Radio Frequency Fingerprint Identification

Home Work 1: RFFI Project Basic Code

†Rohith Yamsani, †Raja Duggirala
†MS in Computer Science, Florida International University, Miami, FL, 33174, USA
Email:ryams001@fiu.edu,rdugg004@fiu.edu

***Abstract*—This study explores advances in signal processing techniques and neural network architectures to improve model performance for time-domain signal analysis. Initially, the Fast Fourier Transform (FFT), Inverse Fast Fourier Transform (IFFT) and Short-Time Fourier Transform (STFT) were used to analyse the time and frequency properties of signals, with specific parameters such as window size and type changed to optimise the transformation results. Following that, modifications were made to a convolutional neural network (myCNN), which included an additional convolutional layer and batch normalisation, enhancing the model's ability to detect complicated patterns in the data. This change resulted in a significant performance improvement, with assessment accuracy improving from 95% to 96.1137% and target domain accuracy rising from 87.5% to 91.212%. These advances show the possibility for deeper structures and advanced signal processing. We also used advanced deep learning techniques such as LSTM, CNN+LSTM, ResNet to check the Evaluation Accuracy, Average accuracy of the training,Accuracy after fine-tuning**

I. Introduction

In the rapidly evolving field of signal processing, the analysis of time-domain signals remains a critical challenge due to the inherent complexity and variability of these signals. Traditional approaches, such as the Fast Fourier Transform (FFT), provide fundamental insights into frequency components, but frequently fall short of capturing dynamic changes in time. This constraint has led to the development of more resilient methodologies and computational models capable of providing increased accuracy and performance in real-time signal processing.
Convolutional neural network(CNN), Long Short-Term Memory networks (LSTMs) and residual networks (ResNets) are few advanced computational models that have gained popularity for their use in signal processing, respectively.

Convolutional Neural Networks (CNNs) are specialised neural networks that handle data with a grid-like architecture, such as pictures. CNNs are very useful for image recognition, classification, and analysis because they can automatically and adaptively learn spatial hierarchies of features via backpropagation. A typical CNN architecture has several layers, including convolutional, pooling, and fully linked layers. The convolutional layers perform a convolution operation on the input and pass the output to the next layer. This process enables the network to efficiently capture spatial and temporal dependencies in an image or signal. Pooling (subsampling or downsampling) decreases the dimensionality of each feature map while retaining the most critical information.

LSTMs, a type of recurrent neural network, are well-suited for time-series data because they can sustain long-term dependencies in sequential information, making them suitable for jobs that require previous information for current decision-making. This feature is particularly valuable for improving temporal analysis capabilities in signal processing.

ResNets, particularly the ResNet-50 design, take an innovative method by leveraging skip connections or shortcuts to jump over some layers. These connections serve to alleviate the vanishing gradient problem, making the network deeper and more robust. ResNet-50, which contains 50 layers, has been widely used for complicated picture recognition tasks and has the ability for adaptation when analysing complex signal patterns.

II. TECHNIQUE

A. Data Preprocessing

To begin, we focus on obtaining a selected subset of IQ signal data from a complex dataset for signal processing. On the first day, it extracts 2x256 dimension data from device '1-10' and stores it in a nested dictionary. This targeted extraction enables a precise study of the signal's properties and prepares the data for the next processing steps.

Then, We illustrate how to extract and process the first sample of IQ data into discrete In-phase (I) and Quadrature (Q) components. It then combines these components into a complex-valued array, x, by multiplying the Q component by the imaginary unit and adding the I component. This complicated array format, which is essential for advanced signal processing jobs, enables further analyses such as Fourier Transform and modulation in communication systems.

From the below plot, We then depict the In-phase (I) and Quadrature (Q) components of signal data to show how they change over time. Plotting these components against a succession of time intervals allows for a clear visual difference between I and Q data, which is critical for analysing signal behaviour and integrity in communication systems.

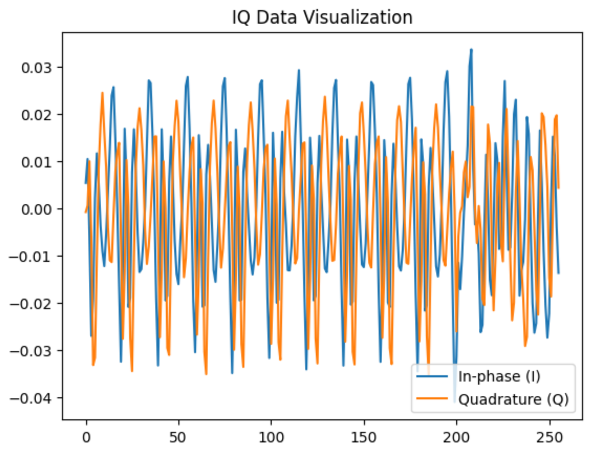


Fig. 1. IQ Data Visulaization.

