CSE 676: Deep Learning, Summer 2023
Project Final Report

Abhijeet, Sugam, asugam
Ayush, Utkarsh, ayushutk
Kumaramangalam, Vangavolu, kumarama

Meta-learning for few-shot Novel Pulmonary Disease Detection

AIM:

This project proposes to apply meta-learning methods on medical chest X-ray datasets for more efficient and reliable few shot disease detection and diagnosis.

Additionally, the project will compare the effectiveness of meta learning with different model architectures in medical domain. Through learning shared representations across various tasks, the model will potentially improve performance on data-limited tasks.

Problem Statement:

When it comes to the accurate and timely detection of pulmonary diseases, there are several challenges that the healthcare and AI industry must address:

1. **Data Collection and Labelled Training Data**: Machine learning models, especially the advanced ones like deep learning, are data-hungry. They require large amounts of labelled data to learn patterns accurately. Collecting data for medical purposes is challenging because it requires patient consent and collaboration from medical institutions. Additionally, labeling this data is often laborious and requires medical expertise. Therefore, generating high-quality labelled data for a diverse range of pulmonary diseases becomes a significant bottleneck.

2. **Privacy and Compliance Issues**: Healthcare data is particularly sensitive due to the private nature of the information it contains. As a result, various laws and regulations protect patient data, such as HIPAA (Health Insurance Portability and Accountability Act) in the U.S., GDPR (General Data Protection Regulation) in the EU, among others. Ensuring privacy and compliance when dealing with patient data presents challenges. For example, anonymizing data to protect patient identities can sometimes remove relevant features for the model.

3. **Generalizing to Novel/Rare Diseases**: Generalization is a significant challenge in machine learning. A model trained on common diseases may not perform well on novel or rare diseases. The fewer data samples available for a specific disease, the harder it is for the model to learn and predict that disease accurately. This was evident during the early stages of the COVID-19 pandemic, where AI models struggled to identify and predict the disease accurately due to the lack of available data.

In the case of Chronic Obstructive Pulmonary Disease (COPD), the symptoms often overlap with other respiratory diseases, making it difficult for models to distinguish it accurately. Also, COPD has a gradual progression, meaning that the disease's early stages may not present as much data for the model to learn from. Therefore, training a model that can accurately identify diseases like COVID-19 or COPD from limited data samples remains a challenging task.

In order to address these challenges, efforts must be directed towards the development of models that can learn effectively from limited data, methods that preserve patient privacy while maintaining data utility, and a more collaborative healthcare data environment. Collaborations between medical institutions, AI researchers, and regulatory authorities can be a way to navigate through these challenges while ensuring ethical standards are met.

Our Thought Process:

Our goal is to develop a model using meta-learning techniques that can quickly adapt to new and unseen pulmonary diseases with limited data. This is a crucial approach in the medical domain given the scarcity of certain conditions and the difficulties in collecting ample amounts of labelled data.

Why Few-Shot Learning?

Medical imaging encompasses various modalities such as X-ray, computed tomography (CT), magnetic resonance imaging (MRI), and ultrasound, among others. These modalities generate massive amounts of

image data, making manual annotation a formidable task. Few-shot learning provides a promising solution by enabling models to learn from a limited number of annotated examples.

Benefits of Few-Shot Learning in Medical Imaging:

- **Limited Labeled Data:** In medical imaging, obtaining a large number of accurately labeled images can be challenging due to factors such as privacy concerns, the need for expert annotation, and the rarity of certain medical conditions. Few-shot learning allows models to learn from a small set of labeled examples, reducing the burden of data collection and annotation.
- **Efficient Model Adaptation:** Medical imaging datasets often suffer from domain shift and concept drift. Few-shot learning techniques excel in adapting to new imaging domains or rare medical conditions, as they can quickly leverage prior knowledge from similar tasks or datasets. This adaptability is crucial for real-world clinical scenarios where rapid deployment and accurate diagnosis are of utmost importance.
- **Cost and Time Savings:** The process of acquiring large-scale medical imaging datasets is resource-intensive and time-consuming. Few-shot learning mitigates these challenges by enabling effective learning from limited labeled data, reducing the cost and time required for dataset collection, annotation, and model training.

Challenges in Few-Shot Learning for Medical Imaging:

- **Data Heterogeneity:** Medical imaging datasets often exhibit high inter- and intra-class variabilities due to differences in imaging protocols, patient demographics, and disease manifestations. Handling this heterogeneity poses a significant challenge for few-shot learning algorithms, as they need to generalize well across diverse image distributions.
- **Limited Support for Rare Conditions:** Some medical conditions are rare, making it difficult to collect a sufficient number of labeled examples. Traditional deep learning approaches struggle to learn from scarce data, resulting in suboptimal performance. Few-shot learning methods can effectively leverage limited labeled examples to generalize well on rare conditions, thereby improving diagnostic accuracy.

Why meta-learning though?

The field of model generalization on a new set of categories based on only a few training data is called few-shot learning. There have been multiple different techniques proposed in literature to facilitate learning from a few datapoints

- Metric Learning: Metric learning aims to learn a similarity metric or distance function that measures the similarity between samples. In few-shot learning, metric learning methods learn an embedding space where samples from the same class are close together, while samples from

different classes are far apart. Prototypical Networks and Siamese Networks are popular metric learning approaches for few-shot learning.

- Data Augmentation: Data augmentation techniques are commonly used in few-shot learning to artificially increase the size and diversity of the training set. By applying various transformations such as rotations, translations, and flips to the few available examples, data augmentation helps the model generalize better to unseen samples. Additionally, generative models like Generative Adversarial Networks (GANs) can be used to synthesize new samples.
- Transfer Learning: Transfer learning leverages knowledge learned from a source domain (where labeled data is abundant) to improve performance on a target domain (with limited labeled data). In few-shot learning, a pre-trained model on a large-scale dataset can be fine-tuned with the few labeled examples from the target domain. This transfer of knowledge allows the model to benefit from the learned representations and generalize well.
- Prototype-based Approaches: Prototype-based methods formulate few-shot learning as a problem of learning prototype representations for each class. Prototypes serve as representatives of the class and can be computed as the mean or centroid of the support set (labeled examples). During inference, new samples are compared to the prototypes to determine their class membership. Prototypical Networks and Matching Networks are examples of prototype-based approaches.
- Bayesian Approaches: Bayesian methods in few-shot learning incorporate prior knowledge and uncertainty estimation into the learning process. Bayesian models can capture the uncertainty of predictions when there is limited labeled data, providing more reliable and calibrated estimates. Bayesian meta-learning and Bayesian neural networks are examples of Bayesian approaches for few-shot learning.
- Meta-Learning: Meta-learning, also known as learning to learn, focuses on training models to acquire learning abilities that enable them to adapt quickly to new tasks. Meta-learning algorithms aim to learn a set of meta-parameters that can be fine-tuned with a few examples from a new task. Model-Agnostic Meta-Learning (MAML) and Reptile are widely used meta-learning algorithms for few-shot learning.

