Deep Learning Course – First Project Report

Convolutional Neural Networks

Marta Szuwarska
01171269@pw.edu.pl
320662

Mateusz Nizwantowski
01161932@pw.edu.pl
313839

April 1, 2025

Contents

1 Introduction

This report presents our first project in the Deep Learning course in Data Science Masters at the Faculty of Mathematics and Information Science. The project aims to introduce us to convolutional neural networks and develop intuitions related to them. We were informed that this work's goal is purely educational and research-focused, so we aim not to create the best recognition model but to learn as much as possible.

For more details about the project and to access the code, please refer to the project repository:
https://github.com/szuvarska/ConvolutionalNeuralNetworks.

1.1 Project Details

The exact topic of this project is *Image classification with convolutional neural networks*. We have been given the data set `CINIC-10`, which we must use for this project. We decided to use PyTorch as a deep learning framework. At the beginning we were experimenting with Tensorflow but we didn't like the API and after some research we realized that PyTorch is crowd favorite. While project is fairly open in its definition there are areas and experiments that we have to cover:

1. Test and compare different network architectures (at least one should be a convolutional neural network).
    - Investigate the influence of the following hyper-parameter change on obtained results:
        - At least 2 hyper-parameters related to training process.
        - At least 2 hyper-parameters related to regularization.
    - Investigate the influence of at least x data augmentation techniques from the following groups:
        - Standard operations ($x = 3$).
        - More advanced data augmentation techniques like cutmix, cutout, or AutoAugment ($x = 1$).
2. Implement one method dedicated to few-shot learning.
    - Reduce the size of the training set and compare the obtained results with those trained on the entire dataset (from the previous point).
3. Consider the application of ensemble (hard/soft voting, stacking).

1.2 Dataset Description

CINIC-10 [1] is a dataset for image classification, and it was designed to serve as a bridge between the widely used CIFAR-10 dataset and the significantly larger ImageNet dataset. Images are split into three subsets: training, validation, and testing. Like CIFAR-10, CINIC-10 includes 10 different classes, which are categories into which the images are grouped. These classes are:

1. **Airplane**
2. **Automobile**
3. **Bird**
4. **Cat**
5. **Deer**
6. **Dog**
7. **Frog**
8. **Horse**
9. **Ship**
10. **Truck**

Each class has 9,000 images in each of the three subsets (training, validation, and testing), so in total, there are 90,000 images per subset and 270,000 images in the whole dataset. The images in CINIC-10 come from two different sources: 60,000 images are from CIFAR-10, and the remaining 210,000 images are from ImageNet, but they have been resized to 32x32 pixels to match the size of the CIFAR-10 images.

CINIC-10 is particularly useful for training and testing image classification models that require more data than CIFAR-10 but are not yet ready for the computational complexity of the full ImageNet dataset, making it an ideal choice for learning deep learning models without extensive computing power.

After some research, we found that the best model trained on this data is `VIT-L/16` [2]. It achieved 95.5% accuracy. Other models worth mentioning are `DenseNet-121` with 91.26%, `ResNet-18` with 90.27%, and `VGG-16` with 87.77% accuracy.

1.3 Project Setup

The project is created using Python Jupyter notebooks. The core packages used in the project include:

- `torch` (2.6.0),
- `torchvision` (0.21.0),

- `matplotlib` (3.10.1),
- `numpy` (2.2.3),
- `timm` (1.0.15),
- `scikit-learn` (1.6.1),
- `seaborn` (0.13.2).

While additional dependencies were used throughout the project, we have chosen not to list them all here, as they are not essential to the main workflow.

Throughout the project, we iterated on various methods. Initially, we had planned to use Marimo notebooks, but after careful consideration - taking into account the tight deadline and our lack of prior experience with this new technology - we decided to stick with the more familiar Jupyter Notebooks.

Additionally, we organized our work with a well - structured directory setup in our GitHub repository to ensure smooth collaboration and clarity. All experiments are fully reproducible, as we set a seed at the beginning of the notebooks.

To replicate the results, execute the notebooks in the order indicated by the numbers in the filenames. Note that the data is not included in the repository; it should be downloaded from the source specified in the previous subsection. To generate augmented versions of the training dataset, run the `src/augmentation.py` script before executing any notebooks in the `notebooks/augmentation` directory.

The recommended execution order is as follows:

1. Notebooks directly in the `notebooks` directory,
2. Notebooks in the `notebooks/few_shot_learning` directory,
3. Notebooks in the `notebooks/augmentation` directory.

However, the order of execution does not affect the final results. To start, ensure that Python is installed and all dependencies listed in the `requirements.txt` file are met.

2 Theory

2.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep learning models that have proven highly effective in visual recognition tasks. A typical CNN consists of multiple layers, including convolutional layers that automatically learn hierarchical features from raw image data. Each convolutional layer applies a set of learnable filters to detect spatial patterns, such as edges and textures. These layers are followed by non-linear activation functions like ReLU, pooling

layers for downsampling, and fully connected layers that ultimately perform classification [3].

The training process of CNNs involves minimizing a loss function through backpropagation, using optimization techniques such as Stochastic Gradient Descent (SGD) or Adam. The softmax function is commonly used in the final classification layer to output class probabilities.

2.2 Transfer Learning with Pre-trained Models

Transfer learning allows models pre-trained on large-scale datasets like ImageNet to be fine-tuned for specific tasks. This approach mitigates the challenges posed by limited data, as the models have already learned useful feature representations that can be adapted to new domains. Popular architectures for transfer learning include VGG, ResNet, EfficientNet, and MobileNet. Fine-tuning involves either retraining the entire model or freezing early layers while adjusting the later layers for the new dataset. This technique significantly reduces training time and improves performance on smaller datasets.

2.3 Few-Shot Learning

Few-shot learning is an area of machine learning focused on training models with a minimal number of labeled examples. This is particularly useful in real-world scenarios where large labeled datasets are scarce. There are multiple approaches to few-shot learning, ranging from conventional CNN-based architectures (including pretrained models) to more advanced techniques such as Prototypical Networks, Siamese Networks, and Model-Agnostic Meta-Learning (MAML).

Prototypical Networks [4] learn a metric space in which the distance between prototypes (representations of classes) is minimized. This model has been effective in few-shot classification tasks, where a new class is represented by only a few labeled examples. Similarly, Siamese Networks [5] use a pair of identical networks to compare two input images and predict whether they belong to the same class. These networks are trained using a contrastive loss function that encourages similar images to have close representations in the feature space.

Model-Agnostic Meta-Learning (MAML) [6] is a meta-learning algorithm designed to enable models to adapt quickly to new tasks with minimal training data. By learning an initialization that can be fine-tuned with a small number of gradient steps, MAML facilitates rapid adaptation in few-shot scenarios.

2.4 Regularization Techniques

To prevent overfitting and improve generalization, regularization techniques such as dropout and L2 regularization (weight decay) can be applied. Dropout randomly disables neurons during training, forcing the network to learn more robust and distributed representations. L2 regularization penalizes large weights, which can help the model generalize better to unseen data.

2.5 Data Augmentation

To improve model generalization and reduce overfitting, one may apply data augmentation techniques that artificially increase the diversity of the training data, making the model more robust to variations in real-world data. Common data augmentation methods include:

- **Random horizontal flip**: Flips the image horizontally with a 50% probability, helping the model become invariant to object orientation.
- **Random rotation**: Applies a random rotation of up to 30 degrees, improving the model's ability to recognize objects from different angles.
- **Random color jitter**: Randomly adjusts brightness, contrast, saturation, and hue, simulating variations in lighting and color conditions.
- **CutMix**[7]: A more advanced augmentation technique that mixes patches from two images. This forces the model to learn more robust features by making it rely on multiple image regions.

3 Experiments

3.1 Preparations to experiments

We didn't start with the experiments right away. Instead, we created a sandbox environment where we were learning and building our knowledge from the ground up. As mentioned previously, we began with Tensorflow. We set up GPU acceleration and followed one or two tutorials on how pipelines in Tensorflow look like. While it worked, we didn't like how it felt, so after quick research (Google's graph of popularity, posts on Reedit and Stack Overflow), we realized that it is not as widely used as we thought and PyTorch is more appreciated by the community. We repeated our initial setup; this time, it was significantly easier because all drivers had already been downloaded.

