Meta-learning Symmetries by Reparameterization for Rotated MNIST

Soutenance projet de DeepLearning

Élèves :
Julien Bertand
Baptiste Coëffier

Enseignants :
Massinissa Hamidi
Dominique Fourer
Farida Zehraoui

15 décembre 2024

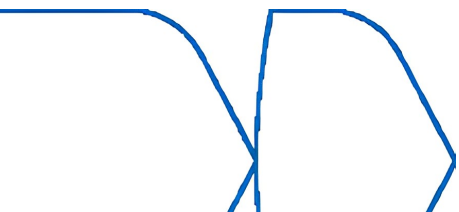

Table des matières

1 Introduction

Ce rapport présente le processus complet d'adaptation d'un code présentant une architecture de méta-apprentissage sur un nouveau jeu de données. Dans le cadre de la matière "Deep Learning", nous avons choisi un des articles proposés par nos enseignants et l'adapter au jeu de données associé. L'article que nous avons choisi est 'Meta-learning symmetries by reparameterization'[1] avec pour jeu de données associé 'Rotated MNIST'.

Nous allons commencer par faire une présentation de l'article. Ensuite, dans un second temps, nous détaillerons les différentes étapes de réalisation du projet. Puis, nous présenterons les résultats de notre travail. Et pour finir, nous entamerons une conclusion. Maintenant nous allons passer à la présentation de l'article.

1.1 Présentation de l'article

L'article 'Meta-Learning Symmetries by Reparameterization' présente une nouvelle méthode pour l'apprentissage automatique des symétries dans les réseaux neuronaux tout en évitant la conception manuelle d'architectures spécifiques. Les symétries, tout comme les translations ou les rotations, jouent un rôle important dans la généralisation et elles permettent de réduire le nombre de paramètres à fournir aux modèles. Pour donner un exemple, les réseaux neuronaux convolutifs ou C.N.N. garantissent une équivariance aux translations, essentielle dans le traitement des images. On peut donc vouloir étendre ces propriétés à d'autres modèles, cependant l'intégration de symétries spécifiques est souvent encore plus complexe.

Les chercheurs, dans l'article, proposent une nouvelle méthode qui se base sur une technique de reparamétrisation des couches du réseau de neurones. Dans ce réseau, les poids sont structurés dans une matrice de "symétrie". Cette matrice permet d'appliquer des transformations spécifiques qui reflètent les symétries présentes dans les données. Cette approche permet au réseau de neurones de s'adapter aux transformations que les données pourraient avoir comme les rotations, réflexions ou permutations.

Cette approche se passe dans un cas de méta-apprentissage qui fonctionne à deux échelles. D'une part, le réseau apprend les motifs de symétrie communs à un ensemble de données et d'autre part, le réseau ajuste ses paramètres. Cette optimisation à double facteur permet de mieux capturer les symétries qui sont sous-jacentes à plusieurs tâches.

Au moment de présenter leurs résultats expérimentaux, les chercheurs montrent que cette méthode permet de récréer des structures existantes comme les réseaux de neurones convolutifs ou bien de créer de nouveaux modèles adaptés à de nouveaux concepts spécifiques. Lors des tests sur des jeux de données générées synthétiquement et sur des problèmes de classification, les chercheurs ont démontré que leur méthode a permis une meilleure généralisation et notamment sur des problèmes avec des symétries complexes.

Après avoir vu plus en détail la méthode utilisée dans l'article, nous allons passer à l'implémentation de la méthode sur un nouveau jeu de données.

2 Réalisation du projet

Dans cette partie, nous allons nous pencher sur tout le processus d'implémentation du code, de la récupération du code des chercheurs à la visualisation des résultats. Nous allons détailler comment nous avons adapté le modèle au jeu de données Rotated MNIST puis nous verrons comment nous avons implémenté les différentes couches du réseau de neurones. Ensuite, nous irons voir comment nous avons implémenté le modèle pour avoir un circuit dédié par angle de rotation. Après cela, nous expliquerons comment nous avons fait pour visualiser et capturer les facteurs d'angle. Pour finir, nous ferons de même pour le mécanisme de "weight sharing".

2.1 Adaptation Rotated MNIST

La première étape dans l'implémentation a été de prendre en main les données. Le code qui nous a permis de générer le dataset Rotated MNIST, nous a été fourni par notre professeur. Ce code fourni utilise le dataset MNIST et applique une rotation sur les images de MNIST puis les sépare dans un train_loader et un test_loader, nous n'avons eu qu'a appelé les fonctions correspondante pour générer les données à utiliser.

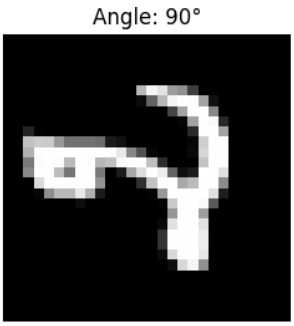

FIGURE 1 – Un exemple Rotated MNIST : chiffre 2 tourné à 90

Afin d'adapter le modèle pour qu'il prenne les données de type Rotated MNIST, il a fallu créer une fonction "load_rmnist_task_data()". En effet, les données de Rotated MNIST contiennent trois informations : les images, leur label et les angles de rotation. Tandis que les données synthétique du papier ne contenaient que les données et les labels. Cette fonction permet de récupérer les informations sur les données et de les préparer dans un tableau afin que le réseau de neurones puissent s'y adapter facilement.

En plus d'adapter comment les données sont données au modèle, il a fallu adapter comment le modèle utilise ces données dans ses boucles d'entraînement et de test. Avant

les boucles récupérait des données directement mais après notre traitement les données sont maintenant dans un tableau il faut d'abord extraire les données du tableau dans une boucle à l'intérieur des fonctions "train()" et "test()" puis d'utiliser les données extraite du tableau directement dans le modèle, plutôt que d'utiliser les données venant de l'extérieur des fonctions.