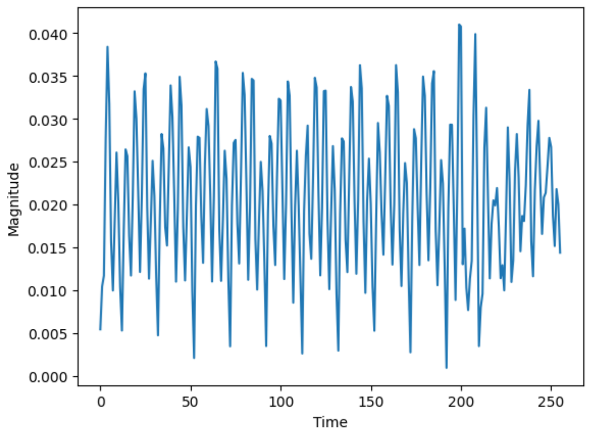


Fig. 2. Signal in time domain

As shown in the fig 2., It shows the magnitude of a complicated signal over time to provide a visual representation of its strength and changes. By charting the absolute values of the complex signal xx, the graph shows the signal's envelope, which is critical for recognising variations and trends in the signal's behaviour. Such visualisations are critical in signal processing for understanding the signal's dynamics, including any anomalies or characteristic features, allowing for more in-depth analysis or diagnostics in communications systems.

The Fast Fourier Transform (FFT) is a fundamental algorithm in signal processing that allows for the efficient computation of the Discrete Fourier Transform (DFT) from a time-domain signal to a frequency-domain representation. FFT reduces computational complexity from O(N2)O(N2) to O(NlogN)O(NlogN), where NN represents the number of samples. It is useful for identifying dominant frequencies, filtering specific frequency components, analysing spectral density, compressing data, and characterising systems. This transformation is critical for analysing and comprehending complicated signals in a variety of scientific and technical applications.

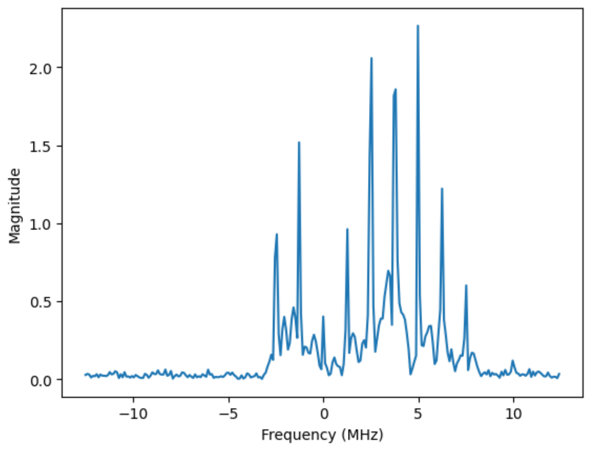


Fig. 3. Fast Fourier Transform (FFT).

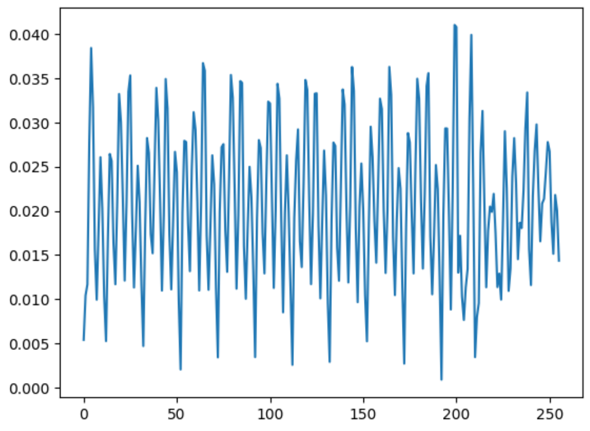


Fig. 4. Inverse Fast Fourier Transform (IFFT)

As shown in the Fig 3. Let's apply the Fourier Transform to a signal to analyse its frequency spectrum. It selects a sample rate of 25 MHz, does the FFT, centres the zero-frequency component for visualisation, and generates frequency bins. The resulting frequency spectrum is then shown, with frequencies in MHz and their accompanying magnitudes, providing information on the energy distribution across different frequencies. This visual representation aids in the identification of the signal's prominent frequencies, which is critical for signal processing and analysis applications.