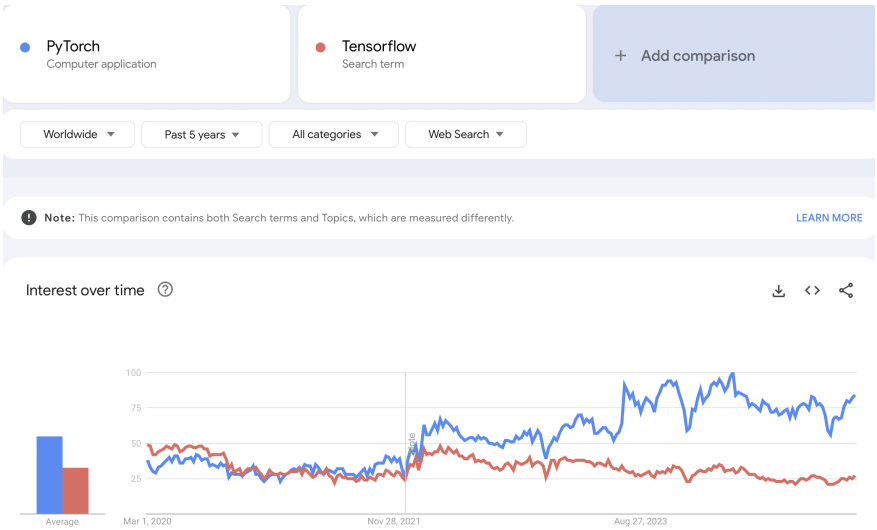
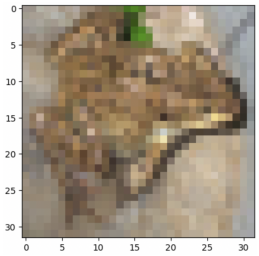
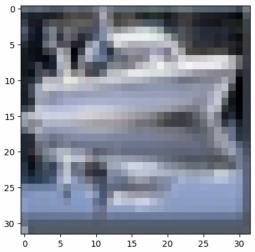


Figure 1: Comparison of interest in PyTorch and Tensorflow based on Google searches

Then, we took a closer look at the dataset we were working with. In the introduction, we already described our findings. During this process, we checked the photos that the dataset consists of and what they look like. We also learned how to technically work with image data, mainly how to use the PyTorch DataLoader class.



(a) Label: Frog



(b) Label: Ship

Figure 2: Example images found in `CINIC-10`

Next, we created a very simple CNN to check if we understand the PyTorch pipeline to create the models and if everything is configured properly. It was working correctly; we used decreasing values of the loss function as an indicator. We made sure that we were using our CUDA devices. With that out of the way, we could focus on creating the models. To understand the theory and ideas behind CNN, we used the website CNN Explainer [8]. This web page has an example architecture that acts as a visualizer, so for our baseline model, we

decided to copy this architecture, and from now on, we will refer to it as a baseline model. Architecture is presented in the Figure 3.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
BaselineModel                            [64, 10]                  --
├─Sequential: 1-1                        [64, 32, 16, 16]          --
│    └─Conv2d: 2-1                       [64, 32, 32, 32]          896
│    └─ReLU: 2-2                         [64, 32, 32, 32]          --
│    └─Conv2d: 2-3                       [64, 32, 32, 32]          9,248
│    └─ReLU: 2-4                         [64, 32, 32, 32]          --
│    └─MaxPool2d: 2-5                    [64, 32, 16, 16]          --
├─Sequential: 1-2                        [64, 32, 8, 8]            --
│    └─Conv2d: 2-6                       [64, 32, 16, 16]          9,248
│    └─ReLU: 2-7                         [64, 32, 16, 16]          --
│    └─Conv2d: 2-8                       [64, 32, 16, 16]          9,248
│    └─ReLU: 2-9                         [64, 32, 16, 16]          --
│    └─MaxPool2d: 2-10                   [64, 32, 8, 8]            --
├─Sequential: 1-3                        [64, 10]                  --
│    └─Flatten: 2-11                     [64, 2048]                --
│    └─Dropout: 2-12                     [64, 2048]                --
│    └─Linear: 2-13                      [64, 10]                  20,490
==========================================================================================
Total params: 49,130
Trainable params: 49,130
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 969.15
==========================================================================================
Input size (MB): 0.79
Forward/backward pass size (MB): 41.95
Params size (MB): 0.20
Estimated Total Size (MB): 42.93
==========================================================================================
```

Figure 3: Architecture of baseline model with 32 hidden units

We wrote functions to train, test and collect metrics. Then, using the baseline model, we tested if there were any bugs. Our first run of this model took 60 seconds, three epochs, and the final test accuracy was 44.1%. While it was not amazing, it meant that everything was working and that we could progress further. We played a little with batch size, but those attempts were very immature, so we won't share them here. We settled for a batch size equal to 64.

At this point, we were in dire need of a refactor. There was a lot of code that had to be copied to the next notebooks. While it was possible, it would be a nightmare to maintain. Notebooks are great while iterating, but it is hard to collaborate while using them, and they have an internal state that can easily introduce bugs while changing code. So, we decided to move all utility functions and model code to .py files. Then, we can import them in notebooks and focus only on experiments we want to perform and not all fuss around it. We repeat this process throughout the project. At first, we build and test in a notebook, then rewrite it to .py files as functions or classes.

Then, we created an improved version of the baseline model. We added one more convolution block; in each of the blocks, we added batch normalization after ReLU, and finally, between convolution blocks, we introduced average

pooling. This model is significantly bigger than the baseline. We are calling it an enhanced model. Both models have *hidden units* as a parameter. This allows us to change the "width" of the network.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
EnhancedModel                            [64, 10]                  --
├─Sequential: 1-1                        [64, 64, 16, 16]          --
│    └─Conv2d: 2-1                       [64, 64, 32, 32]          1,792
│    └─ReLU: 2-2                         [64, 64, 32, 32]          --
│    └─BatchNorm2d: 2-3                  [64, 64, 32, 32]          128
│    └─Conv2d: 2-4                       [64, 64, 32, 32]          36,928
│    └─ReLU: 2-5                         [64, 64, 32, 32]          --
│    └─BatchNorm2d: 2-6                  [64, 64, 32, 32]          128
│    └─MaxPool2d: 2-7                    [64, 64, 16, 16]          --
├─Sequential: 1-2                        [64, 128, 8, 8]           --
│    └─Conv2d: 2-8                       [64, 128, 16, 16]         73,856
│    └─ReLU: 2-9                         [64, 128, 16, 16]         --
│    └─BatchNorm2d: 2-10                 [64, 128, 16, 16]         256
│    └─Conv2d: 2-11                      [64, 128, 16, 16]         147,584
│    └─ReLU: 2-12                        [64, 128, 16, 16]         --
│    └─BatchNorm2d: 2-13                 [64, 128, 16, 16]         256
│    └─MaxPool2d: 2-14                   [64, 128, 8, 8]           --
├─Sequential: 1-3                        [64, 256, 4, 4]           --
│    └─Conv2d: 2-15                      [64, 256, 8, 8]           295,168
│    └─ReLU: 2-16                        [64, 256, 8, 8]           --
│    └─BatchNorm2d: 2-17                 [64, 256, 8, 8]           512
│    └─Conv2d: 2-18                      [64, 256, 8, 8]           590,080
│    └─ReLU: 2-19                        [64, 256, 8, 8]           --
│    └─BatchNorm2d: 2-20                 [64, 256, 8, 8]           512
│    └─MaxPool2d: 2-21                   [64, 256, 4, 4]           --
├─AdaptiveAvgPool2d: 1-4                 [64, 256, 1, 1]           --
├─Sequential: 1-5                        [64, 10]                  --
│    └─Flatten: 2-22                     [64, 256]                 --
│    └─Dropout: 2-23                     [64, 256]                 --
│    └─Linear: 2-24                      [64, 10]                  2,570
==========================================================================================
Total params: 1,149,770
Trainable params: 1,149,770
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 9.79
==========================================================================================
Input size (MB): 0.79
Forward/backward pass size (MB): 234.89
Params size (MB): 4.60
Estimated Total Size (MB): 240.27
==========================================================================================
```

Figure 4: Architecture of enhanced model with 64 hidden units

We trained this bigger model, and we obtained 70% accuracy on the test set. While testing this enhanced model, we noticed that even though we are using GPU, it is not as fast as we expected. We started troubleshooting and realized that one of the CPU cores (threads) is maxed out while other cores and GPUs have low utilization. After a bit of research, we learned that Data Loader

causes bottlenecks. Right now, we will focus on our optimization journey and the steps we performed to reduce the bottlenecks.

3.2 Data Loader optimization

In this subsection, all CPU/GPU usage graphs were taken during training of the baseline model with 10 hidden units, except the last graph, but we will cover it later. In the image 5, we can see that GPU utilization is really low, and only one thread is maxed out. From our experience, it is not worth looking at the percentage usage, especially when only screenshots are provided, because it varies greatly during training. We recommend looking at the "dot" graphs; time is on the x-axis, GPU is at the bottom, CPU is at the top, and the percentage is represented with a number of dots. Additionally, each thread has its own little version of this graph.

