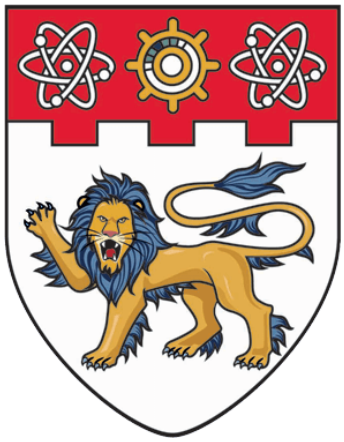

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Final Year Project Report

SCSE20-0322 Federated Learning Study

Submitted in Partial Fulfilment of the Requirements

for the Degree of Bachelor of Computer Science

of the Nanyang Technological University

Submitted By : Aloysius Chan Zhen Wu (U1722355K)

Supervisor : Asst Prof Zhao Jun

Examiner : Prof Cong Gao

School of Computer science and Engineering

2021

Abstract

Federated learning is a hot topic in the recent years due to the increased in emphasis for data privacy. Evidently, expert and specialised domain specific companies harbour large data assets required for a stronger machine learning model and these companies are generally not willing to disclose their restricted data to the public or other competing companies due to privacy infringement policies. Therefore, federated learning is one of the concepts introduced to allow training of machine learning models without direct knowledge of these companies' restricted datasets, maintaining anonymity. However, as model architectures gets more robust and increasingly complex, model architectures will also correspondingly have an increase in its overall size. This causes longer model training durations which can take up a lot of time for the overall federated learning training process. Moreover, it also increases the computational requirements on the federated learning devices.

This project will focus on constructing a federated learning framework to provide a proof of concept that with increasingly complex models, the overall federated learning process requires a much longer time for the training process. This is further emphasized by the lack of strong computational power on the targeted devices. Moreover, further experimentation such as the application of the state-of-the-art deep reinforcement learning (DRL) pruning can be applied to show the effects of model pruning on the whole federated learning process.

The demonstration is done with multiple python scripts to simulate the federated learning framework. Multiple raspberry PI devices will also be used to simulate companies hosting their respective training datasets. For the additional experimentation of DRL pruning, the pruning ratio can be adjusted on the pruning script to change the model architectures to specific targeted sizes.

Acknowledgements

I would like to express my sincerest gratitude to my project supervisor, Professor Zhao Jun, for his guidance, timely updates and checks on my progression of the project, and my in charge, Ms Zhao Yang for her guidance and round the clock response to my queries and problems with regards to the project.

I would also like to thank Ms Zhao Yang for making purchase and preparing the resources required for the project and spending her precious time, even on weekends and late-night hours to provide meaningful advices and clarifying any doubts that I have.

In addition, I would also like to thank my project examiner Professor Cong Gao and Professor Zhao Jun for being lenient and understanding of my unexpected decline in health and approving for an extension of the deadline to the latest possible time.

Table of Contents

Abstract 2

List of Figures 6

1. Introduction 9

1.1 Overview and Motivations 9

1.2 Objective and Scope of project 10

1.3 Report Organisation 11

2. Literature Review 12

2.1 Background Knowledge 12

2.1.1 Convolutional Neural network (CNN) 12

2.1.2 Recurrent Neural Networks (RNN) 13

2.1.3 Activation Functions 14

2.1.4 Federated learning 15

2.1.5 Reinforcement Learning 16

2.1.6 Gaussian Policy 19

2.1.7 Proximal Policy Optimization (PPO) 19

2.1.8 Generalized Advantage Estimation (GAE) 20

2.1.9 Model pruning 21

2.2 Related work 21

2.2.1 Storage Efficient and Dynamic Flexible Runtime channel Prunning via Deep Reinforcement Learning 21

3 System Implementation 22

3.1 Overview 22

3.2 Dataset and train-test split 22

3.3 Training devices specification 25

3.4 Federated Learning framework 26

3.4.1 Orchestrator 27

3.4.2 Chunking Process 29

3.4.3 Model Aggregation 30

3.5.1 Runtime importance predictor 31

3.5.2 Runtime agent 32

3.5.3 Static importance predictor 33

3.5.4 Static agent 34

3.5 Use case Diagram 36

3.6 Sequence Diagram 37

4.1 Demonstration Setup 38
4.1.1 Federated Learning setup and process 38
4.1.2 Deep Reinforcement Learning (DRL) Pruning Process 38
4.2 Demonstration Results 39
5 Conclusion 43
5.1 Project Conclusion 43
5.2 Future Works 43
References 44

List of Figures

Figure 1: Typical CNN architecture example 12
Figure 2: RNN architecture example 13
Figure 3: Sigmoid function 14
Figure 4: Activation functions (ReLU, ELU,SELU) 15
Figure 5: Centralized server approach for federeated learning process 15
Figure 6: Markov Decision Process MDP 16
Figure 7: Discounted cumulative reward 17
Figure 8: State value function 17
Figure 9: Q function or state-action value function 17
Figure 10: Policy gradient 18
Figure 11: Updated policy gradient 18
Figure 12 : Normalized gaussian curves 19
Figure 13: Clipped surrogate objective function 20
Figure 14: Generalized Advantage Estimation (GAE) equation 20
Figure 15: Balanced dataset 23
Figure 16: Dataloader and preprocessing 24
Figure 17: RPI3 specifications 25
Figure 18: RPI 4 specifications 25
Figure 19: Federated learning framework 26
Figure 20: Snippet of websocket consumer handler 27
Figure 21: Snipper of producer handler 28
Figure 22: Chunking code 29
Figure 23: Snippet of the websocket client script 29
Figure 24: Model weight average aggregation DRL framework 30
Figure 25: DRL based runtime pruning framework 31
Figure 26: Subnetwork of runtime importance predictor 31
Figure 27: Runtime agent 32
Figure 28: Static importance predictor 33
Figure 29: SELU equation 34
Figure 30: Static agent 34
Figure 31: Snippet of cifarnet with gating module 35
Figure 32: Use case diagram 36

Figure 33: Client sequence diagram 37
Figure 34: Server sequence diagram 37
Figure 35: Websocket server setup 38
Figure 36: Websocket client setup 38
Figure 37: Dynamic pruning ratio trade off 38

List of Tables

Table 1: MobileNetV1 conventional training results.....39
Table 2: CifarNet conventional training results.....40
Table 3: MobileV1Net Federated learning results.....40
Table 4: Cifarnet federated learning results.....41
Table 5: Cifarnet 50% sparsity trained federated learning results.....42

Introduction

1.1 Overview and Motivations

The modern world is increasingly digitalized than ever before with the internet as the core foundation of it. This has revolutionized the way the world has worked and evolved. To add to this great change, the recent covid-19 pandemic has showed the importance of digitalization. This has spurred many businesses to digitalize their business models and consequently, resulting in a greater uptake of technological means. Moreover, the emerging 5G technology will serve as the catalyst to take technology to even greater heights with potentials such as increased bandwidth, availability and coverage [1]. This will encourage the emergence of the Internet of Things (IoT) which will further improve productivity and drive effortless interaction with machines, creating more enhanced and easier living than ever before.

With the internet, the amount of data created every day is more than 2.5 quintillion bytes of data [2] and this will further accelerate as IoT becomes the new norm. This is great news for data scientists and analysts whereby with the increase in digital information available, larger datasets can be available for model training. However, reality is not as simple with rules and regulations in play. High quality information such as domain specific information collected and housed by domain specific companies are not publicly available due to privacy infringement policies. This results in a waste of valuable information for machine learning models for training.

In response, federated learning is created to adapt to privacy rules. This allows training against private domain specific datasets without knowledge of the dataset contents. This provides a new way to achieve more robust models with more domain specific datasets to provide more distinct feature maps for classification. An example of a much-established framework is the Syft library created by OpenMined [3]. The concept of federated learning is simple, instead of collecting the datasets to a central database, we can decentralize this system by sending the models out to the domain specific companies for training. This concept of decentralized training can also be really useful for applications in IoT.

The problem to federated learning revolves around its application in IoT, where targeted devices for training are smaller in size to accommodate for more human interaction conveniences such as lightweight and compactness in wearables. This results in constraints such as smaller memory capacity and limited computing capabilities which can result in large

federated learning training time. Consequently, making federated learning inapplicable to the context of IoT when considering more complex models.

A possible solution is to perform model pruning, to reduce the size of the model's architecture which consequently reduces the computation capacity of training the model. With smaller models or increased sparsity, we can reduce the number of multiply and accumulate (MAC) operations which will reduce the federated learning training time and solve the problem of applying more complicated models to IoT applications.

1.2 Objective and Scope of project

This Final Year project aims to create a federated learning framework to show the contrast between federated learning training and conventional model training. The results should show the difference in terms of training accuracy and training time. Additionally, state-of-the-art model pruning methodology is also applied on top of this framework for additional experimentation. To achieve model pruning, we will tap on a previous research methodology created that outperforms conventional pruning methods.

Users can create different types of models to host on the server script while additionally, if pruning is performed, they are also able to adjust the pruning ratio to prune the created model. Federated Learning will be done on targeted devices with the copy of the global model and then subsequently, aggregating the new model weights back on the main global model hosted on the server. This aggregated global model is then evaluated to show if there are improvements against the original unaggregated global model. This set of actions are also tested on a 50% sparsity trained model.

1.3 Report Organisation

The report will be organised into 5 sections:

Section 1 : Highlight on the overview, motivations, objectives and scope of this project

Section 2 : Summary on the research carried out about the background knowledge for the project as well as other form of related works.

Section 3 : Project's framework and detailed development process.

Section 4 : Discuss on experimental results to conclude on the effectiveness of Federated learning (FL) and Deep Reinforcement Learning (DRL) experimentation

Section 5 : Conclusion for the report and recommendation for future works.

2. Literature Review

2.1 Background Knowledge

2.1.1 Convolutional Neural network (CNN)

Convolutional Neural Networks (CNN) is a subset of the myriad of machine learning algorithms. Mainly, they are made up of an input layer, one to many hidden layers and a classification layer. Hidden layers include convolutional layers (hence, convolutional networks), pooling layers which are used to reduce the size of the feature maps, dropout layers and many others. The classification layer is typically made up of fully connected layers (dense layers) followed by an activation function. An example can be seen in Figure 1.