From Fig 4. ,The Inverse Fast Fourier Transform (IFFT) is a mathematical procedure that computes the inverse of the Fast Fourier Transform. It translates a function from the frequency domain to the time domain. You can compare the following results to the original data.

$$\text{time}_signal = np.fft.ifft(signal_spectrum)$$

The Short-Time Fourier Transform (STFT) is then used to analyse the temporal evolution of frequencies in a signal, with a Hann window and a segment size of 64. The

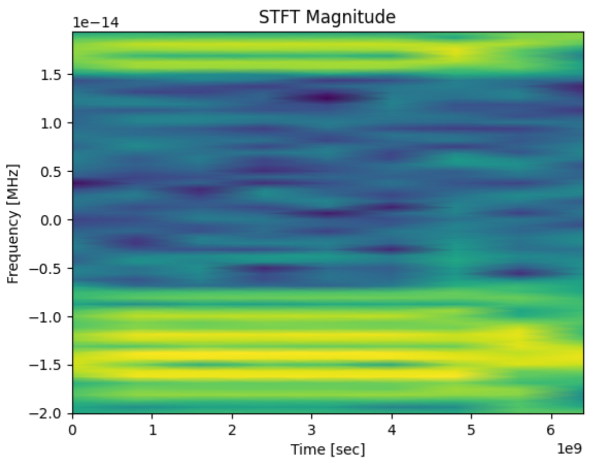


Fig. 5. Inverse Fast Fourier Transform (IFFT)

computed STFT is subsequently transformed into a magnitude spectrogram, which is represented by a pseudo-color graphic. The spectrogram, which plots frequency (in MHz) against time with intensity denoting magnitude, shows how distinct frequency components of the signal change over time, providing critical information for applications such as speech analysis and music processing. The visualisation shows the signal's dynamic fluctuations, making complicated patterns and anomalies easier to grasp.

The above Figure Shows STFT implemetation to view the evolution of frequencies in the signal.

B. Fingerprinting System

At first, The get-data-labels method extracts and organises signal data and labels from a hierarchical dictionary structure, which can accommodate numerous domains and devices as given by domain-list and label-list. It efficiently creates a comprehensive dataset by iterating through each domain and device and storing the appropriate signal data and labels in structured arrays. This strategy ensures that each piece of data is associated with the right label, making it easy to apply in machine learning tasks that require signal analysis.

C. MyCNN

we created a custom dataset and a convolutional neural network (CNN) with PyTorch. The MyDataset class organises data effectively and is compatible with PyTorch's DataLoader, whereas the MyCNN class creates a simple CNN architecture with convolutional layers, batch normalisation, and a classifier sequence. This structure is intended to be adaptable, allowing for simple alterations and upgrades, such as including more complicated designs like LSTMs, ResNets, or transformers to better meet advanced project requirements.

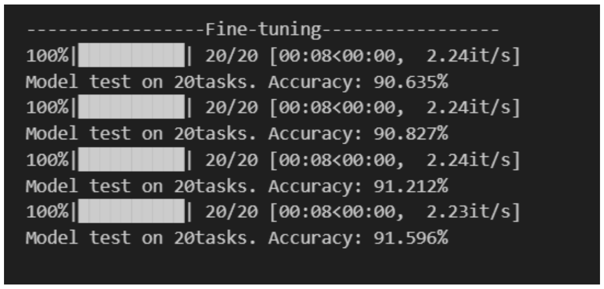
To improve the accuracy of the model using CNN, we added few changes to the myCNN function, Instead of two convolutional layers given in the code, i added one more layer called 'conv3' along with batch normalization 'bn3' was added. By doing this, we can add another layer of feature extraction which can increase the models capacity to learn more complex patterns in the data. This changes aims to increase the model's capacity, optimizes the training and imroves the perfomance on the target domain.

We then organise and prepare a dataset into source and target domains for use in transfer learning or domain adaptation. It entails identifying certain domains as sources and targets, obtaining appropriate data and labels, and processing them to meet the needs of machine learning models. The script ensures that labels are numerically encoded and data is correctly formatted (for example, by altering dimensions and data types), resulting in a structured dataset that can be used to train and evaluate models on fresh, previously unknown data. This strategy is critical for activities that require model robustness under a variety of operational situations.

We then train a convolutional neural network for six epochs, changing parameters based on training data and evaluating performance against a test dataset. It employs data loaders for efficient processing and generates periodic metrics to track the model's progress and generalizability. This method guarantees that the model is properly optimised before deployment.

```
Epoch: 1 Training Loss: 1.390197
Epoch: 1 Training Accuracy: 74.826131
Evaluation Accuracy: 91.6088
Epoch: 2 Training Loss: 0.395708
Epoch: 2 Training Accuracy: 92.341361
Evaluation Accuracy: 94.3117
Epoch: 3 Training Loss: 0.272766
Epoch: 3 Training Accuracy: 94.171059
Evaluation Accuracy: 94.8592
Epoch: 4 Training Loss: 0.214440
Epoch: 4 Training Accuracy: 95.329482
Evaluation Accuracy: 95.6620
Epoch: 5 Training Loss: 0.182416
Epoch: 5 Training Accuracy: 95.981819
Evaluation Accuracy: 96.0474
Epoch: 6 Training Loss: 0.160902
Epoch: 6 Training Accuracy: 96.448976
Evaluation Accuracy: 96.1137
```

