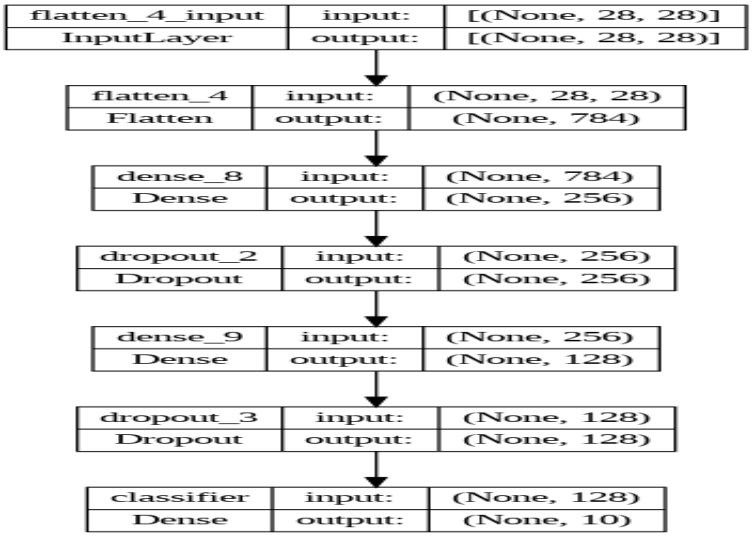

FIGURE 4.3 : *Architecture du modèle basée sur l'augmentation non supervisée des données*

Pour entrainer le modèle sur les données d'apprentissage, nous avons adopté la descente de gradient (Adam) avec une taille de lot (batch size) de 128, un dropout de 0.8 et un taux d'apprentissage (lr) de 1e-2 ainsi que nous avons utilisé la

régularisation L2 avec une valeur de 0.1, un uda-softmax-temp de 0.9 et uda-confidence-thresh de 0.8. L'entraînement est réalisé sur un total de 20 époques.

4.3.2 Résultats obtenus :

Nous représentons graphiquement l'évolution de l'accuracy du modèle UDA entraîné.

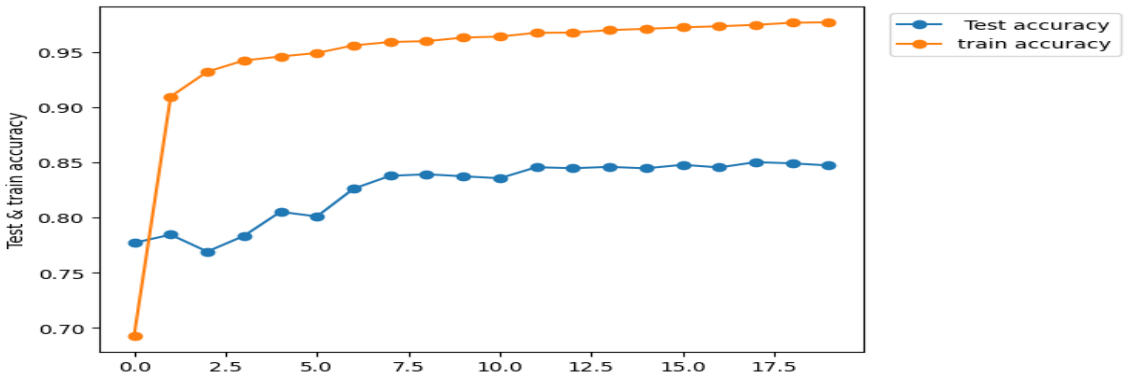

Figure 4.4 : *Evaluation de l'accuracy sur le train et test set*

En analysant l'évolution de l'accuracy entre l'ensemble d'entraînement et l'ensemble de test, nous observons une convergence vers prèsque de 100 % après 17 époques pour les données d'entraînement. Cependant, l'accuracy sur les données de test atteint 88 %. ce qui indique que le modèle UDA est bon.

4.4 Modèle Baseline :

4.4.1 Architecture du modéle baseline

Un modèle de base a été formé sur les 100 images labellisées en utilisant l'architecture suivante :

- Une couche d'entrée Flatten qui prend en entrée une image de 28*28.
- Une couche cachée Dense avec la fonction d'activation Relu.
- Une couche de sortie Dense avec 10 unités .

