Haruka Miura
M2 DCISS 12408660
30/11/2025

S9 Perception Multimodale

TP noté

1, Reconnaissance de caractères

(modification et exécution deTP2_noté > minst > main.py)

Fonctionnement actuel de la fonction train
Cette fonction affiche la valeur de la perte (loss) pour chaque batch, mais elle ne calcule pas la perte totale pour une époque entière, c'est-à-dire la perte sur l'ensemble des données d'entraînement. Ainsi, même si la fonction s'exécute correctement, elle ne montre ni la perte totale par époque ni l'évolution globale de la perte d'apprentissage.

Modifications apportées à la fonction train
La perte de chaque batch est désormais cumulée afin de calculer la perte moyenne par époque. Cette perte moyenne peut ensuite être représentée graphiquement pour visualiser l'évolution de la perte d'entraînement.
Concrètement, une variable train_loss a été ajoutée pour accumuler les pertes :
train_loss += loss.item() * data.size(0)
Cette ligne ajoute au total la perte du batch entier. À la fin, l'instruction
train_loss /= len(train_loader.dataset)
divise la somme des pertes par le nombre total d'échantillons, ce qui permet d'obtenir la perte moyenne sur l'ensemble des données d'entraînement.

Mémo : Pourquoi calculer la moyenne ?
Dans la ligne
train_loss += loss.item() * data.size(0)
la valeur data.size(0) correspond au nombre d'exemples contenus dans le batch actuel. loss.item() représente la perte calculée pour ce batch. En multipliant cette valeur par la taille du batch, on obtient la perte totale pour ce groupe d'images.
Ensuite, diviser par le nombre total d'exemples permet d'obtenir la perte moyenne sur l'ensemble du jeu d'entraînement. Cette moyenne indique à quel point le modèle se trompe, en moyenne, sur toutes les données d'entraînement. C'est cette valeur qui sert à mettre à jour les paramètres (poids) du modèle via optimizer.step().

Fonctionnement actuel de la fonction test
Contrairement aux données d'entraînement, le traitement des données de test n'influencent pas l'ajustement des paramètres du modèle. Elles servent uniquement à évaluer la capacité de généralisation du modèle.
La fonction test calcule la perte totale (test_loss) sur toutes les données de test : elle additionne les pertes de chaque batch puis affiche la perte moyenne. Cela permet d'observer l'évolution de la perte de test à chaque époque. Ici, le cumul de la perte était déjà présent dans le code.

Modifications apportées à la fonction test
La fonction test renvoie maintenant test_loss et accuracy. Après l'affichage de ces valeurs, elles sont ajoutées à un tableau pour conserver leur évolution au fil des époques.

Modifications dans la fonction main
En tenant compte des changements des fonctions train et test, deux tableaux supplémentaires ont été ajoutées : train_losses et test_losses. Les pertes calculées par les deux fonctions sont ajoutées à ces tableaux à chaque époque au moment où la moyenne est calculée.
Ensuite, ces valeurs sont utilisées pour tracer les courbes de perte. Cela permet de visualiser l'évolution de

l’erreur du modèle, tant sur les données d’entraînement que sur les données de test, au fur et à mesure des époques.

Analyse

L’image 1 montre le résultat de l’exécution. La train loss (perte d’entraînement) diminue fortement entre la première et la deuxième époque, puis continue à diminuer de manière plus progressive, pour terminer autour de 0,03. La¥perte de test suit une tendance similaire : une forte baisse au début, puis une diminution plus douce, pour atteindre environ 0,026 à la fin.
La perte d’entraînement diminue régulièrement sans tendre vers zéro de manière excessive. La perte de test reste proche de la perte d’entraînement tout au long de l’apprentissage et ne montre pas de hausse en fin d’entraînement.
Comme l’écart entre les deux courbes reste faible et stable, et que la perte de test ne se met pas à augmenter, le modèle ne présente pas de signes de surapprentissage. Le dropout utilisé (0,25 et 0,5) désactive aléatoirement 25 % des sorties de la couche convolutionnelle et 50 % des sorties de la couche entièrement connectée. Cette technique consiste à éteindre de façon aléatoire certains neurones pendant l’apprentissage afin d’éviter que le modèle ne mémorise trop les données d’entraînement. Elle permet ainsi d’obtenir un modèle plus général, plus performant sur les données de test. Il est donc probable que le dropout ait joué un rôle positif dans la bonne généralisation observée.