We next evaluate a convolutional neural network (CNN) model on a target dataset to see how well it generalises to new, previously unknown data. This evaluation entails configuring a DataLoader for the target data, running model inference in a no-gradient environment to save resources, and calculating accuracy based on the model's predictions. This technique aids in determining the model's usefulness in real-world circumstances and informs any additional modifications or adaptations required for peak performance.



```
-----------------Fine-tuning----------------
100%|██████████| 20/20 [00:08<00:00,  2.24it/s]
Model test on 20tasks. Accuracy: 90.635%
100%|██████████| 20/20 [00:08<00:00,  2.24it/s]
Model test on 20tasks. Accuracy: 90.827%
100%|██████████| 20/20 [00:08<00:00,  2.24it/s]
Model test on 20tasks. Accuracy: 91.212%
100%|██████████| 20/20 [00:08<00:00,  2.23it/s]
Model test on 20tasks. Accuracy: 91.596%
```

Fig. 6. Accuracy after fine tuning

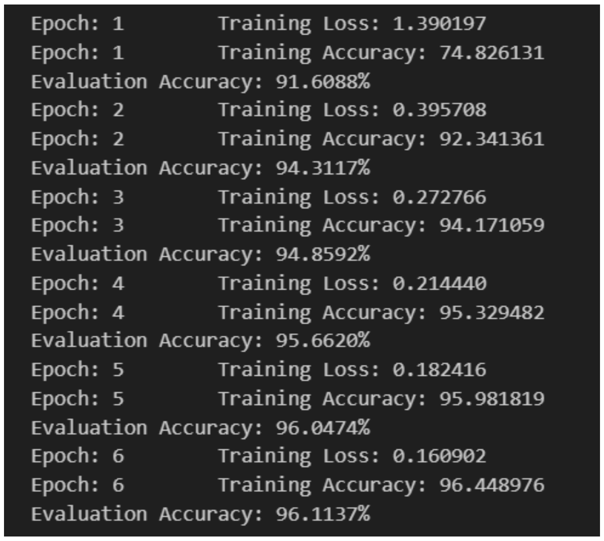
D. *Prototypical Network (PTN)*

Creates a Prototypical Network (PTN) with an existing convolutional neural network model as the feature extractor. This PTN is intended for few-shot learning problems, in which it computes class prototypes from a tiny labelled support set and predicts labels for a query set using distances to the prototypes. The setup is transferring the PTN to a specific computing device (CPU or GPU) and structuring the training process with a cross-entropy loss function and an Adam optimizer, which includes a regularisation term to prevent overfitting. This design equips the PTN for effective training and evaluation of few-shot learning tasks.

We then build up a data loader for Few-Shot Learning (FSL) using the easyfsl library's TaskSampler, which is setup for episodic training. It sets the parameters for "N-Way K-Shot" learning, which include the number of classes, shots, and queries per task, as well as the number of evaluation tasks. This arrangement makes it easier to create bespoke training episodes, which is critical when training models for FSL tasks where models must learn from a limited number of instances per class. The TaskSampler generates these tasks automatically, allowing the models to quickly adapt to new jobs with minimum input.

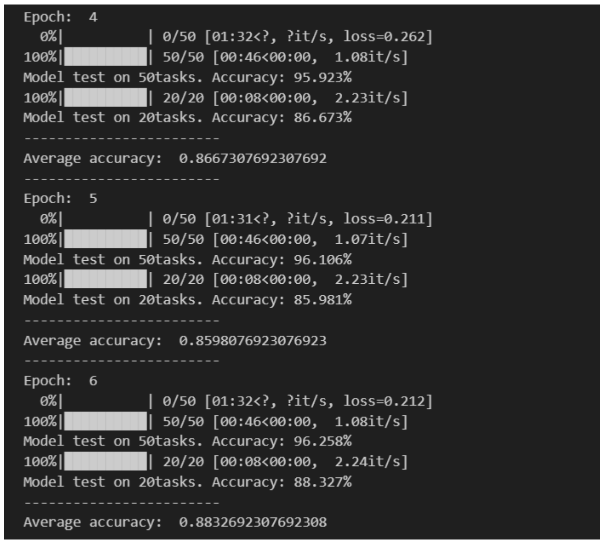
Then, We next set up a training environment for few-shot learning (FSL) using specified hyperparameters for "N-Way K-Shot" tasks. It entails initialising a training dataset with source data and labels, then using a TaskSampler to generate specific training episodes with a predetermined number of classes, support examples, and query examples. The solution employs a proprietary DataLoader designed specifically for episodic training, allowing the model to learn quickly from fewer instances per class, emulating real-world conditions in which only a few data points are available for each category.