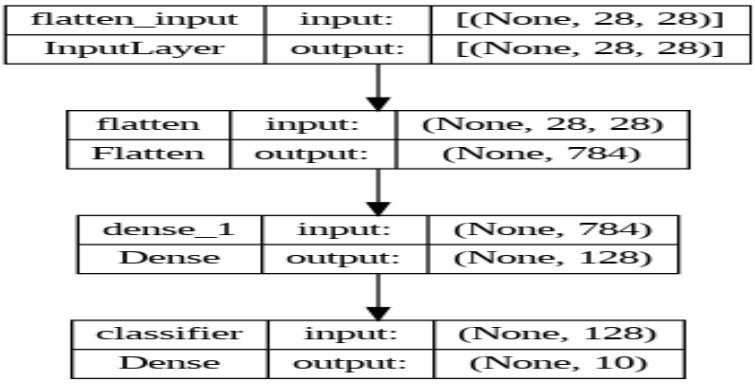

FIGURE 4.5 : *Architecture du modèle de base*

4.4.2 Résultats obtenus :

Aprés avoir entrainé le modèle de base sur les données d'apprentissage (100 images labélisées), nous représentons graphiquement l'évolution de l'accuracy et la fonction de perte du modèle de base sur les données d'entrainement.

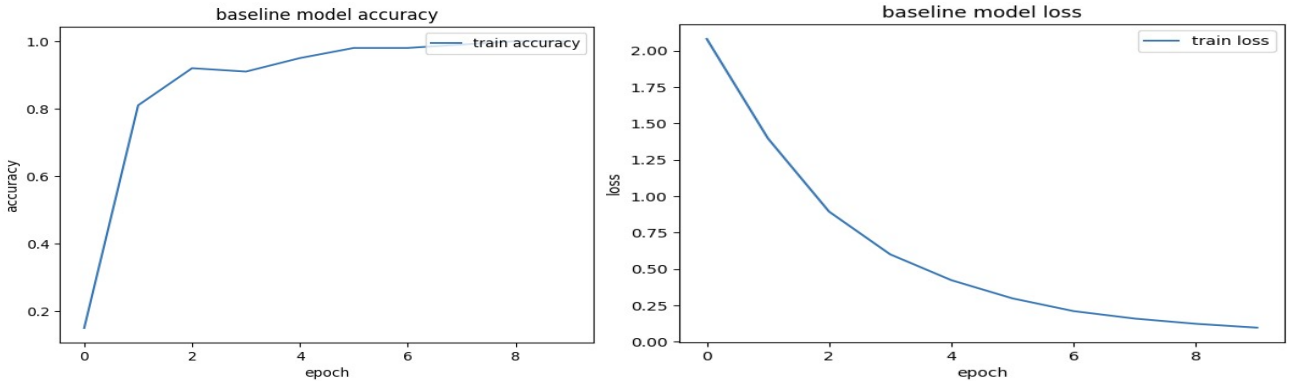

FIGURE 4.6 : *Evaluation de l'accuracy et la fonction de perte du modèle Baseline sur le train set*

En analysant les deux graphes, nous constatons que la fonction de perte diminue et converge vers zéro après 8 époques, et l'accuracy converge vers 1. Cependant, lors de l'évaluation du modèle de base sur les données de test, nous avons obtenu un score d'accuracy de 69% qui est moins généralisé par rapport au modèle basé sur l'augmentation de données non étiquetées.

4.5 Conclusion :

En résumé, l'exploitation de la technique d'augmentation de données non supervisée (UDA) sur notre ensemble de données MNIST, composé de seulement 100 images labellisées et 59 900 images non labellisées, a abouti à des résultats prometteurs. Cette approche s'est avérée particulièrement efficace pour surmonter la contrainte inhérente au nombre limité de données labellisées. Grâce à l'UDA, nous avons pu étendre la diversité et la taille de notre ensemble de données en générant de nouvelles données qui respectent la même distribution que les données existantes. Cette extension a eu un impact significatif sur la capacité de généralisation et la robustesse du modèle. En fin de compte, notre projet de classification d'images de nombres manuscrits a bénéficié de manière significative de cette approche novatrice, ouvrant ainsi la voie à des améliorations continues dans la performance du modèle malgré les contraintes initiales en termes de données labellisées.