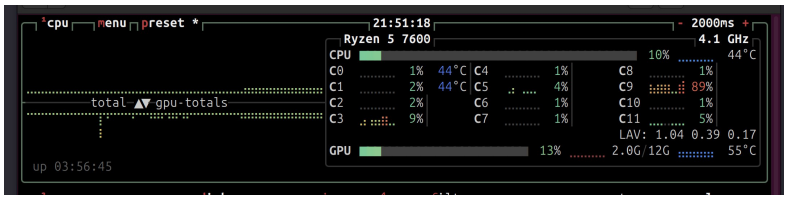


Figure 5: Resource utilization graphs for training with one worker

GPU is waiting (and sitting idle) for the majority of the time. This situation is far from ideal. In this configuration with one threader Data Loader, we get around 19.83s/iter. It will be our point of reference, and we will refer to it as one worker situation. In figure 6, we can see training with two workers. This time, two threads were used, and GPU utilization jumped from 1/2 dots to 2/3. In this configuration, we get around 10.41s/iter.

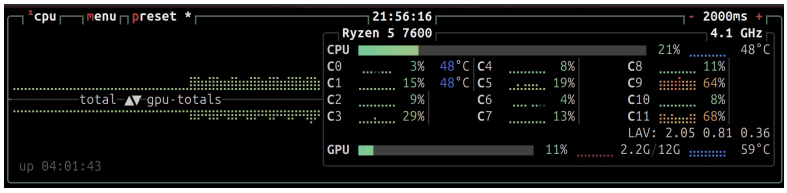


Figure 6: Resource utilization graphs for training with two workers

Training speed is almost doubled. We experimented with how far we could push it. We ended up with six workers. This case can be seen in figure 7. This time, we definitely can see improvement on the graphs, and it is reflected in training time because one iteration takes around 4.83 seconds.

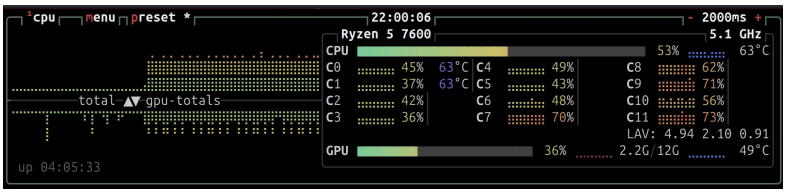


Figure 7: Resource utilization graphs for training with six workers

Reading the documentation, we were testing other options and setting `pin_memory=True` and `persitant_workers=True` gave us a slight but noticeable improvement of 4.23s/iter. Enabling pin_memory allocates page-locked (or "pinned") memory on your CPU, which speeds up data transfers to the GPU, and the persitant_workers reduces the overhead of constantly creating and removing workers. The results of those changes can be seen in figure 9

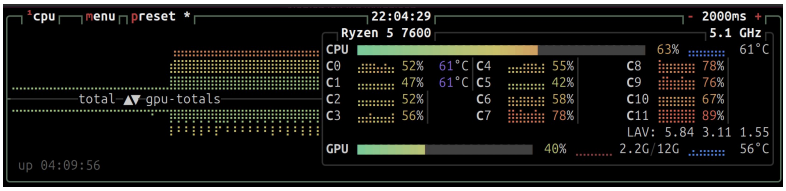


Figure 8: Resource utilization graphs for training with six worker with additional settings

So, in summary, we achieved 5x better performance, but it comes with an asterisk. This gain is possible only when working with small networks (which we and our colleagues are mostly working with during this project) up to around two million parameters (it depends on the hardware used to train). The bigger the networks get, the more strain they put on the GPU. As an example, we prepared one more experiment where we trained an Enhanced model with 128 hidden parameters. It has 4.5 million trainable parameters, and it is enough to fully saturate the 4070 Super - mid-tier graphic cards from 2022. But what is interesting is that this network is only able to do it during training; on inference, where we test our model performance, it only uses around 50% of available computing power. In this configuration, we would achieve negligible improvement by implementing all of the steps described in this subsection. Nonetheless, the majority of our experiments were performed on small networks, and because of this, we were able to run them 4-5 times faster.

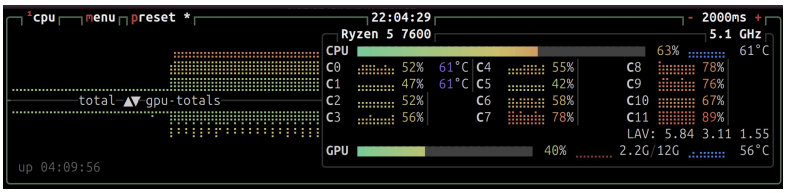


Figure 9: Resource utilization graphs for training with six worker with additional settings

3.3 Training time measurements

We focused on the time aspect of training models. We were curious how time complexity grows with bigger models. In order to measure this, our models (both baseline and enhanced) had a parameter called `hidden_units`. It is an integer that controls "width" and allows us to change the complexity of the network without modifying the underlying architecture.

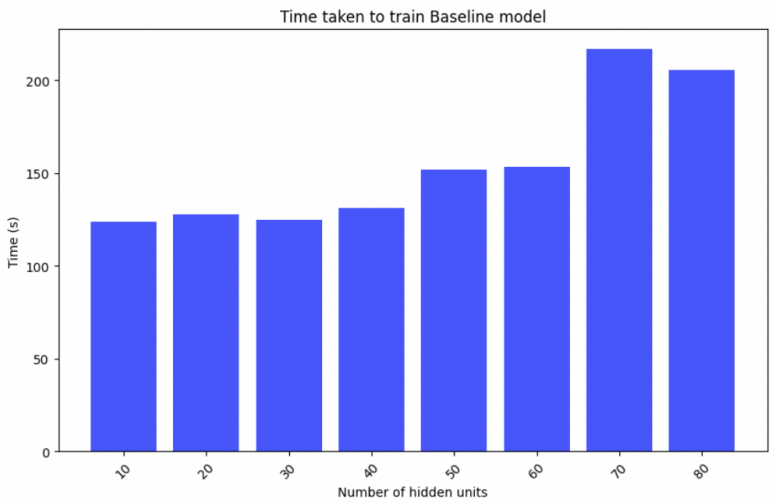


Figure 10: Time of training baseline model for 30 epochs

There is this weird behaviour that the network with 70 hidden units runs for longer than the network with 80 hidden units. We were puzzled by those results, but we re-run this experiment multiple times. The results are in figure 10 and are averaged over 15 runs. We guess GPU caching mechanisms cause this, but we are not sure.

We also compared two version of models and unexpected for us is the fact that while scaling seems to be linear (or at least close to linear), the models have different slope coefficient. This can be seen in figure 11

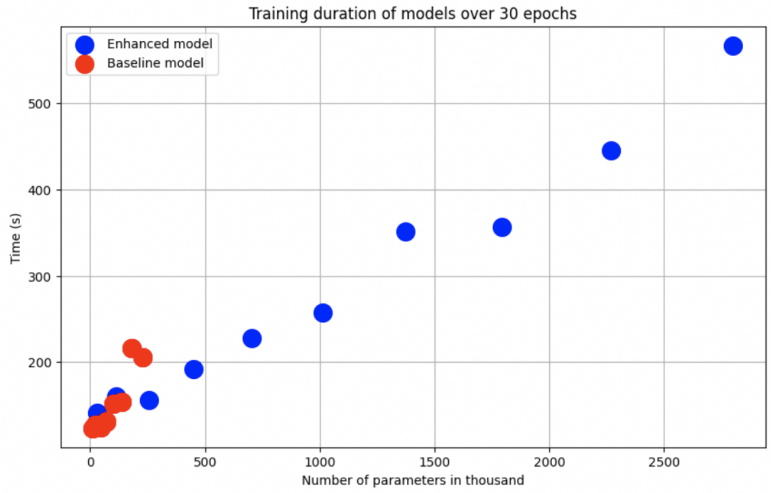


Figure 11: Comparison of training time for baseline and enhanced models for 30 epochs

3.4 Hyper-parameters related to training process

Here, we tested how the learning rate and number of hidden units impact convergence and final score. But at the beginning, we asked ourselves a question: Can we draw conclusions from the results we obtained? It is a question of how stable the training process is, how much it depends on luck and random number generators, and whether the average is a good representative of a run. We created visualizations to help us answer those questions.