Meta learning has been shown to be one of the most efficient ways to deal with few-shot learning.

Meta-learning, often referred to as "learning to learn", is a subfield of machine learning where models are designed to learn quickly when presented with new tasks. In the context of our project, each task could be the classification of a specific pulmonary disease. The model trains on a distribution of tasks, or a dataset containing multiple tasks, and aims to extract common knowledge or patterns across those tasks. This common knowledge then aids the model in generalizing well to new tasks. The advantage of meta-learning is that once the model is trained, it can adapt to new tasks or disease classes with just a few training examples.

We are planning to use two common meta-learning algorithms, namely **Model-Agnostic Meta-Learning (MAML)** and **Reptile**.

- **MAML** operates by learning a model initialization that can be fine-tuned quickly and effectively for a new task using a small number of steps and examples. In the case of classifying pulmonary

diseases, MAML would learn a generic initialization for disease classification and then swiftly fine-tune this base with a few examples of a new disease.

- **Reptile**, on the other hand, is a simpler meta-learning algorithm. It works by performing stochastic gradient descent (SGD) on the parameters of the model for each task and then updating the initial parameters towards the direction of the parameters after the task-specific SGD. The underlying intuition is to move the initial parameters closer to the optimal ones for each task.

We've also considered **few-shot learning**, which is closely related to meta-learning. Few-shot learning refers to the problem of learning from a small number of examples, which is particularly vital in medical imaging, where labeled data is often scarce. In this context, our aim is to develop models that can accurately recognize new classes of diseases with only a few labeled samples. This is achieved by leveraging the knowledge learned from related tasks (or diseases), which the model can generalize from when faced with a new task.

In summary, our goal involves using meta-learning techniques to rapidly adapt a model to diagnose new and unseen pulmonary diseases based on limited data. The methods we're using, MAML, Reptile, and few-shot learning, are designed to learn from a distribution of tasks (diseases) and generalize that knowledge to perform classification on new tasks with few examples. This approach may significantly enhance the flexibility and applicability of our model to real-world, diverse medical scenarios.

Background:

Meta-learning, or "learning to learn", is a concept in machine learning that seeks to design models that can learn new tasks quickly with minimal training data. This is particularly important in applications where data may be scarce or expensive to obtain, such as medical imaging or rare disease detection.

There are three commonly recognized approaches to meta-learning:

Metric-based approaches aim to learn a distance function over objects. This distance function can then be used to compare and classify new objects based on their distance to existing ones. Several models fall under this approach:

- **Convolutional Siamese Neural Network**: This approach, proposed by Koch, Zemel & Salakhutdinov in 2015, uses a twin neural network architecture (hence the term 'Siamese') to learn an embedding space where similar items are close to each other. This network learns to differentiate between categories based on a single example (one-shot learning), which is particularly useful for tasks with limited data.

- **Matching Networks**: Introduced by Vinyals in 2016, Matching Networks are designed for one-shot learning scenarios. They are trained to learn a similarity metric between examples and then use this similarity metric to classify new examples. They do this by

considering all examples in the support set for each prediction and by using an attention mechanism to weigh the influence of each support example.

- **Relational Networks**: Proposed by Santoro in 2018, these networks aim to enable explicit reasoning about the relationships between examples in few-shot learning scenarios. They work by computing relations between pairs of objects and then aggregating these relations to produce an output.

- **Prototypical Networks**: Proposed by Snell, these networks address few-shot learning problems by employing the notion of prototypes. The idea is to represent each class by a prototype, which is the mean of the embedded feature vectors of the examples in that class. When a new example comes in, it is compared to each class prototype and typically classified as belonging to the nearest one.

Model-based meta-learning methods learn to adjust the parameters of the model based on the input data. In this approach, we aim to learn an explicit model or algorithm that will allow the model's parameters to adapt to new tasks effectively. Here are a couple of examples:

- **Memory-Augmented Neural Networks (MANN)**: MANNs augment standard neural networks with a form of external memory, making them better suited to handle new tasks rapidly with few examples. They use a form of differentiable attention over their memory to allow for gradient-based training. This allows the network to learn which information to store in the memory and when to retrieve it, enabling it to adapt its response based on the stored past information.

- **Meta Networks (MetaNet)**: MetaNet learns a meta-level knowledge across tasks and shifts its inductive bias for each task using fast parameterization. This fast parameterization is derived from a meta-learner, which uses the learning experience from previous tasks. The goal is to allow the network to adapt rapidly to new tasks.

Optimization-based meta-learning, also known as gradient-based meta-learning, are approaches that optimize model parameters through gradient-based optimization. They aim to learn a set of initial parameters that can be quickly adapted to new tasks or domains using gradient updates. Here are a couple of examples:

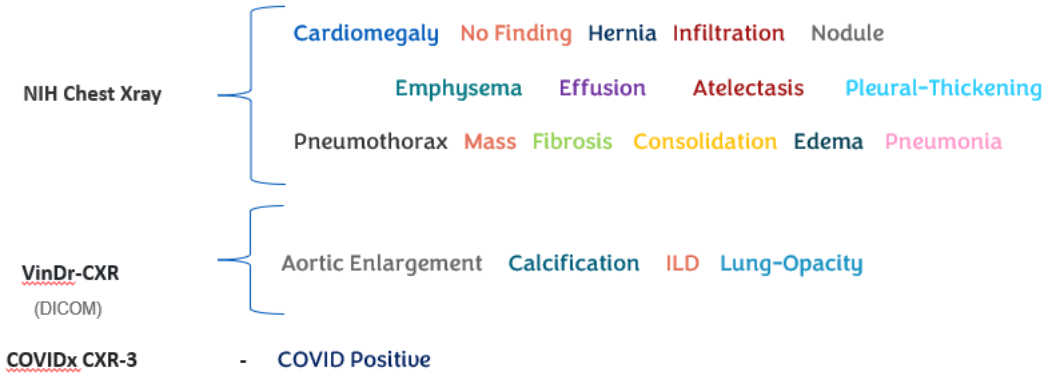
- **Model-Agnostic Meta-Learning (MAML)**: Introduced by Chelsea Finn in 2017, MAML learns an initialization of the model parameters that allows for fast adaptation to new tasks. It achieves this by finding an initial set of parameters such that small updates lead to a significant performance increase on new tasks. MAML's strength lies in its ability to be applied to any model trained with gradient descent.

