Deep Learning: Project 1 Report

Maja Andrzejczuk, Maciej Orsłowski

Contents

1 Research Problem

The research focuses on investigating convolutional neural networks (CNNs), comparing various architectures using the CINIC-10 collection (*CINIC10* 2019). This dataset comprises images categorized into 10 classes, with 9000 images per class per subset. This dataset is already divided into training, testing, and validation sets.

In the experiments, the effects of different augmentation techniques on the photo training process were examined. This included both simple methods such as photo rotation, horizontal flip, and solarization effect, as well as advanced methods like MixUp or CutMix.

Various hyperparameters were tested, including optimizers (SGD, Adam), batch sizes, learning rates, regularization L2, and momentum.

Following the selection of optimal models, ensemble methods were employed to improve forecasting accuracy by combining predictions from multiple models.

2 Instruction of the application

Our application is a Python Notebook (.ipynb format). It requires Python 3.10 and the following packages:

- torch (2.2.1),
- torchvision (0.17.1),
- matplotlib (3.8.3),
- seaborn (0.12.2),
- bayesian-optimization (1.4.3),
- pandas (1.5.0).

Unfortunately, the code can only be run on Windows, due to bayesian-optimization package requiring this OS to work.

All experiments are fully reproducible thanks to seeds being set before each training and optimisation. On top of that, the training is configured to utilise CUDA if a hardware that it runs on supports it, which can significantly shorten training times. The application automatically checks whether PyTorch can use CUDA and sets the device accordingly.

Before running the application, one has to keep in mind that training times are very long, even when using CUDA. Some of the training code can run for many hours, with the entire notebook taking days to complete on an average hardware. To alleviate this, all the results of lengthy operations are saved once these operations are completed. This includes model training, parameter optimisation and final predictors' evaluation. These results can then be loaded, which means a user does not have to run everything in the notebook at once to replicate our experiments. The code which loads the data is provided in the notebook, however some of the save paths might be different for different users as they may depend on a timestamp when the operation was completed (this was put in place for training results, to eliminate the risk of overwriting data).

3 Theoretical Introduction

Convolutional Neural Networks (CNNs) have revolutionized image processing by preserving spatial structures and efficiently capturing patterns through convolutional layers. To enhance model performance, it's beneficial to employ augmentation techniques, hyperparameter optimization and regularization methods, which introduce dataset variations and help preventing overfitting. Additionally, applying Ensembling allows for performance improvement by combining predictions from multiple models.

3.1 CNN

Convolutional neural network (CNN) is a type of neural network that is useful when dealing with images. Standard neural networks can also process images, however they require input to be flattened, which means it loses its spatial structures. On the other hand, CNNs alleviate this problem by processing images as matrices.

A convolutional layer is characterised by a number of input channels, a number of output channels, a filter size, a stride and a padding size. Images going into the network in our case have 3 channels, one for each color (RGB). The number of output channels in the convolutional network should normally be higher or equal to the number of input channels. A filter (or a kernel, as it is named in PyTorch) is a matrix, which usually has an equal width to height, and a depth equal to the number of channels in the input tensor. Its width and height are parameters that can be tweaked when designing a network architecture. Convolutional layers process an image by convolving over all chunks of the image which are of the same size as the filter, and taking a dot product of these chunks and the filter. This can be denoted as simply:

$$w^T x + b$$

where w is the filter, x is a chunk, and b is a bias. This results in a single number, so if applied over all chunks of the image, will eventually output a 2D matrix. Its shape also depends on another parameter - a stride, which is the number of pixels we skip when going from one chunk of the image to the next one (see Figures 1 and 2). Additionally, a padding can be added, which is equivalent to adding 0s around the image, and its size will also affect the shape of the output matrix.