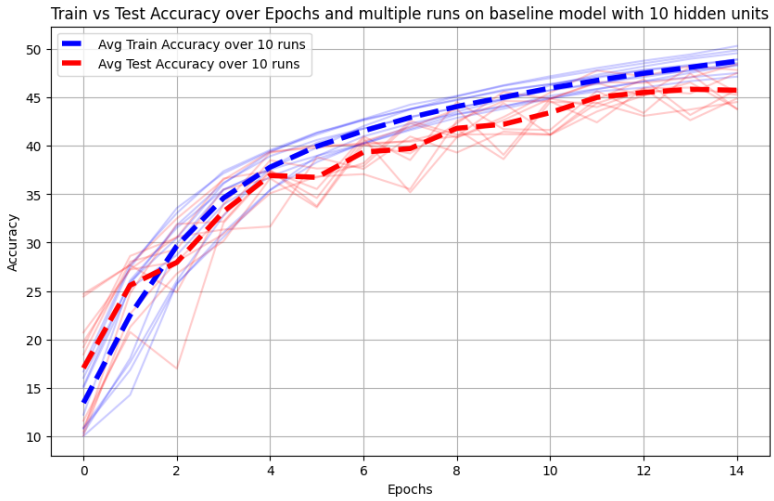


Figure 12: Visualization of how stable is training process

An example can be seen in figure 12. From it, we can draw a conclusion that we can rely on averages. The test accuracy is jagged, but it is because here, we used a high learning rate with Stochastic Gradient Descent. We also created a version of this plot with average and confidence bands, and we used it heavily with Few-Shot learning to characterize models quickly.

The experiments were run on a baseline model. Unsurprisingly, the more hidden units there are, the better, but there are diminishing returns. It tops out at 63-64%.

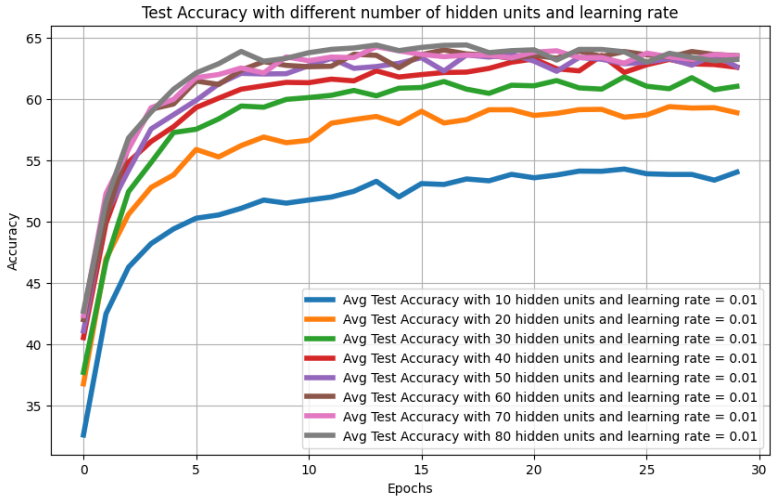


Figure 13: Number of hidden units vs obtained results

We also check how the learning rate impacts training. We found out that no matter the number of hidden units, 30 epochs is enough and even, in some cases, too much for learning rate = 0.01. For learning rate = 0.001, in our configuration, we would need 10-20 more epochs, and it is way too low for learning rate = 0.0001. In the plot, we showed only configurations with 60 or 70 hidden units to avoid poor readability.

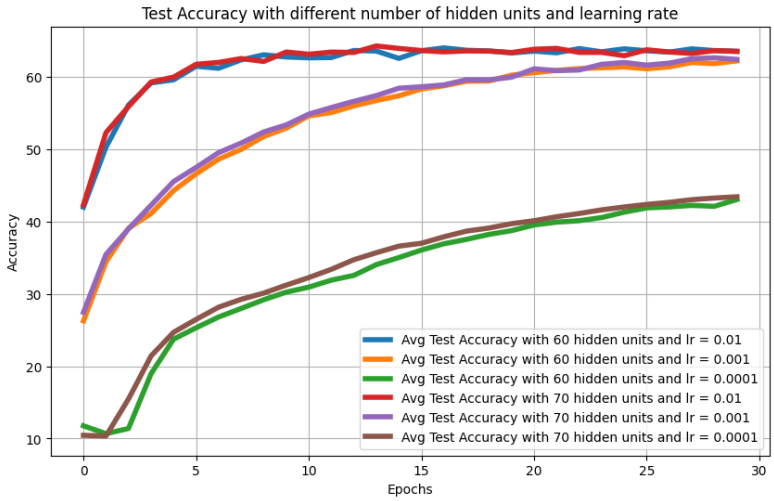


Figure 14: Learning rate and convergence

3.5 Hyper-parameters related to regularization

Regularization is a technique used in neural networks to prevent overfitting and improve generalization. Two common methods are dropout and L2 regularization. Dropout works by randomly disabling a fraction of neurons during training, forcing the network to learn more robust and distributed representations rather than relying on specific neurons. L2 regularization, also known as weight decay, adds a penalty proportional to the squared values of the model's weights to the loss function. This discourages large weight values, leading to a smoother and more stable model. That is why we focused on those two techniques.

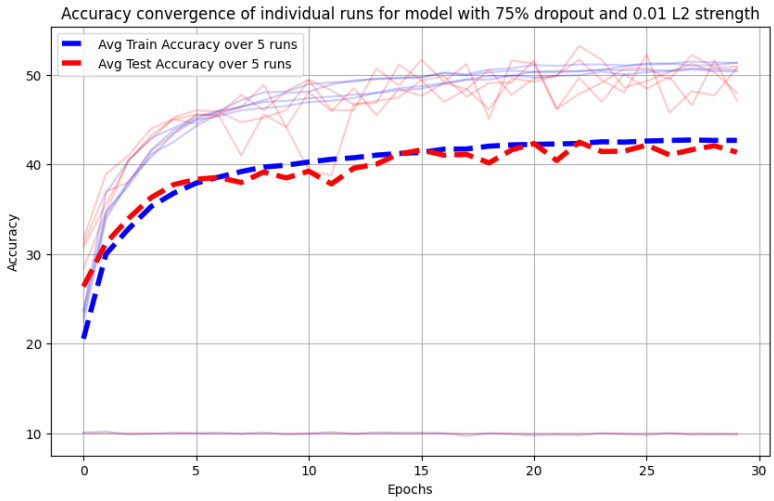


Figure 15: One out of five models collapsed for those extreme parameters

Again, this test featured a baseline model. We checked convergence when using different dropout rates and L2 regularization strengths. In order to converge quickly (this experiment trained 100 different networks), we had to cut some corners, and we used a high learning rate. Unfortunately, this leads, in some extreme cases, to model collapse and random predictions. This is the case when using very high L2 regularization = 0.05 or even L2 strength = 0.01 and dropout = 75%. This can be seen on figure 15

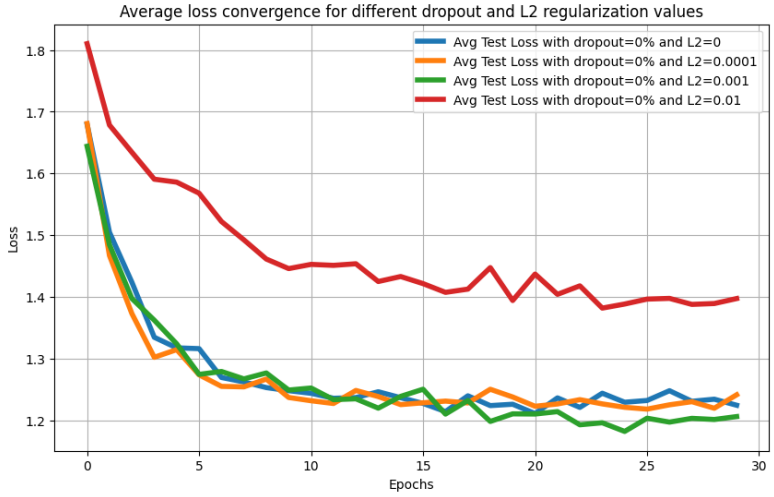


Figure 16: Convergence for models with different regularization strength without dropout

When regularization parameters were not that extreme, we got meaningful results. With dropout set to zero, we can see that the best model is one with regularization = 0.001. It performs marginally better than the model with no regularization and significantly better than a model with too high regularization. Of course, using parameters that cover all available spectra, we didn't obtain the most optimal parameter, but we learned the order of magnitude and where we should look more closely in the future.