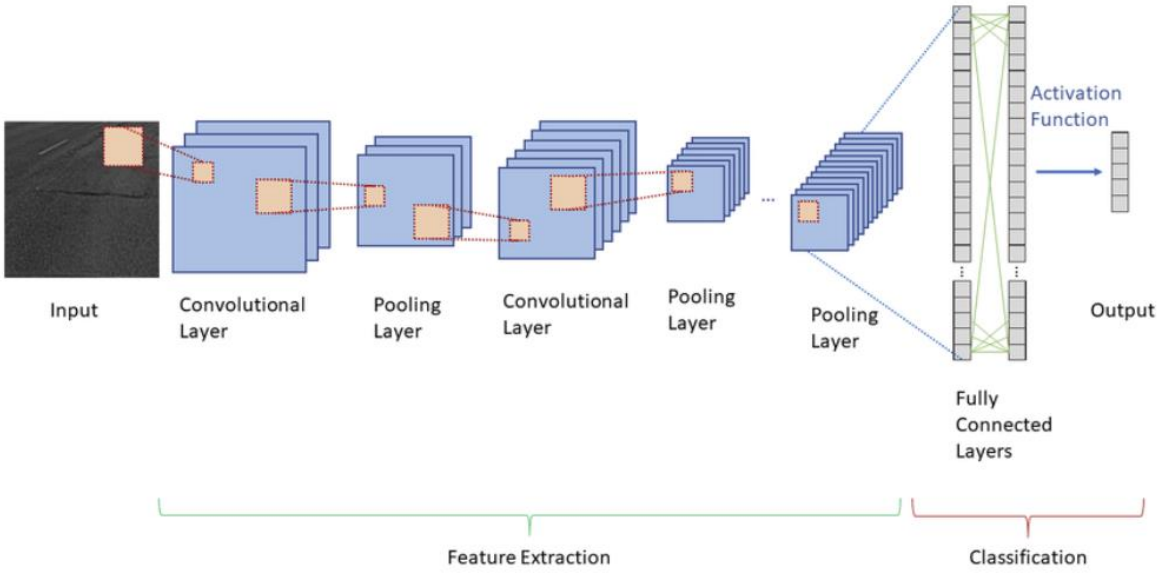

Figure 1: Typical CNN architecture example [4]

In each layer, there can be multiple channels based on the number of feature maps desired. These channels can be defined during the model building process. With larger number of channels or feature maps, the model is able to obtain more information on the input. Typically, the first input channel is 3 which represents the 3 color tones, Red, Green and Blue (1 if we consider grayscale images). Normally, less channels are used in the initial stage of the model as the feature maps can be large and computationally expensive during the filter convolution process, plus the feature maps are more general and less detailed and less class defining. In the later stages of the model, more feature maps are used to identify distinct features that are more class defining. Since the feature maps are also smaller due to the pooling layers, it is also less computationally expensive to do filter convolution.

In most cases, CNN are used for image classification and computer vision tasks depending on the classification layer. In cases where a soft-max function is used at the classification layer, we can perform multiple class classification based on a given input image.

2.1.2 Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) are another set of neural networks that are good for sequence data such as those of time series and natural language (such as word sequences). This means that unlike other neural networks like the Convolutional Neural Network (CNN) or the FCNN (Fully Connected Neural Network), the output is based on the current and previous inputs. A typical RNN can be seen in figure 2. RNN shares parameters across each layer of its network whereas compared to CNN or FCNN, each layer has its own weight parameter. Another notable difference is that the length of the input is not fixed.

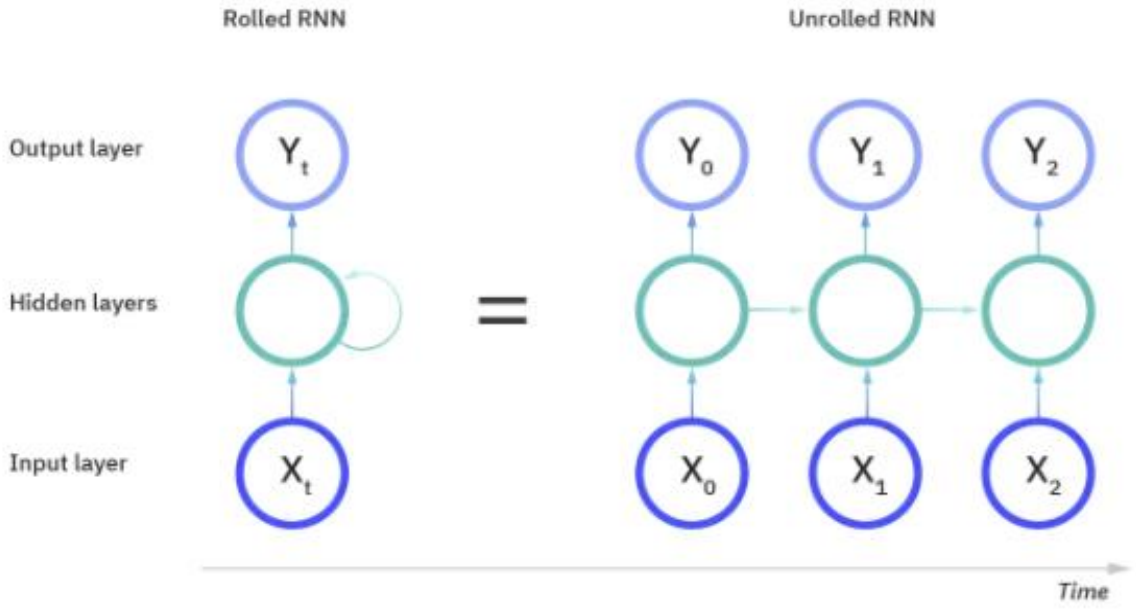

Figure 2: RNN architecture example [5]

RNN trains its weights using backpropagation through time (BPTT) to determine the gradients, largely hinging on the concept of the mathematical chain rule. BPTT, therefore, sums the errors at each time step since they share parameters across each layer. This will ultimately cause BPTT to run into 2 problems, namely, vanishing gradients and exploding gradients. The former is when the gradient is too small and it continues to become smaller at each time step through chain rule multiplication. Constant multiplication of values < 1 will converge to 0. Similarly for the latter, when we perform repetitive multiplication of values >1

will tend to infinity. Therefore, improved versions of the RNN such as the Long-Short Term Memory (LSTM) model is introduced to address these problems.

2.1.3 Activation Functions

Activation functions are used to provide non-linearity in a neural network since each node connection is typically a linear function of output = input*weight + bias. Activation functions should also be differentiable as their derivatives are used in the backpropagation algorithm to adjust the model's weights.

There are many types of activation functions, typically activation functions like sigmoid function or soft-max function are used at the classification layer to provide probabilities for binary or multiclass classification respectively. An example of a sigmoid function can be seen in figure 3.

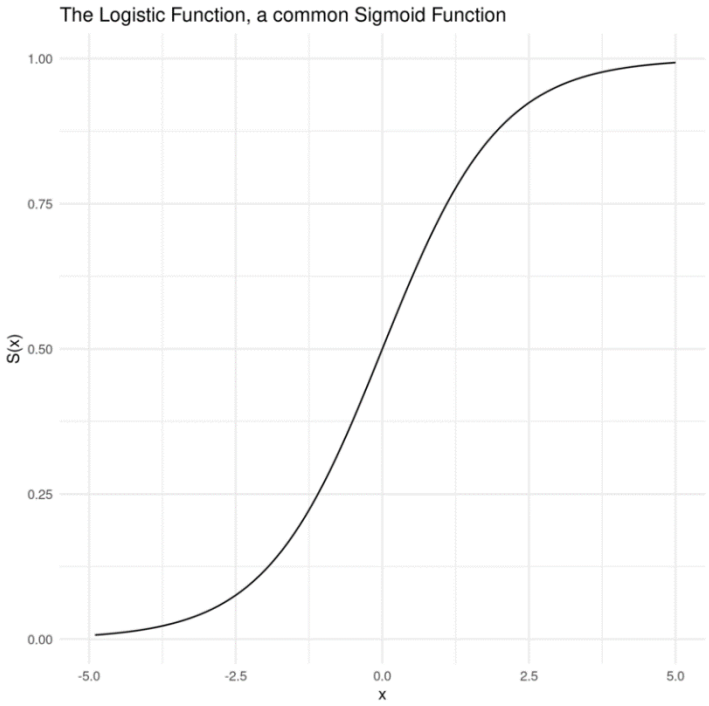

Figure 3: Sigmoid function [6]

Other activation functions also include Rectified Linear Unit (ReLU) or Scaled Exponential Linear Unit (SELU) (both shown in figure 4) that are used with the hidden layers to provide nonlinearity. Despite the fact that ReLU does not suffer from Vanishing gradient issue (gradient is always 1 when inputs are positive), it suffers from dying neurons when the inputs are negative where the derivative of ReLU is zero. In such cases, backpropagation cannot be performed and the learning for that neuron does not happen. Therefore, other modifications

like the SELU are used to provide a small gradient on negative inputs to prevent dying neurons.

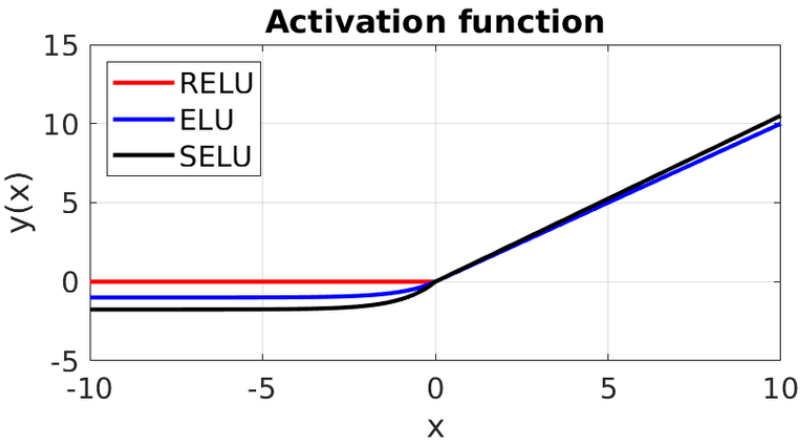

Figure 4: Activation functions (ReLU, ELU,SELU) [7]

2.1.4 Federated learning

Federated learning is the concept of decentralizing the machine learning training process by removing the need to train a specific model at a single location. Instead of the conventional methodology of collecting all the datasets into a single location for the model training, the model is sent over to different sites – different centers and hospitals as per figure 5, where the datasets reside and training is then performed on the respective datasets at their respective locations.