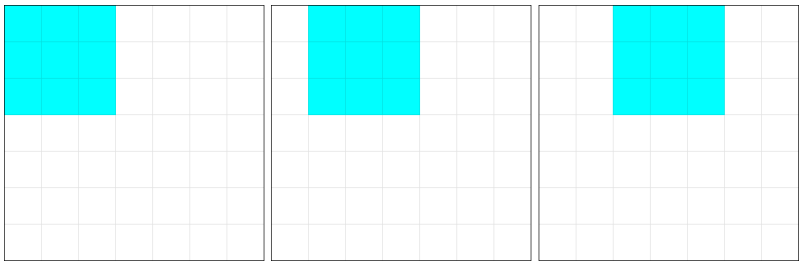

Figure 1: Chunks which filter is applied to with stride equal to 1

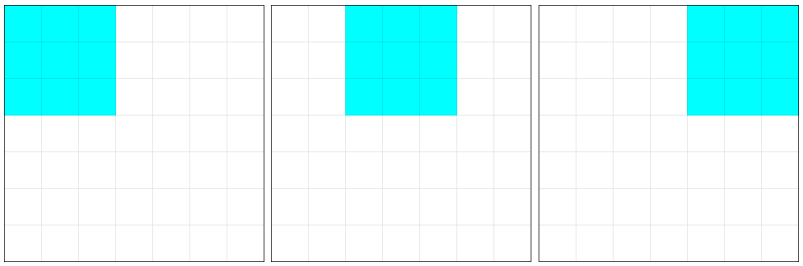

Figure 2: Chunks which filter is applied to with stride equal to 2

Finally, for a square filter, if we denote filter size by f, image height and width by h and w respectively, stride by s, and padding size by p, the output of the convolutional layer is a matrix with its height and width equal to (respectively):

$$h_{out} = \frac{h - f + 2p}{s} + 1$$
$$w_{out} = \frac{w - f + 2p}{s} + 1$$

This operation can be repeated multiple times with multiple different filters, resulting in more output channels of the layer, and can help discover different features in the image. The elements of the matrix can also have an activation function applied to them.

Another concept related to CNNs are pooling layers, which are often used to compress the outputs of convolutional layers. They are similar to convolutional layers in a way that they also process an image by chunks, and from each chunk they output a single number. However, there are many differences, for example, pooling layers do not change the number of channels of the image. They also do not have any parameters - instead of using filter weights and applying dot products, they simply aggregate numbers in the chunk. For example, a max-pool layer would take the maximum of all numbers in a chunk, and an average-pool layer would return an average of all the numbers. Pooling layer hyperparameters are only a filter size and a stride, and these two are often equal to each other, which results in chunks that do not overlap. The output size is calculated analogously to convolutional layers.

Deep neural networks can suffer from the problem called the vanishing gradient, which means that the gradient propagated backwards approaches zero. This results in weights of initial layers not being updated, which makes the training process ineffective. Alternatively, a phenomenon called exploding gradient can occur, which is the opposite of vanishing gradient. Here, the gradients get very large, which can either result in overflows, or make the training unstable and cause gradient descent to diverge. ResNets are a type of deep neural network which attempt to avoid these issues by skipping some of the layers while training (He et al. 2015). ResNet-18, which we use in our experiments, is a type of ResNet which is 18 layers deep.

3.2 Augmentation

Augmentation is a technique used to generate new data by modifying existing data. Augmentation techniques allow for introducing diversity and complexity into the dataset, resulting in improved accuracy of the results. This method of data transformation helps prevent overfitting of the training data and enhances the performance of the model.

In our solution, we employed various augmentations. Some of the simple ones included:

- **Crop and Resize**: This augmentation randomly crops and resizes the image to its original dimensions.
- **Rotation**: Rotates the image with a random probability.
- **HorizontalFlip**: Flips the image horizontally, effectively mirroring it.
- **Invert**: Inverts the colors of the image.
- **Solarize**: Adjusts the image intensity to create a solarization effect.

In augmentation, instead of modifying individual images, one can also employ advanced techniques involving the creation of new data by combining various images into one. Among the popular advanced augmentation techniques are CutMix and MixUp. During these augmentations, two images are merged into one. The resulting image is assigned two class labels corresponding to the images used to create this new image.

- **MixUp**: augmentation technique combines two images by mixing their pixel values based on a certain ratio.
- **CutMix**: replace the removed regions with a patch from another image. The ground truth labels are also mixed proportionally to the number of pixels of combined images.

Although augmentations help in achieving better results by allowing to prevent data overfitting, caution should be taken when using them because it is easy to damage the data. Introducing too many augmentations can cause too much change in the images, only resulting in worse performance. In addition, if the augmentations are too strong, they can cause images to learn augmentation patterns instead of paying attention to details in the original images.