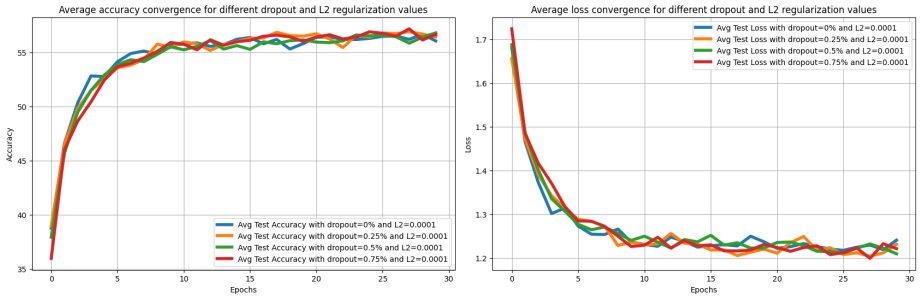


Figure 17: Accuracy and Loss for different dropout rates and L2 = 0.0001

What surprised us is the fact that the dropout (at least in this experiment) did not help at all. To check if there is no mistake, we checked the loss on the train set, but everything looks correct. We are not sure why we got such results.

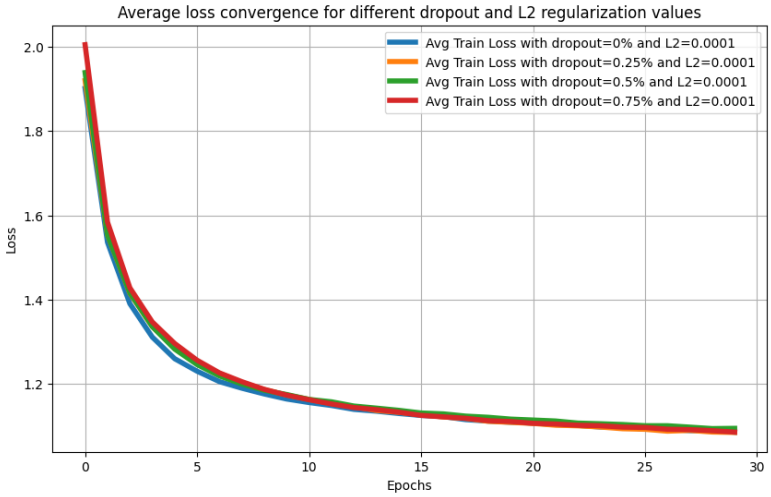


Figure 18: Convergence of loss on train set for example shown in figure 17

The best model we got from this experiment has L2 strength = 0.001 and dropout = 75% and is closely followed by a model with the same L2 value and

dropout equal to 50%.

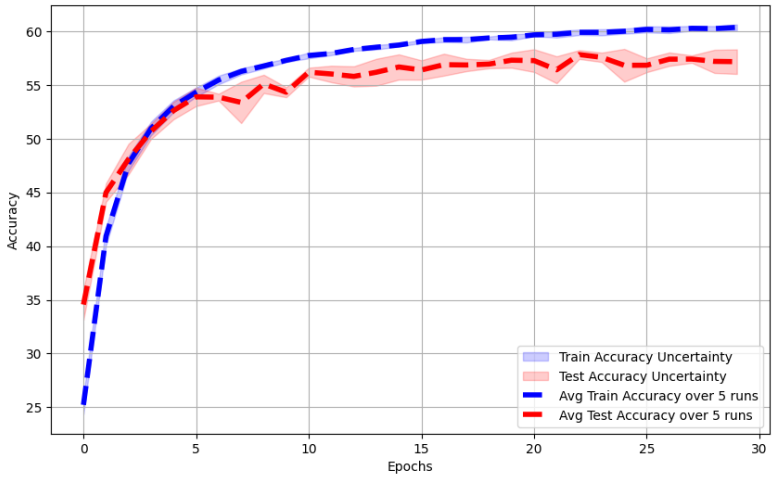


Figure 19: Best model from this experiment has L2 = 0.001 and dropout rate equal 75%

4 Few-Shot Learning

Few-shot learning aims to train models that can generalize effectively from a very limited number of examples. To facilitate this, we developed a specialized data loader that randomly selects a predefined number of images per class (starting with 100) for training and evaluation.

We explored multiple approaches to few-shot learning, ranging from conventional CNN-based architectures (including pretrained models) to more advanced techniques such as Prototypical Networks, Siamese Networks, and Model-Agnostic Meta-Learning (MAML). Unlike standard experiments, where models are evaluated under conventional training conditions, few-shot learning introduces fundamentally different challenges that require dedicated strategies for data preparation, model design, and evaluation.

Due to the distinct nature of these methodologies and their broader implications beyond conventional experimentation, we present our few-shot learning research as a standalone section rather than as a subsection within "Experiments." This structure allows for a more in-depth discussion of the unique characteristics, implementation details, and comparative analysis of different few-shot learning techniques.

4.1 Basic implementation

Our initial approach to few-shot learning involved a simple CNN model consisting of three convolutional layers followed by fully connected layers:

- Three convolutional layers with ReLU activation and max pooling.
- A fully connected layer with 256 neurons and dropout regularization.
- An output layer predicting class probabilities.

This model was trained using the Adam optimizer and cross-entropy loss. We initially set the learning rate to 0.001 and trained for 30 epochs. The performance results are illustrated in Figures 20 and 21.

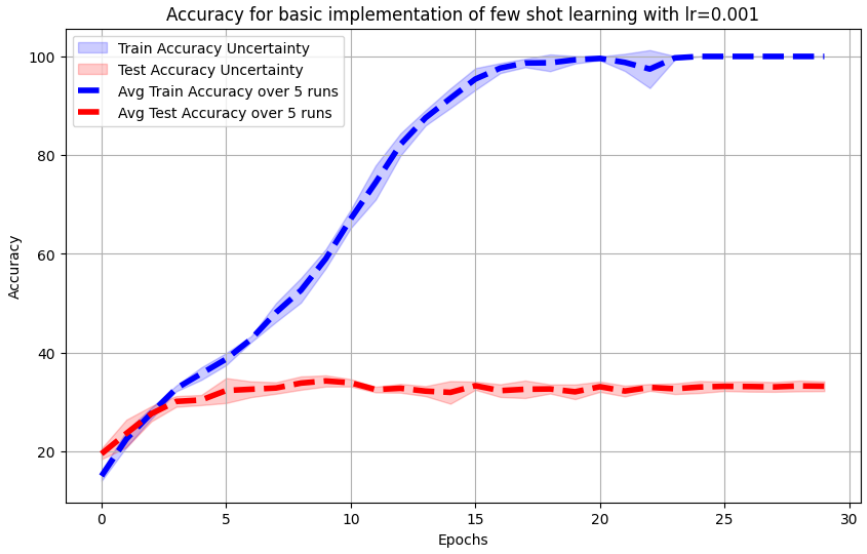


Figure 20: Accuracy over epochs for basic implementation of few-shot learning with lr = 0.001

Figure 21: Loss over epochs for basic implementation of few-shot learning with $\text{lr} = 0.001$

It is evident that the model suffers from severe overfitting. After approximately 15 epochs, training accuracy approaches 100%, whereas test accuracy remains stagnant at around 32%. While some degree of overfitting was anticipated, the extent observed here was unexpectedly high.

To further analyze the model's behavior, we generated the confusion matrix shown in Figure 22.

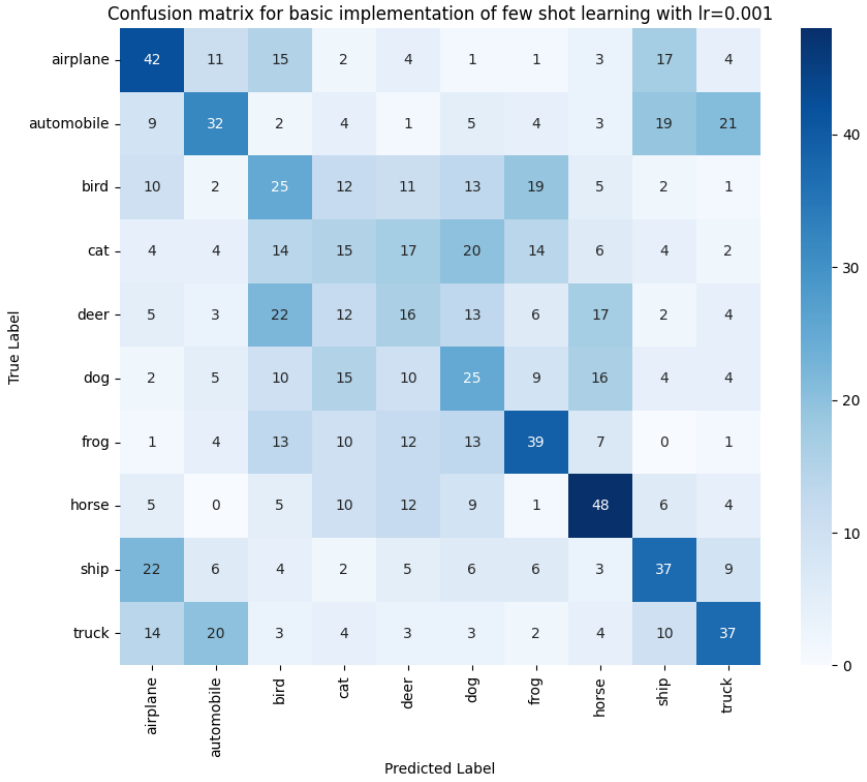


Figure 22: Confusion matrix for basic implementation of few-shot learning with lr = 0.001

As expected, the model correctly differentiates between broad categories such as animals and vehicles. However, frequent misclassifications occurred between visually similar classes, such as automobiles and trucks or ships and airplanes. Interestingly, we were particularly surprised by certain errors among animal classes, such as confusion between birds and frogs, which were not as intuitively similar.