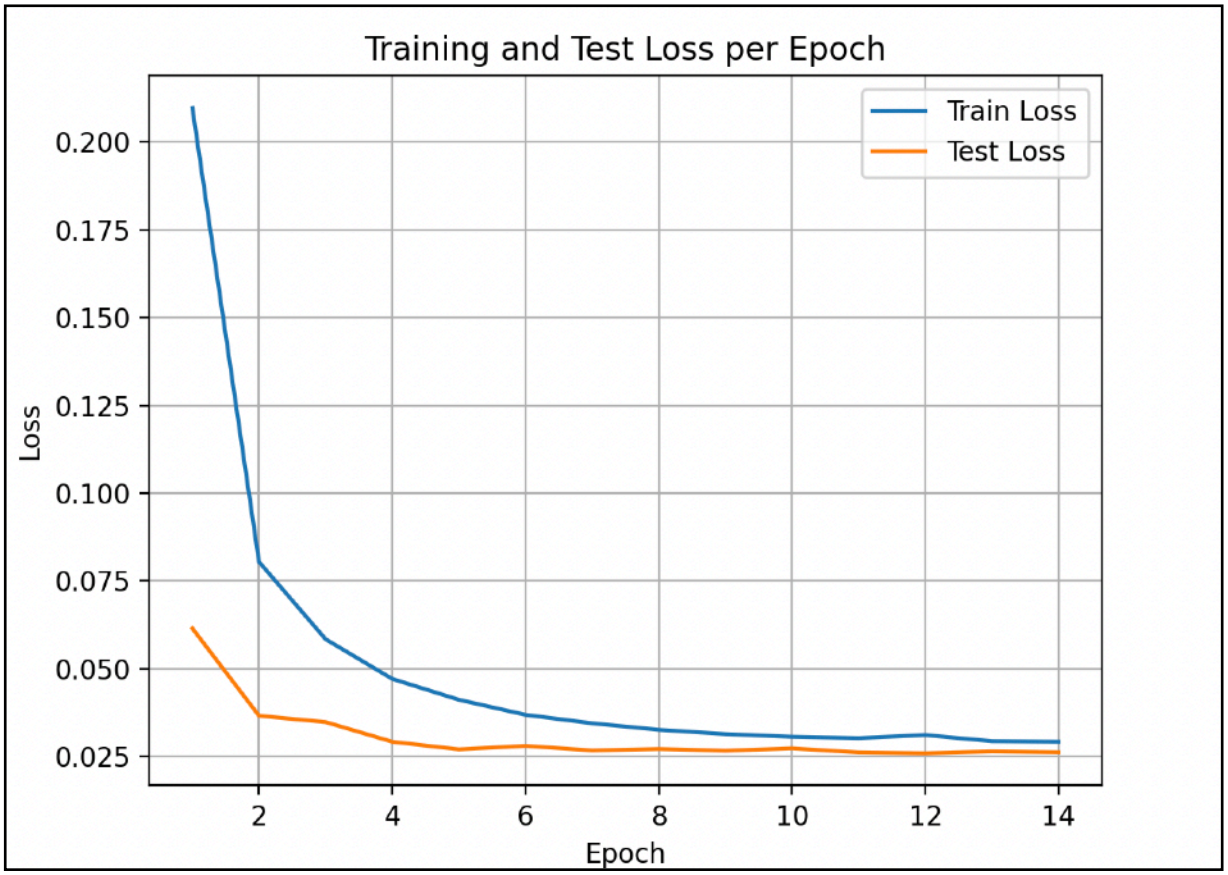

image1

2 , Reconnaissance vocale

2-1 Affichage de l'onde sonore et du spectrogramme Mel

(exécution de TP2_noté > audioMNIST > ShowAudioFile.py)

Dans l'image 2 et l'image 3, la partie supérieure de l'image montre l'onde sonore, c'est-à-dire la variation de l'amplitude du signal en fonction du temps. L'axe des y représente l'amplitude, tandis que l'axe des x correspond au temps.Lorsque l'on place le curseur sur la figure, une valeur du type x, y = (838, 0.404) apparaît. Le nombre 0.404 indique alors l'intensité du signal sonore ou l'une des valeurs normalisées de l'amplitude à ce moment précis.
La partie inférieure présente le spectrogramme Mel. Celui-ci correspond à une transformation du signal audio dans le domaine temps-fréquence. L'axe vertical représente les fréquences Mel, l'axe horizontal représente le temps, et l'intensité ou la couleur indique la puissance du signal (en échelle logarithmique, souvent en décibels).
Dans un spectrogramme Mel, les fréquences classiques (en Hz) sont converties en fréquences Mel. Cette représentation repose sur l'échelle perceptive du système auditif humain et est couramment utilisée en reconnaissance vocale. Elle permet de visualiser, au cours du temps, les composantes fréquentielles du signal, notamment celles auxquelles l'oreille humaine est la plus sensible. Grâce à cela, le modèle peut apprendre à reconnaître les sons à partir des représentations Mel.
Le spectrogramme Mel est très proche des caractéristiques MFCC (Mel-Frequency Cepstral Coefficients). Les MFCC sont des coefficients qui décrivent le signal audio en tenant compte des propriétés perceptives de l'audition humaine : ils extraient principalement les informations fréquentielles utiles pour la compréhension. Cette méthode consiste à appliquer un filtre Mel, à prendre le logarithme du spectre, puis à effectuer une transformation en cosinus discrète (DCT) pour compresser et représenter le signal sous forme d'un vecteur de caractéristiques.
Dans notre cas, l'utilisation de l'échelle Mel met particulièrement en valeur les fréquences sensibles à l'oreille humaine (environ entre 300 Hz et 3 kHz).

En examinant directement le spectrogramme obtenu, on peut noter que :
– la partie supérieure de l'axe des Y (faibles fréquences Mel) doit correspondre aux composantes basses du signal, souvent liées aux voyelles ou aux vibrations des cordes vocales ;
– la partie inférieure (hautes fréquences Mel) doit correspondre aux composantes plus aiguës du signal, comme les consonnes explosives ou les sons de friction.