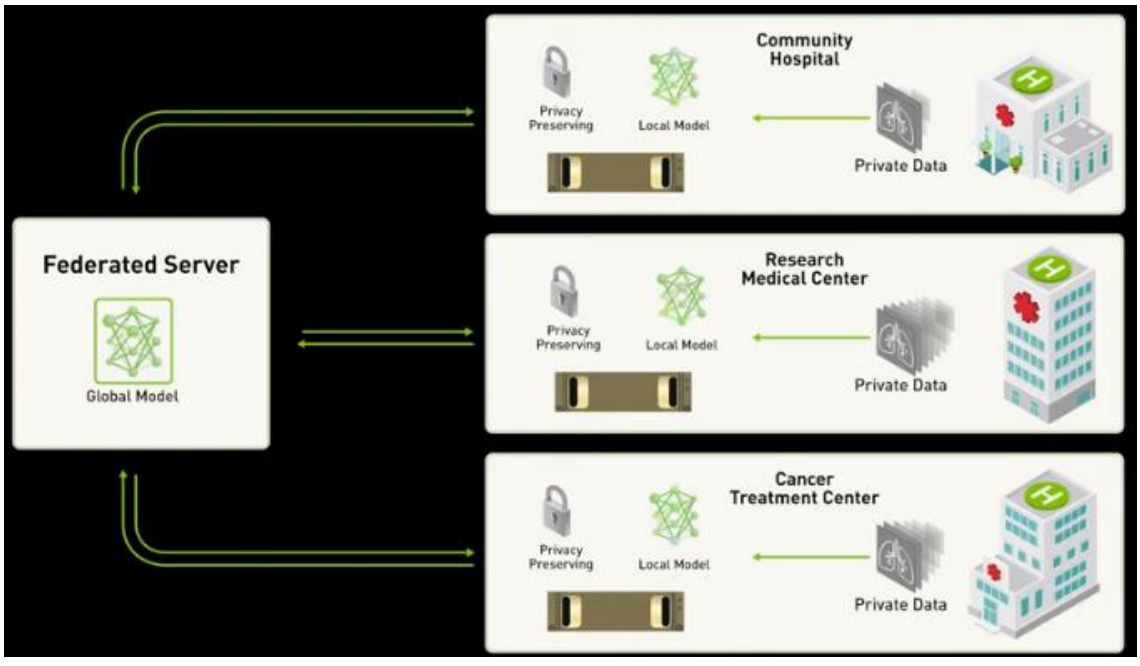

Figure 5: Centralized server approach for federated learning process [8]

2.1.4.1 client server federated approach

A client-server federated approach, similar to the diagram in figure 5, makes use of a centralized server (the federated server) to host the global machine learning model and each participating companies (the various centers and hospital) will be given a copy of this global machine learning model to train on their own private datasets. The locally updated version of the model within the client devices is then transferred back to the centralized server for aggregation where the weights of each locally trained model is averaged and updated against the global model.

2.1.5 Reinforcement Learning

Reinforcement learning is a field of machine learning that does not require human interaction for its machine learning process. In essence, an agent is used to interact with the environment along with a goal that the agent must achieve by taking specific actions available to it. Learning is then achieved through cumulative errors and rewards. The sequence of actions that the agent takes based on its state and environment is called a policy. This whole process can be formalised using a concept called the Markov Decision Process (MDP)

2.1.5.1 Markov Decision Process (MDP)

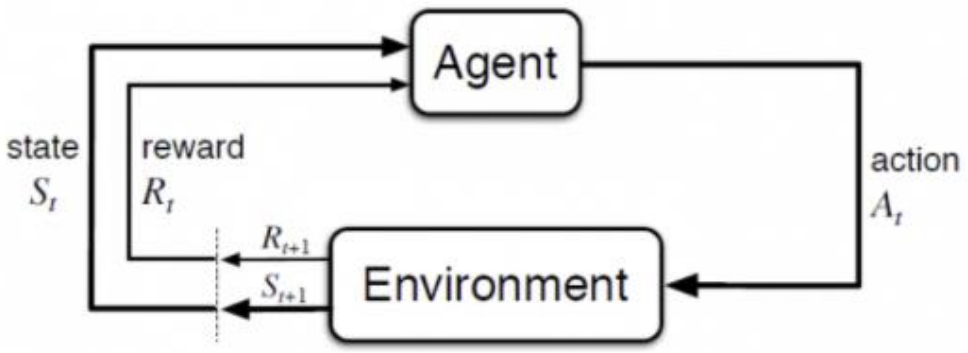

Figure 6: Markov Decision Process MDP [9]

Figure 6 shows a simple MDP whereby agent at a current state at time t, S_t, will have performed an action ,A_t, at time t as well, which will affect the environment the agent is in. This will in turn result in a new state, S_t+1 at time t + 1 and a reward value, R_t+1, will be generated for the agent to understand whether the action taken at time t is desirable or not. The overall objective of the agent is to maximise the total cumulative reward collected until it reaches the goal state.

Given that immediate reward is much favourable over distant future rewards, a discounting factor is thereby introduced as a weighted variable to reduce the weights of the rewards at

future timestamps. The discounted cumulative reward function can be seen in figure 7, where ϒ represents the discounting factor, R_t is the reward obtained at time t and k is the number of timesteps taken to reach the goal state.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Figure 7: Discounted cumulative reward [9]

2.1.5.2 State value function

A state value function will feedback the agent the value of the state the agent is in. In simpler words, how good is the current state for the agent to be in. This can be calculated by the expected reward, Eπ at each state if the agent follows a policy π. An optimal policy can be acquired if the value function is maximised for each state.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}$$

Figure 8: State value function [9]

2.1.5.3 State – action value function

A state-action value function or q-value will take into account an additional factor, that is the agent's action, with respect to the particular state to calculate the expected reward given the state at time t, S_t, action at time t, A_t, under the policy π (figure 9).

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Figure 9: Q function or state-action value function [9]

2.1.5.4 Actor-critic

An actor-critic simply uses a neural network model to help maximize an objective function. An objective function is a function that is defined by the total amount of rewards achieved based on the policy taken by the agent. The actor-critic works towards finding a set of weights of its neural network to help maximize this objective function. Therefore, in order to maximize this objective function, we have to look at the gradient of the objective function or also known as the policy gradient in figure 10.

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau^i)$$

Figure 10: Policy gradient [10]

With reference to figure 10, the policy gradient simply states that the gradient of the objective function is the average of all m episodes following a policy πϴ where each episode of m is the sum of the steps that compose it (T). At each of this step t, we simply compute the derivative of the logarithmic of the policy and multiply it by the return reward R. In other words, if the total return reward, R, is high given the trajectories, m, then the agent is more confident that theta, policy gradient, is progressing in the right direction.

With the knowledge of the policy gradient, we can add in a baseline, b_t , for reference for the model and rewrite the whole equation as in figure 11.

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T}\nabla_\theta \underbrace{\log \pi_\theta(a_t|s_t)}_{\text{Actor}}\underbrace{(Q(s_t,a_t) - V_\phi(s_t))}_{\text{Critic}}$$

Figure 11: Updated policy gradient [10]

In comparison to figure 10, the updated policy gradient revises the return reward to the difference between the Q value of the state (Return reward) and the state value (baseline, b_t). This gives us the actor and critic component of what the actor and critic agent aims to implement.

2.1.6 Gaussian Policy

In gaussian policy, the policy takes on a continuous action space as compared to the usual discrete action space. This continuous action space is dictated by the use of a gaussian distribution for which we get our set of actions an agent can take by sampling this distribution.

A gaussian policy is, therefore, parameterized by a mean (mu) and a standard deviation (sd) or variance σ (sd squared) . With reference to figure 12, when the standard deviation is low, the gaussian curve has a smaller width, this means that the action space is more concentrated to a specific action region. In other words, the standard deviation is used to control the degree of exploration of the agent in its actions taken. The mean controls where the gaussian curve is centered at, which therefore, controls where the best action should be concentrated at.

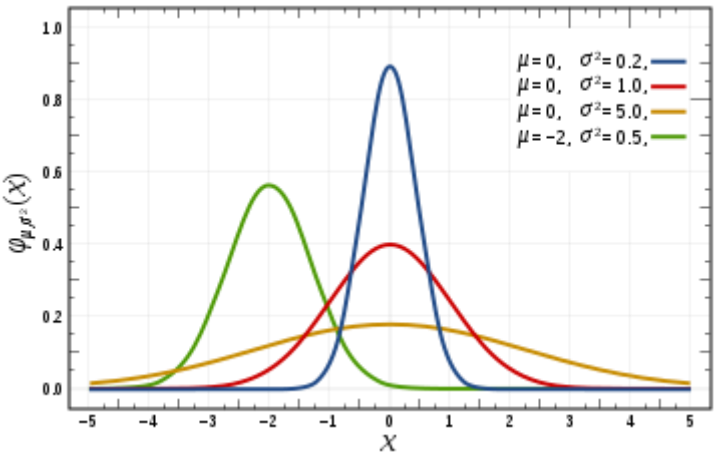

Figure 12 : Normalized gaussian curves [11]

2.1.7 Proximal Policy Optimization (PPO)

PPO is used to optimize the actor-critic model or in other words, optimize the policy objective function. The main idea of PPO is to prevent the agent from having too large a policy update. Too large an update can cause too much variability in the agent training and too small an update can cause the agent training to be too slow. Therefore, by limiting the update at each training step, we can improve the stability of the agent. This is done so by introducing a clipped surrogate objective function as seen in figure 13.

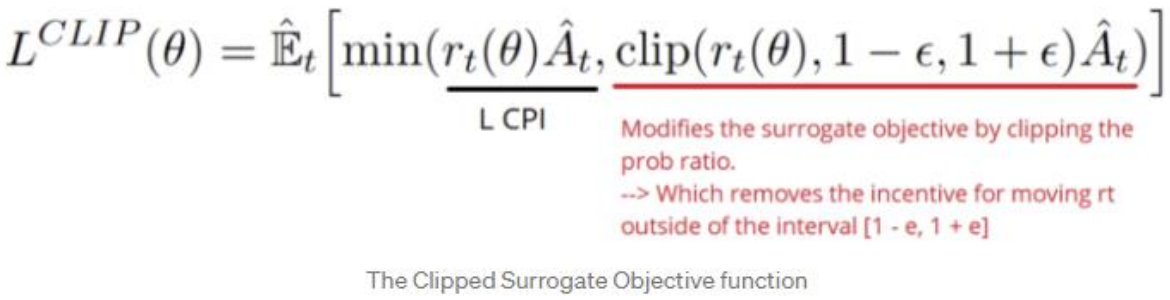

Figure 13: Clipped surrogate objective function [12]

A_t hat, represents the advantage of taking a particular action over other actions at the current state and r_t represents the ratio of the new policy over the old policy. r_t is used in replacement of the logarithmic probability of policy since they both resemble the same measurement of calculating the probability of action taken at the state t. The clip function is used to limit the update of the policy with epsilon as a hyperparameter. This means that the agent cannot update beyond the clipping points which establishes that the new policy cannot be too much better than the old policy. The minimum of both is then taken to update the policy objective function.

2.1.8 Generalized Advantage Estimation (GAE)

GAE creates an estimator with lower variance at the cost of adding some form of bias. GAE is typically used to update the policy gradient due to its significantly lower variance. The equation of GAE can be seen in figure 14.

$$\hat{A}_t^{\text{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Figure 14: Generalized Advantage Estimation (GAE) equation [13]

γ and λ are essentially discounting factors of the value function δ. By controlling γ, we control the discounting factor of the future rewards, thus, giving more value to immediate rewards from immediate actions which creates a larger bias but lowers the variance of the rewards since less future actions are effectively considered. This is also dependent on the number of steps in the future we are considering given the limit of ι, in computational case, cannot be infinity . λ works as the secondary discounting factor on top of γ on the whole value function.

