Université de Montréal
MILA

Rapport Devoir 3

Alex Maggioni (20266243)

Aaron Courville

May 1, 2024

1 SimCLR

Sélection des Transformations pour l'Augmentation des Données

Le choix des augmentations de données a été fortement inspiré par les détails fournis dans la section *A. Data Augmentation Details* de l'annexe du papier de référence. Il est crucial de reconnaître que la diversité et la composition des augmentations sont essentielles pour apprendre des représentations de qualité. Selon le papier — *We observe that no single transformation suffices to learn good representations [...] When composing augmentations, the contrastive prediction task becomes harder, but the quality of representation improves dramatically.*

Une composition particulièrement efficace mise en avant est celle associant le recadrage aléatoire (random cropping) et la distorsion des couleurs (random color distortion). Entre deux découpes aléatoires d'une même image, il est souvent observé que les deux segments partagent un histogramme de couleurs extrêmement similaire, permettant ainsi au modèle de *tricher* en apprenant ses représentations. Pour contrer cela, il est impératif de composer le recadrage avec une distorsion des couleurs. Selon le papier — *Therefore, it is critical to compose cropping with color distortion in order to learn generalizable features [...] Stronger color augmentation substantially improves the linear evaluation of the learned unsupervised model.*

Les transformations spécifiques que j'ai utilisées comprennent le recadrage aléatoire redimensionné, le renversement horizontal, l'application aléatoire de *Color Jitter* et de *Gaussian Blur*, ainsi que la conversion en niveaux de gris. Ces méthodes ont été directement inspirées de l'annexe A du papier, garantissant ainsi que notre approche reste fidèle aux méthodes éprouvées pour obtenir des représentations robustes et généralisables.

En plus de ces transformations, une normalisation des données est appliquée en utilisant les moyennes et écarts-types calculés à partir du jeu d'entraînement. Cette étape est cruciale pour assurer que les données d'entrée sont bien centrées et normalisées, rendant ainsi le modèle plus stable et performant durant l'apprentissage.

Voici le code que j'utilise pour configurer mes transformations :

```
data_transforms = transforms.Compose([
    transforms.RandomResizedCrop(size=(size, size)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomApply([transforms.ColorJitter(0.8, 0.8, 0.8, 0.2)], p=0.8),
    transforms.RandomGrayscale(p=0.2),
    transforms.RandomApply([transforms.GaussianBlur(11, sigma=(0.1, 2.0))], p=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.44058355689048767, 0.42731037735939026, 0.38579756021499634],
                         std=[0.2686561346054077, 0.2612513303756714, 0.2684949040412903])
])
```

Perte SimCLR

La fonction de perte SimCLR est fondamentale pour l'apprentissage des représentations par contraste dans notre modèle. Selon le papier, la perte est définie comme suit :

Calculer les similarités par paires $s_{i,j}$ pour chaque $i, j \in \{1, \ldots, 2N\}$, où

$$s_{i,j} = \frac{z_i^\top z_j}{\|z_i\| \|z_j\|}.$$

Définir la fonction de perte $\ell(i,j)$ pour une paire (i,j) comme suit :

$$\ell(i,j) = -\log\left(\frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq j]} \exp(s_{i,k}/\tau)}\right).$$

La perte totale $\mathcal{L}$ est calculée en moyennant les pertes sur toutes les paires positives :

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

Mettre à jour les réseaux f et g pour minimiser $\mathcal{L}$.

Mon implémentation de cette perte est structurée comme décrit ci-dessous :

```
def loss(self, features):
    # Les caractéristiques d'entrée sont un tenseur de torch avec une forme de (2*batch_size, out_dim)
    # Les paires positives sont (features[i] et features[i+batch_size]) pour tout i

    # Étape 1
    batch_indices = torch.arange(self.args.batch_size).repeat(self.args.n_views)
    labels = torch.eq(batch_indices.unsqueeze(0), batch_indices.unsqueeze(1)).float().to(self.args.device)

    # Étape 2
    normalized_features = F.normalize(features, dim=1)
    similarity = torch.mm(normalized_features, normalized_features.T)

    # Étape 3
    diagonal_mask = torch.eye(labels.size(0), dtype=torch.bool).to(self.args.device)
    labels_filtered = labels.masked_select(~diagonal_mask).view(labels.size(0), -1)
    similarity_filtered = similarity.masked_select(~diagonal_mask).view(similarity.size(0), -1)

    # Étape 4
    positive_pairs = similarity_filtered[labels_filtered.bool()].view(labels.size(0), -1)
    negative_pairs = similarity_filtered[~labels_filtered.bool()].view(labels.size(0), -1)

    # Étape 5
    logits = torch.cat([positive_pairs, negative_pairs], dim=1) / self.args.temperature
    zero_labels = torch.zeros(logits.size(0), dtype=torch.long).to(self.args.device)

    return self.criterion(logits, zero_labels)
```

Étape 1 : Les `batch_indices` créent un vecteur d'indices pour chaque élément dans le batch, et `labels` forme une matrice indiquant les paires positives et négatives. Les éléments de la diagonale, bien que tous à 1 initialement, seront éliminés dans l'étape suivante pour éviter la comparaison de chaque image avec elle-même. Ici l'utilisation de `torch.arange` et `repeat` pour créer `batch_indices`, et leur mise en forme ultérieure par broadcasting pour générer la matrice `labels`, réduit le besoin de boucles explicites pour identifier les paires positives et négatives.