Résultat pour la prononciation de "zero[/zɪə.roʊ/]"
Données : 0_george_34.wav
L'image 2 montre le résultat de l'exécution. Dans les basses fréquences (0 à 1 kHz), on observe une amplitude particulièrement forte dans les segments vocaliques (/ɛ/ et /oʊ/), ce qui apparaît sous la forme de zones plus sombres sur le spectrogramme.
Dans les hautes fréquences (au-delà de 3 kHz), l'influence de la consonne fricative /z/ se manifeste par des zones légèrement plus claires, correspondant à une amplitude plus faible.

Résultat pour la prononciation de "five[/faɪv/]"
Données : 5_jackson_5.wav
L'image 3 montre le résultat de l'exécution. Dans les basses fréquences (0 à 1 kHz), les segments vocaliques /aɪ/ présentent une amplitude élevée, ce qui se traduit par des zones sombres sur le spectrogramme.
Dans les hautes fréquences (au-delà de 3 kHz), les consonnes fricatives /f/ et /v/ apparaissent sous forme de composantes fréquentielles plus aiguës. La couleur devient plus claire, mais l'énergie est concentrée dans les hautes fréquences.

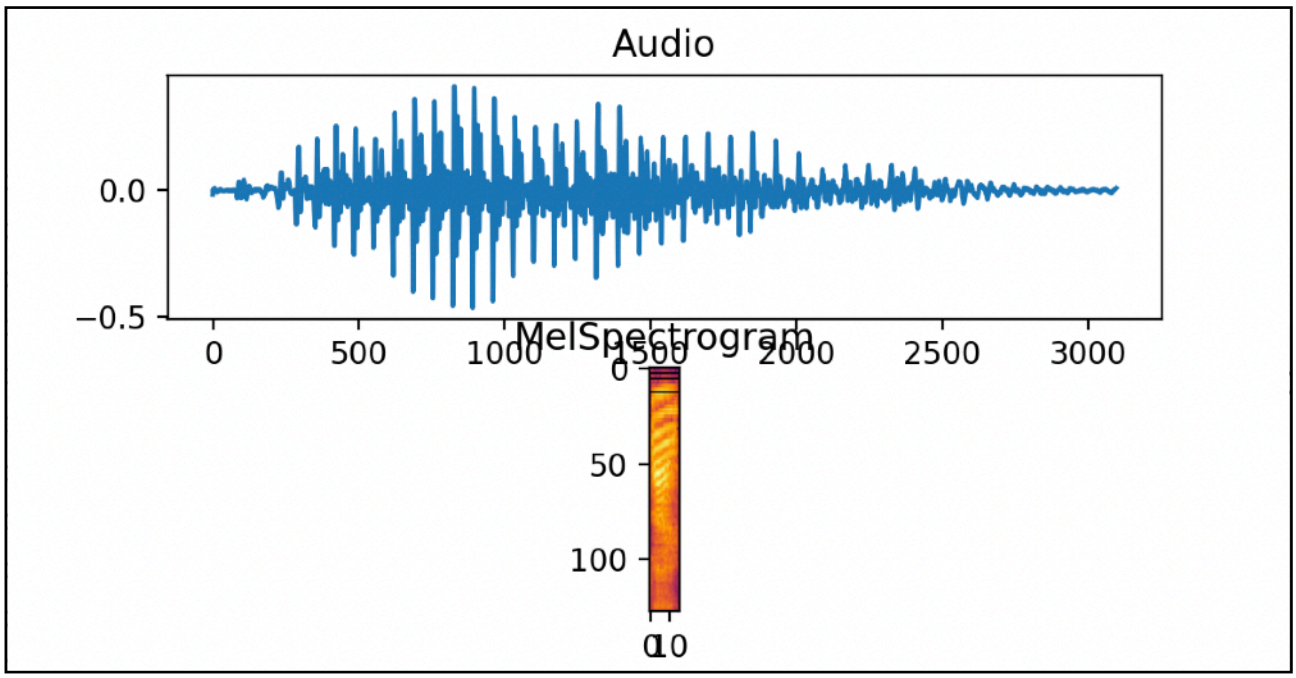

image2

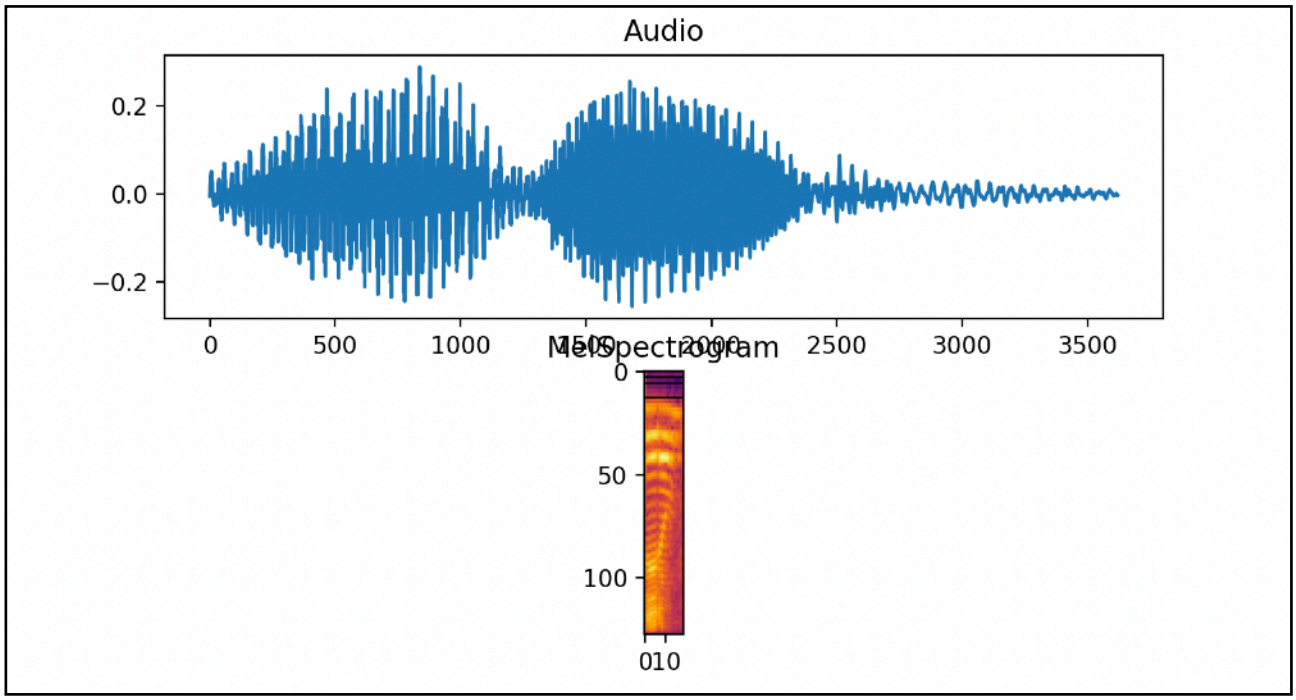

image3

2-2 Exécution de l'apprentissage avec *MainAudioMNIST.py*

(Exécution de MainAudioMNIST.py à l'aide des données contenues dans TP2_noté > audioMNIST > original)