4.6 Code :

```
(x_train, y_train), (x_test, y_test) =
↪  tf.keras.datasets.mnist.load_data()

# Rescale the images from [0,255] to the [0.0,1.0] range.
x_train, x_test = x_train[..., np.newaxis]/255.0, x_test[...,
↪  np.newaxis]/255.0

print("Number of original training examples:", len(x_train))
print("Number of original test examples:", len(x_test))

from scipy.ndimage.interpolation import shift

# Method to shift the image by given dimension
def shift_image(image, dx, dy):
    image = image.reshape((28, 28))
    shifted_image = shift(image, [dy, dx], cval=0,
    ↪  mode="constant")
    return shifted_image

idxs = np.ones(100)
for digit in range(10):
  digit_args = np.argwhere(y_train == digit)[:10,0]
  np.random.shuffle(digit_args)
```

```
  idxs[digit*10:(digit+1)*10] = digit_args
np.random.shuffle(idxs)
idxs = list(idxs.astype("int"))

x_train_labeled = x_train[idxs]
y_train_labeled = y_train[idxs]
x_train_unlabeled_original = np.delete(x_train.copy(), idxs,
 ↪  axis=0)
x_train_unlabeled =
 ↪  x_train_unlabeled_original[:int(x_train_unlabeled_original.shape[0]/2)]
# Creating Augmented Dataset
x_train_augmented = []
for image in x_train_unlabeled:
    x_train_augmented.append(np.expand_dims(shift_image(image, 3,
     ↪  3), axis=-1))
x_train_augmented = np.array(x_train_augmented)

digit_idxs = {}
for digit in range(10):
  digit_args = np.argwhere(y_train == digit)[:,0]
  np.random.shuffle(digit_args)
  digit_idxs[digit] = list(digit_args.astype('int'))

elt_per_class = int(min([len(idxs) for idxs in
 ↪  digit_idxs.values()]) / 2)

digit_data = {}
digit_label = {}
for digit in np.unique(y_train):
  digit_data[digit] = ( x_train[
   ↪  digit_idxs[digit][:elt_per_class] ] , x_train[
   ↪  digit_idxs[digit][elt_per_class:elt_per_class*2] ])
  digit_label[digit] = ( y_train[
   ↪  digit_idxs[digit][:elt_per_class] ] , y_train[
   ↪  digit_idxs[digit][elt_per_class:elt_per_class*2] ])

original_data = None
original_labels = None
for digit in np.unique(y_train):
  if original_data is None:
    original_data = digit_data[digit][0]
    original_labels = digit_label[digit][0]
  else:
```

```
    original_data = np.concatenate((original_data, digit_data[digit][0]))
    original_labels = np.concatenate((original_labels, digit_label[digit][0]))

augmented_data = None
augmented_labels = None
for digit in np.unique(y_train):
  if augmented_data is None:
    augmented_data = digit_data[digit][1]
    augmented_labels = digit_label[digit][1]
  else:
    augmented_data = np.concatenate((augmented_data, digit_data[digit][1]))
    augmented_labels = np.concatenate((augmented_labels, digit_label[digit][1]))

permutation = np.random.permutation(original_data.shape[0])
augmented_data, original_data, augmented_labels, original_labels = augmented_data[permutation], original_data[permutation], augmented_labels[permutation], original_labels[permutation]

inputs = keras.Input(shape=(28,28), name="digits")
x1 = keras.layers.Dense(128, activation="relu")(inputs)
outputs = keras.layers.Dense(10, name="predictions")(x1)
model = keras.Model(inputs=inputs, outputs=outputs)

model = keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu', kernel_regularizer=keras.regularizers.l2(5e-4),
        #kernel_initializer=keras.initializers.he_normal
                              ),
        tf.keras.layers.Dense(10, name="classifier")
])

optimizer = keras.optimizers.Adam()
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
train_acc_metric = keras.metrics.SparseCategoricalAccuracy()
val_acc_metric = keras.metrics.SparseCategoricalAccuracy()
batch_size = 128