3.3 Hyperparameters

In the process of training models, a crucial aspect is the selection of appropriate learning parameters. These parameters can be related to the training process itself or associated with regularization. During the training process, finding optimal hyperparameters is important to enhance the model's performance, these include:

- Optimiser: an algorithm used to update network weights in each iteration, for example Adam, SGD or RMSprop.
- Batch size: determines the number of samples to be processed before updating internal model parameters.
- Learning rate: determines the step size in each iteration while approaching optimal weights. It controls how much the weights will change during one parameter update.
- L2 regularization coefficient: L2 Regression, also known as Ridge Regression, is a method that involves imposing a penalty on the loss function for large coefficient values. This way, L2 regularization encourages weights to remain as small as possible, thereby helping to reduce

the complexity of the model and prevent overfitting in machine learning models. The formula for L2 regularization is given by:

$$\text{L2 Penalty} = \lambda \sum_{j=1}^{p} \beta_j^2$$

Where:

$$\sum_{j=1}^{p} \beta_j^2$$

represents the sum of squares of all model coefficients (β_j) and λ is the regularization coefficient, controlling the strength of regularization. A higher value of λ results in stronger regularization.

- Dropout: refers to the practice of disregarding certain nodes in a layer at random during training. A dropout is a regularization approach that prevents overfitting. The nodes are dropped by a dropout probability of p.
- Momentum: determines how much model parameter updates take into account previous changes in weights, accelerating the convergence of the optimization algorithm.

In machine learning, various methods exist for selecting appropriate hyperparameters, including:

- Grid search: involves determining specific values for each parameter and then searching through all possible combinations.
- Random search: consists in setting for each parameter a range of values and then searching for random combinations of parameters.
- Bayesian optimization: is an advanced method of parameter selection. It consists in setting for each parameter a specific range of values and random selection of initial parameters, and then using a probabilistic model to determine the next parameters worth testing. This method is very effective because it uses information from the previous iteration.
- Gradient-based optimization: involves the use of an additional network whose learning objective would be to determine the optimal hyperparameters.

Each of these methods has its advantages and is suited to different scenarios in optimizing model performance.

3.4 Ensembling

Individual models may not get good enough results. Ensembling is a technique for improving performance by using a combination of results from multiple models. By taking the predictions of multiple models and coalescing them into a common result, we aim to improve performance. One of the popular ensembling techniques includes hard voting and soft voting.

The hard voting method involves counting the final predictions of each model and taking the class with the highest number of votes as the final prediction. An important aspect of hard voting is to consider the situation in which a tie occurs. The implementation should indicate how the predictor should behave in a situation where there is no clear winning class. For example, you can choose the prediction of the model that was most certain of its decision. The advantage of hard voting is its simplicity of implementation, which assumes a recount of votes. The inaccuracy of hard voting is that it does not take into account the certainty of the models' predictions, but only their final decision.

The soft voting method is an ensemble technique similar to hard voting. In this method, we consider the confidence of each model for a specific class. When determining the final prediction, we take the calculated class probabilities from each model and compute the average probability for each class. Then, having a unified list of probabilities for the belongingness of the image to specific classes, we choose the class with the highest probability as the final prediction.

4 Experiments

4.1 Network Architectures

One of the most important choices to make when using neural networks is the choice of their architecture. More complex networks might have a higher potential in terms of accuracy, but they also come with a larger risk of overfitting.

In this research, we focused on convolutional neural networks. Among the tested networks, we have our own custom CNN architecture, and two variants of ResNet-18 - one pre-trained, and one initialised with random weights. The aim of this experiment was to see how each architecture performs, which would help us to pick the best ones for later experiments.

4.1.1 Setup

Our custom CNN (named CustomCNN) comprises two pairs of convolutional layers and two fully-connected layers. All convolutional layers use a 3x3 kernel and a ReLU activation function. In the first pair of convolutional layers, the first one has three channels on the input, corresponding to RGB channels on the input image, and has 24 channels on the output. The second layer has 24 channels on the input and on the output. Similarly, in the second pair, the first layer has 24 channels on the input and 48 on the output, while the second one has again the same number of channels on the input and on the output. Each pair is followed by a 2x2 max-pool layer.

The convolutional section of the network produces tensors of a size 48x5x5, which are flattened before going into fully-connected layers. The first of these has 256 neurons and uses a ReLU activation function, while the one that follows is an output layer, with 10 outputs and a softmax activation. The number of outputs can be customised when initialising the network.

Our architecture also supports dropout, the probability of which is 0 by default but can be configured when creating a network. If specified, neurons can be randomly deactivated right after flattening and in the 256-neuron fully-connected layer.