Ce programme réalise l'ensemble du processus consistant à convertir les données audio en spectrogrammes Mel, puis à les utiliser pour entraîner et évaluer le modèle.
En exécutant la commande *python MainAudioMNIST.py*, on utilise un total de 3000 enregistrements.

Dans la version “originale” du découpage des données, ces 3000 échantillons sont répartis selon un ratio *train:test = 3:1*, ce qui correspond à 2250 données d’entraînement et 750 données de test. Les fichiers proviennent de 6 locuteurs différents, mélangés aussi bien dans le train que dans le test.

L’image 4 montre le résultat de l’exécution. selon le résultats à la 30e époque,
Loss : entre 0.03 et 0.04 à la fin de l’apprentissage
Accuracy : entre 0.92 et 0.97

Interprétation de l’apprentissage du modèle
La loss diminue jusqu’à une valeur très faible (0.03–0.04), ce qui indique que l’erreur entre les prédictions du modèle et les étiquettes de vérité est devenue très petite.
L’accuracy augmente jusqu’à atteindre 0.92–0.97, montrant que le modèle obtient de très bonnes performances sur les données de test.
Le modèle a appris efficacement les caractéristiques du jeu de données. Les résultats montrent qu’en utilisant des représentations en spectrogramme Mel, il est possible de reconnaître correctement les chiffres prononcés.
La transformation de l’audio à Mel-spectrogramme fournit donc des caractéristiques pertinentes pour la classification.