2.1.9 Model pruning

Model pruning is the process of removing weights layer by layer in a neural network. This allows the trimming of weights that are not significant in the objective of the neural network, thereby reducing the number of learnable parameters. This helps to reduce the size of the model and also reduce the time taken for computational process (Multiply and accumulate operations (MACs) decreases). The decision to trim the weights can be done by sorting the weights in a descending order and pruning off the weights based on a predefined sparsity level. Sparsity level is simply the percentage of weights to be pruned. A threshold can also be used to prune off weights below the threshold as they would be deemed less significant.

A pruned model is especially useful during deployment phase due to the faster prediction process with less MACs in the model. Also, the smaller sized model can help to account for memory constraints during deployment on targeted devices.

2.2 Related work

2.2.1 Storage Efficient and Dynamic Flexible Runtime Channel Pruning via Deep Reinforcement Learning

Jianda Chen [14] proposes a deep reinforcement learning (DRL) based framework to perform efficient runtime channel pruning of convolutional neural networks. This framework allows model pruning in 2 cases, dynamic pruning based on input instance at runtime and static pruning. The static pruning component provides an additional edge over existing runtime pruning methods to reduce the parameters storage consumption of the runtime approach, therefore, allowing a trade-off choice between better dynamic flexibility or more efficient storage consumption. This provides greater allowance for more intensive pruning on the model.

The above study focused on the framework implementation of DRL channel pruning. This framework is, therefore, greatly tapped upon to aid as additional experimentation for this project to prove the efficacy of this framework on the federated learning use case.

3 System Implementation

3.1 Overview

Federated learning makes use of the concept of decentralizing the machine learning process. Therefore, to demonstrate the federated learning process, this project designs a federated learning framework, implemented by python Websocket library to establish communication between a single federated learning server script with multiple client scripts. Each client script is initiated using a raspberry PI to establish a communication with the server. Within the server script, the DRL framework is optionally added to help in model pruning before distribution of the original model copy to the clients. Within the Client script, the user is able to select the type of model to request from the server and subsequently perform training on the received copy of the global model. The time taken for the full federated learning process is then recorded on the client side. The full federated learning process includes the time taken to receive the model from the server, the time taken to train the model locally on the client device and the time taken to send the trained model from the client device back to the server for aggregation against the global model. Therefore, do note that the full federated learning process does not include the privacy preserving technique as that should be additive to the overall time taken and independent of the model architectural size.

3.2 Dataset and train-test split

In lieu of the covid-19 pandemic, the covid-19 dataset is chosen from [15] which consist of about 3616 covid-19 positive samples and about 18000 covid-19 negative samples. The dataset chosen consist of x-ray images for covid-19 classification. A supervised learning training is used to distinguish whether the x-ray image is covid-19 positive or negative (binary classification).

The total number of images used is 7232 and a further 70-30 train-test split is performed on these set of images. To ensure that the model is not biased against either class, an equal number of positive and negative covid-19 images is used. This is done by down sampling the majority category using random sampling.

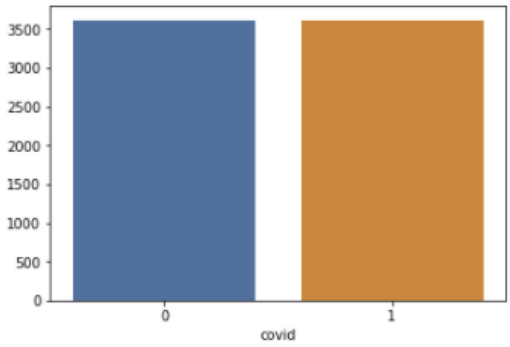

Figure 15: Balanced dataset

Figure 15 shows the count plot of the 2 classes. 0 represents that the x-ray image does not have covid-19 (negative) and 1 represents that the x-ray image has covid-19 (positive).

The breakdown of the dataset is as follows:

- 30% for testing dataset which is 2168 images.
- 70% for training dataset which is 5064 images.
    - 30% for global model base training on the server side which is 2170 images.
    - 20% for RPI 3 training on the client side which is 1447 images.
    - 20% for RPI 4 training on the client side which is 1447 images.

A 70-30 train test split is decided upon to account for more percentage of the images for testing. This is to allow a better accuracy representation against the trained dataset. If the number of testing dataset is too small, a high accuracy can be easily achieved and this accuracy will not be accurate in judging the model's actual performance.

3.3 Pre-processing techniques and dataset setup

There are 2 types of models used in this project, namely the MobileV1Net and the Cifarnet. The reason for using these 2 models is in line with the implementation of the DRL pruning which will be explained in the latter section of the report. Since the input shapes of the models are different due to the difference in model architectural sizes, pre-processing has to be done on the input images.

Pre-processing techniques include the resizing of the images to shape 32x32x3 for the Cifarnet and shape 224x224x3 for MobileV1Net. A center crop is applied to the image to crop off the padded white corners in some of the images. Subsequently, the image is normalized before being used as input for the model training.

A data loader is also used to load the images and split the dataset into specific batch sizes. This will allow the tuning of the batch size of the dataset to accommodate for memory limitation on the training device. Figure 16 shows the implementation of the data loader with the pre-processing of the images using the torch and torchvision library.

```
batch_size = 16
train_img_base_path = './train_covid_folder/'
test_img_base_path = './test_covid_folder/'


transform_img = transforms.Compose([
    transforms.Resize(32),
    transforms.CenterCrop(32),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485,0.456,0.406],
                         std=[0.229,0.224,0.225])
])
train_dataset = torchvision.datasets.ImageFolder(
    root = train_img_base_path,
    transform=transform_img
)
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True
)

test_dataset = torchvision.datasets.ImageFolder(
    root=test_img_base_path,
    transform=transform_img
)
test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=batch_size,
    shuffle=True
)
```

Figure 16: Data loader and pre-processing

3.4 Training devices specification

Locally on the server, the model is trained using an NVIDIA GeForce GTX 1050 TI GPU. On the client side, a Raspberry PI 3 (RPI3) and a Raspberry PI 4 (RPI4) is used. Their respective specifications can be seen in figure 17 and figure 18 respectively.

```
pi@raspberrypi:~ $ lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Vendor ID:             ARM
Model:                 4
Model name:            Cortex-A53
Stepping:              r0p4
CPU max MHz:           1200.0000
CPU min MHz:           600.0000
BogoMIPS:              38.40
```

Figure 17: RPI3 specifications

```
pi@raspberrypi:~ $ lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Vendor ID:             ARM
Model:                 3
Model name:            Cortex-A72
Stepping:              r0p3
CPU max MHz:           1500.0000
CPU min MHz:           600.0000
BogoMIPS:              270.00
```

Figure 18: RPI 4 specifications

The RPI4 has a slightly faster CPU max frequency than the RPI3, therefore, training on the RPI4 is much faster as compared to the training on the RPI3. This can be observed in the demonstration results later.

3.5 Federated Learning framework

The federated learning framework implemented is seen in figure 19. This will depict the flow of the machine learning model from server to client and vice versa.

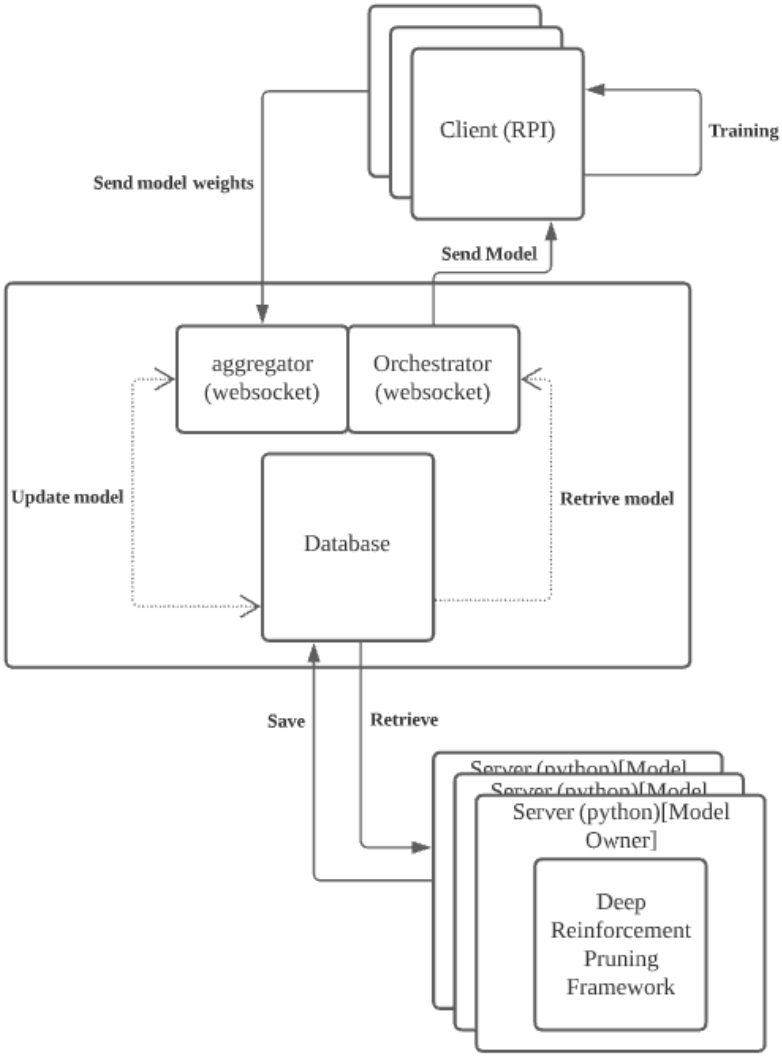

Figure 19: Federated learning framework

On the server side, a user can create a machine learning model with the options of DRL pruning. This newly created model is then labelled as the global model which can then be trained against a set of local training dataset for global model base training. After training, the global model can then be stored in a database folder to be hosted on the Websocket server or the orchestrator.

On the client side, a user can initiate the federated learning process by initiating the Websocket client script. This will query the Websocket server for a copy of the exposed global model. Upon receiving the copied global model, the client, in this case represented by

a RPI, is able to perform further training of the copied global model against its own local training dataset. This will subsequently update the weights of the copied global model. After which, the updated global model is send back to the server side for aggregation.

Back on the server side, the Websocket server or the aggregator will perform the aggregation of the received updated global model and the hosted global model. Aggregation involves averaging the weights of both models for each layer. Finally, the aggregated model is overwritten as the new global model.

3.4.1 Orchestrator

The orchestrator and aggregator make use of a consumer and producer handler to handle concurrent communication between multiple clients against a single server. This is achieved via the python's asynchronous process.