Other than our custom network, ResNet-18 was tested. We tested separately a randomly initialised ResNet-18, and a default pre-trained version of it provided by PyTorch, which, according to its documentation, had been pre-trained on the ImageNet-1k dataset (Paszke et al. 2019). We manually added a final layer to each of these networks to set their number of outputs to 10.

The loss function used throughout the training process was the cross-entropy loss. To ensure a fair comparison, all networks have been trained using the same hyperparameters, which are as follows:

- batch size: 128,
- optimiser: SGD,
- learning rate: 10^{-3},
- momentum: 0.9,
- number of epochs: 50.

There was however an exception in the case of a pre-trained ResNet-18, which had its pre-trained layers frozen for the first 10 epochs to tune the manually-added final layer. For these initial epochs, a smaller learning rate of 10^{-4} was used, however the rest of the parameters were the same as listed above.

4.1.2 Results

The results are visualised on Figure 3. Looking at how loss function values changed on training and validation data, it is clear that both ResNet-18 networks were strongly overfitted. This is also reflected by accuracy values - on the training data it approached 1 at the end, while on the validation data it was significantly lower - at around 0.65 for a pre-trained ResNet-18 and at around 0.53 for the variant with random weights.

Another aspect worth noting is the performance of our custom network. It was able to achieve similar accuracy on the validation data as randomly initialised ResNet-18, while at the same time not overfitting quite as much - while the training data accuracy was significantly higher than for validation data, validation loss values were steadily shrinking for the whole duration of training. This behaviour was probably thanks to its shallow architecture. It is a known fact that deep models, like ResNet-18, are more prone to overfitting, and this was apparent in our experiment.

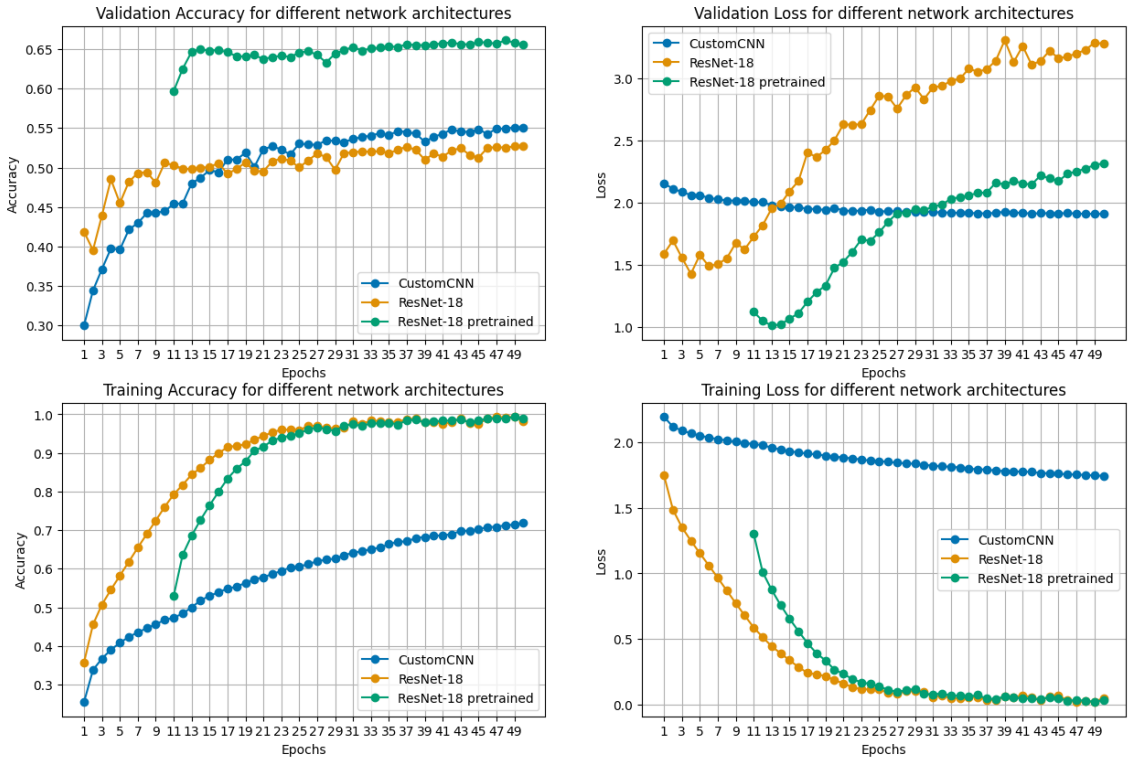

Figure 3: Statistics per epoch for different network architectures

4.2 Data Augmentations

Augmentations are a crucial aspect of training models based on image data. They aid in increasing the diversity of training data, enabling models to learn various features of images. In our approach, we divided our research into three main categories: simple augmentations, advanced augmentations, and combinations of these techniques.