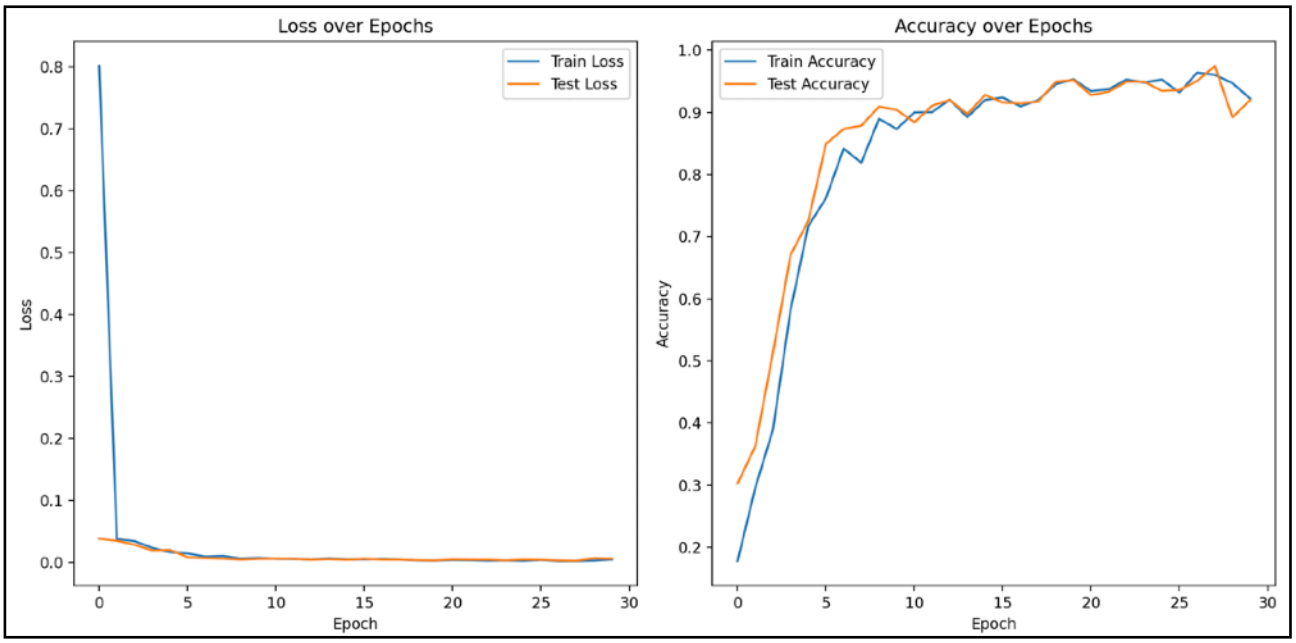

image4

2-3 Optimisation

Le modèle utilise l’optimiseur suivant :
model_optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
Ici, la méthode de mise à jour des paramètres est SGD (Stochastic Gradient Descent), ou descente de gradient stochastique. Cette méthode consiste à mettre à jour les paramètres à partir du gradient calculé sur des batchs tirés aléatoirement. Cette approche permet de réduire le coût de calcul et d’accélérer l’apprentissage.
Learning rate (lr = 0.001)
Ici, le learning rate détermine la taille du pas pour la mise à jour des paramètres.
– S’il est trop grand, le modèle risque d’osciller et de ne jamais converger.
– S’il est trop petit, la convergence devient très lente et l’entraînement prend beaucoup de temps.
Ici, la valeur 0.001 permet un apprentissage stable.

Momentum(momentum=0.9)
Le paramètre momentum introduit une inertie dans la mise à jour des poids : 90 % du gradient précédent est ajouté au gradient actuel. Cela permet. de lisser les mises à jour et d'accélérer la convergence. En résumé, chacun de ces paramètres contribue à mettre à jour les poids du modèle sur la base des gradients calculés à chaque étape de rétropropagation.

2-4 Exécution de l'apprentissage avec MainAudioMNIST.py

(Exécution du script en utilisant les données contenues dans TP2_noté > audioMNIST > not_original)

Dans cette configuration, la répartition des données est différente de celle de "original".
Un locuteur parmi les six est entièrement réservé au test : 10 chiffres × 50 enregistrements = 500 fichiers uniquement pour le test.
Les cinq autres locuteurs fournissent les 2500 fichiers du train.
En exécutant python SplitTrainTest.py, la console affiche :

```
Lecture des fichiers dans recordings/...
Total de fichiers trouvés: 3000
Nombre de locuteurs: 6

=== Séparation des locuteurs ===
Locuteurs pour le train (5): ['george', 'jackson', 'lucas', 'nicolas', 'yweweler']
Locuteurs pour le test (1): ['theo']

Nombre de fichiers dans train: 2500
Nombre de fichiers dans test: 500
Ratio réel train/test: 5.00
```

Résultats
L'image 5 montre le résultat de l'exécution.
Loss de fin d'époque : 0.67–0.87, nettement plus élevé que dans original
Accuracy : 0.56–0.71, également plus faible que dans original

Comparaison entre original et not_original

On observe une différence notable entre les deux expériences original et not_original, tant au niveau de la perte que de la précision. Dans la version original, les ensembles de train et de test contiennent des données provenant des mêmes locuteurs. Le modèle peut donc apprendre non seulement les caractéristiques acoustiques liées aux chiffres, mais aussi les particularités vocales des locuteurs (timbre, rythme, articulation). Comme les voix présentes dans le test ont déjà été rencontrées dans le train, le modèle bénéficie d'une cohérence inter-locuteurs et parvient à de très bonnes performances.
En revanche, dans la version not_original, les locuteurs du train et ceux du test sont totalement séparés. Le modèle ne rencontre jamais la voix du locuteur utilisé pour le test.
Ainsi, même si la perte d'entraînement diminue fortement et approche de zéro, la performance en test reste limitée.
Cela s'explique par le fait que le modèle apprend des caractéristiques fortement dépendantes des locuteurs du train. Lorsqu'il est confronté à un locuteur totalement nouveau (voix différente, prononciation différente), ses représentations apprises ne s'adaptent pas suffisamment, ce qui entraîne une chute de la précision. En d'autres termes, le modèle sur-apprend les caractéristiques spécifiques aux locuteurs du train, il généralise mal à une voix inconnue, et les variations naturelles de la parole deviennent un défi majeur pour la reconnaissance automatique.
Cela illustre un phénomène classique appelé variabilité de la parole , montrant que la reconnaissance vocale nécessite des modèles capables de s'adapter à une grande diversité de voix.