The orchestrator script runs on the Websocket server script which consist of a consumer and producer handler. This implementation is inspired from the Syft library which makes use of the asynchronous process of python. The consumer handler (figure 20) is in charge of incoming Websocket connection and queueing the message from the client in a First In First Out (FIFO) manner.

The producer handler (figure 21) is in charge of processing the message it receives from the queue. A switch statement or in python's case, 'if' statements are used to interface between different message types.

```
#consumer handler
async def _consumer_handler(self,websocket : websockets.WebSocketCommonProtocol):
    """This handler listens for messages from WebsocketClientWorker
           objects.
       Args:
           websocket: the connection object to receive messages from and
               add them into the queue.
    """
    try:
        self.connected_clients.add(websocket)
        while True:
            msg = await websocket.recv()

            await self.broadcast_queue.put(msg)
    except websockets.exceptions.ConnectionClosed:
        self._consumer_handler(websocket)
```

Figure 20: Snippet of Websocket consumer handler

```python
#producer handler
async def _producer_handler(self, websocket : websockets.WebSocketCommonProtocol):
    """This handler listens to the queue and processes messages as they
            arrive.
       Args:
            websocket: the connection object we use to send responses
                       back to the client.
    """

    while True:
        #get the message from the queue
        message = await self.broadcast_queue.get()
        print(message)

        if isinstance(message, str):
            self.nextMsg = False
            message = message.strip()

        if message == 'mvl':
            print('Fetching mvlnet on request...')
            # critical section
            await self.lock.acquire()
            chunk_dict = self.process_chunking('mvl')
            self.lock.release()
            for chunk_idx, byte in chunk_dict.items():
                await websocket.send(byte)
            await websocket.send('end')

        elif message == 'cfn':
            print('Fetching cifarnet on request...')
            # critical section
            await self.lock.acquire()
            chunk_dict = self.process_chunking('cfn')
            self.lock.release()
            for chunk_idx, byte in chunk_dict.items():
                await websocket.send(byte)
            await websocket.send('end')
```

Figure 21: Snipper of producer handler

The Websocket server runs in an asynchronous thread for which, the keyword ‘await’ provides context switching from one process to another process. This allows multiple clients to connect to a single Websocket server and share on the same resources created in the thread. In order to prevent the case of race condition where a client copies over an older version of the global model, a mutex (lock) is used to safeguard this critical section of access to the global model.

3.4.2 Chunking Process

Since the model can be large in size, the Websocket does not provide enough bandwidth to transfer the whole file in 1 attempt. Therefore, a chunking process is implemented as seen in figure 22. This will help to split the model save file into smaller byte sized chunks before sending them over via the Websocket to the client side.

```
chunking_split = 2**19
chunk_dict = {}
counter = 0
for i in range(0,len(byte_data),chunking_split):
    print('This is the size of the {} byte : {}'.format(counter, sys.getsizeof(byte_data[i:i+chunking_split])))
    chunk_dict[counter] = byte_data[i:i+chunking_split]
    counter += 1
```

Figure 22: Chunking code

Over on the client side, the Websocket client will initiate the communication to the Websocket server. Subsequently, retrieving a copy of the global model from the Websocket server (figure 23). Due to the chunking process, a while loop is used to establish a continuous connection to the server to receive the chunking bytes of the model's save file. A response 'end' is used to signal the end of model data byte transmission. Subsequently, the Websocket client is able to continue the training process on the received global model.

```
async def connect_recv():
    print('entering connect loop?')
    global model_type
    global model_save_path
    uri = 'ws://192.168.1.149:8765'
    if os.path.isfile(model_save_path):
        os.remove(model_save_path)
    async with websockets.connect(uri) as websocket:

        if model_type == 'cfn':
            print('connected but no send?')
            await websocket.send('cfn')
        elif model_type == 'mvl':
            await websocket.send('mvl')

        model_byte_dict = {}
        counter = 0
        while(1):
            response = await websocket.recv()
            if response == 'end':
                break
            else:
                print('This is the size of the {} byte : {}'.format(counter,sys.getsizeof(response)))
                model_byte_dict[counter] = response
                counter += 1

        with open(model_save_path,'wb') as outs:
            for chunk_id,m_bytes in model_byte_dict.items():
                outs.write(m_bytes)
```

Figure 23: Snippet of the Websocket client script

3.4.3 Model Aggregation

After training on the client device, the trained model is then sent back to the Websocket server for model aggregation. Likewise, the chunking process is used to send the model bytes back to the server side.

In this case, average aggregation on the models' weights is performed layer by layer for each model's state dictionary. This can be seen in figure 24 where the weights sdMain and sdTemp are averaged equally. In a more practical scenario, the trained model received from a client may have a lower performance compared to the global model due to the different quality types of training datasets. This may cause the global model to decline in performance after aggregation. Hence, a weighted average can be used instead to distinguish different model quality received from the client.

```
def aggregation_cs(self,model_type):

    if model_type == 'mv1':
        PATH = "./database/mobilenetv1_recv.pth"
        Main_PATH = "./database/mobilenetv1_global.pth"
        model_temp = get_mv1net()
        model_main = get_mv1net()
    elif model_type == 'cfn':
        PATH = "./database/cifarnet_recv.pth"
        Main_PATH = "./database/cifarnet_global.pth"
        model_temp = get_cifarnet()
        model_main = get_cifarnet()


    with open(PATH, 'wb') as model_data:
        for chunk_idx, model_bytes in self.modelBytes.items():
            model_data.write(model_bytes)

    # average the weights (Needs some error checking)
    model_temp.load_state_dict(torch.load(PATH))
    if os.path.isfile(Main_PATH):
        model_main.load_state_dict(torch.load(Main_PATH))

    sdTemp = model_temp.state_dict()
    sdMain = model_main.state_dict()

    for key in sdTemp:
        sdMain[key] = (sdMain[key] + sdTemp[key]) / 2

    # Check that can load into the model
    model_main.load_state_dict(sdMain)

    # save new model
    torch.save(model_main.state_dict(), Main_PATH)

    self.nextMsg = False
    self.agg = False
    return 'Aggregation Done! no errors found!'
```

Figure 24: Model weight average aggregation DRL framework

3.5 Deep reinforcement Learning (DRL) pruning framework

The DRL framework as depicted in figure 25 shows the pruning process of the model. This framework is the exact framework as described in [14].

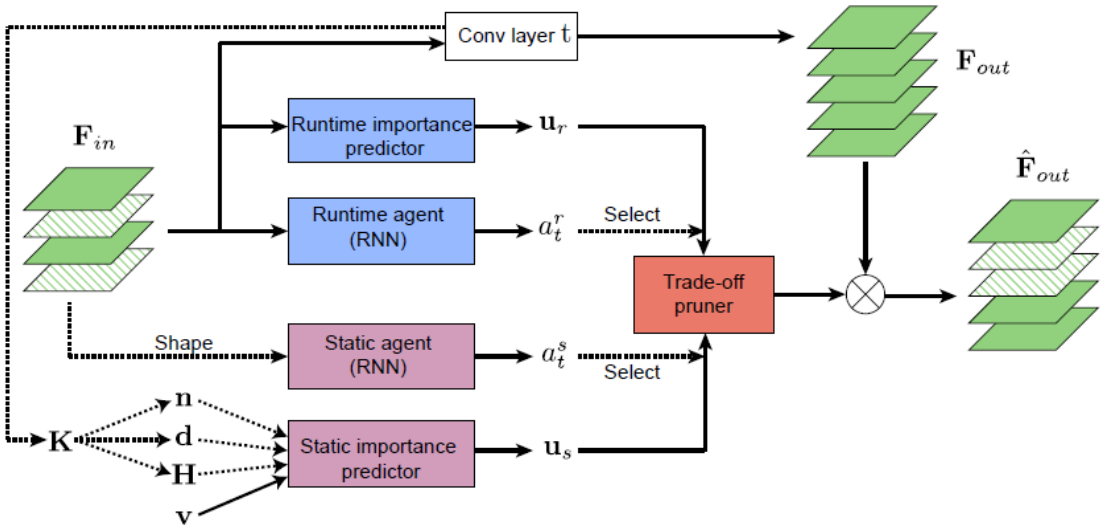

Figure 25: DRL based runtime pruning framework [14]

This framework takes into account 2 types of instances, namely, static and runtime. In order to prune a convolutional layer, the results from both the static and runtime instances will be taken into account via the trade-off pruner which will decide on which channels in the model to prune.

3.5.1 Runtime importance predictor

The runtime importance predictor is part of the runtime instance that produces the runtime channel importance, this importance value is used to determine the number of active channels (active ratio) in each layer which will be one of the requirements of the trade-off pruner to decide on which channels to prune. The runtime importance predictor consist of a subnetwork within the original model that takes the input of feature map Fin and computes the runtime channel importance of each layer in the original model. This subnetwork consist of a global pooling layer (F.avg_pool2d) and a fully connected layer (self.gate) as seen in Figure 26.

```
#self.gate = nn.Linear(self.in_channels,self.out_channels)
downsampled = F.avg_pool2d(x, x.shape[2])
downsampled = downsampled.view(x.shape[0], x.shape[1])
gates = self.gate(downsampled)
```

Figure 26: Subnetwork of runtime importance predictor

Since the runtime importance predictor requires the input feature map, this means that the runtime importance value varies over different input instances (different batches of the dataset).

3.5.2 Runtime agent

The runtime agent is the actor-critic agent in this framework. With reference to figure 27, the runtime agent is made up of an encoder and multiple decoders. The encoder consist of a Recurrent Neural Network (RNN) that takes in an input size of 128. This input represents the state, for which in this case, is the number of input channels at each layer of the model. Since the number of input channels varies in each layer of the model, the encoder is used to project into a fixed length vector with dimensions of 128 (adjustable hyperparameter). This fixed length vector will be the state representation of the DRL in the context of the runtime pruning.

```
class GaussianActorCritic(nn.Module):
    def __init__(self, rnn_hidden_size, prunable_layers_n_channels, sigma=0.05):
        super(GaussianActorCritic, self).__init__()
        self.prunable_layers_n_channels = prunable_layers_n_channels #Number of pruneable layers + channels
        self.rnn_hidden_size = rnn_hidden_size #fixed size of 128
        self.sigma = sigma

        self.layer_index_size = len(prunable_layers_n_channels)
        self.encoder_size = self.rnn_hidden_size# could be different
        self.rnn_input_size = self.encoder_size + 1# 1 for budget
        self.rnn = nn.RNN(self.rnn_input_size, self.rnn_hidden_size, num_layers=1)

        #-----------------------------Creating encoder for each layer-----------------------------------
        # Since each layer has different number of channels, need to craete 1 encoder for each layer
        self.encoders = []
        for in_channels in self.prunable_layers_n_channels:
            self.encoders.append(self.create_encoder(in_channels))
        self.encoders = nn.ModuleList(self.encoders)
        #-----------------------------------------------------------------------------------------------
        #-------------------------------------Creating decoders-----------------------------------------
        #Decoder 1) decoder_mu
        self.decoder_mu = nn.Sequential(
            nn.Linear(self.rnn_hidden_size, 1),
        )
        # Decoder 2) decoder_log_sigma
        self.decoder_log_sigma = nn.Linear(self.rnn_hidden_size, 1)
        # Decoder 3) decoder_value
        self.decoder_value = nn.Linear(self.rnn_hidden_size, 1)


        self.decoder_mu[0].bias.data.fill_(0.)
        self.decoder_mu[0].weight.data *= 0.2
        self.decoder_log_sigma.bias.data.fill_(-4.)
```

Figure 27: Runtime agent

The output of the RNN of the encoder is then fed into 3 separate linear layers which output the mean, standard deviation and a trainable value. The actor will require the mean and

standard deviation outputs to form a Gaussian policy where the continuous action is sampled from. While the critic will require the trainable value which represents the predicted future discounted accumulated reward to help with the policy training. Proximal Policy Optimization (PPO) is then used to optimize the actor-critic agent. Note that the actor and critic agent will output a sampled action and a predicted future discounted accumulated reward for each layer of the model.