```

```
labels_per_batch=10
unlabels_per_batch=59
all_x = None
for i in range(int(100/labels_per_batch)):
    if all_x is None:
        all_x
        ↪ np.concatenate((x_train_labeled[i*labels_per_batch:i*labels_per_batch
        ↪ x_train_unlabeled[i*unlabels_per_batch :
        ↪ i*unlabels_per_batch+unlabels_per_batch],
        ↪ x_train_augmented[i*unlabels_per_batch:
        ↪ i*unlabels_per_batch+unlabels_per_batch]))
    else:
        all_x = np.concatenate((all_x,

        ↪ x_train_labeled[i*labels_per_batch:i*labels_per_batch+labels_per_bat
        ↪ x_train_unlabeled[i*unlabels_per_batch :
        ↪ i*unlabels_per_batch+unlabels_per_batch],
        ↪ x_train_augmented[i*unlabels_per_batch:
        ↪ i*unlabels_per_batch+unlabels_per_batch]))

@tf.function
def test_step(x, y):
    val_logits = model(x, training=False)
    val_acc_metric.update_state(y, val_logits)

val_dataset = tf.data.Dataset.from_tensor_slices((x_test,
↪ y_test))
val_dataset = val_dataset.batch(64)
epochs = 10
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))

    # Iterate over the batches of the dataset.
    for step in range(math.ceil(all_x.shape[0]/batch_size)):
        x_batch_train =
        ↪ all_x[step*batch_size:step*batch_size+labels_per_batch]
        y_batch_train =
        ↪ y_train_labeled[step*labels_per_batch:step*labels_per_batch+labels_p

        # x_train_unlabeled[i*unlabels_per_batch :
        ↪ i*unlabels_per_batch+unlabels_per_batch]
        # x_train_augmented[i*unlabels_per_batch:
        ↪ i*unlabels_per_batch+unlabels_per_batch]
```

```
        # Open a GradientTape to record the operations run
        # during the forward pass, which enables
        ↪ auto-differentiation.
        with tf.GradientTape() as tape:

            # Run the forward pass of the layer.
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            logits = model(x_batch_train, training=True)  #
            ↪ Logits for this minibatch
            # Compute the loss value for this minibatch.
            loss_value = loss_fn(y_batch_train,
            ↪ logits[:labels_per_batch])

        # Use the gradient tape to automatically retrieve
        # the gradients of the trainable variables with respect
        ↪ to the loss.
        grads = tape.gradient(loss_value,
        ↪ model.trainable_weights)

        # Run one step of gradient descent by updating
        # the value of the variables to minimize the loss.
        optimizer.apply_gradients(zip(grads,
        ↪ model.trainable_weights))

        # Update training metric.
        train_acc_metric.update_state(y_batch_train,
        ↪ logits[:labels_per_batch])

        # Log every 200 batches.
        if step % 200 == 0:
            print(
                "Training loss (for one batch) at step %d: %.4f"
                % (step, float(loss_value))
            )
            print("Seen so far: %s samples" % ((step + 1) * 64))

    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
    print("Training acc over epoch: %.4f" % (float(train_acc),))

    # Reset training metrics at the end of each epoch
```

```
    train_acc_metric.reset_states()

for x_batch_val, y_batch_val in val_dataset:
    val_logits = model(x_batch_val, training=False)
    # Update val metrics
    val_acc_metric.update_state(y_batch_val, val_logits)
val_acc = val_acc_metric.result()
val_acc_metric.reset_states()
print("Validation acc: %.4f" % (float(val_acc),))

def _kl_divergence_with_logits(p_logits, q_logits):
  p = tf.nn.softmax(p_logits)
  log_p = tf.nn.log_softmax(p_logits)
  log_q = tf.nn.log_softmax(q_logits)

  kl = tf.reduce_sum(p * (log_p - log_q), -1)
  return kl

  def get_ent(logits, return_mean=True):
  log_prob = tf.nn.log_softmax(logits, axis=-1)
  prob = tf.exp(log_prob)
  ent = tf.reduce_sum(-prob * log_prob, axis=-1)
  if return_mean:
    ent = tf.reduce_mean(ent)
  return ent

def get_tsa_threshold(schedule, global_step, num_train_steps,
↪  start, end):
    step_ratio = float(global_step) / float(num_train_steps)
    if schedule == "linear_schedule":
        coeff = step_ratio
    elif schedule == "exp_schedule":
        scale = 5
        # [exp(-5), exp(0)] = [1e-2, 1]
        coeff = tf.exp((step_ratio - 1) * scale)
    elif schedule == "log_schedule":
        scale = 5
    # [1 - exp(0), 1 - exp(-5)] = [0, 0.99]