- **Reptile**: Reptile, like MAML, aims to find a good parameter initialization, but it does so in a simpler way. It performs Stochastic Gradient Descent (SGD) on each task separately and then updates the initial parameters towards the direction of the task-specific optimal parameters. It does not require second-order derivatives like MAML, which makes it computationally more efficient.

Each of these approaches has its own advantages and challenges, and the choice of approach can depend on the specific task and the available data.In summary, model-based and optimization-based meta-learning methods offer different ways to tackle the challenge of learning new tasks quickly with minimal examples. They do so by either learning to adjust the model's parameters in response to new tasks (model-based) or by finding a parameter initialization that can be rapidly fine-tuned for new tasks (optimization-based). These techniques are critical in applications where data is scarce or expensive to collect, like medical imaging or rare disease detection.

Dataset:

In our project, we have amalgamated data from three distinct and extensive datasets: the NIH Chest X-ray dataset, the VinDrCXR dataset, and the COVIDx CXR-3 dataset. Combined, these datasets provide a rich and diverse set of instances across 20 unique classes. This comprehensive dataset is instrumental in creating a robust and versatile model capable of classifying a wide range of pulmonary conditions.



NIH Chest Xray Dataset contains 5309 X-ray images with 14 unique disease labels from unique patients.

The disease (or condition) labels are.
Atelectasis, Consolidation, Infiltration, Pneumothorax, Edema, Emphysema, Fibrosis, Effusion, Pneumonia, Pleural thickening ,Cardiomegaly ,Nodule/Mass ,Hernia, No Findings

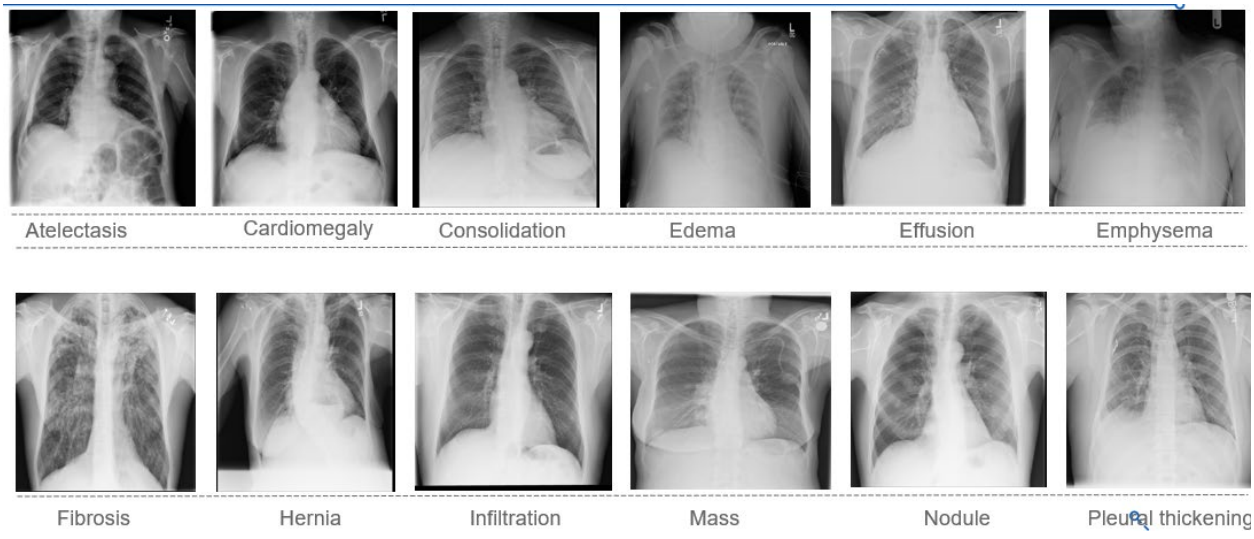
This sample dataset is very much unbalanced, and we might evaluate the option of adding another chunk of the original dataset to this sample if the results are not promising with the baseline model. The count of the images across different labels is mentioned below:

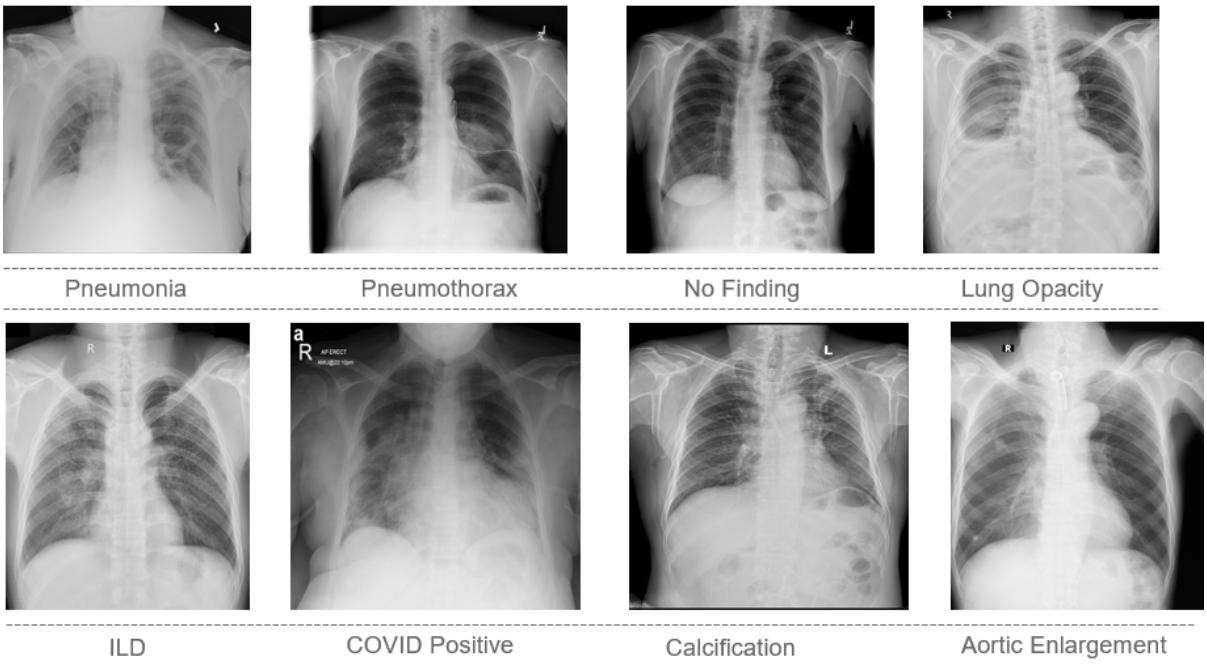
Test set classes: Atelectasis , Hernia, Edema

Train set classes: Remaining 17

15 images per each class – Support 5, Query 10

DATASET SAMPLES:





Neural Network Architecture:

Our neural network architecture called MyAlexNet, which is based on the popular AlexNet architecture is modified in such a way that out dataset gives appropriate result:

Network Structure:

- The network consists of two main parts: the "features" module and the "classifier" module.
- The "features" module contains a sequence of convolutional and pooling layers, responsible for extracting hierarchical features from input images.
- The "classifier" module consists of fully connected layers, responsible for mapping the extracted features to the desired number of output classes.