Lors de notre implémentation, nous avons rencontrés une erreur dans l'appel de la couche qui applique les filtres dans le réseau (layers.ShareLinearFull) sur l'incompatibilité des dimensions des tenseurs. Nous avons du retirer la taille des filtres appliqués sur l'image (kernel_size) de la taille des images pour corriger ce problème de dimensionnement.

Avec ces quelques modifications, nous avons réussi à faire utiliser les données Rotated MNIST au modèle fourni.

2.2 Modèle à circuit unique

Après avoir adapter les données au modèles nous avons modifié le modèle afin d'utilisé le mécanisme de "weight sharing". Pour se faire nous avons utilisé les fonctions défini dans le fichier "layers.py" du code fourni par les chercheurs. Les chercheurs ont implémenter eu même différents type de layer à utiliser pour leur modèle et les ont implémenter de telle sorte à ce qu'il soit plus souple au changement que les layers que l'on peut retrouver dans dans bibliothèque classique. Nous avons donc utilisé la couche "ShareConv2d" pour les couches de convolutions car c'est une couche à poids partagée spécifique pour les images en deux dimensions. Pour la fin de réseau qui permet de prédire la classe de l'image nous avons utilisé la classe déjà implémenter "ShareLinearFull".

Lors de l'utilisation de la fonction "ShareConv2d", nous avons rencontrés un bug et nous avons donc dû modifier le return de cette fonction car la fonction appelé dans le return ne supportait pas le jeu de données pour le "forward". Nous avons donc appelé une autre fonction, celle implémenter dans le module "torch.nn.functional", conv2d() afin de régler le problème du "forward".

2.3 Modèle à circuit dédié

Après avoir crée un modèle à couche partagé pour chaque angle de rotation sur le dataset Rotated MNIST, nous sommes passés à l'implémentation de circuit dédié par rotation avec une couche d'alignement en sortie. Pour se faire, nous avons séparer les deux architectures avec un nouveau paramètre d'entrée entre ["shared", "dedicated"] pour choisir au lancement le type de modèle à utiliser.

Pour l'implémentation de se modèle, nous avons utilisé la même structure que le modèles a circuit partagé en enlevant la couche de classification en triant les données à envoyé à chaque modèle en fonction des angles. Le modèle au moment de l'entraînement appelle

les différents réseaux qui s'entraîne sur un angle de rotation spécifique et en sortie de tous ces réseaux, il y a une couche d'alignement qui utilise la couche layers.ShareLinearFull avec comme différence qu'elle a un nombre de paramètre d'entrée multiplié par le nombre de réseau par rapport à cette même couche dans le réseau partagé. Cela permet de relié les circuits dédié à une rotation d'angle dans un même grand circuit et d'avoir un sortie des données uniforme.

Après l'implémentation des différents modèles nous allons passer à la visualisation des différents facteurs.

2.4 Visualisation des angles

Dans le but de mieux comprendre les données et de vérifier leur distribution, une visualisation est réalisée pour représenter le nombre d'exemples associés à chaque angle de rotation. Cette étape est nécessaire pour analyser les résultats. Pour cela, on modifie les fonctions train() et test() pour qu'un graphique en barres soit généré avec Matplotlib puis transmis dans les logs de Wandb. Ça permet de voir combien d'images le modèle a eux par angles différents, que ce soit pour les entraînements ou les tests. Comment on fait ? A chaque tâche, on regarde chaque angle associé à l'image et on la stocke dans un dictionnaire, ça permet d'avoir un dictionnaire avec tous les angles que le modèle rencontre. Chaque fois qu'on rencontre un angle, on incrémente de un le nombre d'image associée à cet angle, aussi simplement que ça. Voici ce que donne sur une run du code :

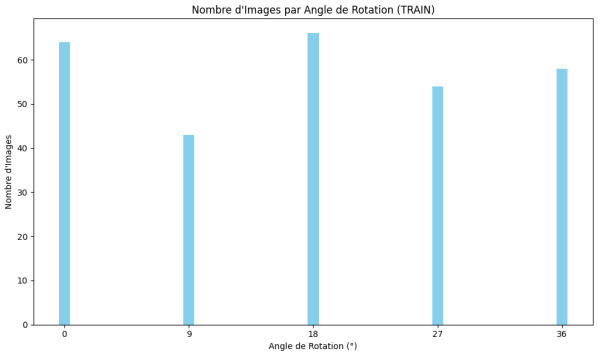

FIGURE 2 – Un exemple du nombre d'images par angle de rotation pour le train

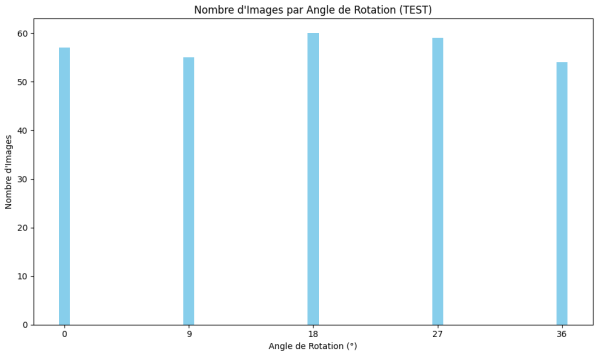

FIGURE 3 – Un exemple du nombre d'images par angle de rotation pour le test

Passons maintenant à la seconde étape pour capturer les facteurs de variations des angles, qui est le nombre d'erreurs commis par le modèle pour chaque angle de rotation. Cette fois si, ça se passe dans test(). Ça ne change pas beaucoup de la manière juste au dessus car on crée un dictionnaire pour les erreurs avec comme clé, chaque angle rencontré par le modèle et si une prédiction est fausse, on incrémente de un le nombre de prédiction fausse associé à l'angle. Pour chaque angle, on va créer un graphique représentant le nombre d'erreur sur un angle tout le long des steps qu'on affiche avec Wandb.