Étape 2 : Les caractéristiques sont normalisées pour que chaque vecteur ait une norme unitaire, permettant ainsi le calcul des similarités cosinus. Cette normalisation, combinée à l'opération de multiplication de matrices (`torch.mm`) entre les vecteurs normalisés, permet de calculer efficacement toutes les similarités en une seule opération matricielle hautement optimisée.

Étape 3 : Pour respecter la contrainte que chaque exemple ne doit pas être comparé avec lui-même, nous utilisons un masque diagonal pour filtrer les éléments indésirables de la matrice de similarité. L'utilisation d'un masque diagonal (`torch.eye`) optimise l'utilisation de la mémoire et accélère les calculs en évitant les opérations redondantes. Cette opération est rendue possible par l'utilisation de `masked_select`, qui sélectionne uniquement les éléments pertinents de la matrice, évitant ainsi des calculs inutiles.

Étape 4 : Utilisation de masques booléens pour séparer les similarités en paires positives et négatives. Cette opération est directement dérivée des labels filtrés, permettant un traitement vectorisé.

Étape 5 : Concaténation et normalisation par la température des logits des paires positives et négatives pour leur passage à la fonction de perte d'entropie croisée, avec les étiquettes à zéro indiquant que la vraie classe pour chaque exemple est la paire positive. Le recours à `torch.cat` pour concaténer les logits et à `torch.zeros` pour préparer les étiquettes pour la perte est particulièrement efficace pour préparer les données pour les opérations finales sans recalculs inutiles.

En conséquence, l'implémentation respecte fidèlement la formulation de la perte décrite dans le papier original. L'adoption de techniques de vectorisation a considérablement optimisé le processus d'entraînement, permettant au modèle de compléter 70 époques en seulement 6 heures sur un GPU de type L4.

Résultats

Les résultats obtenus à partir de l'entraînement de notre modèle SimCLR sont illustrés ci-dessous, à travers deux figures clés qui résument la courbe de perte pendant l'entraînement et l'exactitude top-k lors de l'évaluation avec une sonde linéaire.