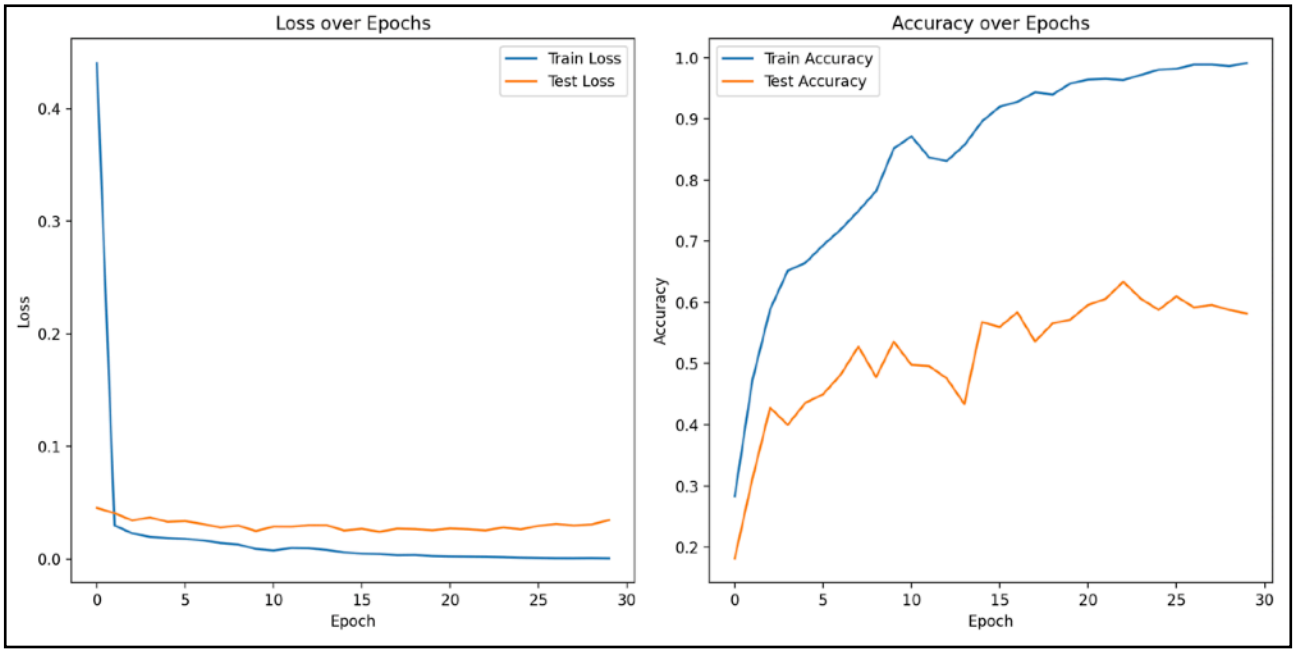

image5

3, Perception Multimodale

3-1 Early Fusion

(Exécution de TP2_noté > audioMNIST > early.py)

Dans cette approche, l'intégration des modalités (image + audio) est réalisée dès le début du modèle, c'est-à-dire immédiatement après l'entrée. Lors de la première convolution effectuée par le CNN, les deux types d'informations sont déjà fusionnés dans un seul tenseur.
Concrètement, dans le code, la fonction __getitem__ constitue le point de fusion.
C'est là que le DataLoader dédié à l'early fusion est construit ici.

image_stucked = torch.cat((image, spectrogram_resized), dim=0)
return image_stucked, label

L'image et le spectrogramme sont ainsi concaténés en un objet 2D à deux canaux, transmis ensuite au CNN :

nn.Conv2d(in_channels=2, out_channels=10, …)

Dès cette première couche, les filtres du CNN sont donc entraînés sur un signal fusionné. Les caractéristiques apprises à ce stade reflètent des sorties issues simultanément de l'image et du signal audio.
Dans les couches suivantes du CNN :

nn.Flatten()
nn.Linear(…)

les représentations utilisées proviennent entièrement de cette feature map fusionnée (image × audio). À ce niveau, les deux modalités ne sont plus distinguées : le classifieur exploite un vecteur de caractéristiques complètement intégré.

Calcul de la perte
Lors de la rétropropagation, l'erreur se propage jusqu'aux filtres du CNN, qui sont ajustés pour renforcer ou

atténuer les motifs pertinents dans l'entrée combinée des deux modalités. Avec optimizer.step(), la mise à jour des poids reflète cette co-apprentissage : le modèle détermine automatiquement la contribution relative de chaque modalité.

Analyse
L'image 6 montre le résultat de l'exécution. Il est probable que le modèle apprenne très rapidement, notamment durant les trois premiers epochs. Plusieurs facteurs peuvent expliquer ce phénomène :
les deux modalités sont fusionnées dès l'entrée et peuvent se compléter mutuellement ;
la corrélation entre les deux modalités est élevée, ce qui facilite des prédictions précises dès le début de l'entraînement.
Au fil des epochs, la perte continue de diminuer sans signe de surapprentissage manifeste : le modèle présente un apprentissage stable sur les données d'entraînement.
Le fait d'atteindre plus de 99 % de précision montre que le modele possède une excellente capacité d'adaptation aux données du train.
Cependant, les résultats obtenus sur les données de test montrent une perte plus élevée et une précision légèrement inférieure à celles observées sur les données d'entraînement. Cela suggère que, malgré la forte performance en apprentissage, la généralisation reste plus difficile lorsque le modèle est confronté à des exemples nouveaux.