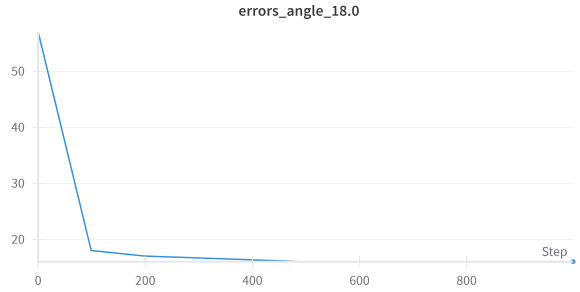

FIGURE 4 – Un exemple du nombre d'erreur de prédiction pour les images à 18°

2.5 Visualisation weight sharing

L'étude des poids appris par le modèle est une étape fondamentale pour comprendre son comportement et interpréter les caractéristiques qu'il extrait des données. Les poids des filtres convolutifs représentent des motifs visuels comme des lignes ou courbes que le modèle utilise pour différencier les chiffres. En visualisant ces poids, on visualise aussi les motifs. Cette analyse permet de comparer les architectures circuit unique et circuit dédié. Dans l'architecture partagée, les poids doivent capturer des caractéristique générales applicable à chaque rotation. Dans l'architecture dédiée, chaque circuit doit apprendre des caractéristiques spécifiques à une rotation. Pour faire cela, on a implémenter la fonction visualize_weights() qui permet de générer des images représentant les poids convolutifs du modèle. Pour l'architecture circuit unique, les poids sont extraits de la première couche

convolutive alors que pour le circuit dédié, les poids sont extrait séparément pour chaque circuit. Chaque filtre est représenté sous forme de matrice 2D qui est ensuite affichée sous forme de heat maps où les couleurs indiquent l'intensité du poids. On génère les graphiques avec Matplotlib à nouveau, qu'on transmet à Wandb.

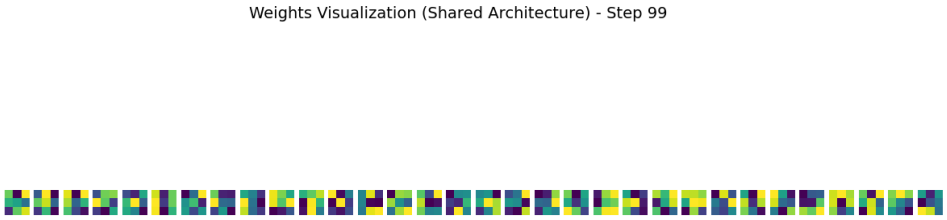

FIGURE 5 – Un exemple de visualisation des filtres pour l'architecture partagée

Après avoir analysé les poids appris par les différentes architectures, on va maintenant s'intéresser à la manière dont ces poids influencent le traitement des données dans le réseau. Les poids ne sont qu'une partie du mécanisme d'apprentissage, les activations intermédiaires, produites lorsque les donnée traversent le modèle, permette de comprendre comment les caractéristiques sont extraites et transformées à chaque étape. La visualisation des activations complète l'analyse des poids avec un aperçu fonctionnel du comportement de nos deux architectures. Pour ce faire, on a implémenté la fonction visualize_activations() qui capture et affiche les activations après chaque couche en fonction de l'architecture choisie. Pour le circuit unique, les données sont propagées couche par couche, les activations sont enregistrées après les couches importantes. Pour le circuit dédié, les activations de chaque circuit sont visualisées séparément. Ici les activations sont en 3D, on les moyenne sur les cannaux pour obtenir une image 2D et mieux visualiser les informations. Comme pour les poids, on affiche les activations sous formes de heat maps.

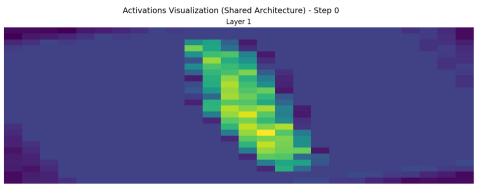

FIGURE 6 – Un exemple de visualisation des activations pour l'architecture circuit unique à 0 steps

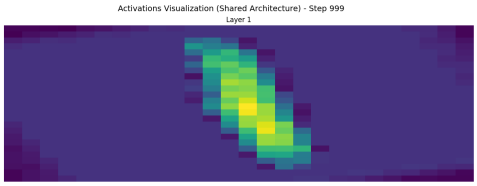

FIGURE 7 – Un exemple de visualisation des activations pour l'architecture circuit unique à 999 steps

Ces deux fonctions marchent très bien pour l'architecture circuit unique mais quand on test pour l'architecture circuit dédiée, le code plante et nous avons pas réussi à débugger les différentes erreur pour faire fonctionner sur ce modèle à temps.