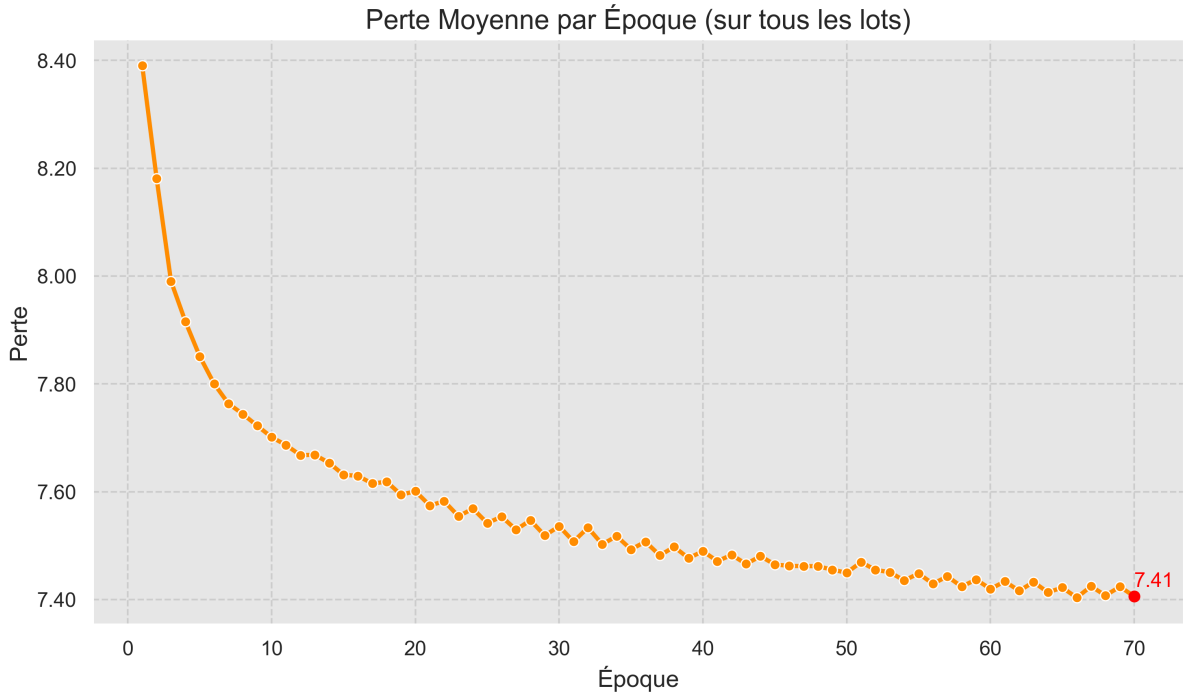


Figure 1: Évolution de la perte moyenne par époque. La courbe montre une convergence stable, avec une perte finale moyenne de 7.41 après 70 époques.

La Figure 1 montre la courbe de perte moyenne par époque pour toutes les batchs, affichant une réduction significative et stable de la perte au fil des époques. Cette courbe démontre l'efficacité de notre approche de formation, caractérisée par un `batch_size` de 3072, un taux d'apprentissage de 0.0003, une décroissance de poids de 10^{-4}, utilisant l'optimiseur AdamW et un ajustement du taux d'apprentissage par CosineAnnealingLR. Les performances atteintes confirment les attentes établies par les discussions sur Piazza et les benchmarks théoriques.