Convolutional Layers:

- The network starts with a 2D convolutional layer with 64 filters, a kernel size of 11, a stride of 4, and padding of 2.
- Batch normalization is applied after the first convolutional layer.
- ReLU activation function is applied after each convolutional layer.
- Max pooling with a kernel size of 3 and stride of 2 is performed after the first and second convolutional layers.
- The subsequent layers consist of convolutional layers with decreasing filter sizes: 192, 384, 256, and 256.
- Batch normalization is applied after each convolutional layer in the "features" module.
- ReLU activation function is applied after each convolutional layer.

Fully Connected Layers:

- The output of the convolutional layers is flattened and passed through the fully connected layers in the "classifier" module.
- Dropout is applied (if specified) after the first and second fully connected layers.
- The first fully connected layer has 4096 output units.
- ReLU activation function is applied after each fully connected layer.
- The last fully connected layer produces the final output logits, with the number of units equal to the specified "num_classes" parameter.

```
<bound method Module.parameters of MyAlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (5): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=3, bias=True)
  )
)>
```

Regularization:

- Two types of regularization are implemented: L1 regularization and L2 regularization.
- The regularization strength can be controlled using the "reg_strength" parameter.
- L1 regularization penalizes the L1 norm of the network parameters.
- L2 regularization penalizes the L2 norm of the network parameters.
- The regularization losses are computed separately in the "l1_regularization_loss" and "l2_regularization_loss" methods, respectively.
- The regularization losses are multiplied by the "reg_strength" value and added to the overall loss during training.

Forward Pass:

- The forward method performs the forward pass through the network.
- The input is passed through the "features" module, then flattened and passed through the "classifier" module.
- The output of the last fully connected layer is returned as the final logits.
- The provided architecture can be used for classification tasks with a specified number of classes. It also supports the option of using dropout regularization during training and applying L1 or L2 regularization to the network parameters.

Our structure has the same idea of Alexnet but we have made few differences from the original structure:

- Batch Normalization: Our AlexNet structure includes batch normalization layers after each convolutional layer and before the activation function. Batch normalization helps in normalizing the inputs to each layer, improving the stability and speed of training.
- Softmax Activation: By the virtue of CrossEntropyLoss we are using SoftMax Activation function.

Methodology:

Meta Learning methods

- MAML
- Reptile

MAML:

The goal of MAML is to learn a good initialization of model parameters that can quickly adapt to new tasks with limited training samples.

MAML addresses the challenge of learning from small datasets, which often leads to poor generalization and limited performance. It adopts a meta-learning framework, where the model is trained on a distribution of tasks rather than a single task. This allows the model to learn general representations that can be fine-tuned quickly for new tasks.

The MAML algorithm involves two steps: an inner loop and an outer loop. In the inner loop, the model parameters are updated using a small number of samples from a specific task, aiming to achieve good performance on that task. The gradients obtained during this inner loop update are used to compute the updated model parameters. In the outer loop, these updated parameters are evaluated on a different set of tasks to measure their generalization ability. The parameters are then adjusted using these evaluation results, optimizing for good performance across multiple tasks.

The authors demonstrate the effectiveness of MAML on a variety of tasks, such as regression, classification, and reinforcement learning. They compare MAML with other meta-learning approaches and show that it achieves better performance and faster adaptation. MAML is also shown to be model-agnostic, meaning it can be applied to different types of deep neural networks without modifications.

Overall, the paper presents Model-Agnostic Meta-Learning (MAML) as a powerful technique for fast adaptation of deep networks. It offers a promising approach to address the challenge of learning from limited data and demonstrates its effectiveness across various tasks and models.

The MAML (Model-Agnostic Meta-Learning) algorithm is a meta-learning technique designed to enable rapid adaptation to new tasks with limited data. Meta-learning, also known as "learning to learn," focuses on training models that can learn new tasks quickly and effectively.

MAML specifically addresses the problem of few-shot learning, where the model needs to generalize from a small number of examples or adapt to new tasks with limited data. It aims to learn a good initialization for model parameters that can be quickly fine-tuned or adapted to new tasks with just a few gradient steps.

Here's a high-level overview of how the MAML algorithm works:

1. Initialization: The algorithm starts by initializing a base model with some random parameters.
2. Task sampling: For each iteration, a set of tasks is sampled from a task distribution. Each task consists of a small, labeled dataset for training and a small, labeled dataset for evaluation.
3. Inner loop: For each task, the base model is copied, and the model parameters are fine-tuned on the training dataset (usually with gradient descent) for a few iterations. This process is called the inner loop or inner optimization.
4. Outer loop: After the inner loop optimization, the model's parameters are updated based on the performance of the fine-tuned models on the evaluation datasets. The goal is to find model parameters that can be quickly adapted to new tasks.
5. Repeat: Steps 3 and 4 are repeated for multiple iterations, with the model gradually improving its ability to adapt to new tasks.