3 Résultats

Maintenant que notre modèle avec ses 2 architecture est bien implémenté avec en plus des outils qui permettent d'analyser le modèle, nous allons passer à l'analyse des résultats. Tout d'abord nous avons les courbes qui était déjà implémenté dans le modèle du papier. Ce sont steps_per_sec, train_loss, test_loss et test_err. Le premier permet de mesurer la vitesse d'exécution du programme. Le deuxième est la perte moyenne sur toutes les requêtes pour l'entraînement. Le troisième c'est la même chose mais pour le test. Puis le dernier, c'est la différence entre la borne supérieure de l'intervalle de confiance à 95% et la perte moyenne, qui donne l'étendue de l'intervalle de confiance pour la perte sur l'ensemble de la requête. Pour ce qui est des résultats que nous avons implémentés, il y a le nombre d'images par angles pour l'entraînement et le test, les erreurs de prédiction pour chaque rotation d'image. Et pour finir, nous avons la visualisation des poids des filtres convolutifs et la visualisation des activations intermédiaires. Tous sont visibles sur Wandb.

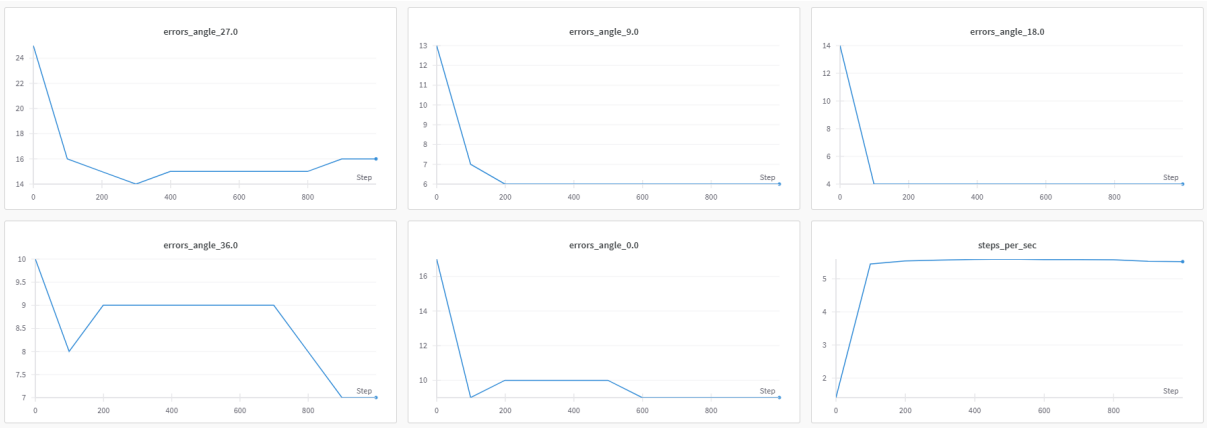

FIGURE 8 – Un exemple de visualisation des erreurs de prédiction et la vitesse d'exécution sur Wandb

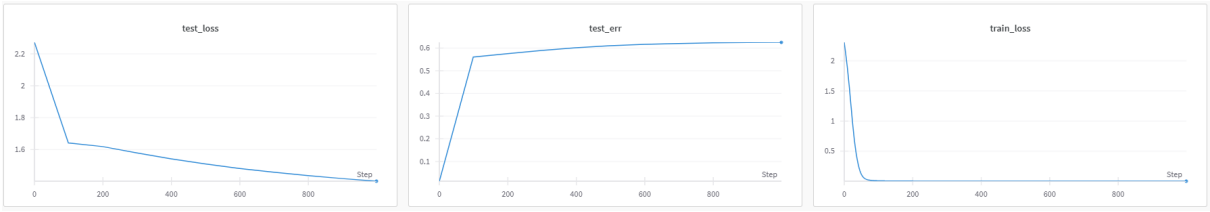

FIGURE 9 – Un exemple de visualisation des pertes moyennes du modèle à circuit unique sur Wandb

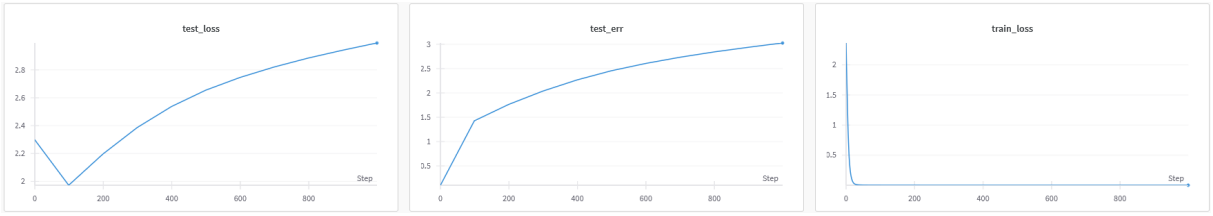

FIGURE 10 – Un exemple de visualisation des pertes moyennes du modèle à circuit dédié sur Wandb

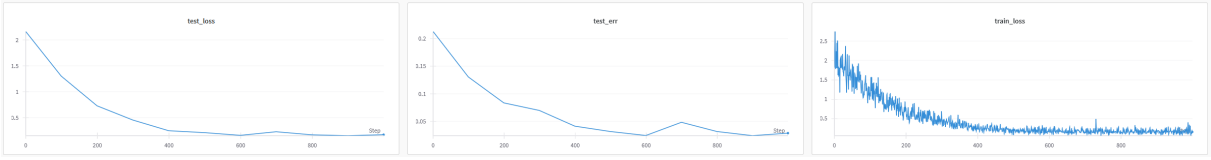

FIGURE 11 – Un exemple de visualisation des pertes moyennes du modèle original sur Wandb

Sur les trois figures précédentes, nous pouvons tirer des conclusions sur l'implémentation de nos modèles par rapport au modèle original. Sur le test_loss, on peut observer que le circuit unique suit une courbe similaire à l'implémentation originale alors que les circuits dédiés, eux, commencent par améliorer les prédictions puis l'erreur ne fait qu'augmenter. De même pour le test_err, le circuit unique fait de plus en plus d'erreur mais la différence reste minime après 100 epochs alors que les circuits dédiés eux suivent une courbe similaire mais la différence est assez marqué entre chaque centaine d'epochs et le modèle original, lui, ne fait que s'améliorer. Sur le dernier graphique, train_loss, on peut remarque que les courbes des trois graphiques ont la même forme mais la courbe du modèle original est bien plus lente et remplie d'irrégularités.