4.2.1 Setup

The augmentation experiments were performed on a ResNet-18 model, which was not pretrained beforehand. We conducted experiments on datasets employing simple augmentations, each applied with a certain probability. Example augmentations used can be observed in Fig. 7. The configuration for simple augmentations was as follows:

- RandomResizedCrop with a size of (32, 32)
- RandomRotation within a range of 0 to 180 degrees.
- RandomHorizontalFlip with a probability of 0.2
- RandomInvert with a probability of 0.2
- RandomSolarize with a probability of 0.2 and a threshold of 100

In addition to these simple augmentations, we also explored more advanced techniques. In our advanced augmentation solution, we use both MixUp and CutMix techniques, randomly assigning one of these methods to each image.

After obtaining the initial augmentation results, we noticed a significant disparity between the performance of simple augmentations compared to advanced augmentations. We concluded that applying an excessive amount of augmentation could lead to considerable image distortion. Therefore, we also experimented with an alternative approach where we utilized only rotation and horizontal flipping. This resulted in the following augmentation set:

- RandomRotation within a range of 0 to 180 degrees,
- RandomHorizontalFlip with a probability of 0.2.

Figure 4: Original photo without any augmentation

Figure 5: Rotation + Resize and Crop effect

Figure 6: Rotation + Solarize effect

Figure 7: Original photo vs augmented photos.

4.2.2 Results

The results from the initial augmentation experiment are depicted in the charts labelled as Fig. 8 and Fig. 9. We can compare our results to those previously obtained on the ResNet-18 model using unaugmented images (ref. 3).Each type of augmentation improves predictors, aiding in alleviating the problem of data overfitting. ResNet-18 model achieved the best results using advanced augmentations.

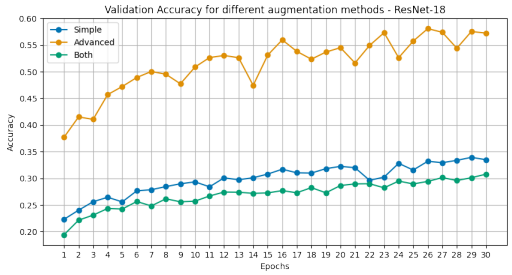

Figure 8: Validation accuracy per epoch for different augmentations

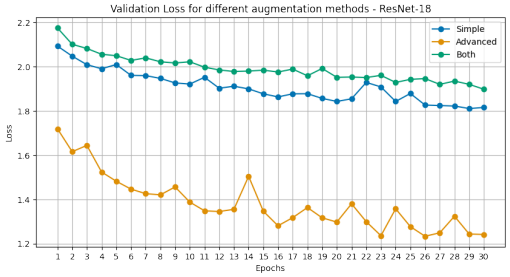

Figure 9: Validation loss per epoch for different augmentations

We noticed that when using simple augmentations, we achieved poorer results. The most detrimental impact on our data was observed with the random crop and resize operation, due to the already limited resolution of our low-dimensional training images. Following this operation, the resulting images were so degraded that even for a human observer, recognizing the original photo would be challenging (ref. 5). We believe that the images are small enough that this augmentation method is not advisable in this process. Therefore, we decided to train models without this type of augmentation and try use only random rotation and horizontal flip.