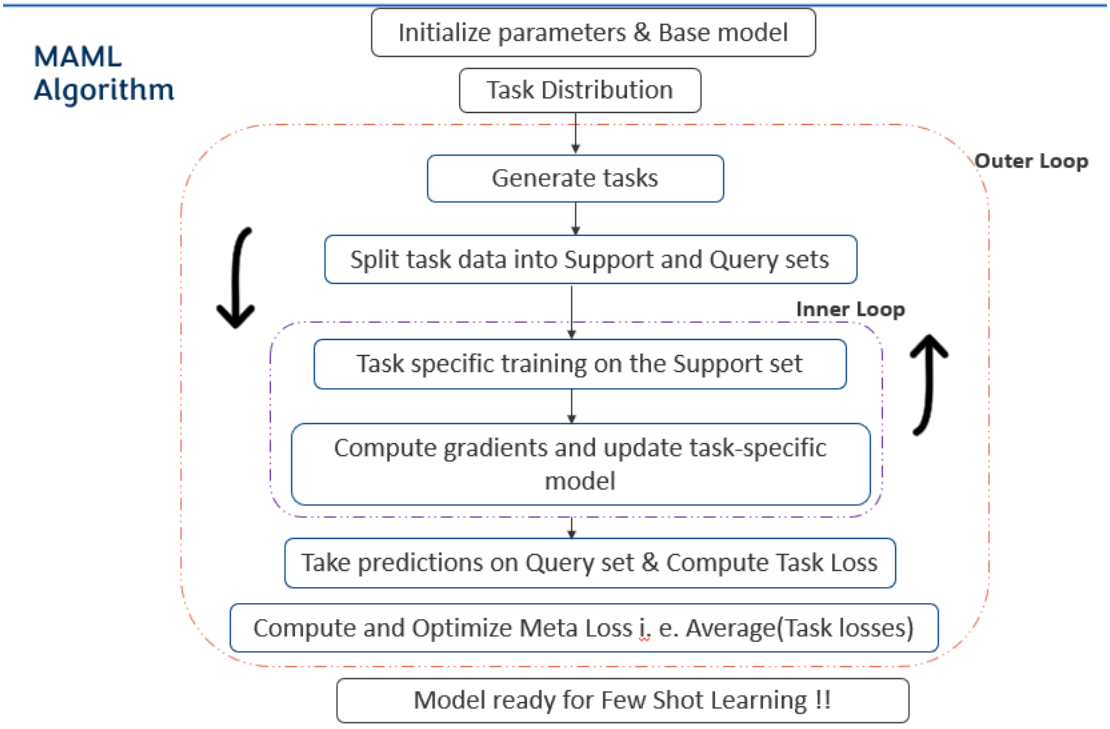


Fig: Flowchart and working of MAML

By iteratively updating the model's initialization based on how well it can adapt to different tasks, MAML learns a set of parameters that facilitate fast adaptation to new tasks with limited data. It can be applied to various machine learning models and has been used in computer vision, natural language processing, reinforcement learning, and other domains.

Algorithm 2 MAML for Few-Shot Supervised Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K datapoints $\mathcal{D} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (2) or (3)
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 2 or 3
11: **end while**

We aim to train models that can achieve rapid adaptation, a problem setting that is often formalized as few-shot learning. In this section, we will define the problem setup and present the general form of our algorithm. The goal of few-shot meta-learning is to train a model that can quickly adapt to a new task using only a few datapoints and training iterations. To accomplish this, the model or learner is trained during a meta-learning phase on a set of tasks, such that the trained model can quickly adapt to new tasks using only a small number of examples or trials. In effect, the meta-learning problem treats entire tasks as training examples. In this section, we formalize this metalearning problem setting in a general manner, including brief examples of different learning domains.

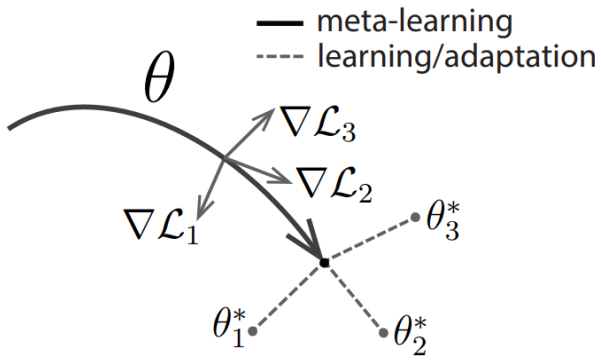


Figure 1. Diagram of our model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks.

CODE SNIPPET (MAML)

```
def compute_loss(model, data, params=None):
    # Assuming data is a tuple of (inputs, targets)
    inputs, targets = data
    # Use the provided model parameters or the current model parameters
    if params is None:
        outputs = model(inputs)
    else:
        outputs = model(inputs, params)

    loss = nn.CrossEntropyLoss()(outputs, targets)
    return loss

def inner_loop(self, task_support_set):

    fast_weights = [w.clone() for w in self.model.parameters()]


    for t_epoch in range(self.num_inner_updates):
        loss = compute_loss(model, task_support_set, params=fast_weights)
        gradients = torch.autograd.grad(loss, fast_weights, create_graph=True)
        # We set create_graph=True to allow for gradient calculations in the next step
        # Now we update the fast weights using the calculated gradients (applying the inner-loop learning rate)
        fast_weights = [w - inner_lr * g for w, g in zip(fast_weights, gradients)]

    return fast_weights
```

```
    def outer_loop(self):
        self.optimizer.zero_grad()
        task_losses = []

        for task in range(self.tasks):
            #print("\nWorking with task: {}".format(task + 1))
            support_set_X_batch, support_set_y_batch, query_set_X_batch, query_set_y_batch = self.dataloaders[task]

            if len(query_set_X_batch.shape) == 5:
                support_set_X_batch = support_set_X_batch.squeeze(0)

            if len(query_set_X_batch.shape) == 5:
                query_set_X_batch = query_set_X_batch.squeeze(0)

            support_set_X_batch =support_set_X_batch.to(self.device)
            support_set_y_batch = support_set_y_batch.to(self.device)
            query_set_X_batch = query_set_X_batch.to(self.device)
            query_set_y_batch = query_set_y_batch.to(self.device)
            #print("shapes support set ", support_set_X_batch.shape, support_set_y_batch.shape, sep = ' : ')
            fast_weights = self.inner_loop((support_set_X_batch, support_set_y_batch))


            loss = compute_loss(self.model, tuple(query_set_X_batch,query_set_y_batch), params=fast_weights)
            task_losses.append(loss)
            # This is where the second order derivatives are calculated.
            # We calculate the gradients of the query set loss with respect to the original model parameters
            meta_gradients = torch.autograd.grad(loss, self.model.parameters(), retain_graph=True)
            for p, g in zip(self.model.parameters(), meta_gradients):
                if p.grad is None:
                    p.grad = g
                else:
                    p.grad += g

        meta_loss = sum(task_losses) / len(task_losses)
        meta_loss.backward()
        self.optimizer.step()

        return meta_loss.item()
```

REPTILE:

The Reptile algorithm is another meta-learning technique that focuses on few-shot learning and rapid adaptation to new tasks. Like MAML, Reptile aims to learn a good initialization for model parameters that can be quickly fine-tuned to new tasks.

Here's an overview of how the Reptile algorithm works:

1. Initialization: Similar to MAML, Reptile starts by initializing a base model with random parameters.
2. Task sampling: For each iteration, a set of tasks is sampled from a task distribution. Each task consists of a small labeled dataset for training and a small labeled dataset for evaluation.
3. Inner loop: For each task, the base model's parameters are updated using gradient descent on the training dataset. However, instead of updating the parameters fully, Reptile performs a partial update by taking a step in the direction of the task-specific model parameters.
4. Outer loop: After the inner loop optimization, the base model's parameters are updated by averaging the partial updates across all the tasks. This update reinforces the common patterns across tasks, aiming to improve the model's generalization ability.
5. Repeat: Steps 3 and 4 are repeated for multiple iterations, gradually improving the base model's initialization for fast adaptation.