FIGURE 12 – Un exemple du nombre d'image par angle pour l'entraînement et le test sur Wandb

Sur ces graphes représentant la répartition des angles dans les différents dataloader, on peut observer une certaine disparité. Cette disparité s'explique parce que les dataloader prennent aléatoirement des données dans le dataset Rotated MNIST.

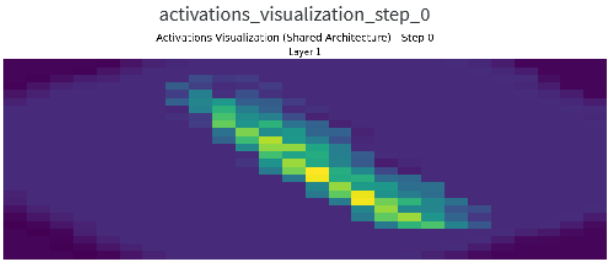

FIGURE 13 – Un exemple de visualisation des activations initiales sur Wandb

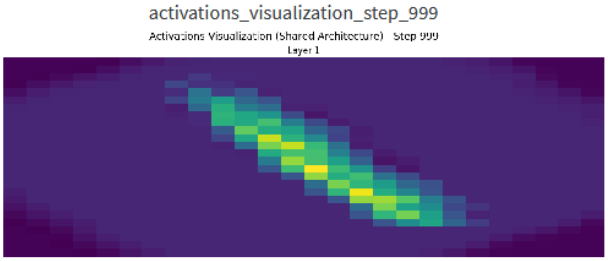

FIGURE 14 – Un exemple de visualisation des activations finales sur Wandb

Les deux figures précédentes représentent est une heatmap de l'importance des poids des filtres sur une des images du dataloader de test. Sur ces figures, on peut observer un changement de l'importance des poids, représenter par le changement de couleur.

Dans le code original, il y a aussi l'affichage dans le terminal de l'avancement de l'exécution du code en cours ainsi que la vitesse d'exécution et Val Loss qui est la même donné que test_loss.

```
Step: 0. Steps/sec: 1.41. Val Loss: 2.2721
Step: 99. Steps/sec: 5.45. Val Loss: 1.6420
Step: 199. Steps/sec: 5.53. Val Loss: 1.6189
Step: 299. Steps/sec: 5.56. Val Loss: 1.5788
Step: 399. Steps/sec: 5.58. Val Loss: 1.5418
Step: 499. Steps/sec: 5.59. Val Loss: 1.5097
Step: 599. Steps/sec: 5.58. Val Loss: 1.4819
Step: 699. Steps/sec: 5.58. Val Loss: 1.4578
Step: 799. Steps/sec: 5.57. Val Loss: 1.4368
Step: 899. Steps/sec: 5.53. Val Loss: 1.4183
Step: 999. Steps/sec: 5.52. Val Loss: 1.4019
```

FIGURE 15 – Un exemple de l'affichage dans le terminal

Pour terminer sur nos résultats, on peut dire que le résultat de l'implémentation à circuit unique n'est pas très éloignés des résultats initiaux des chercheurs ce qui est très encourageant. De l'autre côté, pour le réseau à circuit dédié par angle de rotation, les performances ne sont pas très éloignés mais le problème est que plus il apprend, moins il est performant. Cela est peut être dû a un sur apprentissage qu'il faudrait régler en ayant plus de données pour chaque angle.

4 Conclusion

Après avoir montré les résultats que nous avons obtenus, nous allons passer à la conclusion.

Après avoir lu l'article et nous l'être approprié, nous avons adapter le code du modèle des chercheurs au jeu de données Rotated MNIST puis nous avons implémenter deux architectures : un réseau de neurones convolutif à poids partagés et un réseau de neurones convolutif à circuit dédié par angle de rotation. Ensuite, nous avons montré les résultats de nos implémentations que nous avions réussi à avoir une implémentation plutôt bonne du réseau de neurones convolutif avec un circuit unique, tandis que l'implémentation avec un circuit dédié à chaque angle, elle, n'avait pas eu de résultats satisfaisant et que cela pouvait être du à un problème de sur apprentissage.

Pour conclure, nous avons implémenter un modèle de meta learning par reparamétrisation pour l'adapter à un nouveau jeu de données et grâce à cela, nous avons pu montrer que ce modèle est facilement modulable car il permet de recréer plusieurs architecture comme les réseaux de neurones convolutifs mais aussi que ce modèle peut accueillir plusieurs type de données comme les données synthétique utilisé par les chercheurs ou Rotated MNIST comme nous l'avons implémenter.

5 Bibliographie

[1] Chelsea Finn Allan Zhou Tom Knowles. *Meta-learning symmetries by reparameterization*. URL : https://arxiv.org/pdf/2007.02933. (Date de publication : 30/03/2021).