We then fine-tune a model for few-shot learning tasks by training it in multiple domains. It generates training episodes using a custom loader and updates the model's parameters multiple times over various epochs. Each epoch is made up of many training episodes, which are then evaluated on a test set to track performance changes. This strategy improves the model's capacity to generalise successfully to new, previously unknown data.



```
Epoch: 1        Training Loss: 1.390197
Epoch: 1        Training Accuracy: 74.826131
Evaluation Accuracy: 91.6088%
Epoch: 2        Training Loss: 0.395708
Epoch: 2        Training Accuracy: 92.341361
Evaluation Accuracy: 94.3117%
Epoch: 3        Training Loss: 0.272766
Epoch: 3        Training Accuracy: 94.171059
Evaluation Accuracy: 94.8592%
Epoch: 4        Training Loss: 0.214440
Epoch: 4        Training Accuracy: 95.329482
Evaluation Accuracy: 95.6620%
Epoch: 5        Training Loss: 0.182416
Epoch: 5        Training Accuracy: 95.981819
Evaluation Accuracy: 96.0474%
Epoch: 6        Training Loss: 0.160902
Epoch: 6        Training Accuracy: 96.448976
Evaluation Accuracy: 96.1137%
```

Fig. 7. Evaluation Accuracy of enhanced CNN after adding layer



```
Epoch:  4
  0%|          | 0/50 [01:32<?, ?it/s, loss=0.262]
100%|██████████| 50/50 [00:46<00:00,  1.08it/s]
Model test on 50tasks. Accuracy: 95.923%
100%|██████████| 20/20 [00:08<00:00,  2.23it/s]
Model test on 20tasks. Accuracy: 86.673%
------------------------
Average accuracy:  0.8667307692307692
------------------------
Epoch:  5
  0%|          | 0/50 [01:31<?, ?it/s, loss=0.211]
100%|██████████| 50/50 [00:46<00:00,  1.07it/s]
Model test on 50tasks. Accuracy: 96.106%
100%|██████████| 20/20 [00:08<00:00,  2.23it/s]
Model test on 20tasks. Accuracy: 85.981%
------------------------
Average accuracy:  0.8598076923076923
------------------------
Epoch:  6
  0%|          | 0/50 [01:32<?, ?it/s, loss=0.212]
100%|██████████| 50/50 [00:46<00:00,  1.08it/s]
Model test on 50tasks. Accuracy: 96.258%
100%|██████████| 20/20 [00:08<00:00,  2.24it/s]
Model test on 20tasks. Accuracy: 88.327%
------------------------
Average accuracy:  0.8832692307692308
```

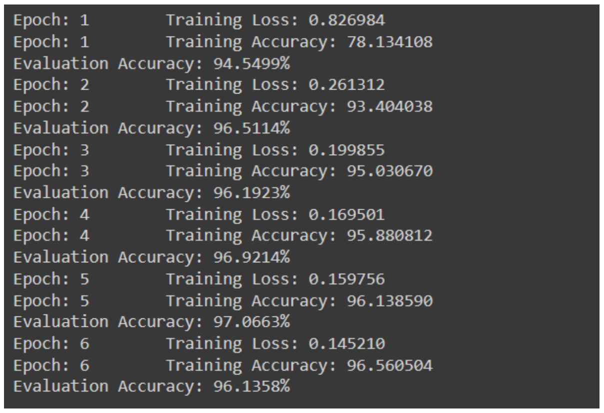
Fig. 8. Average accuracy of enhanced CNN after adding layer

III. Results

A. *Enhanced CNN*

To enhance accuracy, I made a few adjustments to the myCNN function. Instead of the two convolutional layers specified in the code, I added a third layer called 'conv3', as well as batch normalisation 'bn3'. By doing so, we can add another layer of feature extraction, increasing the model's ability to learn more complicated patterns in the data. This update attempts to expand the model's capacity, optimise training, and improve performance in the target area.

At epoch 6, The evaluation accuracy was 95% prior to making any changes to the code; after the changes, it increased to 96.1137%. Also, the target-loader's average accuracy was 0.87423 prior to any adjustments, but it increased to 0.88326



```
Epoch: 1        Training Loss: 0.826984
Epoch: 1        Training Accuracy: 78.134108
Evaluation Accuracy: 94.5499%
Epoch: 2        Training Loss: 0.261312
Epoch: 2        Training Accuracy: 93.404038
Evaluation Accuracy: 96.5114%
Epoch: 3        Training Loss: 0.199855
Epoch: 3        Training Accuracy: 95.030670
Evaluation Accuracy: 96.1923%
Epoch: 4        Training Loss: 0.169501
Epoch: 4        Training Accuracy: 95.880812
Evaluation Accuracy: 96.9214%
Epoch: 5        Training Loss: 0.159756
Epoch: 5        Training Accuracy: 96.138590
Evaluation Accuracy: 97.0663%
Epoch: 6        Training Loss: 0.145210
Epoch: 6        Training Accuracy: 96.560504
Evaluation Accuracy: 96.1358%
```

Fig. 9. Evaluation accuracy of ResNet

after the alterations.

