Introduction to Machine Learning - Milestone 2

Emma Toutel, Alicia Bächler and Paul Giard

June 1, 2025

1 Introduction

In this project, we explore the application of deep learning models for classifying images. We are working with one of the database in the MedMNIST collection named DermaMNIST. It focuses on medical images of skin patches with a possible condition. They are divided into 7 different classes, which correspond to different types of skin diseases or lesions. Using PyTorch, the goal was to implement, train, and evaluate two types of deep learning models: Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN). Additionally, we want to compare their performance based on accuracy and macro F1-score.

2 Method

The DermaMNIST consists of 9'012 images divided into a training set of 7'007 images and a test set of 2'005 images. In addition to that, 15% of the training set is randomly selected, in order to avoid bias, and is used to form the validation set. Each image is a 28x28 pixel greyscale image, with 3 colour channels. The labels of the classes have been cast into integers in $\{0,1,\ldots,6\}$.

The system implemented is composed of three major components: two architectures model, Multilayer Perceptron and Convolutional Neural Network, and a training framework in the Trainer class. The models are structured to deal with different types of input. The training is designed to accommodate both scenarios allowing flexibility to experiment.

Furthermore, we normalised the data which led to better convergence and smoother training curves.

2.1 MLP

The first model, MLP, represents a typical fully connected neural network architecture. This is well suited for vectorised input like flattened images. The MLP is fabricated from a stack of linear fully connected layers with hidden layer that can be configured via the `hidden_layers` parameter, such as the default is [256,128]. Each layer is followed by a batch normalisation, ReLU activation and dropout regularisation. Normalising the batches ensures the distribution of the layer's input remains stable, for better convergence and improved speed. The ReLU's non-linearity enables the network to model complex non-linear functions. The dropout condition helps prevent over-fitting of the data during training by randomly zeroing some elements of the input tensor with a probability of p which we define. Finally, the output is a linear layer producing logits (raw prediction scores) for the 7 classes.

2.2 CNN

The second model implemented is a Convolutional Neural Network, CNN, designed to handle image data by preserving its 2D spatial structure —unlike MLPs, which flatten inputs. It detects local patterns with increasing precision as the network deepens. We ensured the input shape was (N, 3, 28, 28), where N is the batch size, 3 the RGB channels, and 28×28 the image dimensions. The model begins with two convolutional layers. The first layer takes the 3-channel input and produces 6 feature maps using a 5×5 kernel, which is a small sliding window that moves across the image to detect patterns, like edges or textures. We also applied a padding of 2 to maintain the original spatial dimension. The second convolutional layer uses the same kernel size and padding but increases the number of channels from 6 to 16, allowing the network to capture more complex features. After each convolutional layer we applied a ReLU activation function to introduce non-linearity so that our model could learn more complex patterns. Finally we did pooling in order to select the learned features the model needed to keep. We used max pooling after the first convolution, which selects the maximum value from each patch, and adaptive max pooling after the second convolution to ensure the output has a fixed spatial size of 7×7, regardless of the input size, so that it can pass through the four fully connected layers following. These layers enable the model to learn global patterns and ultimately output class scores used for classification.

A second type of CNN, the ResNet18 model was also built as a way to compare our implementation to a 'tried and true' one. ResNet implements many more layers and processes than our CNN, with around 100 times more trainable parameters than our CNN. This has two effects on computation. Resnet takes much longer than our CNN model to train (over 15 minutes on a GPU), however it converges to a good fit much faster (10 times less epochs) than CNN. As ResNet18 is not our own model, its results will not be presented here.

2.3 Trainer

The Trainer class gives a general training guideline that works for both the MLP and the CNN models. It contains all tasks related to training, validation and predictions. It does not include details such as device management and data convergence. During the initialisation, Trainer is defined with its different objects like the model, learning rate, batch size, number of epochs, cross entropy loss and the choice of optimiser. Multiple optimisation algorithms such as AdamW, Adam and SGD are established. A learning rate scheduler is also included, which halves the learning rate every 10 epochs. This is meant to encourage convergence and prevent overshooting later on.

Training is achieved in two ways: `train_all` and `train_one_epoch`. The `train_all` function runs the training loop over a defined number of epochs, and for each epoch, the `train_one_epoch` implements iteration over batches of data. For each batch, the chosen model processes the input data and computes the loss with cross-entropy. Then it runs a back propagation to calculate gradients, it updates the weights using the chosen optimiser where the default is AdamW and finally it resets gradients before the next batch is processed. During the training process, the loss can be monitored with printed results along the way. The model is set to training mode and then set to evaluation mode when using the validation or testing set, this guarantees coherence. The `predict_torch` method uses `torch.no_grad()` which can locally disable the gradient while computing. In the end, it returns predicted labels. Additionally, Trainer includes two functions to bridge between NumPy arrays and PyTorch tensors. The `fit()` function is doing it with the training data and also trains the model and returns predictions, while `predict()` uses the test or validation datasets.