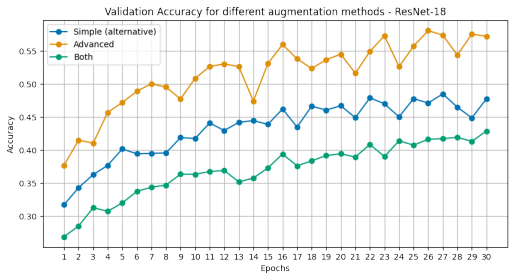

Figure 10: Validation accuracy per epoch for different augmentations using alternative approach

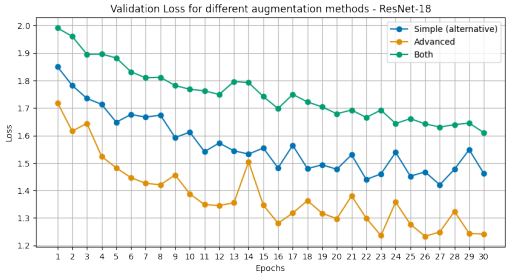

Figure 11: Validation loss per epoch for different augmentations using alternative approach

As seen in the charts 10 and 11, by employing an alternative approach with fewer simple

augmentations, we achieved better results for the simple augmentation technique and for the model incorporating both types of augmentations during training.

4.3 Hyperparameters Tuning

Choosing the right hyperparameters is a key part of training a model. There are multiple approaches that can be used for this task, such as random search or grid search. In our experiments, we chose a method called Bayesian optimisation, which produces better results than random search by, in each step, using prior knowledge about the parameters already tested in an attempt to find the most optimal ones.

4.3.1 Setup

For this experiment, we decided to optimise hyperparameters for our custom network and for ResNet-18 (not pre-trained). Due to a difference in parameters used by different network weights optimisers, we also conducted hyperparameter optimisation separately for SGD and Adam. The other optimised hyperparameters and their search spaces were as follows:

- batch size: $\{2^5, 2^6, \ldots, 2^9\}$,
- learning rate: $[10^{-4}, 10^{-2}]$,
- L2 regularisation coefficient: $[10^{-5}, 10^{-2}]$,
- dropout probability (CustomCNN only): $[0, 0.2]$,
- momentum (SGD only): $[0, 0.99]$
- beta1 (Adam only): $[0, 0.99]$
- beta2 (Adam only): $[0, 0.99]$

The Bayesian Optimisation algorithm was configured to sample 4 random hyperparameter combinations and then run 10 iterations of optimisation, which gives a total of 14 trainings per network-optimiser pair. That means there were 56 networks trained in total, each one trained for 15 episodes. During the training, we used advanced augmentations, because they proved to be the most effective method of data augmentation in the previous experiment.

4.3.2 Results

Final Bayesian optimisation results, along with their accuracy values on the validation data, are presented in Table 1. When it comes to our custom architecture, SGD proved to be underwhelming. In fact, most iterations (which are not shown in this report) ended with similar accuracy scores, proving that this weight optimisation method perhaps had little potential in the case of CustomCNN. Hence, we abandoned it going forward.

Table 1: The best results of Bayesian Optimisation for each run

Network Optimiser	CustomCNN SGD	CustomCNN Adam	ResNet-18 SGD	ResNet-18 Adam
batch size	32	64	32	64
learning rate	3.158×10^{-3}	10^{-4}	3.338×10^{-3}	10^{-4}
L2	3.855×10^{-3}	10^{-5}	1.746×10^{-3}	10^{-5}
dropout	0.05463	0.2	-	-
momentum	0.273	-	0.8547	-
beta1	-	0.99	-	0.99
beta2	-	0.99	-	0.99
Valid. Acc.	**0.3484**	**0.4052**	**0.461**	**0.4892**

Other results gave us a solid basis for training further models. We trained CustomCNN with Adam, ResNet-18 with SGD, ResNet-18 with Adam and ResNet-18 pre-trained with Adam, using parameters found through Bayes optimisation (Table 1)[1]. This time, to tune the models to the best of their abilities, we run the training process for 150 epochs. The results of their training are visualised on Figures 12 and 13.

[1]ResNet-18 pre-trained used the same parameters as randomly initialised ResNet-18