For fine tuning, the accuracy before the code modifications was 87.5%, but after the necessary code changes in the myCNN function, it increased to 91.212%. This shows us that the code's accuracy has grown.

B. ResNet

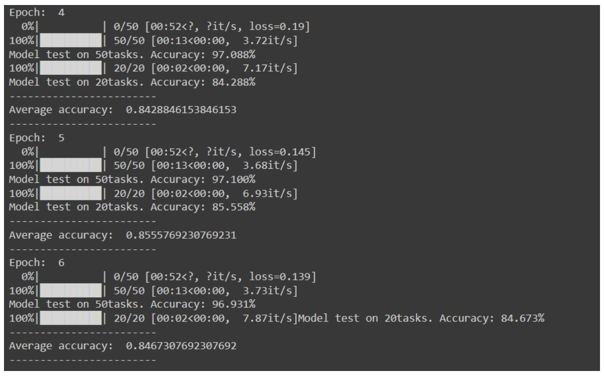
The ResNet implementation we created is a custom version of ResNet that does not exactly correlate to any conventional ResNet versions like ResNet-18, ResNet-34, and so on, which are normally specified in libraries such as PyTorch's 'torchvision.models'. We used a sequence of Residual Blocks to conduct convolutions and add the input (or a downsampled version of it) back into the output.

Here's an overview of how we implementated ResNet:

1. ResidualBlock: ResidualBlock is a basic building block in ResNet topologies that includes two convolutional layers, batch normalisation, and ReLU activation. This is comparable to the blocks used by ResNet-18 and ResNet-34.

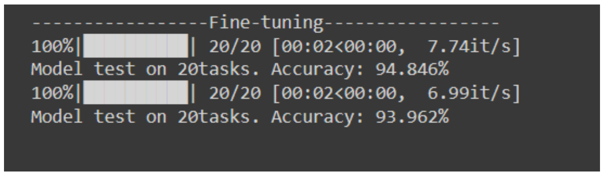
2. Custom layers and configurations: You configure the network with the array '[2, 2, 2, 2]', which specifies the number of blocks in each layer. This configuration is comparable to ResNet-18, which also follows this architecture, but with a slightly different structure in the deeper layers.

3. Adjustments for Input Channels and Class Numbers: - Your model begins with a convolution layer optimised for 1-channel images, unlike normal ResNet models that are often pre-trained on 3-channel (RGB) images. This customisation is beneficial for non-standard colour collections, such as medical or grayscale photos. The number of output classes is set to 130, suggesting that it is customised for a specific classification purpose in addition to the standard 1000 classes in ImageNet.



```
Epoch:  4
  0%|          | 0/50 [00:52<?, ?it/s, loss=0.19]
100%|██████████| 50/50 [00:13<00:00,  3.72it/s]
Model test on 50tasks. Accuracy: 97.088%
100%|██████████| 20/20 [00:02<00:00,  7.17it/s]
Model test on 20tasks. Accuracy: 84.288%
------------------------
Average accuracy:  0.8428846153846153
------------------------
Epoch:  5
  0%|          | 0/50 [00:52<?, ?it/s, loss=0.145]
100%|██████████| 50/50 [00:13<00:00,  3.68it/s]
Model test on 50tasks. Accuracy: 97.100%
100%|██████████| 20/20 [00:02<00:00,  6.93it/s]
Model test on 20tasks. Accuracy: 85.558%
------------------------
Average accuracy:  0.8555769230769231
------------------------
Epoch:  6
  0%|          | 0/50 [00:52<?, ?it/s, loss=0.139]
100%|██████████| 50/50 [00:13<00:00,  3.73it/s]
Model test on 50tasks. Accuracy: 96.931%
100%|██████████| 20/20 [00:02<00:00,  7.87it/s]Model test on 20tasks. Accuracy: 84.673%
------------------------
Average accuracy:  0.8467307692307692
------------------------
```

Fig. 10. Average accuracy of ResNet



```
-----------------Fine-tuning-----------------
100%|██████████| 20/20 [00:02<00:00,  7.74it/s]
Model test on 20tasks. Accuracy: 94.846%
100%|██████████| 20/20 [00:02<00:00,  6.99it/s]
Model test on 20tasks. Accuracy: 93.962%
```

Fig. 11. Fine tuning after training For ResNet

4. No use of bottleneck blocks: Standard deeper ResNet models (such as ResNet-50 and ResNet-101) employ bottleneck blocks with 1x1 convolutions at the beginning and end of each block to reduce and then increase dimensions, respectively, with a 3x3 convolution in between. Your model contains only basic blocks, which is typical of shallower ResNets (ResNet-18, ResNet-34).