        coeff = 1 - tf.exp((-step_ratio) * scale)
    return coeff * (end - start) + start

def anneal_sup_loss(sup_logits, sup_labels, sup_loss,
↪  global_step, nbr_steps, num_classes, tsa_schedule):
```

```
196   tsa_start = 1. / num_classes
197   eff_train_prob_threshold = get_tsa_threshold(
198       tsa_schedule, global_step, nbr_steps,
199       tsa_start, end=1)
200   one_hot_labels = tf.one_hot(
201       sup_labels, depth=num_classes, dtype=tf.float32)
202   sup_probs = tf.nn.softmax(sup_logits, axis=-1)
203   correct_label_probs = tf.reduce_sum(
204       one_hot_labels * sup_probs, axis=-1)
205   larger_than_threshold = tf.greater(
206       correct_label_probs, eff_train_prob_threshold)
207   loss_mask = 1 - tf.cast(larger_than_threshold, tf.float32)
208   loss_mask = tf.stop_gradient(loss_mask)
209   sup_loss = sup_loss * loss_mask
210   avg_sup_loss = (tf.reduce_sum(sup_loss) /
211                   tf.maximum(tf.reduce_sum(loss_mask), 1))
212   return sup_loss, avg_sup_loss
213
214 import copy
215
216 def create_train(hyper, epochs=10, pretraining=False,
    ↪  verbose=False):
217
218     hyper = copy.deepcopy(hyper)
219     model = keras.Sequential([
220         tf.keras.layers.Flatten(input_shape=(28, 28)),
221         tf.keras.layers.Dense(256, activation='relu',
          ↪  kernel_regularizer=keras.regularizers.l2(hyper["l2"]),
222                               #
                                   ↪  kernel_initializer=keras.initializers.he_norma
223                              ),
224         tf.keras.layers.Dropout(hyper["dropout"]),
225         tf.keras.layers.Dense(128, activation='relu',
          ↪  kernel_regularizer=keras.regularizers.l2(hyper["l2"]),
226                               #
                                   ↪  kernel_initializer=keras.initializers.he_norma
227                              ),
228         tf.keras.layers.Dropout(hyper["dropout"]),
229                               #
                                   ↪  kernel_initializer=keras.initializers.he_norma
230
231         tf.keras.layers.Dense(10, name="classifier")
232     ])
```

```
233
234     hyper["optimizer"] = keras.optimizers.Adam()
235     hyper["loss_fn"] =
        ↪  keras.losses.SparseCategoricalCrossentropy(from_logits=True)
236     hyper["train_acc_metric"] =
        ↪  keras.metrics.SparseCategoricalAccuracy()
237     hyper["val_acc_metric"] =
        ↪  keras.metrics.SparseCategoricalAccuracy()
238
239     val_dataset = tf.data.Dataset.from_tensor_slices((x_test,
        ↪  y_test))
240     val_dataset = val_dataset.batch(128)
241
242     if pretraining:
243         print("Starting Pretraining")
244         pretrain(model, x_train_labeled, y_train_labeled,
            ↪  val_dataset, epochs)
245     print("Starting Training")
246     acc,history =train(model,
        ↪  x_train_labeled,original_data,augmented_data,
        ↪  val_dataset, hyper, epochs, verbose=verbose )
247     print(history)
248     import pandas as pd
249     df1= pd.DataFrame(data = history)
250     plt.plot(df1.epoch, df1.val_acc, marker='o', label=' Test
        ↪  accuracy')
251     plt.plot(df1.epoch, df1.train_acc, marker='o', label='train
        ↪  accuracy')
252     plt.xlabel('Epoch')
253     plt.ylabel('Test & train accuracy')
254     plt.legend(bbox_to_anchor=( 1.35, 1.))
255     plt.show()
256 def pretrain(model, x_train, y_train,val_dataset, epochs=10,
    ↪  verbose=False):
257     model.compile(
258         optimizer=tf.keras.optimizers.Adam(),
259
            ↪  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
260         metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],
261     )
262
263     model.fit(
264         x_train, y_train,
```

```
265         epochs=epochs,
266         validation_data=val_dataset,
267         verbose=0
268     )
269     print("Pretraining Validation Accuracy: %.4f \n\n" %
        ↪ model.evaluate(val_dataset, batch_size=128)[1])
270
271 def
    ↪ train(model,x_train_labeled,original_data,augmented_data,val_dataset,
    ↪ hyper, epochs=10, verbose=False):
272
273     @tf.function
274     def test_step(x, y):
275         val_logits = model(x, training=False)
276         val_acc_metric.update_state(y, val_logits)
277
278     unlabels_per_batch = hyper["unlabels_per_batch"]
279     batch_unique_labels = hyper["batch_unique_labels"]