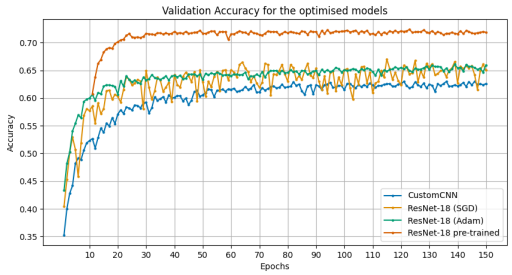

Figure 12: Validation accuracy per epoch for the final models

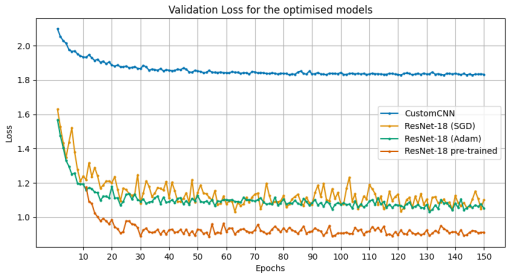

Figure 13: Validation loss per epoch for the final models

Overall, all models were able to eventually achieve satisfactory results with final validation accuracy values exceeding 0.6 in every case (CustomCNN: 0.6260, ResNet-18 with SGD: 0.6501, ResNet-18 with Adam: 0.6596, ResNet-18 pre-trained: 0.7187). Thanks to augmentations used, and possibly also some of the parameters, no models shown any signs of overfitting, with validation loss values steadily shrinking, unlike in the first experiment where after a few epochs, validation loss would start rapidly growing.

Unsurprisingly, the best scores were achieved by a pre-trained ResNet-18. However, since it is a pre-trained model, we were not allowed to use it to make final predictions, so we did not analyse it further. Nonetheless, these experiments proved that even though the model had been pre-trained on a large dataset, tuning it further did greatly improve its performance on our data.

The other models were able to achieve similar results to one another. Interestingly, while our custom CNN did have much higher validation loss values than ResNets, its final accuracy was not much worse than these of the other models. This proves that a well-designed shallow architecture was indeed good enough for this task. Another aspect to point out is that Adam guaranteed a much more stable training process than SGD, which can be clearly observed when comparing how both metrics changed for ResNet-18 SGD and other models.

4.4 Ensembling

Ensembling techniques allow for improving performance by combining multiple models to achieve a single optimal solution. In our solution, we implemented two types of ensembling: hard voting and soft voting.

4.4.1 Setup

In our ensembling technique, we employed 3 models: CustomCNN, ResNet-18 trained with SGD, and ResNet-18 trained with Adam (both ResNets had random starting weights). Using these models, we implemented predictors using two ensembling technics to get opimalized prediction.

In soft voting, predictions from multiple models are combined by averaging their individual predictions for each class. This approach considers the collective opinion of the models, giving each model's prediction an equal weight in determining the final outcome

In hard voting, we first calculate the argmax for each model's predictions and then count the votes to determine the winning label. What was important for us during the project development was implementing a function that resolves tie situations between models. We decided that in the event of a tie, the deciding vote would go to the model with the highest confidence (the highest probability that the image belongs to the specified class).

4.4.2 Results

The results we achieved in the ensemble process are visible in the chart shown in Fig. 14 and in the final table 2. In the chart, we presented the results with a breakdown of accuracy for each class to observe how models perform for each class and identify classes for which models perform poorly. Such a comparison is useful when we aim for specific accuracy of models in particular classes.

Thanks to such a comparison, we can see whether the use of ensemble technics performs better and in which classes it helps. In the chart, we observe that the utilization of soft ensembling or hard ensembling improved the accuracy across nearly all classes. Using soft ensembling as a predictor gives the best results