3.5.3 Static importance predictor

The static importance predictor is part of the static instance that produces the static channel importance. This importance value, likewise with the runtime channel importance, is used to determine the number of active channels (active ratio) in each layer. The difference is that the static importance predictor takes in input of the following, namely, Frobenius norm (K), Euclidean distance (d) , Hessian diagonal (H) and a learnable parameter (v). The 3 feature vectors n, d and H are generated from the original model while the learnable parameter gets updated during further training of the original model via backpropagation. From figure 28, k is represented by self.filter_norm, d is represented by filter_dist, H is represented by taylor_first (due to the computation complexity of Hessian distance, first order Taylor expansion is used instead) and lastly, v is represented by static_gate.

```
#-----------------static importance predictor-----------------------
# self.predict_gate_with_filter_dist = nn.Sequential(
#     nn.Linear(4, 100),
#     nn.SELU(),
#     nn.Linear(100, 1)
# )

filter_dist = self.filter_dist[..., None]
filter_norm = self.filter_norm[..., None]

static_gates = self.static_gate.clone()
taylor_first = self.taylor_first[..., None]
static_gates = self.predict_gate_with_filter_dist(
    torch.cat(
        (
            static_gates[..., None],
            filter_dist,
            filter_norm,
            taylor_first,
        ),
        dim=1
    )
) # [out_channel, 1]
static_gates = static_gates.squeeze(dim=1).unsqueeze(dim=0) # [1, out_channel]
```

Figure 28: Static importance predictor

The static importance predictor is a subnetwork that consist of 2 fully connected layers with a Scaled Exponential Linear Unit (SELU) in the middle.

$$\mathrm{SELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$$

Figure 29: SELU equation [16]

SELU is used in place of Rectified Linear Unit (ReLU) due to the output of 0 if x is less than or equal to 0. This can result in 0 output and stop the backpropagation gradient.

3.5.4 Static agent

The static DRL agent takes in the full shape of the original model as the state representation, regardless of the output values and input data to the model. With reference to figure 30, this state input is then passed to the encoder (self.static_encoder), layer by layer, consisting of a fully connected layer and a ReLU activation function. Subsequently, the output of the encoder is passed into a LSTM model (self.static_rnn) before passing to the decoder (self.static_decoder_mu). The mean and standard deviation generated by the decoder is used to sample the action from the Gaussian policy and this action is also used to represent the sparsity ratio at each respective layer based on the layer input index.

```
def predict_action(self, budget, layer_index, static_hidden, with_noise=True, is_train_gagent=None):

    # self.static_rnn = nn.LSTM(self.rnn_hidden_size, self.rnn_hidden_size, num_layers=1)
    #
    # self.static_encoder = nn.Sequential(
    #     nn.Linear(1, self.rnn_hidden_size),
    #     nn.ReLU(),
    # )
    #
    # self.static_decoder_mu = nn.Sequential(
    #     nn.Linear(self.rnn_hidden_size, 1),
    # )

    x = torch.ones((static_hidden[0].shape[1], 1), device=static_hidden[0].device) \
            * self.prunable_layers_n_channels[layer_index] \
            / self.max_prunable_layers_n_channels
    x = self.static_encoder(x)

    x, static_hidden = self.static_rnn(x.unsqueeze(0),
                                       static_hidden)
    x = x.squeeze(0)
    static_action_mu = self.static_decoder_mu(x)
    static_action_mu = static_action_mu + budget
    static_action_log_sigma = self.static_decoder_log_sigma(x)
    static_value = self.static_decoder_value(x)
    gaussian_dist = torch.distributions.normal.Normal(static_action_mu,
            scale=torch.exp(static_action_log_sigma)
            )
    if with_noise:
        action = gaussian_dist.rsample().detach()
    else:
        action = static_action_mu.detach()
```

Figure 30: Static agent

Unlike the runtime agent, the reward of the static agent is based on the model's accuracy and parameters budget. However, likewise, PPO is used to optimize the static agent.

3.5.5 Gating module

In order to determine the number of prune-able channels, a gating module is used over every Convolutional - Batch Normalization - ReLU (ConvBnRelu) layer. This gating module will help to calculate the respective parameters for the static and runtime pruning agent, subsequently, providing the active ratio of each layer in the model. Figure 31 shows a CifarNet with the implementation of a gating module over each layer.

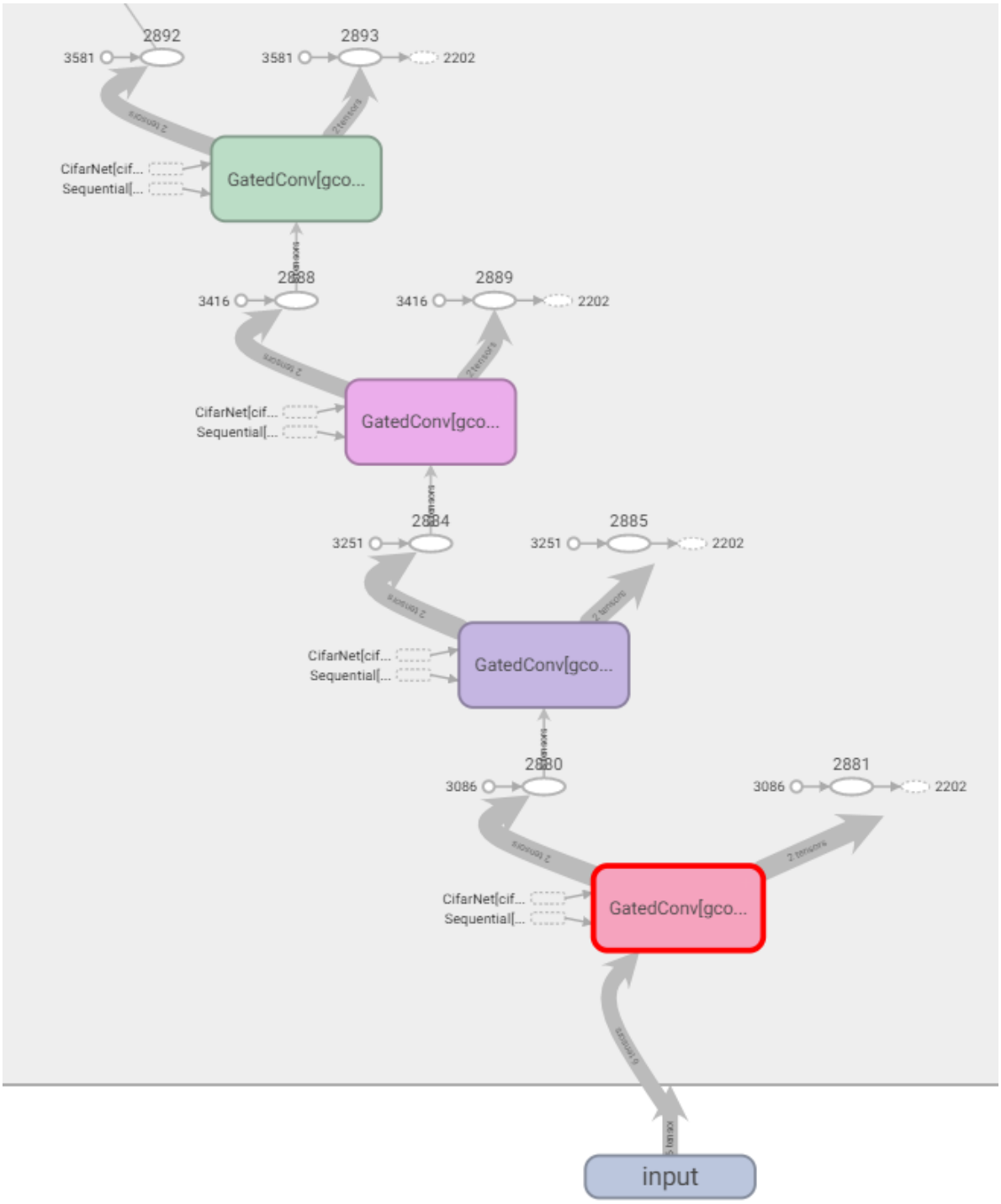

Figure 31: Snippet of Cifarnet with gating module

3.6 Use case Diagram

Figure 32 shows the use case diagram of the federated learning framework. There are 2 interface devices here, namely the client and server device. At any point in time, multiple users are able to create a new global model and update the model on the server device's database. Concurrently, multiple clients are also able to interface with the server device to obtain a copy of the global model for localized training in the client device. The locally trained model can then be sent back to the server device for aggregation as seen in the 'include' arrow after 'Train on copied global model'.