A Annexe

Lien github où vous trouverez les différents fichiers à télécharger pour faire tourner le programme : `https://github.com/GlobeTique77/metalearning-symmetries_AP2425`

Code principal :

```python
import argparse
import os
import time
import scipy.stats as st
import wandb
import numpy as np
import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
import higher
import matplotlib.pyplot as plt

import layers_2425_AP as layers
from inner_optimizers import InnerOptBuilder
from rotated_mnist_main.rotated_mnist import flattened_rotMNIST

OUTPUT_PATH = "./outputs/rotated_mnist_outputs"


def load_rmnist_task_data(loader, num_tasks, k_spt, k_qry):
    """Transformation des donn es pour cr er les ensembles support et requ te"""
    task_data = []
    for batch_idx, (images, labels, angles) in enumerate(loader):
        x_spt, y_spt, angles_spt = images[:k_spt], labels[:k_spt], angles[:k_spt]
        x_qry, y_qry, angles_qry = images[k_spt:k_spt + k_qry], labels[k_spt:k_spt +
            k_qry], angles[k_spt:k_spt + k_qry]

        task_data.append((x_spt, y_spt, angles_spt, x_qry, y_qry, angles_qry))
        if len(task_data) >= num_tasks:
            break

    return task_data

def visualize_weights(net, architecture, step_idx):
    """Visualise et log les poids du mod le dans WandB"""
    with torch.no_grad():
        if architecture == "shared":
            # Cas o  l'architecture est partag e
            shared_layer = net[0]  # La premi re couche partag e
            weights = shared_layer.weight.cpu().numpy()  # Poids des filtres convolutifs

            # Cr er une figure avec les filtres
            num_filters = weights.shape[0]
            fig, axes = plt.subplots(1, num_filters, figsize=(15, 5))
            for i, ax in enumerate(axes):
                ax.imshow(weights[i, 0], cmap="viridis")  # Poids du filtre
                ax.axis("off")
            plt.suptitle(f"Weights Visualization (Shared Architecture) - Step {step_idx}"
                , fontsize=14)
        elif architecture == "dedicated":
            # Cas o  il y a des circuits d di s
            circuits = net.circuits
            fig, axes = plt.subplots(len(circuits), 1, figsize=(10, len(circuits) * 3))
            for i, circuit in enumerate(circuits):
                dedicated_weights = circuit[0].weight.cpu().numpy()
```

```
                ax = axes[i]
                ax.imshow(dedicated_weights[0, 0], cmap="viridis")
                ax.axis("off")
                ax.set_title(f"Circuit {i + 1} Weights")
            plt.suptitle(f"Weights Visualization (Dedicated Architecture) - Step {step_idx}", fontsize=14)
        else:
            raise ValueError("Architecture non reconnue pour la visualisation des poids.")

        # Log directement dans WandB
        wandb.log({f"weights_visualization_step_{step_idx}": wandb.Image(fig)}, step=step_idx)
        plt.close(fig)


def visualize_activations(net, x_sample, architecture, step_idx):
    """Visualise et log les activations interm diaires du mod le dans WandB"""
    with torch.no_grad():
        if architecture == "shared":
            activations = []
            current_input = x_sample
            for layer in net:
                current_input = layer(current_input)
                if isinstance(layer, nn.ReLU):  # Prendre les activations apr s chaque ReLU
                    activations.append(current_input.cpu().numpy())

            # Cr er un graphique pour chaque couche
            num_activations = len(activations)
            fig, axes = plt.subplots(1, num_activations, figsize=(15, 5))

            # G rer le cas o  il n'y a qu'une seule activation
            if num_activations == 1:
                axes = [axes]

            for i, activation in enumerate(activations):
                # R duire les dimensions pour rendre compatible avec imshow
                # Prendre la moyenne sur les canaux pour obtenir une image 2D
                activation_2d = activation[0].mean(axis=0)  # Moyenne sur les canaux

                ax = axes[i]
                ax.imshow(activation_2d, cmap="viridis", aspect="auto")
                ax.axis("off")
                ax.set_title(f"Layer {i + 1}")
            plt.suptitle(f"Activations Visualization (Shared Architecture) - Step {step_idx}", fontsize=14)

        elif architecture == "dedicated":
            circuits = net.circuits
            num_circuits = len(circuits)
            fig, axes = plt.subplots(num_circuits, 1, figsize=(10, num_circuits * 3))

            # G rer le cas o  il n'y a qu'un seul circuit
            if num_circuits == 1:
                axes = [axes]

            for i, circuit in enumerate(circuits):
                activations = circuit(x_sample).cpu().numpy()

                # R duire les dimensions pour rendre compatible avec imshow
                activation_2d = activations[0].mean(axis=0)  # Moyenne sur les canaux

                ax = axes[i]
                ax.imshow(activation_2d, cmap="viridis", aspect="auto")
```

```
                ax.axis("off")
                ax.set_title(f"Activations from Circuit {i + 1}")
            plt.suptitle(f"Activations Visualization (Dedicated Architecture) - Step {step_idx}", fontsize=14)
        else:
            raise ValueError("Architecture non reconnue pour la visualisation des activations.")

        # Log directement dans WandB
        wandb.log({f"activations_visualization_step_{step_idx}": wandb.Image(fig)}, step=step_idx)
        plt.close(fig)


def train(step_idx, task_data, net, inner_opt_builder, meta_opt, n_inner_iter, architecture, weight):
    """Main meta-training step for RMNIST."""
    qry_losses = []
    meta_opt.zero_grad()
    angle_errors = {}  # Dictionnaire pour stocker les erreurs par angle
    angle_counts = {}  # Dictionnaire pour stocker le nombre d'images par angle

    for task in task_data:
        x_spt, y_spt, angles_spt, x_qry, y_qry, angles_qry = task

        task_num = x_spt.size(0)
        inner_opt = inner_opt_builder.inner_opt

        # Copie du bout de test() pour compter les images par angle
        for angle in angles_qry:
            angle_val = angle.item()
            if angle_val not in angle_errors:
                angle_errors[angle_val] = 0
                angle_counts[angle_val] = 0

        with higher.innerloop_ctx(
            net,
            inner_opt,
            copy_initial_weights=False,
            override=inner_opt_builder.overrides,
        ) as (fnet, diffopt):
            # Inner-loop updates on the support set
            for _ in range(n_inner_iter):
                spt_pred = fnet(x_spt)
                spt_loss = F.cross_entropy(spt_pred, y_spt)  # Classification loss
                diffopt.step(spt_loss)

            # Query set evaluation
            qry_pred = fnet(x_qry)
            qry_loss = F.cross_entropy(qry_pred, y_qry)
            qry_losses.append(qry_loss.detach().cpu().numpy())
            qry_loss.backward()

            # Comptage des images par angle
            qry_pred_classes = torch.argmax(qry_pred, dim=1)  # Pr dictions des classes
            for pred, label, angle in zip(qry_pred_classes, y_qry, angles_qry):
                angle_val = angle.item()
                angle_counts[angle_val] += 1  # Incr menter le compteur d'images

    # /!\ Pour visualiser le weight sharing de train, ne pas mettre en m me temps que le test.
    # **Appel    visualize_weights et visualize_activations**
    # if weight and (step_idx == 0 or (step_idx + 1) % 100 == 0):
    #     # Visualiser les poids appris
    #     visualize_weights(net, architecture, step_idx)