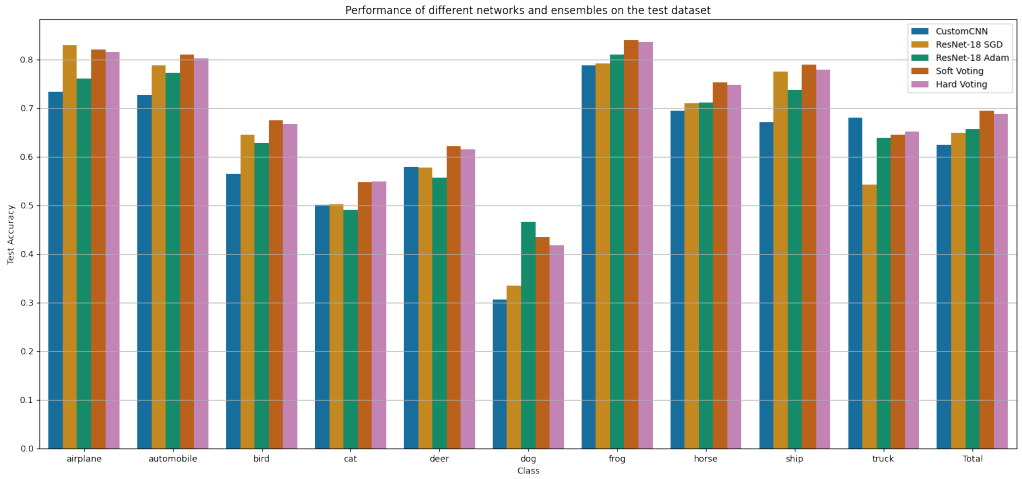

Figure 14: Test accuracy on final models per class and total

It was interesting to observe how ensembling works when predicting the "dog" class. We see that two models, ResNet with the SGD optimizer and our Custom Network, perform poorly, achieving an accuracy of around 35% for this class. We see that the best performance in predicting this class is achieved by ResNet-18 with the Adam optimizer, reaching results around 45%. We notice that despite including two 'worse' models, ensembling technic does not achieve significantly worse results than the best model.

Table 2: Test Accuracy of Different Predictors

Place	Predictor	Test Accuracy
1	Soft Voting	0.694
2	Hard Voting	0.688
3	ResNet-18 Adam	0.657
4	ResNet-18 SGD	0.649
5	CustomCNN	0.624

As evident from Table 2, all the final results achieved satisfactory performance, including our custom model which attained a total accuracy of 62.4%. Interestingly, our custom model achieved the highest accuracy for the 'truck' class among all predictors. The best results were achieved by predictors utilizing ensemble techniques. Soft-voting, with an accuracy of 69.4%, was selected as the optimal predictor for class prediction on the CIDER dataset.

5 Conclusions

This project helped us gain knowledge and practical skills in dealing with convolutional neural networks. During this project, we were able to design our own CNN architecture and compare it to ResNet-18. To our surprise, the results of our custom-designed architecture were not far off of those achieved by ResNets.

We also tested various image augmentation techniques, including some advanced ones like CutMix and MixUp. We discovered that even though augmentations can significantly prevent models from overfitting, adding too many of them can lessen data quality, thus hindering the training process.

The next step was to optimise network parameters. This was the most time-consuming part of the project, which unfortunately meant we had to put some limitations in place to allow us to conduct this experiment within a reasonable timeframe. Ideally, we would have wanted to begin each optimisation with more random samples, and run it for significantly more iterations. This would have allowed us to obtain even better results, and perhaps even surpass the 0.7 test accuracy threshold we got so close to in the end. Still, the results we obtained in this experiment were satisfactory, and they proved that hyperparameter optimisation is an important aspect of machine learning which cannot be neglected.

Our final predictor, which was a soft-voting ensemble of the three best networks, was able to achieve the accuracy of 0.694 on the testing dataset. While there was room for improvement, this

result exceeded our expectations, and we consider it satisfactory.

References

CINIC10 (2019).
URL: *https://www.kaggle.com/datasets/mengcius/cinic10?fbclid=IwAR3rPMLK4IJZLcVMCwFvcIvSOGjNcoDSk*

He, K., Zhang, X., Ren, S. & Sun, J. (2015), 'Deep residual learning for image recognition'.

Nouman (2021), 'Writing CNNs from Scratch in PyTorch'.
URL: *https://blog.paperspace.com/writing-cnns-from-scratch-in-pytorch/*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. & Chintala, S. (2019), PyTorch: An Imperative Style, High-Performance Deep Learning Library, *in* H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox & R. Garnett, eds, 'Advances in Neural Information Processing Systems 32', Curran Associates, Inc., pp. 8024–8035.
URL: *http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf*

Rosebrock, A. (2023), 'PyTorch: Training your first Convolutional Neural Network (CNN) - PyImageSearch'.
URL: *https://pyimagesearch.com/2021/07/19/pytorch-training-your-first-convolutional-neural-network-cnn/*

Samar, A. (2023), 'Fine-Tuning a Pre-Trained ResNet-18 Model for Image Classification with PyTorch'.
URL: *https://alirezasamar.com/blog/2023/03/fine-tuning-pre-trained-resnet-18-model-image-classification-pytorch/*

Sangdoo Yun, Dongyoon Han, S. J. O. S. C. J. C. Y. Y. (2019), 'Cutmix: Regularization strategy to train strong classifiers with localizable features'.

Tam, A. (2023), 'Using dropout regularization in PyTorch models'.
URL: *https://machinelearningmastery.com/using-dropout-regularization-in-pytorch-models/*

Tong Yu, H. Z. (2020), 'Hyper-parameter optimization: A review of algorithms and applications'.