Additionally, we experimented with alternative learning rate values. With lr = 0.0001, the model struggled to learn effectively, achieving only around 60% accuracy on the training set and showing no improvement on the test set. Conversely, with lr = 0.01, the model failed to converge altogether.

4.2 FewShotResNet

To improve upon the baseline CNN, we explored the use of ResNet-18, a well-established deep convolutional neural network. The model was initialized with ImageNet-pretrained weights to leverage transfer learning, and its final fully connected layer was replaced to match the number of classes in CINIC-10.

A key consideration was determining how many layers to fine-tune. We experimented with different configurations, progressively unfreezing layers from the later stages of ResNet-18. The training was conducted using either Adam or SGD optimizers, with different learning rates. The results are summarized in Table 1.

Table 1: Best accuracy values for the train and test datasets, computed over five runs for the FewShotResNet model with different hyperparameter settings.

No.	**Learning Rate**	**Optimizer**	**Unfrozen Layers**	**Cosine Classifier**	**Accuracy**	
					Train	**Test**
1	0.01	Adam	2	no	94%	41%
2	0.001	Adam	2	no	100%	44%
3	0.0001	Adam	2	no	100%	41%
4	0.001	SGD	2	no	100%	43%
5	0.001	Adam	3	no	93%	43%
6	0.001	Adam	1	no	94%	36%
7	0.001	Adam	2	yes	90%	44%

The highest test accuracy of 44% was achieved with a learning rate of 0.001, the Adam optimizer, and two unfrozen layers. Introducing the Cosine Classifier had negligible impact on performance, so we decided to discard it in subsequent experiments. Figures 23 and 24 show the accuracy and loss curves for the best-performing model without the Cosine Classifier.

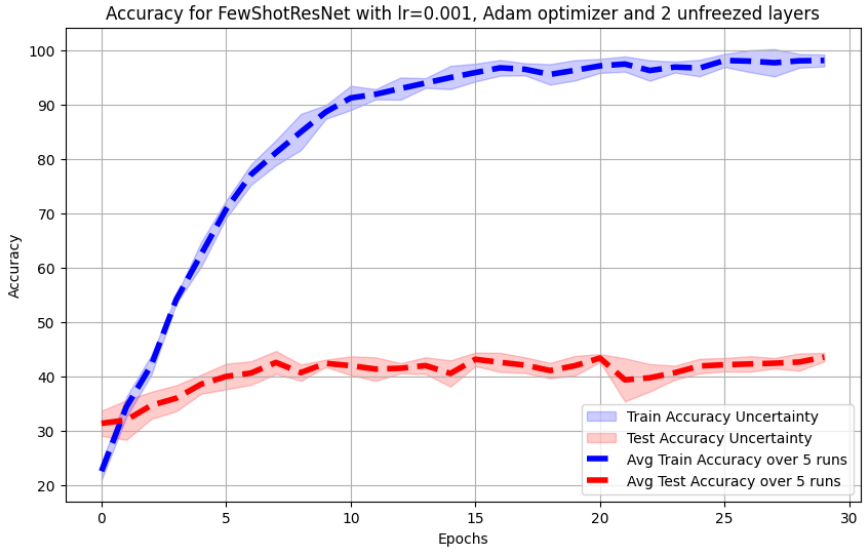


Figure 23: Accuracy over epochs for FewShotResNet model with lr = 0.001, Adam optimizer and 2 unfrozen layers

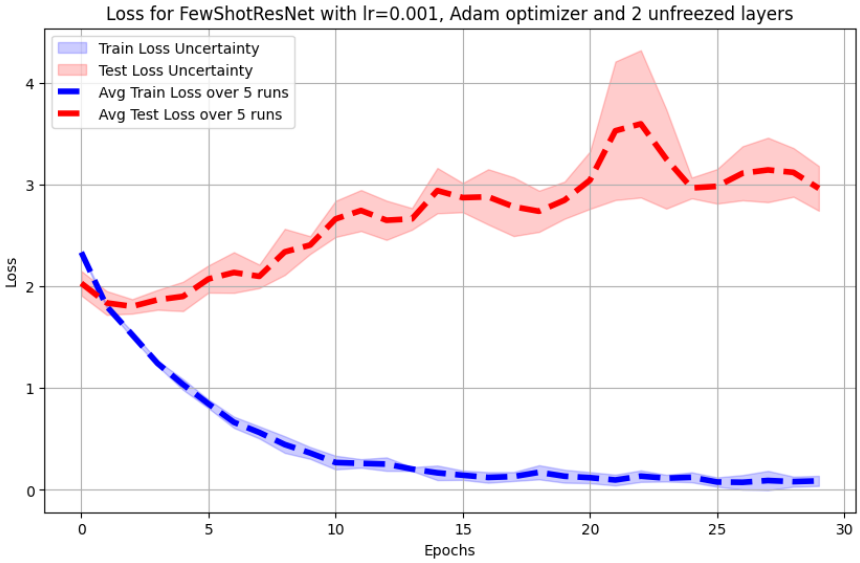


Figure 24: Loss over epochs for FewShotResNet model with lr = 0.001, Adam optimizer and 2 unfrozen layers

Examining these plots, we observe that the training loss consistently decreases and stabilizes after approximately 15 epochs. However, the test loss behaves erratically, fluctuating and generally increasing. Meanwhile, training accuracy improves steadily, whereas test accuracy plateaus after around five epochs.

This behavior was unexpected and difficult to interpret. Similar patterns were observed across different hyperparameter configurations, suggesting a fundamental limitation in the model's ability to generalize with limited data.

4.3 FewShotEfficientNet

We also experimented with EfficientNet-B0, a more recent architecture known for its efficiency and strong performance across various image classification tasks. EfficientNet employs a compound scaling strategy, balancing depth, width, and resolution to maximize performance while maintaining computational efficiency.

Building on our previous findings, we unfroze the last two layers and used the Adam optimizer. We tested learning rates of 0.001, 0.0001, and 0.00001. The highest learning rate (0.001) prevented convergence entirely, yielding results similar to a randomly initialized model. Lower learning rates allowed the model to reach 100% accuracy on the training dataset; however, test accuracy remained significantly lower—approximately 30% for lr = 0.0001 and about 17% for lr = 0.00001.

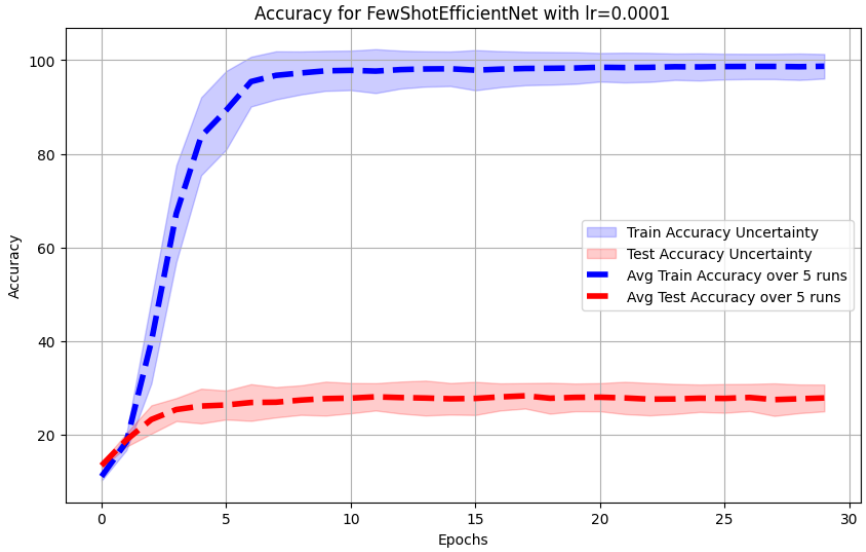


Figure 25: Accuracy over epochs for FewShotEfficientNet model with lr = 0.0001, Adam optimizer and 2 unfrozen layers