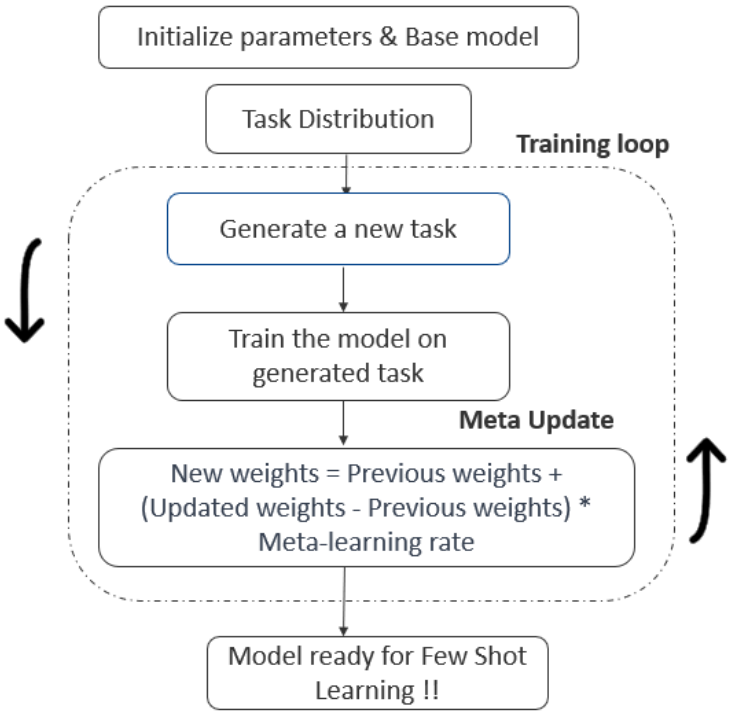


Fig: Flowchart and working of Reptile

The key difference between MAML and Reptile lies in the update mechanism. MAML performs a full update using the gradients from the inner loop, whereas Reptile takes a partial update towards the task-specific parameters. This difference affects how the algorithms adapt to new tasks and the generalization properties they exhibit.

Reptile has been used in various domains, including computer vision and reinforcement learning, and it provides an alternative approach to few-shot learning and meta-learning problems.

CODE SNIPPET (REPTILE)

```python
class ReptileCloneScheduled:
    def __init__(self, name, model,dataloaders, test_dataloader=None, tasks=100, n_shot=3, epochs=50,
        self.model = model
        self.device = device
        #print(self.model)
        #print(self.device)
        self.model = self.model.to(self.device)
        self.name = name
        self.tasks = tasks
        self.n_shot = n_shot
        self.epochs = epochs
        self.dataloaders = dataloaders
        #self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr_outer)
        self.test_dataloader = test_dataloader
        self.inner_lr = inner_lr
        self.meta_lr = meta_lr
        self.inner_steps = inner_steps
        self.inner_optimizer = optim.SGD(self.model.parameters(), lr=self.inner_lr)

    def inner_loop(self, task_support_set):

        X_train, y_train = task_support_set

        for t_epoch in range(self.inner_steps):
            self.model.train()
            preds = self.model(X_train)
            loss = nn.CrossEntropyLoss()(preds, y_train)
            self.inner_optimizer.zero_grad()
            loss.backward()
            self.inner_optimizer.step()

        return loss.item()
```

```python
def outer_loop(self, meta_lr_weight):

    meta_weights = {name: param.clone() for name, param in self.model.named_parameters() if 'running_mean' not in name and 'running_var'
    task_losses = 0.0
    update_directions = {name: 0 for name, _ in meta_weights.items()}

    for task in range(self.tasks):
        if task%1==0:
            print("\nWorking with task: {}".format(task + 1))
        support_set_X_batch, support_set_y_batch, query_set_X_batch, query_set_y_batch = self.dataloaders[task]

        if len(query_set_X_batch.shape) == 5:
          support_set_X_batch = support_set_X_batch.squeeze(0)

        if len(query_set_X_batch.shape) == 5:
          query_set_X_batch = query_set_X_batch.squeeze(0)

        support_set_X_batch =support_set_X_batch.to(self.device)
        support_set_y_batch = support_set_y_batch.to(self.device)
        query_set_X_batch = query_set_X_batch.to(self.device)
        query_set_y_batch = query_set_y_batch.to(self.device)
        #print("shapes support set ", support_set_X_batch.shape, support_set_y_batch.shape, sep = ' : ')
        task_losses += self.inner_loop((support_set_X_batch, support_set_y_batch))

        for name, param in self.model.named_parameters():
          if 'running_mean' not in name and 'running_var' not in name:
              update_directions[name] += (param.detach() - meta_weights[name])

    update_directions = {name: direction / self.tasks for name, direction in update_directions.items()}
    # Apply meta update
    with torch.no_grad():
        for name, param in self.model.named_parameters():
            if 'running_mean' not in name and 'running_var' not in name:
                param += meta_lr_weight * self.meta_lr * update_directions[name]
```

Experimental Setup

In order to conduct our study, we designed a series of experimental settings that involved the use of several base models: AlexNet, ResNet50, Viz-Transformers, and DensNet. We crafted 100 tasks utilizing the 17 training classes available in our dataset. Each task contained precisely three classes, chosen at random from the available 17.

Due to the constraints of the project, specifically time limitations, we opted to conduct 5-shot learning, where each task used exactly five images from each class for training.

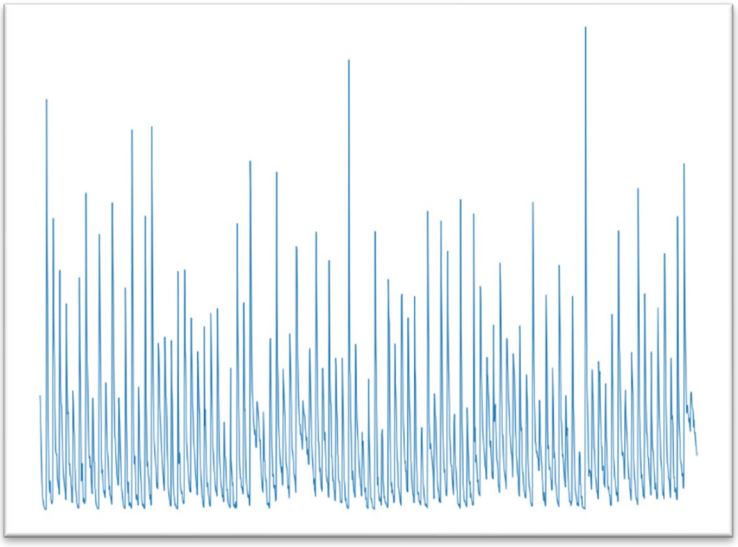
Baseline Creation: To establish a benchmark for comparison, we fine-tuned the base models with all 100 tasks in an iterative manner. For the test task, the baseline model was further fine-tuned using a few-shot learning technique, where it was trained on a small number of images from the final test classes. The testing was then conducted on the entire dataset, with the exception of the few images used for training.