280     labels_per_batch = hyper["labels_per_batch"]
281     hyper["labels_per_batch"]
282
283     optimizer = hyper["optimizer"]
284     loss_fn = hyper["loss_fn"]
285     train_acc_metric = hyper["train_acc_metric"]
286     val_acc_metric = hyper["val_acc_metric"]
287     tsa_schedule = hyper["tsa_schedule"]
288     num_classes = hyper["num_classes"]
289     uda_softmax_temp = hyper["uda_softmax_temp"]
290     uda_confidence_thresh = hyper["uda_confidence_thresh"]
291     ent_min_coeff = hyper["ent_min_coeff"]
292     lamd = hyper["lamd"]
293     history={'val_acc':[],'train_acc':[],'epoch':[],'loss':[]}
294     for epoch in trange(epochs):
295         if(verbose):
296             print("\nStart of epoch %d" % (epoch,))
297         start_time = time.time()
298
299         # Iterate over the batches of the dataset.
300         for step in
            ↪ range(int(original_data.shape[0]/unlabels_per_batch)):
301             sup_images = x_train_labeled[int(step %
                ↪ batch_unique_labels)*labels_per_batch:int(step %
                ↪ batch_unique_labels)*labels_per_batch+labels_per_batch]
```

```
302                 ori_images = original_data[step*unlabels_per_batch:
                    ↪  step*unlabels_per_batch+unlabels_per_batch]
303                 aug_images = augmented_data[step*unlabels_per_batch:
                    ↪  step*unlabels_per_batch+unlabels_per_batch]
304
305                 # x_batch_train =
                    ↪  all_x[step*batch_size:step*batch_size+batch_size]
306                 x_batch_train = np.concatenate((sup_images,
                    ↪  ori_images, aug_images))
307                 y_batch_train = y_train_labeled[int(step %
                    ↪  batch_unique_labels)*labels_per_batch:int(step %
                    ↪  batch_unique_labels)*labels_per_batch+labels_per_batch]
308
309                 # x_train_unlabeled[i*unlabels_per_batch :
                    ↪  i*unlabels_per_batch+unlabels_per_batch]
310                 # x_train_augmented[i*unlabels_per_batch:
                    ↪  i*unlabels_per_batch+unlabels_per_batch]
311                 # Open a GradientTape to record the operations run
312                 # during the forward pass, which enables
                    ↪  auto-differentiation.
313                 with tf.GradientTape() as tape:
314
315                     # Run the forward pass of the layer.
316                     # The operations that the layer applies
317                     # to its inputs are going to be recorded
318                     # on the GradientTape.
319                     logits = model(x_batch_train, training=True)  #
                        ↪  Logits for this minibatch
320                     # Compute the loss value for this minibatch.
321                     sup_logits = logits[:labels_per_batch]
322                     sup_loss = loss_fn(y_batch_train, sup_logits)
323
324                     # Use TSA training loss
325                     if tsa_schedule:
326                         sup_loss, avg_sup_loss =
                            ↪  anneal_sup_loss(sup_logits,
                            ↪  y_batch_train, sup_loss, epoch, epochs,
                            ↪  num_classes, tsa_schedule)
327                     else:
328                         avg_sup_loss = tf.reduce_mean(sup_loss)
329                     total_loss = avg_sup_loss
330
331                     # Get loss from the unlabled data
```

```
332
333                 # logits of unlabled real images
334                 ori_logits = logits[labels_per_batch :
                    ↪  labels_per_batch + unlabels_per_batch]
335                 # logits of unlabeled augmented images
336                 aug_logits = logits[labels_per_batch +
                    ↪  unlabels_per_batch : ]
337
338                 #Sharpening predictions for the kl divergence
339                 if uda_softmax_temp != -1:
340                     ori_logits_tgt = ori_logits /
                        ↪  uda_softmax_temp
341                 else:
342                     ori_logits_tgt = ori_logits
343
344                 # Calculate KL divergence
345                 aug_loss = _kl_divergence_with_logits(
346                     p_logits=tf.stop_gradient(ori_logits_tgt),
347                     q_logits=aug_logits)
348
349                 if uda_confidence_thresh != -1:
350                     ori_prob = tf.nn.softmax(ori_logits, axis=-1)
351                     largest_prob = tf.reduce_max(ori_prob,
                        ↪  axis=-1)
352                     loss_mask = tf.cast(tf.greater(
353                         largest_prob, uda_confidence_thresh),
                            ↪  tf.float32)
354                     loss_mask = tf.stop_gradient(loss_mask)
355                     aug_loss = aug_loss * loss_mask
356
357                 if ent_min_coeff > 0:
358                     per_example_ent = get_ent(ori_logits)
359                     ent_min_loss =
                        ↪  tf.reduce_mean(per_example_ent)
360                     total_loss = total_loss + ent_min_coeff *
                        ↪  ent_min_loss
361
362                 avg_unsup_loss = tf.reduce_mean(aug_loss)
363
364                 total_loss += lamd * avg_unsup_loss
365
366
367             # Use the gradient tape to automatically retrieve
```

```
368             # the gradients of the trainable variables with
                ↪  respect to the loss.
369             grads = tape.gradient(total_loss,
                ↪  model.trainable_weights)
370
371             # Run one step of gradient descent by updating
372             # the value of the variables to minimize the loss.
373             optimizer.apply_gradients(zip(grads,
                ↪  model.trainable_weights))
374
375             # Update training metric.
376             train_acc_metric.update_state(y_batch_train,
                ↪  logits[:labels_per_batch])
377
378             if verbose:
379                 # Log every 200 batches.
380                 if step % 50 == 0:
381                     print(
382                         "Training loss (for one batch) at step
                        ↪  %d: %.4f"
383                         % (step, float(total_loss))
384                     )
385
386                     print("Seen so far: %s samples" % ((step + 1)
                        ↪  * batch_size))
387
388         # Display metrics at the end of each epoch.
389         train_acc = train_acc_metric.result()
390         history['train_acc'].append(float(train_acc))
391         history['epoch'].append(epoch)
392         history['loss'].append(float(total_loss))
393         print("Training acc over epoch: %.4f" %
            ↪  (float(train_acc),))
394
395         # Reset training metrics at the end of each epoch
396         train_acc_metric.reset_states()
397
398
399         # Run a validation loop at the end of each epoch.
400         for x_batch_val, y_batch_val in val_dataset:
401             test_step(x_batch_val, y_batch_val)
402
403         val_acc = val_acc_metric.result()
```

```
404         val_acc_metric.reset_states()
405         print("Validation acc: %.4f" % (float(val_acc),))
406         history['val_acc'].append(float(val_acc))
407         if verbose:
408             print("Time taken: %.2fs \n" % (time.time() -
                ↪  start_time))
409     return float(val_acc) ,history
410 
411 val_dataset = tf.data.Dataset.from_tensor_slices((x_test,
    ↪  y_test))
412 val_dataset = val_dataset.batch(128)
413 labels_per_batch = 50
414 batch_size = 128
415 
416 hyper = {
417     "labels_per_batch":labels_per_batch,
418     "batch_size" : batch_size,
419     "batch_unique_labels":100/labels_per_batch,
420     "unlabels_per_batch":int((batch_size-labels_per_batch)/2),
421     "nbr_batch" : 100,
422     "lamd" : 1,
423     "tsa_schedule":"exp_schedule",
424     # tsa_schedule=""
425     "num_classes" : 10,
426     "uda_softmax_temp" : 0.9,
427     "uda_confidence_thresh" : 0.8,
428     "ent_min_coeff" : 0.5,
429 
430     "num_classes" : 10,
431     "decay_steps" : 1000,
432     "moving_average_decay":0.9999,
433     "initial_learning_rate":1e-2,
434     "dropout":0.8,
435     "l2": 0.1
436 }
437 create_train(hyper, epochs=20, pretraining=False, verbose=False)
```

Bibliographie

[1] N. K. Spyros Gidaris, Praveer Singh, "Unsupervised representation learning by pre-dicting image rotations," p. 16.

[2] M. K. Takeru Miyato, Shin-ichi Maeda and S. Ishii†, "Virtual adversarial training : A regularization method for supervised and semi-supervised learning," p. 16.

[3] E. H. M.-T. L. e. Q. V. L. Qizhe Xie1, Zihang Dai1, "Unsupervised data augmentation for consistency training," p. 20.