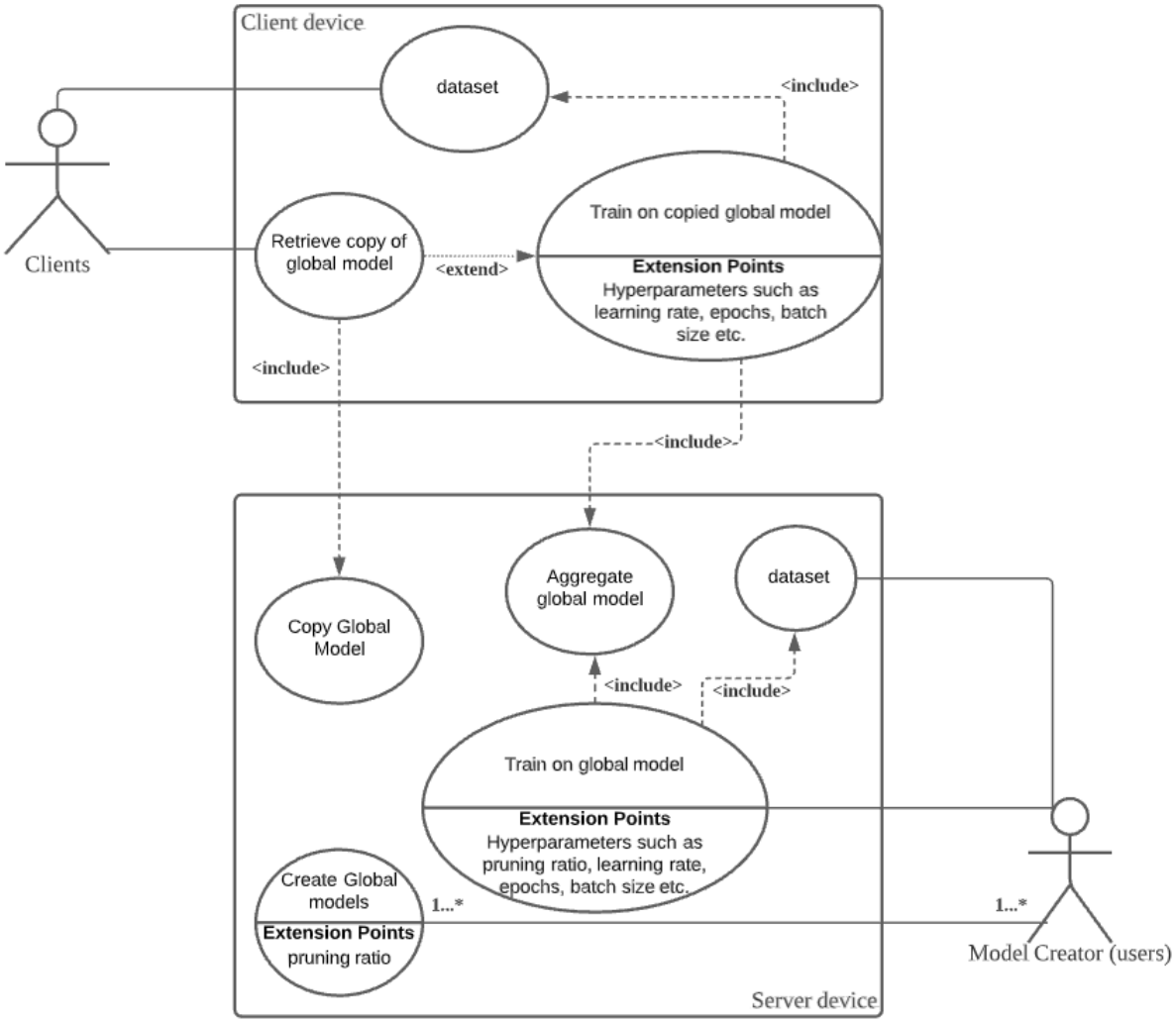

Figure 32: Use case diagram

3.7 Sequence Diagram

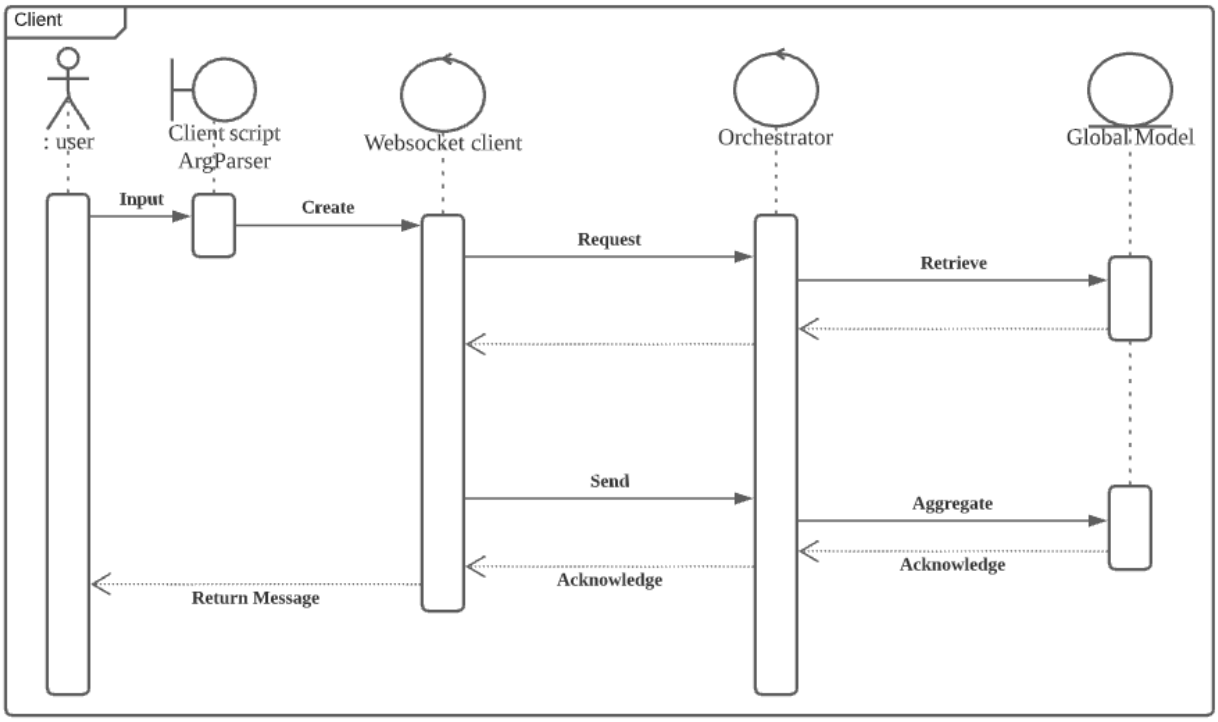

Figure 33: Client sequence diagram

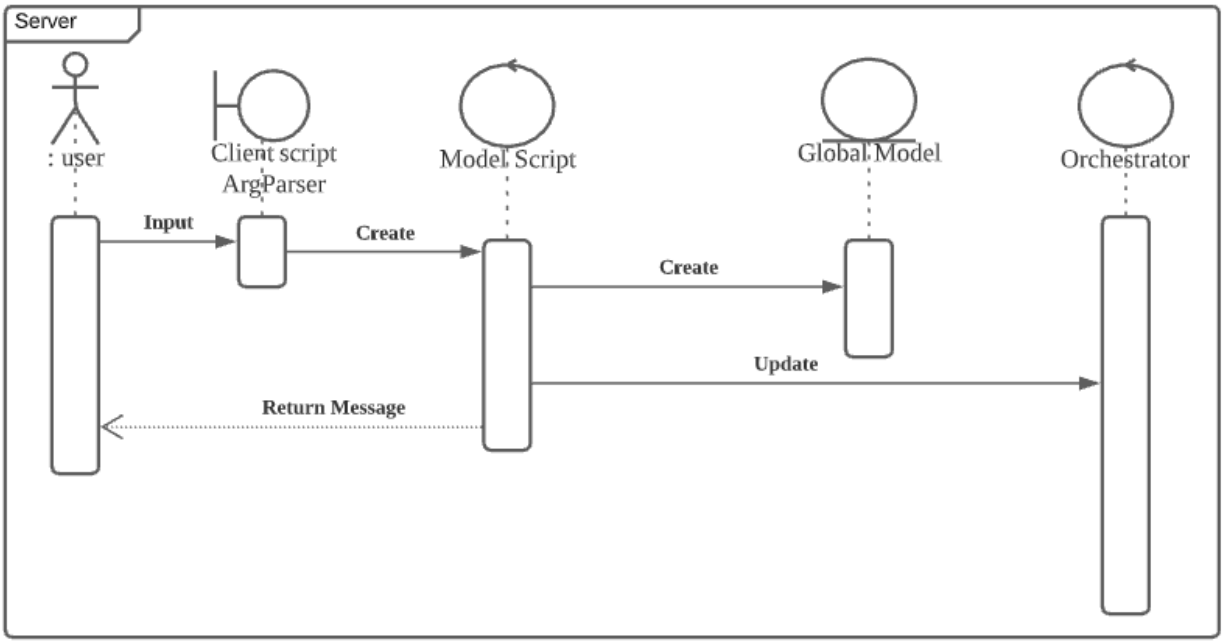

Figure 34: Server sequence diagram

4.1 Demonstration Setup

4.1.1 Federated Learning setup and process

In order to establish communication between the client and server Websocket scripts, we have to first initiate the orchestrator script. This will start the Websocket server and allow it to listen for requests on its allocated port number given the device's ipv4 address (figure 35).

```
server_worker = WebsocketServerWorker(host="192.168.1.149",port=8765)
server_worker.start()
```

Figure 35: Websocket server setup

The Uniform Resource Identifier (uri) address for the client script has to be configured to the server's ip4 address before the connection between the Websocket server and Websocket client can be established using a Websocket common protocol. (Figure 36)

```
uri = 'ws://192.168.1.149:8765'
if os.path.isfile(model_save_path):
    os.remove(model_save_path)
async with websockets.connect(uri) as websocket:

    if model_type == 'cfn':
        print('connected but no send?')
        await websocket.send('cfn')
    elif model_type == 'mvl':
        await websocket.send('mvl')
```

Figure 36: Websocket client setup

4.1.2 Deep Reinforcement Learning (DRL) Pruning Process

Using the argparser for the DRL script, the user is able to adjust 2 parameters, namely the budget and dynamic pruning ratio. The budget specifies the target sparsity of the model while the dynamic pruning ratio specifics the sensitivity of the dynamic pruning. When the pruning ratio is set to 1, dynamic pruning is more favoured over static pruning and vice versa for the case of pruning ratio when it is set to 0. This is illustrated by the code statements in figure 37.

```
valid_action = 1. - (1. - valid_action - overlap_inactive_ratio) * dynamic_prune_ratio - overlap_inactive_ratio
static_valid_action = 1. - (1. - static_valid_action - overlap_inactive_ratio) * (1. - dynamic_prune_ratio) - overlap_inactive_ratio
```

Figure 37: Dynamic pruning ratio trade off

From figure 37, we can see that when the dynamic pruning ratio is set to 1, sparsity from dynamic pruning is set to the predicted sparsity (action) from the runtime agent's actor while

the sparsity of static pruning is set to be a larger ratio than the predicted sparsity from the static agent.

4.2 Demonstration Results

From Table 1 and 2, we can see that conventional training, with localized datasets and model, will produce an average testing accuracy of 92% on the MobileNetV1 and 83.85% on the Cifarnet. The total time taken for training is very fast at 5.4 minutes and 2.18 minutes on the MobileNetV1 and Cifarnet respectively due to the training of the dataset using a GPU. This results mainly serves as a baseline accuracy for comparison against the federated learning results.