Model-Agnostic Meta-Learning (MAML): In implementing MAML, we utilized both inner and outer learning rates of 0.0001 to promote stability during the learning process. We set the inner epochs to 2 and the outer meta-epoch to 10. To enhance the adaptability and performance of our model, we experimented with both second-order gradient calculations and first-order approximations.

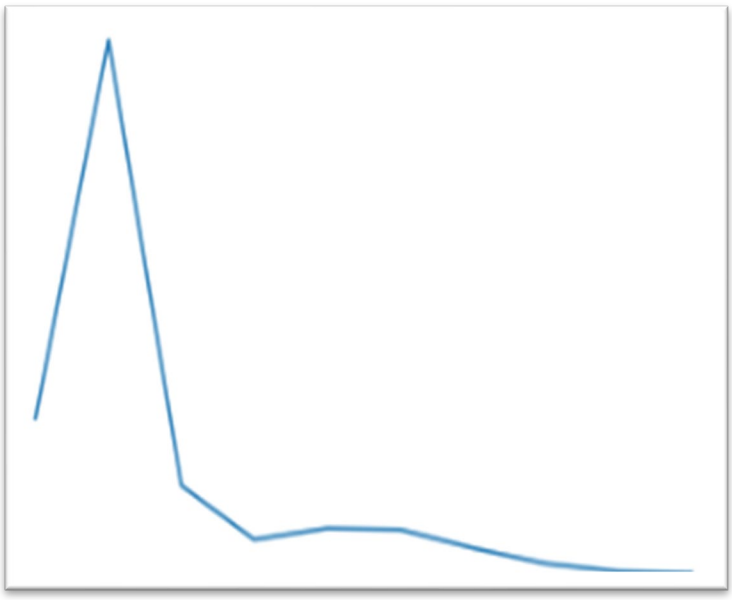
Reptile: In our usage of the Reptile algorithm, we adopted a similar learning rate to that used in MAML. For the meta learning rate, we followed the original authors' recommendation and implemented a linear scheduler.

In summary, our experimental setup encompassed multiple models and learning strategies, aimed at understanding how different configurations could influence performance. The results from these experiments will guide our next steps in developing a more refined model that can accurately diagnose various pulmonary conditions with minimal training data.

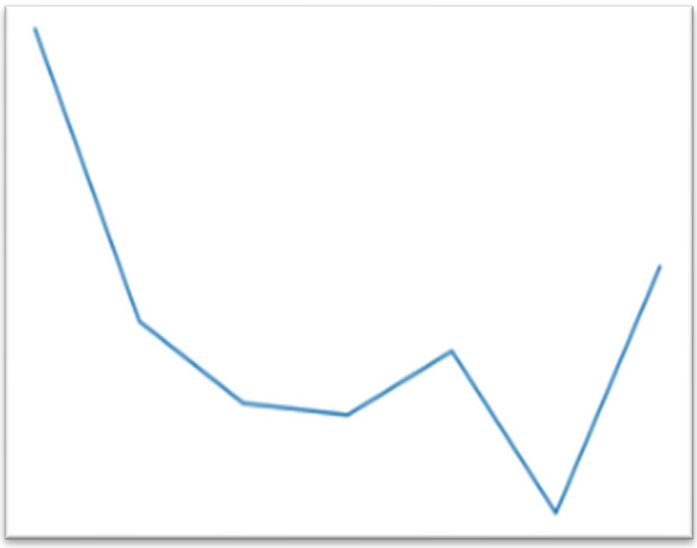
Results:



Baseline



MAML



REPTILE

The Meta Loss Vs Epoch graph illustrates the results of our baseline, MAML, and Reptile models. Upon observation, it may appear that the Baseline Losses are somewhat erratic.

This variability arises due to the nature of the tasks involved in our training set. We have 100 tasks, and for each task, the model loss decreases. However, the introduction of a new task causes a temporary spike in the loss. This is attributed to the fact that the model essentially 'starts afresh' with each new task, an event commonly referred to as catastrophic forgetting. Consequently, the model doesn't appear to effectively retain the knowledge gleaned from previous tasks.

In contrast, both MAML and Reptile exhibit a decrease in meta loss over epochs, showcasing their ability to learn and adapt across multiple tasks without succumbing to catastrophic forgetting.

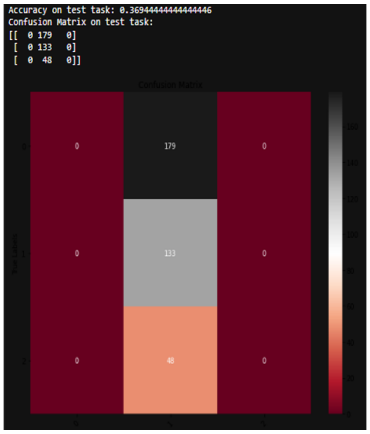
In every meta epoch, we execute 'n' inner epochs for each task. In this case, n equals 2.

We have incorporated mechanisms like saving the model that performs best according to a chosen metric, as well as early stopping. The latter helps prevent overfitting by stopping the training when the model ceases to improve significantly, thereby optimizing computational resources and ensuring model performance.

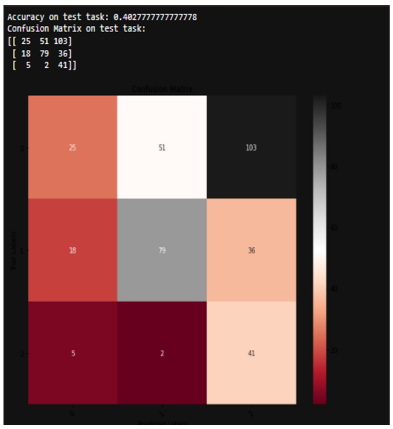
Results over Epochs.