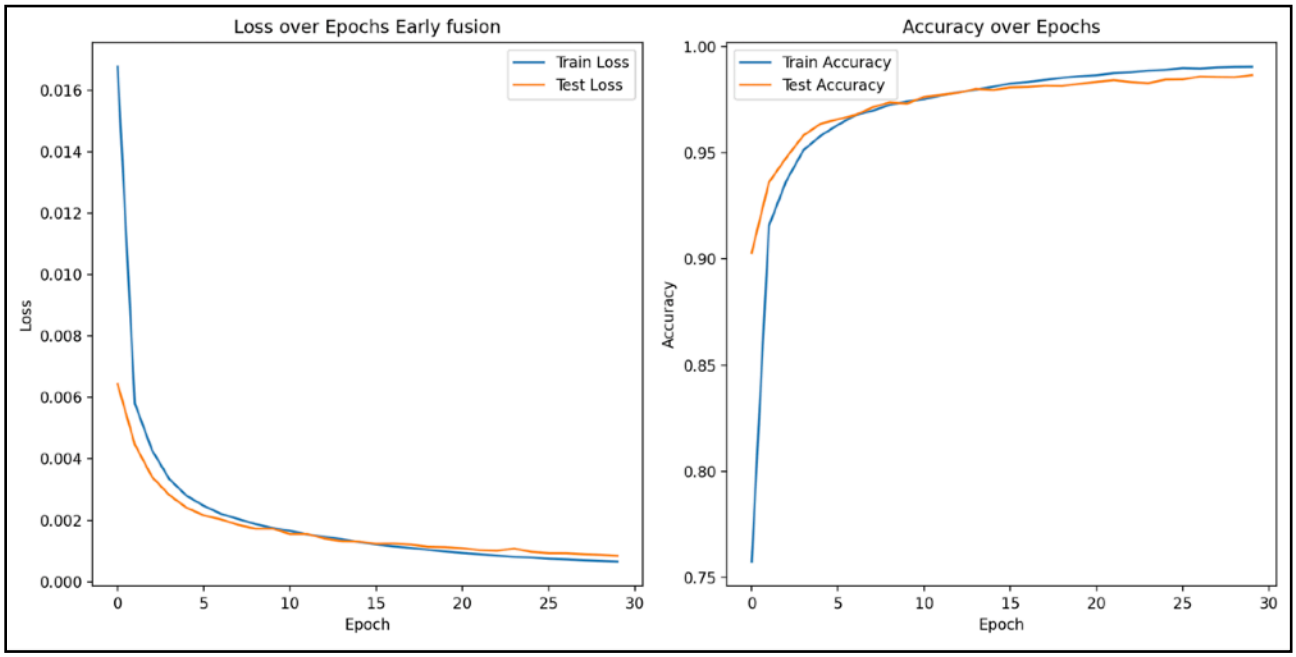

image6

3-2. Late Fusion

(Exécution de TP2_noté > audioMNIST > late.py)

Dans le cas de late fusion, l'image et le son sont traités séparément par deux CNN, puis leurs vecteurs de caractéristiques sont combinés à un stade avancé du réseau. Ce ne sont donc pas les données brutes qui sont fusionnées, mais les caractéristiques extraites par les CNN.
Pour ImageCNN, l'entrée est une image MNIST (1×28×28) et la sortie est un vecteur de caractéristiques de 10 dimensions, représentant les informations propres au domaine visuel. Pour AudioCNN, l'entrée est un spectrogramme Mel (1×H×W) et la sortie est également un vecteur de 10 dimensions, qui encode les motifs fréquentiels et temporels caractéristiques du signal audio.
Dans la méthode forward, la fusion entre les chiffrevisuelles et sonores s'effectue au niveau suivant dans LateFusionCNN.forward() :

img_feat = self.image_cnn(image)

aud_feat = self.audio_cnn(audio)
fusion = torch.cat((img_feat, aud_feat), dim=1)

Ainsi, contrairement à la early fusion, où l'intégration des modalités se fait immédiatement après l'entrée, late fusion s'effectue après l'abstraction produite par les CNN. Les caractéristiques combinées sont donc des représentations de haut niveau : 10 dimensions pour l'image + 10 dimensions pour l'audio = 20 dimensions d'information. Le réseau final est le suivant :

nn.Linear(20, 64) → ReLU → nn.Linear(64, 10)

Ce dernier module produit la prédiction sur les 10 classes possibles, déterminant ainsi le chiffre final reconnu. Ces prédictions sont ensuite comparées à l'étiquette réelle à chaque batch pour calculer la précision et la perte.

Analyse

L'image 7 montre le résultat de l'exécution. Au premier epoch, la perte de l'entraînement est élevée (environ 0,048 au début) et la précision faible (autour de 0,135). Cela indique que le modèle n'a pas encore appris à extraire correctement les caractéristiques pertinentes des données.La faible précision initiale peut s'expliquer par le fait que, dans late fusion, les deux modalités sont d'abord traitées indépendamment. Au début, les CNN ne maîtrisent pas encore les caractéristiques propres à chaque modalité. Le modèle LateFusionCNN n'a donc pas encore appris la relation entre les deux sources d'information ni trouvé une manière optimale de les combiner.
À mi-parcours, au fur et à mesure des epochs, le modèle apprend progressivement comment intégrer efficacement les caractéristiques visuelles et sonores, ce qui améliore les performances. Il doit commencer également à montrer une certaine capacité de généralisation sur les données inconnues. En fin d'entraînement, l'amélioration devient plus lente. Le modèle atteint déjà de bonnes performances et la marge d'optimisation restante se réduit. Finalement, la perte tombe en dessous de 0,01 et la précision atteint environ 0,898.
Le fait que les valeurs du test suivent une tendance similaire à celles de l'entraînement suggère que ce comportement n'est pas dû au surapprentissage. Dans une approche où chaque modalité est d'abord apprise séparément puis fusionnée, il est normal que la convergence prenne davantage de temps, et le modèle doit trouver un équilibre adéquat entre les deux modalités avant d'obtenir des performances optimales.