Training	**MobileNetV1**			
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	66.3187778	1.1053	0.7815956	0.440913892
2	63.92443347	1.0654	0.8708531	0.304412269
3	62.27911448	1.038	0.9026461	0.227890352
4	68.81009173	1.1468	0.9316746	0.168901776
5	62.66356277	1.0444	0.9533965	0.123828678
Testing				
average testing accuracy			0.920664207	
Total time taken to train 5 epochs			5.4 minutes	

Table 1: MobileNetV1 conventional training results

Training	**CifarNet**			
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	28.06868	0.46781	0.681477093	0.591589687
2	25.56546	0.42609	0.746248025	0.508224192
3	25.1093	0.41849	0.782977883	0.445121939
4	26.15879	0.43598	0.811413902	0.394664159
5	26.38925	0.43982	0.836097946	0.349969937
Testing				
average testing accuracy			0.838560886	
Total time taken to train 5 epochs			2.1883 Minutes	

Table 2: Cifarnet conventional training results

From table 3 and 4, we can observe the results of training the models on the RPI 3 and 4. Since the RPI3 and RPI 4 does not have GPU attachments, training is done on the CPU instead. Since a CPU is used for training, the time taken to train the models is extremely long, with the training on Cifarnet taking almost 5 hours per epoch. We can see that because MobileV1Net is more complex and larger than the Cifarnet, the time taken to train per epochs is exponentially larger. This sprouts the potential of model pruning to further reduce the model's size to allow shorter training time on the client devices.

MobileV1Net				
Training on RPI4 (RPI2 dataset)				
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	17299.50954	288.33	0.890884	0.275339503
2	17261.12594	287.69	0.9274862	0.174445548
total	34560.63548	576.02		
Time taken for model exchange (respective to RPI)				
Receive (seconds)		Send(seconds)		
1.351117849		4.125297308		
Training on RPI4 (RPI1 dataset)				
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	17205.64267	286.76	0.8874309	0.275306751
2	17183.27774	286.39	0.9452118	0.137147786
total	34388.92041	573.15		
Time taken for model exchange (respective to RPI)				
Receive (seconds)		Send(seconds)		
1.322829247		2.506022215		
Testing				
PRE-aggregation average testing accuracy			0.873155	
POST-aggregation average testing accuracy			0.8805351	

Table 3: MobileV1Net Federated learning results

CifarNet				
Training on RPI4 (RPI2 dataset)				
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	490.2108574	8.1702	0.752071823	0.508139617
2	457.5087585	7.6251	0.810773481	0.400429041
3	454.1792889	7.5697	0.863259669	0.310019921
4	458.171263	7.6362	0.90538674	0.231418697
5	462.3364992	7.7056	0.927486188	0.197411778
total	2322.406667	38.7068		
Time taken for model exchange (respective to RPI)				
Receive (seconds)		Send(seconds)		
0.538524866		2.876970291		
Training on RPI3 (RPI1 dataset)				
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	751.4112048	12.524	0.725598527	0.547204065
2	806.1196063	13.435	0.800184162	0.414800825
3	808.5076244	13.475	0.861418048	0.313341523
4	809.2686036	13.488	0.905847145	0.23154976
5	812.9181862	13.549	0.916206262	0.197832373
total	3988.225225	66.471		
Time taken for model exchange (respective to RPI)				
Receive (seconds)		Send(seconds)		
1.088569164		2.716389179		
Testing				
PRE-aggregation average testing accuracy			0.742619926	
POST-aggregation average testing accuracy			0.816420664	

Table 4: Cifarnet federated learning results

Evidently, after model aggregation, the models generally have an improved accuracy over the base global model (pre-aggregation). However, the accuracy from federated learning is slightly less than the conventionally trained models due to the effects of weights averaging. This means that this is the compromise that federated learning have to bear over conventional model training. For the results for MobileV1Net, the model is too big for computation on the RPI3 which caused the RPI3 to crash upon training. Therefore, the RPI4 is used instead to train on both client side datasets instead.

From table 5, we can see the results of the model trained with 50% sparsity target. Although we are able to show that the accuracy from federated learning is higher than that from the unpruned federated learning training, we are unable to observe a shorter training time on the 50% sparsity trained model. Various reasons include the failure of the runtime and static agent in pruning the model's parameters or the increased computation resulted from the additional gating modules added to each layer. Therefore, we are unable to concretely justify

that a pruned model will definitely provide faster performance on the federated learning process and additional experimentations with other pruning methodology should be employed in future works.

CifarNet	**0.5 sparsity**			
Training on RPI4 (RPI2 dataset)				
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	849.4626577	14.158	0.872238	0.313938
2	911.9996545	15.2	0.877072	0.301372
3	908.5168126	15.142	0.877762	0.286802
4	856.2906399	14.272	0.884669	0.275404
5	879.9152596	14.665	0.895718	0.257529
total	4406.185024	73.437		
Time taken for model exchange (respective to RPI)				
Receive (seconds)		Send(seconds)		
1.70998621		5.56490922		
Training on RPI3 (RPI1 dataset)				
	Train time			
Epochs	seconds	minutes	Accuracy	loss
1	1196.670482	19.945	0.875	0.29027
2	1348.986524	22.483	0.872698	0.292417
3	1354.562961	22.576	0.888122	0.268003
4	1255.679081	20.928	0.895718	0.263787
5	1303.367886	21.723	0.874309	0.283186
total	6459.266934	107.655		
Time taken for model exchange (respective to RPI)				
Receive (seconds)		Send(seconds)		
4.455176115		9.565993309		
Testing			-	
PRE-aggregation average testing accuracy			0.87607	
POST-aggregation average testing accuracy			0.885609	

Table 5: Cifarnet 50% sparsity trained federated learning results [17] [4]

5 Conclusion

5.1 Project Conclusion

From the demonstration results, we can see that because of model weight averaging, the accuracy of the model is not optimal as compared to conventional training. There is also increased training overhead due to model transfer and the total time taken for training is very much dependent on the computational capabilities of the client devices. For the DRL portion, even though the accuracy of the model via federated learning is higher than that of normal federated learning, but the overall training time did not reduce despite training the base model on 50% sparsity. Therefore, future works can involve other pruning methodology to further justify the effects of model pruning in a federated learning framework.

5.2 Future Works

Due to time constraints for this project and the steep learning curve for the state-of-the-art reinforcement learning concepts, other pruning methodologies are not tested. Therefore, in future works, other pruning methodology can be tested to further justify that model pruning can help to reduce the overall federated learning training time. We can also contrast different types of pruning methods and identify which types of pruning method best suits for re-training on client devices.

Currently, the implementation is a simplex communication with the Websocket client as the initiator. We can create a duplex communication by using an intermediate server such as a Kafka server. We can then establish communication with the Kafka broker to access the global model hosted in the Kafka server. In this case, we can easily interface with multiple global model creators.

Additionally, to make the application more realistic, targeted client devices can include smart phones and microcontrollers. Load testing can also be experimented on to simulate large communication bandwidth against the server.

Finally, we can also test on dynamic weighted aggregation based on different model quality by experimenting on methods that can determine the value of the received model quality. This can include measurements such as the judgement on model's performance metrics or the deviation of the model weights from the global models'. Subsequently, performing weighted aggregation based on this determined quality value.

References

[1] Intel, “Understanding the advantages of 5G,” Intel, [Online]. Available: https://www.intel.com/content/www/us/en/wireless-network/5g-benefits-features.html.

[2] B. Marr, “How much data do we create everyday? The mind-blowing stats everyone should read,” Forbes, 21 May 2018. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=1ad68bb960ba.

[3] OpenMined, “Syft,” 13 November 2020. [Online]. Available: https://github.com/OpenMined/PySyft.

[4] D. Arya, H. Maeda, S. K. Ghosh and D. Toshniwal, “Transfer Learning-based Road Damage Detection for mutiple countries,” ResearchGate, 2020.

[5] IBM Cloud Education, “Recurrent Neural Networks,” IBM , 14 September 2020. [Online]. Available: https://www.ibm.com/cloud/learn/recurrent-neural-networks.

[6] T. Wood, “What is the sigmoid function?,” DeepAI, [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/sigmoid-function.

[7] A. Marchisio, M. A. Hanif, S. Rehman and M. Martina, “A methodology for automatic selection of activation functions to design hybrid deep neural networks,” ResearchGate, 2018.

[8] N. Rieke, “What is Federated learning?,” Nvidia, 13 October 2019. [Online]. Available: https://blogs.nvidia.com/blog/2019/10/13/what-is-federated-learning/.

[9] A. Choudhary, “Nuts and bolts of reinforcement learning : model based planning using dynamic programming,” 18 September 2018. [Online]. Available: https://www.analyticsvidhya.com/blog/2018/09/reinforcement-learning-model-based-planning-dynamic-programming/.

[10] Z. Salloum, “Introduction to actor-critic in reinforcement learning,” 18 June 2019. [Online]. Available: https://towardsdatascience.com/introduction-to-actor-critic-7642bdb2b3d2.

[11] Wikipedia, “Gaussian function,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_function. [Accessed 10 april 2021].

[12] T. Simonini, “Proximal Policy Optimization (PPO) with Sonic the Hedgehog 2 and 3,” 3 September 2018. [Online]. Available: https://towardsdatascience.com/proximal-policy-optimization-ppo-with-sonic-the-hedgehog-2-and-3-c9c21dbed5e.

[13] Mike, “Generalized Advantage Estimator explained,” [Online]. Available: https://notanymike.github.io/GAE/.

[14] J. Chen, S. Chen and S. J. Pan, “Storage efficient and dynamic flexible runtime channel pruning via deep reinforcement learning,” Nanyang Technological University of Singapore, Singapore, 2020.

[15] Kaggle, “COVID-19 RADIOGRAPHY DATABASE (Winner of the COVID-19 Dataset Award by Kaggle Community),” [Online]. Available: https://www.kaggle.com/tawsifurrahman/covid19-radiography-database.

[16] C. Hansen, “Activation Functions Explained - GELU, SELU,ELU,ReLU and more,” 22 August 2019. [Online]. Available: https://mlfromscratch.com/activation-functions-explained/#/.

[17] IBM cloud education, “Convolutional Neural Networks,” IBM, 20 October 2020. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks.

[18] P. Pedamkar, “What is reinforcement learning?,” EDUCBA, [Online]. Available: https://www.educba.com/what-is-reinforcement-learning/.

[19] Floydhub, “An introduction to Q-learning: reinforcement learning,” 15 May 2019. [Online]. Available: https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/.

[20] Coursera, “Gaussian policies for continuous actions,” Coursera, [Online]. Available: https://www.coursera.org/lecture/prediction-control-function-approximation/gaussian-policies-for-continuous-actions-T8mUv.

[21] S. Paul, “Model Pruning in deep learning,” 22 June 2020. [Online]. Available: https://towardsdatascience.com/scooping-into-model-pruning-in-deep-learning-da92217b84ac.