TEST TASK CLASSIFICATION REPORT:
TRAIN EPOCHS = 2

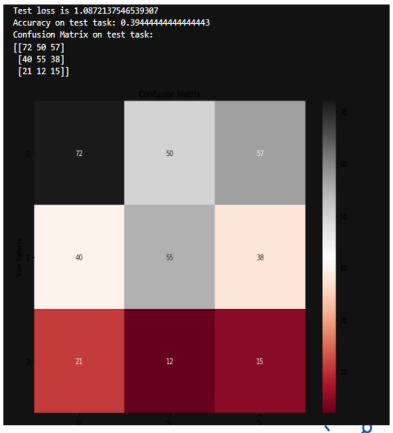
For the test task, we had the following test-set distribution:
Atelectasis: 179
Hernia: 133
Edema: 48



Baseline



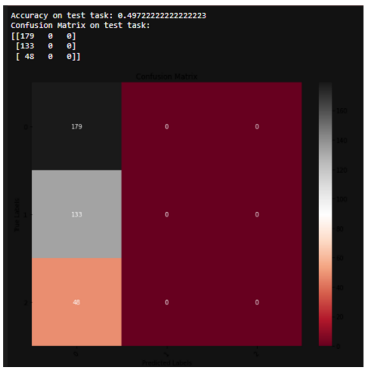
MAML



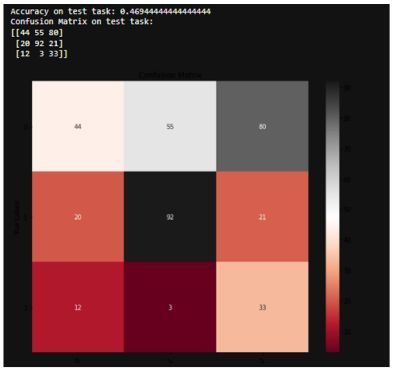
REPTILE

TEST TASK CLASSIFICATION REPORT:
TRAIN EPOCHS = 5

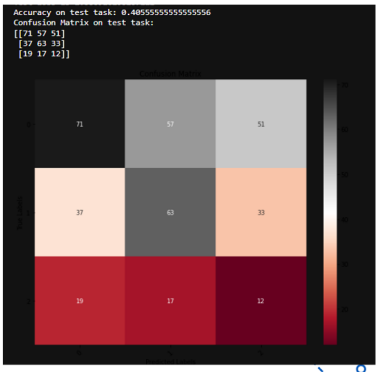
For the test task, we had the following test-set distribution:
Atelectasis: 179
Hernia: 133
Edema: 48



Baseline



MAML



REPTILE

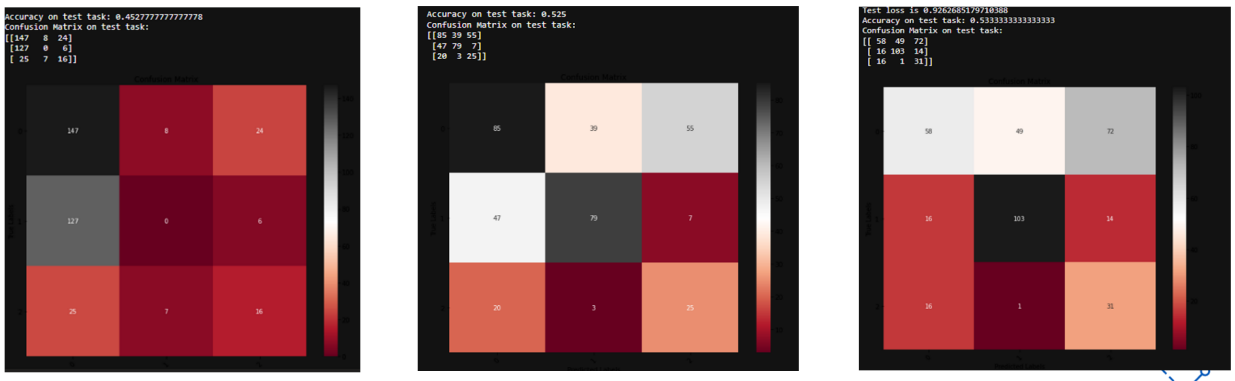
TEST TASK CLASSIFICATION REPORT:
TRAIN EPOCHS = 50

For the test task, we had the following test-set distribution:
Atelectasis: 179
Hernia: 133
Edema: 48

Baseline MAML REPTILE

TEST TASK CLASSIFICATION REPORT:
TRAIN EPOCHS = 100

For the test task, we had the following test-set distribution:
Atelectasis: 179
Hernia: 133
Edema: 48



Baseline MAML REPTILE

In our study, we encounter a test scenario that simulates the occurrence of new lung diseases, each represented by just five patients. Our observations indicate that, despite extensive training on 100 tasks, the baseline model struggles to form a reliable understanding of the classification decision boundary when only a few examples are provided.

The F1 score, a measure of model performance that combines precision and recall, experiences a slight improvement after 50 to 100 training epochs for the test task. However, the model still misclassifies the majority of the images, underlining its inadequate learning from the limited samples.

On the other hand, both MAML (Model-Agnostic Meta-Learning) and Reptile exhibit more effective adaptability to the new task. This efficiency holds even when the testing dataset is heavily imbalanced, an often challenging scenario in machine learning. As the number of training epochs increases from 2 to

100, the performance of both MAML and Reptile improves further, underscoring the benefits of more extensive training in meta-learning contexts.

In a head-to-head comparison, Reptile outshines MAML in terms of performance. Furthermore, the simplicity of Reptile, which only necessitates first-order gradient calculations, makes it a more straightforward algorithm to implement.

Our implementation of the Reptile algorithm required significantly less hyperparameter tuning to produce good results compared to MAML, reinforcing its practicality. This finding suggests that, in scenarios requiring rapid adaptation to new tasks with minimal examples, meta-learning techniques like Reptile could offer more efficient and effective solutions than traditional models.

Contributions:

Team Member	Contribution (%)	UBIT Name
Abhijeet Sugam	30	**asugam**
Ayush Utkarsh	40	**ayushutk**
Kumaramangalam, Vangavolu	30	**kumarama**

REFERENCES:

1. Medium.com
2. Implementing Focal Loss for multi-class classification in PyTorch? : MLQuestions (reddit.com)
3. Geekforgeek.org
4. Focal-loss-pytorch-implementation/function.py at master · namdvt/Focal-loss-pytorchhttps://github.com/namdvt/Focal-loss-pytorch-implementation/blob/master/function.pyimplementation · GitHub
5. Class slides
6. Piazza
7. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks (arxiv.org)
8. torch.Tensor — PyTorch 2.0 documentation
9. torchvision.ops.focal_loss — Torchvision 0.15 documentation (pytorch.org)
10. A1 report (asugam)
11. NIH Chest X-rays | Kaggle
12. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks by Chelsea Finn, Pieter Abbeel, Sergey Levine. https://arxiv.org/pdf/1703.03400.pdf
13. On First-Order Meta-Learning Algorithms by Alex Nichol and Joshua Achiam and John Schulman from Open AI
14. https://arxiv.org/pdf/1803.02999.pdf
15. https://lilianweng.github.io/posts/2018-11-30-meta-learning/
16. HOW TO TRAIN YOUR MAML by Antreas Antoniou https://arxiv.org/pdf/1810.09502.pdf
17. https://openai.com/research/reptile