The device (CPU or GPU) is automatically inferred from the model's parameters, and all data is moved accordingly to maintain compatibility.

2.4 Hyper-parameter selection

One of the main challenges when working with PyTorch and model optimization is selecting the hyper-parameters to find the best model. In order to do so, we created a dictionary of all possible model combinations and ran a few hundred tests to find the best ones, using a set seed to ensure reproducibility. Computing was done on Google Colab in order to speed up times and have access to a GPU. The results of these hyper-parameter tests and the code used to implement them are available in the `Bonus` directory of the submitted project.
As a result, the default parameters for the MLP are already set up in the project, to run the default parameters for the CNN, simply run the code with a `-default_cnn` flag.

3 Results

3.1 MLP

The highest validation F1-score achieved was 0.498 in experiment 114. Applying those hyper-parameters to the test set we obtain the results displayed in Table 1. The best hyper-parameters were with hidden layers [256, 128], learning rate 0.01, optimiser AdamW, batch size 64 and weight decay 1e-5.

	Accuracy %	**F1-Score**
Training set	97.41	0.9630
Test set	74.47	0.4912

Table 1: Average Performance for MLP model with the Best Hyper-parameters

3.2 CNN

The highest validation set F1-score of 0.535676 in experiment 132 with the hyper-parameters: kernel 5, dropout 0.5, learning rate 0.001, optimiser AdamW, batch size 64 and padding 2.

	Accuracy %	**F1-Score**
Training set	88.83	0.7471
Test set	73.40	0.4815

Table 2: Average Performance for CNN model with the Best Hyper-parameters

4 Discussion

It is important to notice before starting the discussion of our results that our original dataset is unbalanced. The Class 5 (Melanocytic nevus) for example has 6,034 images, which is over 58 percent of the dataset while Class 3 (Dermatofibroma) has only 103 images, which represent less than 1.2 percent of our dataset. We have to take it into account when looking at our indicators. Indeed, accuracy can be misleading in class-imbalanced datasets, the F1 score is more relevant so that our model be good to recognize all classes, especially rare ones.

4.1 MLP

From the results and our subsequent observations, the model clearly struggles with over-fitting the data. This very likely comes from the limited size of our dataset. Having too many hyper-parameters can also lead the model to fit the training set too well but does not reflect general trends. Additionally, fine tuning the hyper-parameters also complicated our task because of the massive amount of different combinations possible, rendering it a very waste and computationally expensive task.

We were able to notice that both too high (0.1) or too low (0.0001) learning rates badly capture the data. The high learning rates would be too extreme and destabilise the training whereas the ones that are too small do not converge fast enough or might converge on a local minima.

The use of smaller batch sizes (like 32 or 16) typically improved the generalisation allowing the model to learn more broadly. However, it does increase the amount of computation needed, increasing the training time. This would probably be more noticeable on larger datasets.

To further improve the model, synthetic data augmentation could help combat the lack of data and its diversity. Adding an L_2 regularisation could reduce over-fitting and an ensemble method which combines outputs from multiple MLP configurations might also lead to more robust results.

4.2 CNN

We can first mention that CNN did not outperform MLP when used in the test set (CNN F1-score 0.48 $<$ 0.49 for MLP, and the accuracy was also lower). It is surprising because we could have expected a better generalization on the validation set for CNN looking at the very high accuracy scores for MLP that are signs of overfitting.

The choice of the hyper-parameters helped a lot to stabilize the the training: The AdamW optimizer really improved the training and validation results (compared to Adam); it helps the model to update its weights during the training and can reduce the problem of over-fitting. Choosing the right learning rate was also determinant (0,001); smaller ones led to slow convergence while larger rates lead to a training less stabilised. The choice of adaptive pooling was relevant, it is more robust and flexible than fixed kernel-size and it eliminated problems due to dimensionality. Finally the impact of batch size was less important; Smaller batches (32) are expected to help with generalization but here it didn't lead to outperforming the larger one (64).

The CNN made us face lots of challenges: it had a long computation time due to its number of convolution operations and number of batches. Its architecture was very sensitive to varying hyper-parameters: changing kernel size, padding, or dropout often caused large differences in validation performance, making CNNs harder to improve reliably without extensive search.

To further increase CNN performance, strategies such as data augmentation, deeper architectures could be explored.

4.3 Trainer

The problems faced were mainly linked to data conversion between NumPy arrays and PyTorch as well as the rigidity caused by the fixed step decay scheduler which might not be practical for all models.

The addition of early stopping based on the loss could help avoid calculations of unnecessary epochs when the model has stopped improving. Or modifying the `StepLR` to a more advanced one might lead to better generalisation. Tensor board logging could make analysis more efficient by allowing the tracking and visualisation of the loss and accuracy.

5 Conclusion

The size and diversity of the data set is finally a limiting factor in the performance of our methods : We have a rather small number of data and it is unbalanced making the training process without over-fitting difficult. These models could not currently be used by medical professionals to help predict skin conditions. However, it might be possible in the near future if the dataset used is much larger than the one we are currently using and more balanced.