To summarise, we used ResNet that was specifically designed for single-channel input and a limited number of output classes.

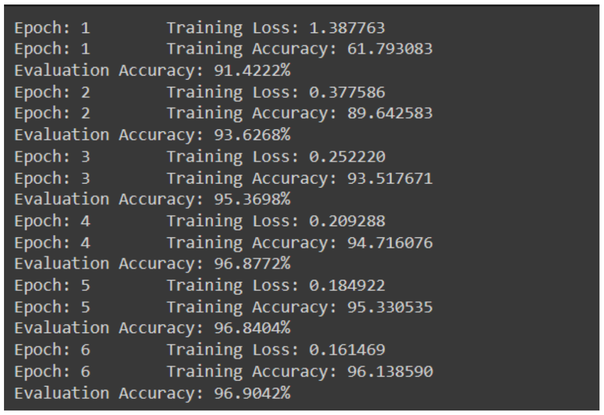
C. ResNet 50

Reasons to implement Standard ResNet 50:

Block Type and Configuration: The standard ResNet-50 employs "bottleneck" blocks, which have three convolutional layers (1x1, 3x3, and another 1x1) to lower, then expand, the dimensions, resulting in a greatly reduced number of parameters and processing complexity. Your implementation employs a simpler "basic block" design with two 3x3 convolutional layers, as is characteristic with shallower ResNets such as ResNet-18 and ResNet-34.

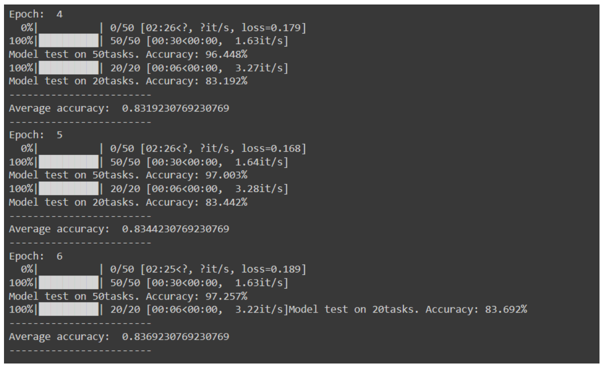
Layer count: ResNet-50's four primary layers are configured with [3, 4, 6, 3] blocks. While you can specify this configuration in your custom implementation by supplying these values to the layers option, the block architecture (using basic blocks instead of bottleneck blocks) differs from that of a genuine ResNet-50.

Block Expansion Factor: ResNet-50 employs a bottleneck block with an expansion factor of four. This means that each



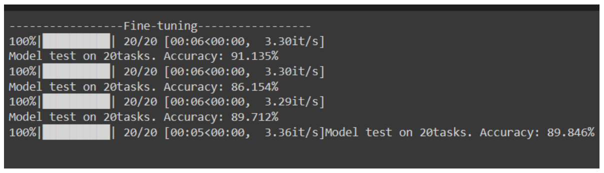
```
Epoch: 1        Training Loss: 1.387763
Epoch: 1        Training Accuracy: 61.793083
Evaluation Accuracy: 91.4222%
Epoch: 2        Training Loss: 0.377586
Epoch: 2        Training Accuracy: 89.642583
Evaluation Accuracy: 93.6268%
Epoch: 3        Training Loss: 0.252220
Epoch: 3        Training Accuracy: 93.517671
Evaluation Accuracy: 95.3698%
Epoch: 4        Training Loss: 0.209288
Epoch: 4        Training Accuracy: 94.716076
Evaluation Accuracy: 96.8772%
Epoch: 5        Training Loss: 0.184922
Epoch: 5        Training Accuracy: 95.330535
Evaluation Accuracy: 96.8404%
Epoch: 6        Training Loss: 0.161469
Epoch: 6        Training Accuracy: 96.138590
Evaluation Accuracy: 96.9042%
```