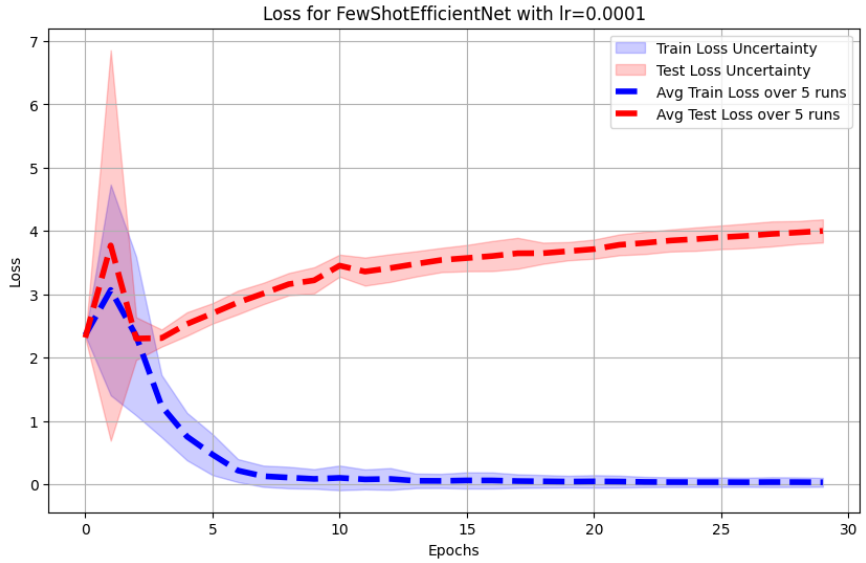


Figure 26: Loss over epochs for FewShotEfficientNet model with lr = 0.0001, Adam optimizer and 2 unfrozen layers

As shown in Figures 25 and 26, the model exhibits a behavior similar to the previous approach. Despite achieving perfect training accuracy, its generaliza-

tion capability remains poor, as indicated by the low test accuracy and erratic loss trends. This suggests that even with a highly efficient architecture like EfficientNet, few-shot learning remains a challenging problem, likely requiring more sophisticated adaptation strategies.

4.4 FewShotConvNeXt

To further explore modern architectures for few-shot learning, we experimented with ConvNeXt-Tiny, a convolutional model inspired by transformer-based architectures such as Vision Transformers (ViTs). ConvNeXt retains the advantages of convolutional neural networks while incorporating design principles from transformers, including large kernel sizes and improved normalization techniques. This makes it a strong candidate for few-shot learning tasks.

For this model, we experimented with different optimizers (Adam and SGD) and learning rate scheduling strategies to stabilize training and enhance generalization.

Table 2: Best accuracy values for training and test datasets obtained with the FewShotConvNeXt model under different configurations over five runs.

No.	**Learning Rate**	**Optimizer**	**Scheduling**	**Accuracy**	
				Train	**Test**
1	0.001	Adam	no	98%	50%
2	0.0001	Adam	no	100%	60%
3	0.00001	Adam	no	100%	59%
4	0.0001	Adam	yes	100%	63%
5	0.0001	SGD	no	100%	60%
6	0.0001	SGD	yes	98%	59%

As shown in Table 2, most test accuracy results clustered around 60%, with the highest accuracy of 63% achieved using an initial learning rate of 0.0001, the Adam optimizer, and learning rate scheduling. Based on these findings, we adopted these parameter values for subsequent experiments unless stated otherwise.

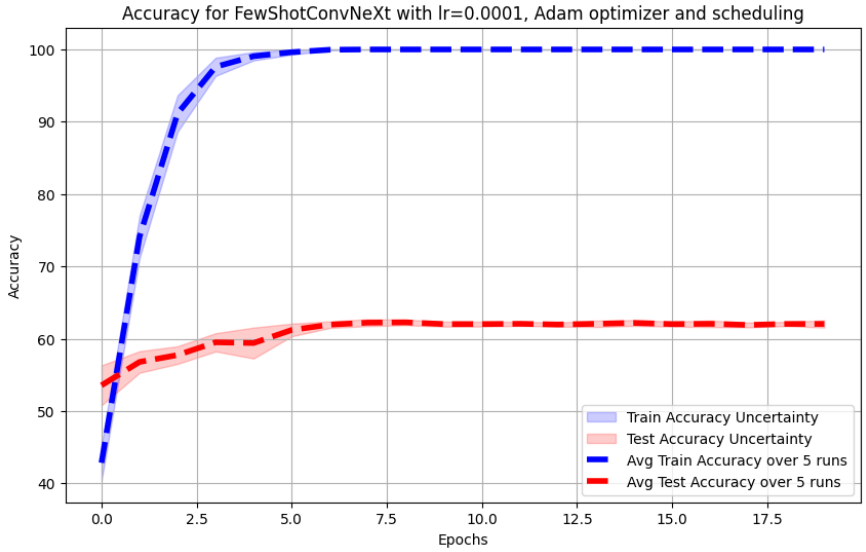


Figure 27: Accuracy over epochs for FewShotConvNeXt model with lr = 0.0001, Adam optimizer and scheduling.

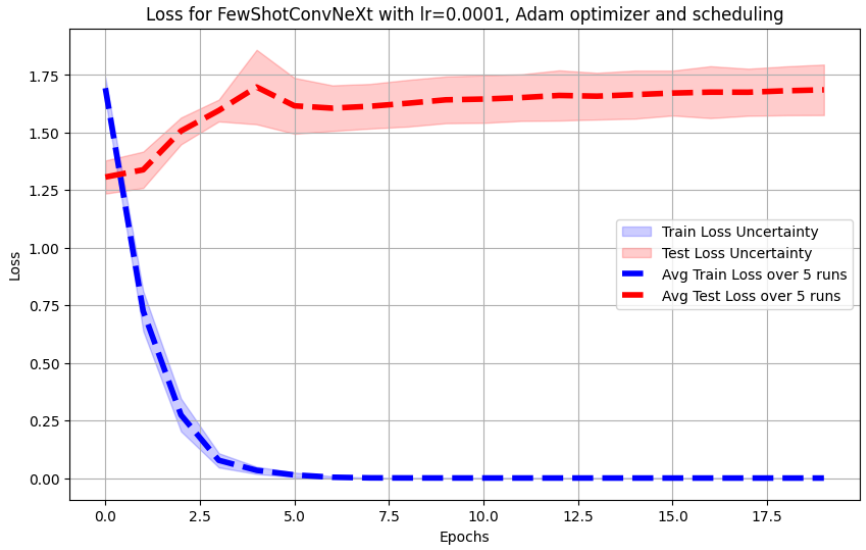


Figure 28: Loss over epochs for FewShotConvNeXt model with lr = 0.0001, Adam optimizer and scheduling.

Figures 27 and 28 reveal more pronounced changes in training accuracy during the initial epochs compared to previous experiments. The training accuracy

increases rapidly, while the loss value decreases sharply. For the test dataset, however, both accuracy and loss stabilize after a relatively short period. This suggests that while the model learns effectively from the training data, its ability to generalize remains a challenge, similar to previous architectures.

4.5 Other methods

In addition to the previously discussed models, we explored alternative few-shot learning approaches, including a Prototypical Network based on FewShotConvNeXt, a Siamese Network, and Model-Agnostic Meta-Learning (MAML) based on ResNet-18. Unfortunately, none of these methods yielded promising results, so we limited our experimentation.

For all models, we used the Adam optimizer. The Prototypical and Siamese Networks were trained with a learning rate of 0.0001, while for MAML, we used an inner learning rate of 0.01 and an outer learning rate of 0.001. Additionally, we applied learning rate scheduling for the Prototypical Network. In all cases, both training and test accuracy hovered around 50%, except for MAML, where test accuracy was significantly lower. The results are summarized in Table 3.

Table 3: Accuracy values for train and test datasets computed for Prototypical Network based on FewShotConvNeXt, Siamese Network, and Model-Agnostic Meta-Learning (MAML) based on ResNet18 (single runs).

No.	**Method**	**Learning Rate**	**Optimizer**	**Scheduling**	**Accuracy**	
					Train	**Test**
1	Prototypical	0.0001	Adam	yes	47%	46%
2	Siamese	0.0001	Adam	no	49%	51%
3	MAML	0.01+0.001	Adam	no	100%	14%

As shown in Table 3, the Prototypical and Siamese Networks achieved train and test accuracy values close to 50%, suggesting that they struggled to learn meaningful representations beyond chance-level performance. MAML, on the other hand, achieved 100% training accuracy but exhibited severe overfitting, with test accuracy dropping to just 14%. This highlights a key challenge in meta-learning approaches: while they can effectively memorize training data, their generalization to unseen examples remains problematic.