```

```
    #     # Visualiser les activations interm diaires pour un   chantillon   (query set)
    #     x_sample = task_data[0][3][:1]  # Prendre un   chantillon   du query set
    #     visualize_activations(net, x_sample, architecture, step_idx)

    # G n rer le graphique en barres pour le nombre d'images par angle
    angles = list(angle_counts.keys())
    counts = list(angle_counts.values())
    plt.figure(figsize=(10, 6))
    plt.bar(angles, counts, color='skyblue')
    plt.xlabel("Angle de Rotation (  )")
    plt.ylabel("Nombre d'Images")
    plt.title("Nombre d'Images par Angle de Rotation (TRAIN)")
    plt.xticks(angles)
    plt.tight_layout()
    # Log du graphique directement dans wandb
    wandb.log({"angle_counts_barplot1": wandb.Image(plt)}, step=step_idx)
    plt.close()

    # Meta-optimization step
    metrics = {"train_loss": np.mean(qry_losses)}
    wandb.log(metrics, step=step_idx)
    meta_opt.step()

def test(step_idx, task_data, net, inner_opt_builder, n_inner_iter, architecture, weight)
    :
    """Main meta-testing step for RMNIST"""
    qry_losses = []
    angle_errors = {}  # Dictionnaire pour stocker les erreurs par angle
    angle_counts = {}  # Dictionnaire pour stocker le nombre d'images par angle

    for task in task_data:
        x_spt, y_spt, angles_spt, x_qry, y_qry, angles_qry = task

        task_num = x_spt.size(0)
        inner_opt = inner_opt_builder.inner_opt

        # Initialisation des erreurs par angle
        for angle in angles_qry:
            angle_val = angle.item()
            if angle_val not in angle_errors:
                angle_errors[angle_val] = 0
                angle_counts[angle_val] = 0

        with higher.innerloop_ctx(
            net, inner_opt, track_higher_grads=False, override=inner_opt_builder.
                overrides,
        ) as (fnet, diffopt):
            # Inner-loop updates on the support set
            for _ in range(n_inner_iter):
                spt_pred = fnet(x_spt)
                spt_loss = F.cross_entropy(spt_pred, y_spt)
                diffopt.step(spt_loss)

            # Query set evaluation
            qry_pred = fnet(x_qry)
            qry_loss = F.cross_entropy(qry_pred, y_qry)
            qry_losses.append(qry_loss.detach().cpu().numpy())

            # Comptage des erreurs par angle
            qry_pred_classes = torch.argmax(qry_pred, dim=1)  # Pr dictions des classes
            for pred, label, angle in zip(qry_pred_classes, y_qry, angles_qry):
                angle_val = angle.item()
                angle_counts[angle_val] += 1  # Incr menter compteur d'images
                if pred != label:  # Si pr diction incorrecte
                    angle_errors[angle_val] += 1

```

```
    avg_qry_loss = np.mean(qry_losses)
    _low, high = st.t.interval(
        0.95, len(qry_losses) - 1, loc=avg_qry_loss, scale=st.sem(qry_losses)
    )
    test_metrics = {"test_loss": avg_qry_loss, "test_err": high - avg_qry_loss}
    wandb.log(test_metrics, step=step_idx)

   # Log des erreurs et du nombre d'images par angle
    for angle in angle_counts.keys():
        wandb.log({f"errors_angle_{angle}": angle_errors[angle]}, step=step_idx)

    # G n rer le graphique en barres pour le nombre d'images par angle
    angles = list(angle_counts.keys())
    counts = list(angle_counts.values())
    plt.figure(figsize=(10, 6))
    plt.bar(angles, counts, color='skyblue')
    plt.xlabel("Angle de Rotation (  )")
    plt.ylabel("Nombre d'Images")
    plt.title("Nombre d'Images par Angle de Rotation (TEST)")
    plt.xticks(angles)
    plt.tight_layout()
    # Log du graphique directement dans wandb
    wandb.log({"angle_counts_barplot2": wandb.Image(plt)}, step=step_idx)
    plt.close()

    # /!\ Pour visualiser le weight sharing de test, ne pas mettre en m me temps que le
        train.
    # **Appel    visualize_weights et visualize_activations**
    if weight and (step_idx == 0 or (step_idx + 1) % 100 == 0):
        # Visualiser les poids appris
        visualize_weights(net, architecture, step_idx)

        # Visualiser les activations interm diaires pour un   chantillon   (query set)
        x_sample = task_data[0][3][:1]  # Prendre un   chantillon   du query set
        visualize_activations(net, x_sample, architecture, step_idx)


    return avg_qry_loss

def build_model(architecture, device, num_tasks):
    """Construire le mod le selon l'architecture sp cifi e."""
    if architecture == "shared":
        # Toutes les rotations aplaties en entr e d'un unique circuit
        net = torch.nn.Sequential(
            layers.ShareConv2d(1, 32, kernel_size=3),
            nn.ReLU(),
            nn.Flatten(),
            layers.ShareLinearFull(32 * 26 * 26, 10)
            # 10 classes pour RMNIST, 32 filtres, 26=28 (taille image)-3 (taille filtre)
                +1
        ).to(device)
    elif architecture == "dedicated":
        # Chaque angle de rotation avec un circuit  d  di   (avec une couche d'alignement
            en sortie)
        circuits = nn.ModuleList()
        for _ in range(num_tasks):
            circuit = nn.Sequential(
                layers.ShareConv2d(1, 32, kernel_size=3),
                nn.ReLU(),
                nn.Flatten()
            )
            circuits.append(circuit)

        # Couche d'alignement en sortie
        align_layer = layers.ShareLinearFull(32 * 26 * 26 * num_tasks, 10)