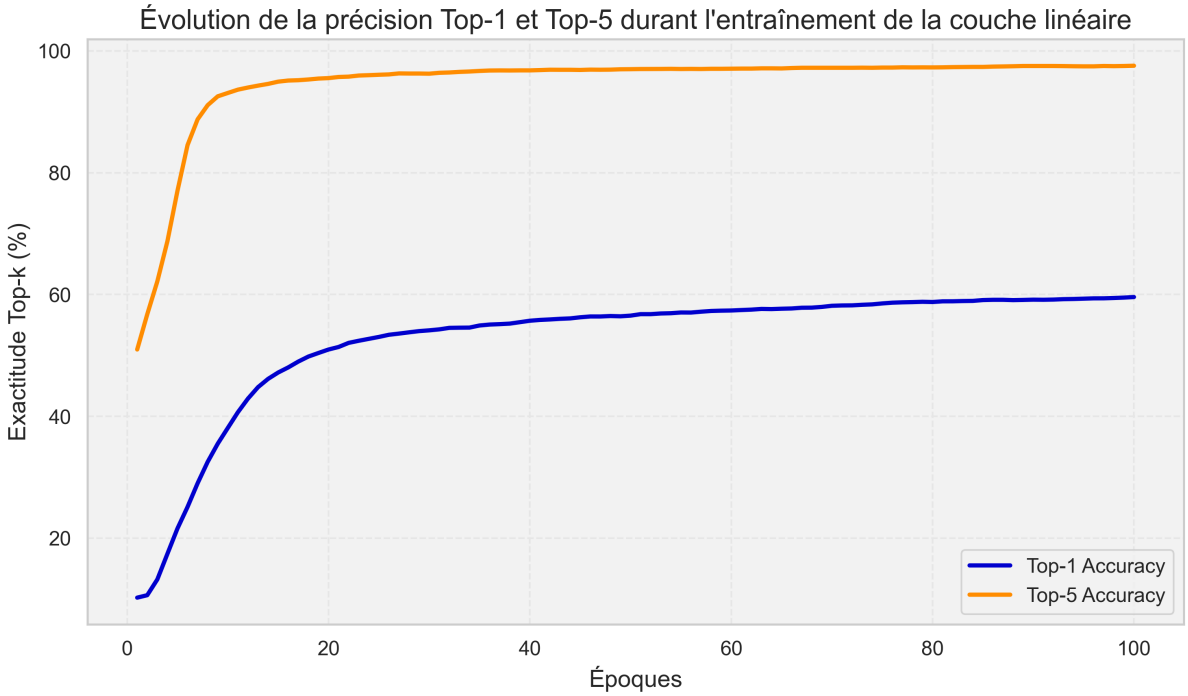


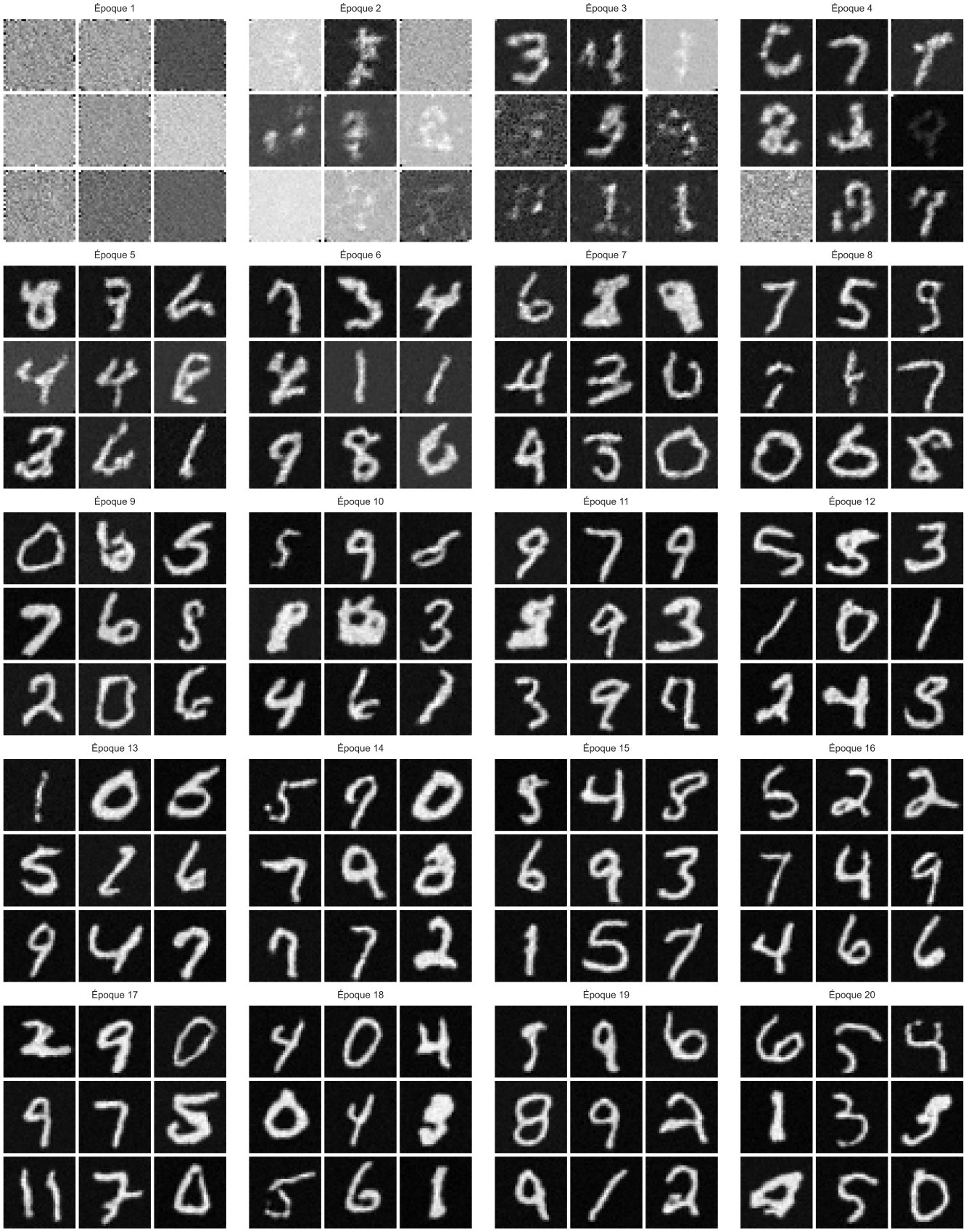
Figure 2: Évolution de la précision Top-1 et Top-5 lors de l'entraînement de la sonde linéaire sur les représentations obtenues par SimCLR.

Après l'entraînement de SimCLR, une sonde linéaire a été entraînée sur les représentations apprises pour évaluer la qualité des caractéristiques extraites. Comme montré dans la Figure 2, nous observons une augmentation rapide de la précision Top-1 et Top-5 dans les premières époques, suivi d'une stabilisation. Cette progression indique une extraction efficace de caractéristiques utiles pour la classification.

En termes de performances finales sur le jeu de données de test, nous avons atteint une précision Top-1 de 57.09% et une précision Top-5 de 96.75%. Ces résultats soulignent l'efficacité de l'apprentissage non supervisé par SimCLR pour extraire des caractéristiques qui sont significatives pour des tâches de classification downstream, tout en démontrant une excellente gestion des ressources computationnelles et une convergence rapide lors de l'entraînement. Encore une fois, les performances atteintes confirment les attentes établies par les discussions sur Piazza et les benchmarks théoriques.

2 DPPM

Voici l'ensemble des échantillons générés au cours de chacune des 20 époques, illustrés ci-dessous :



Cette figure montre une amélioration progressive et notable de la qualité des générations au fil des époques. Comme anticipé, les premières images générées ressemblent principalement à du bruit. Cependant, dès l'époque 9, nous observons une nette amélioration, les échantillons devenant visuellement acceptables. Cette tendance à l'amélioration se poursuit au cours des époques suivantes. L'échantillon de l'époque 19 a une qualité particulièrement élevée.