4.6 FewShotConvNeXt with regularization

Since most of our models exhibited significant overfitting, we explored the impact of regularization techniques, specifically dropout and weight decay. We tested various values for both parameters and evaluated their effect on the FewShotConvNeXt model. The results are summarized in Table 4.

Table 4: The best accuracy values for train and test datasets computed for FewShotConvNeXt model with different regularization parameters over 5 runs.

No.	**Dropout**	**Weight Decay**	**Accuracy**	
			Train	**Test**
1	-	-	100%	61%
2	0.25	-	100%	60%
3	0.5	-	100%	61%
4	0.75	-	100%	59%
5	-	1e-5	100%	60%
6	-	1e-4	100%	61%
7	-	1e-3	100%	60%
8	-	1e-2	100%	59%
9	-	1e-1	92%	54%
10	-	1e-0	51%	47%

Overall, the test accuracy remained around 60%, similar to the results obtained without regularization. However, extremely high weight decay values led to a noticeable drop in performance. Notably, when weight decay was set to 10^{-1} or higher, the training accuracy significantly decreased, suggesting that excessive regularization impeded the model's ability to learn useful representations.

These findings suggest that while regularization helps prevent overfitting in some cases, excessive weight decay may hinder the model's ability to learn effectively.

4.7 FewShotConvNeXt with a smaller training dataset

In all previous experiments, we used a training dataset containing 100 images per class. To assess the impact of reducing the dataset size, we conducted an experiment with only 16 images per class while keeping the test dataset unchanged at 100 images per class. The results of this experiment are shown in Figures 29 and 30.

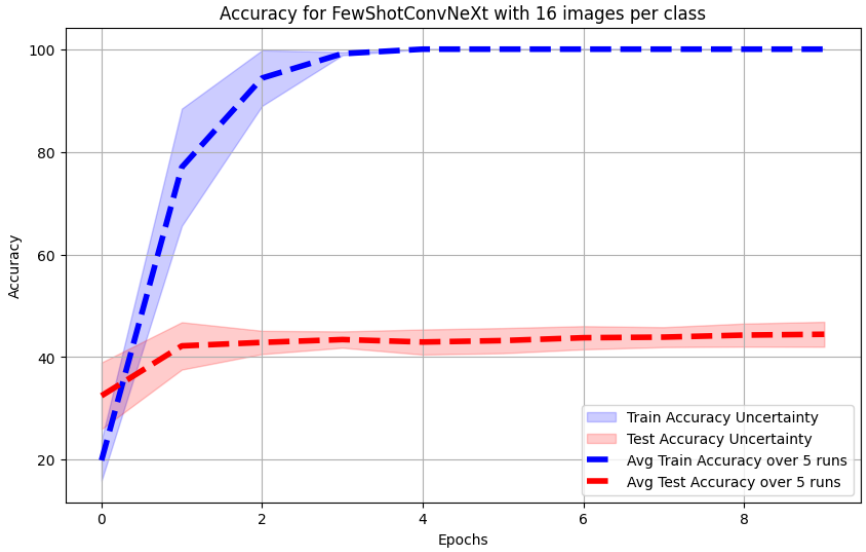


Figure 29: Accuracy over epochs for FewShotConvNeXt model with lr=0.0001, Adam optimizer and scheduling, trained on a dataset containing 16 images per class.

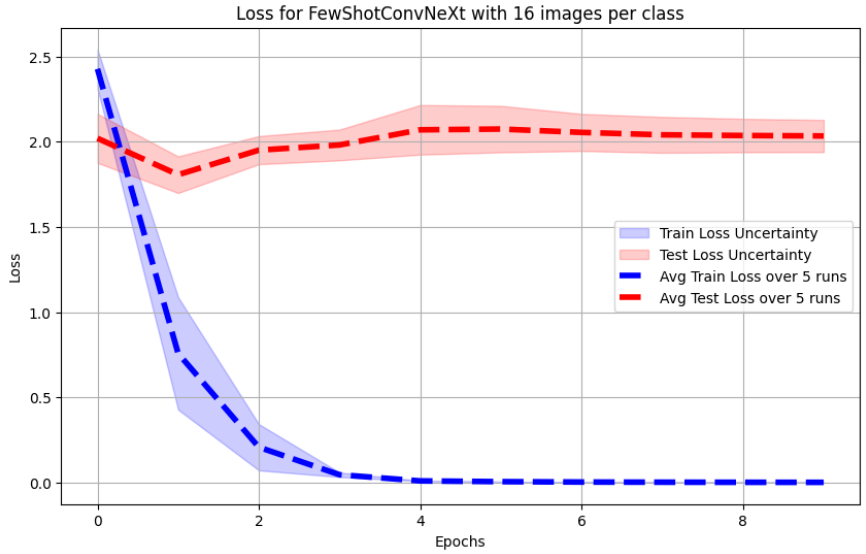


Figure 30: Loss over epochs for FewShotConvNeXt model with lr=0.0001, Adam optimizer and scheduling, trained on a dataset containing 16 images per class.

As expected, reducing the training dataset size resulted in a performance

decline. The test accuracy dropped from approximately 60% (with 100 images per class) to around 40% (with 16 images per class). This highlights the model's increased difficulty in generalizing when trained on fewer examples, reinforcing the importance of dataset size in few-shot learning scenarios.

4.8 FewShotConvNeXt with augmentation

Data augmentation is a crucial technique in machine learning, especially when training models on limited datasets. By applying a variety of transformations to the original data, we can artificially increase the size of the dataset and improve the model's ability to generalize to unseen examples. This is particularly important in few-shot learning tasks, where the number of labeled examples is limited.

In this work, several augmentation methods were employed, including basic techniques such as random horizontal flips, random rotations, and color jittering. Additionally, more advanced technique CutMix was applied to further enhance the model's robustness and performance.

These augmentations were applied to the CINIC-10 dataset, with each method tested individually, as well as a combination of the three basic techniques (flip, rotation, and jitter). The results obtained when training the model on augmented datasets are presented in Table 5.

Table 5: Best accuracy values for train and test datasets computed for the FewShotConvNeXt model trained with different augmentation techniques (averaged over 5 runs).

No.	**Augmentation**	**Accuracy**	
		Train	**Test**
1	Flip	100%	61%
2	Rotation	100%	55%
3	Jitter	100%	59%
4	Mixed	100%	54%
5	CutMix	100%	39%

In most cases, data augmentation did not improve model performance, and in some cases, it even led to worse results compared to training on the non-augmented dataset. The best performance was observed when using horizontal flipping, which yielded similar results to the baseline. The worst accuracy was obtained with CutMix, suggesting that this technique may not be well-suited for the FewShotConvNeXt model in this context.

5 Summary

We really enjoyed the project; we learned much and now know more about neural networks. We familiarized ourselves with the PyTorch framework and

explored various aspects of image recognition using deep learning. While we didn't manage to implement all the ideas, we created a set of abstractions that, in theory, allow us to test it easily. We might give it a try in the future, but for now, we are happy with the results we obtained.

That being said, if we could change one aspect of the project, we would make it more specific. As it stands, the scope feels quite broad, making the project somewhat generic. Additionally, we felt that some form of competition was missing, which could have made the experience even more engaging.

References

[1] L. N. Darlow, E. J. Crowley, A. Antoniou, and A. J. Storkey, "CINIC-10 is not imagenet or CIFAR-10," *CoRR*, vol. abs/1810.03505, 2018.

[2] H. M. D. Kabir, M. Abdar, S. M. J. Jalali, A. Khosravi, A. F. Atiya, S. Nahavandi, and D. Srinivasan, "Spinalnet: Deep neural network with gradual input," *CoRR*, vol. abs/2007.03347, 2020.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[4] J. Snell, K. Swersky, and R. Zemel, "Prototypical networks for few-shot learning," in *Proceedings of Neural Information Processing Systems (NeurIPS)*, 2017.

[5] D. Chicco, H. Sutherland, and G. Jurman, "Siamese neural networks: An overview," *BioData Mining*, vol. 13, no. 1, pp. 1–17, 2020.

[6] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[7] S. Yun, D. Han, S. Oh, S. Chun, C. Kim, S. Yoo, and J. Kim, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 6023–6032, 2019.

[8] Z. J. Wang, R. Turko, O. Shaikh, H. Park, N. Das, F. Hohman, M. Kahng, and D. H. Chau, "CNN explainer: Learning convolutional neural networks with interactive visualization," *CoRR*, vol. abs/2004.15004, 2020.