```

```
304         class DedicatedNetwork(nn.Module):
305             def __init__(self, circuits, align_layer):
306                 super(DedicatedNetwork, self).__init__()
307                 self.circuits = circuits
308                 self.align_layer = align_layer
309 
310             def forward(self, x):
311                 # Appliquer chaque circuit d di  sur l'entr e
312                 outputs = []
313                 for circuit in self.circuits:
314                     outputs.append(circuit(x))
315                 # Concatenation des sorties des circuits
316                 concatenated = torch.cat(outputs, dim=-1)
317                 # Passer par la couche d'alignement
318                 return self.align_layer(concatenated)
319 
320         net = DedicatedNetwork(circuits, align_layer).to(device)
321     else:
322         raise ValueError("Architecture inconnue : " + architecture)
323     return net
324 
325 def main():
326     parser = argparse.ArgumentParser()
327     parser.add_argument("--init_inner_lr", type=float, default=0.01)
328     parser.add_argument("--outer_lr", type=float, default=0.0001)
329     parser.add_argument("--k_spt", type=int, default=1)
330     parser.add_argument("--k_qry", type=int, default=19)
331     parser.add_argument("--lr_mode", type=str, default="per_layer")
332     parser.add_argument("--num_inner_steps", type=int, default=1)
333     parser.add_argument("--num_outer_steps", type=int, default=1000)
334     parser.add_argument("--inner_opt", type=str, default="maml")
335     parser.add_argument("--outer_opt", type=str, default="Adam")
336     parser.add_argument("--device", type=str, default="cpu")
337     parser.add_argument("--num_tasks", type=int, default=5) #Nombre d'angles diff rents
338     parser.add_argument("--per_task_rotation", type=int, default=9 ) #D calage de l'
        angle
339     parser.add_argument("--batch_size", type=int, default=1024)
340     parser.add_argument("--architecture", type=str, choices=["shared", "dedicated"],
        default="shared")
341     parser.add_argument("--view_weight_sharing", type=bool, default=False) #voir weigt
        sharing ou non
342 
343     args = parser.parse_args()
344 
345     if not os.path.exists(OUTPUT_PATH):
346         os.makedirs(OUTPUT_PATH)
347 
348     wandb.init(project="rmnist_meta_learning", dir=OUTPUT_PATH)
349     wandb.config.update(args)
350     cfg = wandb.config
351 
352     device = torch.device(cfg.device)
353 
354     #Cr ation donn es RMNIST
355     train_loader, test_loader = flattened_rotMNIST(
356         num_tasks=cfg.num_tasks,
357         per_task_rotation=cfg.per_task_rotation,
358         batch_size=cfg.batch_size
359     )
360 
361     #Transformation des donn es pour adaptation
362     train_task_data = load_rmnist_task_data(train_loader, cfg.num_tasks, cfg.k_spt, cfg.
        k_qry)
363     test_task_data = load_rmnist_task_data(test_loader, cfg.num_tasks, cfg.k_spt, cfg.
        k_qry)
364 
```

```
    # Define model
    net = build_model(cfg.architecture, device, cfg.num_tasks)

    inner_opt_builder = InnerOptBuilder(
        net, device, cfg.inner_opt, cfg.init_inner_lr, "learned", cfg.lr_mode
    )

    if cfg.outer_opt == "SGD":
        meta_opt = optim.SGD(inner_opt_builder.metaparams.values(), lr=cfg.outer_lr)
    else:
        meta_opt = optim.Adam(inner_opt_builder.metaparams.values(), lr=cfg.outer_lr)

    start_time = time.time()
    for step_idx in range(cfg.num_outer_steps):
        train(
            step_idx,
            train_task_data,
            net,
            inner_opt_builder,
            meta_opt,
            cfg.num_inner_steps,
            cfg.architecture,
            cfg.view_weight_sharing
        )

        if step_idx == 0 or (step_idx + 1) % 100 == 0:
            val_loss = test(
                step_idx,
                test_task_data,
                net,
                inner_opt_builder,
                cfg.num_inner_steps,
                cfg.architecture,
                cfg.view_weight_sharing
            )

            steps_p_sec = (step_idx + 1) / (time.time() - start_time)
            wandb.log({"steps_per_sec": steps_p_sec}, step=step_idx)
            print(f"Step: {step_idx}. Steps/sec: {steps_p_sec:.2f}. Val Loss: {val_loss
                :.4f}")

if __name__ == "__main__":
    main()
```

Listing 1 – Fonction Python