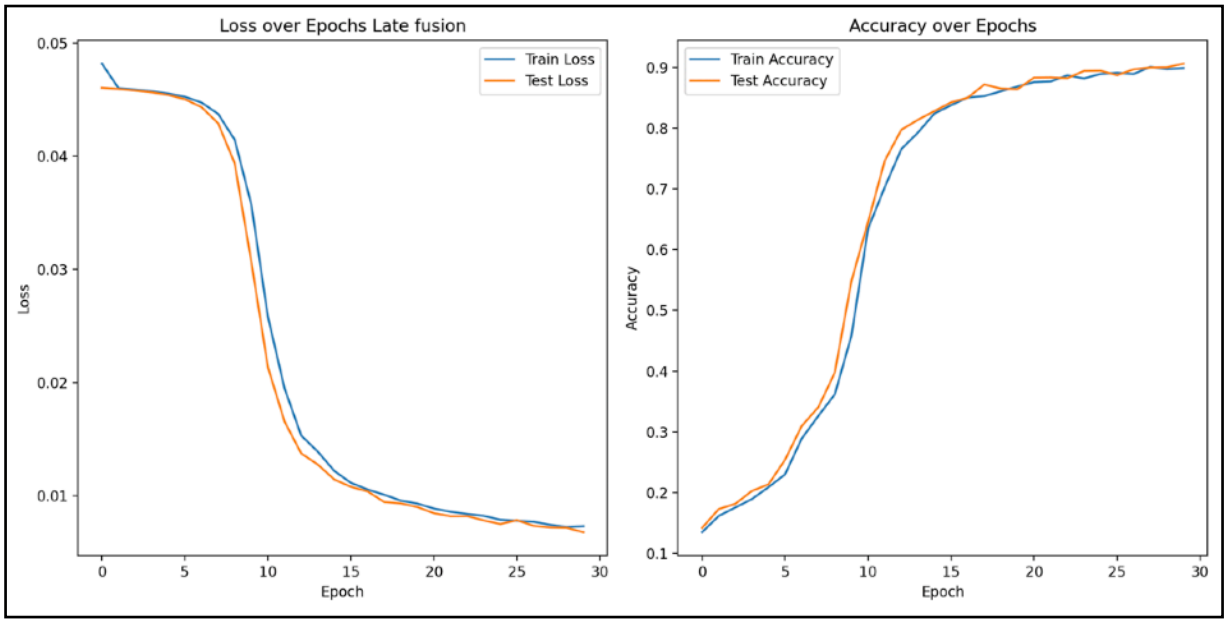

image7

4. Synthèse

Dans cette évaluation, nous avons d'abord observé l'évolution de la perte sur les données de test à chaque epoch, tout en ajustant les paramètres du modèle grâce à la rétropropagation effectuée sur la perte calculée lors de l'entraînement. Pour chaque batch, la perte obtenue est multipliée par la taille du batch (le nombre d'échantillons), ce qui permet d'obtenir la perte totale pour ce batch. En divisant cette somme par le nombre total d'exemples, on obtient la perte moyenne, qui indique à quel point le modèle commet des erreurs sur les données d'apprentissage. Cette valeur moyenne a ensuite été utilisée pour les courbes tracées à chaque epoch.
Selon la composition des données d'entraînement et de test, des résultats différents peuvent être observés même avec le même modèle. Lorsque le modèle dépend trop des caractéristiques propres aux locuteurs présents dans les données d'entraînement, il ne parvient pas à généraliser correctement aux locuteurs différents contenus dans les données de test. Cela se produit lorsque le modèle apprend de manière excessive certains motifs spécifiques qui ne sont pas applicables au test set. À l'inverse, lorsque les locuteurs diffèrent entre l'entraînement et le test, le modèle est contraint d'apprendre des caractéristiques plus générales du signal vocal. Cela peut améliorer les performances à long terme, mais conduit souvent à une précision moins correcte dans un premier temps.
L'apprentissage multimodal peut suivre plusieurs approches, selon le moment où les modalités sont intégrées et la manière dont cette fusion influence le processus d'apprentissage. Le choix de la méthode devrait donc dépendre des caractéristiques de chaque modalité et de la nature de la tâche. Dans notre cas, early fusion a finalement donné les meilleurs résultats, atteignant une précision de 99 %. Il est probable que ce haut niveau de performance provienne du fait que le modèle apprend simultanément les « motifs temporels » du signal audio et les « motifs spatiaux » de l'image. Comme les motifs temporels du signal audio et les motifs spatiaux de l'image sont déjà combinés dans l'entrée, cette fusion précoce a permis au modèle d'apprendre directement sur les caractéristiques intégrées, facilitant ainsi l'extraction de représentations pertinentes pour la tâche.