Fig. 12. Evaluation accuracy of ResNet 50



```
Epoch:  4
  0%|          | 0/50 [02:26<?, ?it/s, loss=0.179]
100%|██████████| 50/50 [00:30<00:00,  1.63it/s]
Model test on 50tasks. Accuracy: 96.448%
100%|██████████| 20/20 [00:06<00:00,  3.27it/s]
Model test on 20tasks. Accuracy: 83.192%
------------------------
Average accuracy:  0.8319230769230769
------------------------
Epoch:  5
  0%|          | 0/50 [02:26<?, ?it/s, loss=0.168]
100%|██████████| 50/50 [00:30<00:00,  1.64it/s]
Model test on 50tasks. Accuracy: 97.003%
100%|██████████| 20/20 [00:06<00:00,  3.28it/s]
Model test on 20tasks. Accuracy: 83.442%
------------------------
Average accuracy:  0.8344230769230769
------------------------
Epoch:  6
  0%|          | 0/50 [02:25<?, ?it/s, loss=0.189]
100%|██████████| 50/50 [00:30<00:00,  1.63it/s]
Model test on 50tasks. Accuracy: 97.257%
100%|██████████| 20/20 [00:06<00:00,  3.22it/s]Model test on 20tasks. Accuracy: 83.692%
------------------------
Average accuracy:  0.8369230769230769
------------------------
```

Fig. 13. Average accuracy of ResNet 50 model



```
-----------------Fine-tuning-----------------
100%|██████████| 20/20 [00:06<00:00,  3.30it/s]
Model test on 20tasks. Accuracy: 91.135%
100%|██████████| 20/20 [00:06<00:00,  3.30it/s]
Model test on 20tasks. Accuracy: 86.154%
100%|██████████| 20/20 [00:06<00:00,  3.29it/s]
Model test on 20tasks. Accuracy: 89.712%
100%|██████████| 20/20 [00:05<00:00,  3.36it/s]Model test on 20tasks. Accuracy: 89.846%
```

Fig. 14. Fine tuning after training For ResNet 50

block's last convolutional layer has four times as many filters as the middle convolution layer. This functionality is missing from your implementation since it utilises a fixed number of filters throughout each block and does not contain the expansion factor found in ResNet-50 models.

Initial Layer Modifications: The change to accept 1-channel input is unique to your needs and differs from the normal 3-channel RGB input that ResNet-50 expects.

D. Difference between Resnet we implemented vs ResNet 50

The key differences between a custom ResNet implementation and the conventional ResNet-50 architecture are the block structure, expansion factor, and overall complexity.

Block Structure:
Custom. ResNet often employs simpler "basic blocks" composed of two 3x3 convolutional layers. These are more straightforward and are typically seen in shallower versions such as ResNet-18 or ResNet-34.
ResNet-50 employs "bottleneck blocks" made out of three layers: a 1x1 convolution to lower depth (channel-wise), a 3x3 convolution at this reduced depth for processing, and a final 1x1 convolution to extend the depth again, frequently by a factor of four. This arrangement aims to reduce computational overhead while maintaining or improving depth.

Expansion Factor:

Custom ResNet: Does not usually use an expansion factor, hence the number of output channels is frequently the same as the number of input channels for each block unless expressly changed.

ResNet-50 has an expansion factor of 4 in its bottleneck blocks, which means that the final 1x1 convolution quadruples the number of channels relative to the input, improving the network's ability to learn more complicated features while not significantly increasing computing demands.

Architectural Complexity:
Custom ResNet is less difficult due to the usage of basic blocks, making it better suited to less demanding applications or smaller datasets.
ResNet-50 is more sophisticated because to its three-layer bottleneck structure, making it more suited for demanding tasks such as large-scale image recognition, which require deep and nuanced feature extraction capabilities.

To summarise, ResNet-50 is designed for depth and efficiency at large scales, with bottleneck blocks used to limit computational costs while expanding network depth and capacity. In contrast, a custom ResNet may use simpler configurations that are better suited to less difficult jobs or computing efficiency.

IV. Conclusion:

Some of the work we did during the project is:

Few Shot Learning: The setup for few-shot learning, which allows a model to learn from a small quantity of data, was described in depth. The preparation of data loaders for this type of learning, as well as the training of models employing Prototypical Networks to predict fresh data from a small number of instances, were discussed.

Differences in Network Design: Differences between custom-built network models and common models such as ResNet-50 were discussed. The emphasis was on how these models are built differently, specifically how ResNet-50 uses more sophisticated blocks for data processing, allowing it to perform better on demanding tasks.

TABLE I
COMPARISON OF ACCURACIES BETWEEN ENHANCED CNN, CUSTOM RESNET AND RESNET 50

Model	***Enhanced Accuracies***	***Average Accuracy***	***Fine tuning***
Enhanced CNN	96.1137%	88.3269%	91.596%
Custom ResNet	96.1258%	85.5557%	93.962%
ResNet 50	96.9042%	83.6923%	91.135%

Model Evaluation and Tuning: This section included testing and fine-tuning models to increase performance. This entailed running models on new data, tweaking parameters such as the learning rate, and employing strategies to keep the model from overfitting.

Table 1 include all the results we got from different models we used.