Blood Cell Classification Using Deep Learning: MLNN and CNN Architectures

Miguel Cabral Pinto & Tiago Silva

University of Coimbra

Abstract. Blood cell classification is a critical task in medical diagnosis and hematological analysis. The present work conducts a comparative study between two deep learning architectures for blood cell classification supported by the BloodMNIST dataset. We implement and evaluate two neural network architectures: a Convolutional Neural Network (CNN) and a Multi-Layer Neural Network (MLNN). Besides implementation details we created an evaluation framework composed by three dimensions: quality (accuracy, precision, recall, F1-score), efficacy (sensitivity, specificity), and efficiency (training time, inference time). Experimental results support the theoretical superiority of CNN architectures for image-based classification tasks, showing significantly better performance than traditional fully-connected networks although the imbalanced nature of the dataset poses challenges for both architectures.

Keywords: Blood Cell Classification · Deep Learning · Multi-Layer Neural Networks · Convolutional Neural Networks · Medical Imaging · BloodMNIST

1 Methodology

1.1 Dataset

We utilize the BloodMNIST dataset from the MedMNIST collection. The dataset characteristics are:

- **Image Size**: 28×28 pixels, RGB (3 channels)
- **Classes**: 8 blood cell types
- **Training Set**: 11,959 images
- **Validation Set**: 1,712 images
- **Test Set**: 3,421 images
- **Total**: 17,092 images

Because the dataset is already preprocessed and normalized, we focus on the model architectures and training procedures rather than data augmentation techniques. As expected in medical datasets, class imbalance exists, with some cell types being underrepresented. This is addressed in the evaluation phase using metrics sensitive to class distribution to ensure a more robust assessment of model performance.

1.2 Multi-Layer Neural Network Model

Architecture The MLNN architecture uses only fully-connected layers:

Algorithm 1 MLNN Architecture

1: **Input:** $x \in \mathbb{R}^{B\times3\times28\times28}$
2: **Flatten:** $x \in \mathbb{R}^{B\times2352}$ (where $2352 = 3 \times 28 \times 28$)
3: $\mathbf{FC_1}$**:** Linear($2352 \to h_1$) + ReLU
4: $\mathbf{FC_2}$**-**$\mathbf{FC_n}$**:** Linear($h_{i-1} \to h_i$) + ReLU (for n hidden layers)
5: **Output Layer:** Linear($h_n \to 8$)
6: **Output:** Logits $\in \mathbb{R}^{B\times8}$

The size of the hidden layers in the network is calculated using the average of the input and output dimensions:

$$h = \frac{n_{input} + n_{classes}}{2} = \frac{2352 + 8}{2} = 1180 \tag{1}$$

This heuristic provides reasonable capacity without excessive parameters. For a single hidden layer configuration, it results in 1,180 hidden units, creating a bottleneck architecture ($2352 \to 1180 \to 8$) that forces the network to learn compressed representations. While this approach can work for simple classification tasks, it fundamentally treats each pixel as an independent feature, ignoring the 2D spatial relationships that characterize cell morphology.
The MLNN architecture serves primarily as a pedagogical baseline to demonstrate why CNNs are preferred for image classification. The lack of translation invariance, the enormous first-layer parameter count (2.77M parameters just in the first layer), and the immediate loss of spatial structure highlight the architectural disadvantages of this approach when compared to convolutional ones. As such, this work validates theoretical understanding through empirical evidence.

Training Configuration Table 1 presents the complete hyperparameter configuration used in our experiments, chosen to both experiment with different training dynamics and provide a solid baseline for comparison with the CNN architecture.

Table 1: Experimental Hyperparameters

Parameter	**MLNN-Test1**	**MLNN-Test2**	**MLNN-Test3**
Learning Rate	0.001	0.01	0.001
Optimizer	Adam	SGD w/ Momentum (0.9)	Adam
Loss Function	CrossEntropyLoss	CrossEntropyLoss	MeanSquaredError
Epochs	15	15	15
Hidden Layers	1	1	1
Hidden Size	1180	1180	1180
Activation Function	ReLU	ReLU	ReLU
Initialization	He	He	He

Here are the main reasons behind our choices:

- CrossEntropyLoss was chosen as the standard loss function for multi-class classification, combining LogSoftmax and NLLLoss for numerical stability. This is particularly important given the class imbalanced nature of the dataset, CrossEntropyLoss poses a powerful approach to handle such scenarios by adjusting class influence through customized weighting if necessary.
- The state of the art Adam optimizer handles the varying gradient magnitudes across different network depths was selected due to its innovative adaptive learning rate mechanism, combining the benefits of RMSProp and momentum-based SGD. To compare with the forementioned momentum-based SGD optimizer, we also implemented the same MLNN using this optimizer (with momentum set to 0.9) and a higher learning rate (0.01 because in contrast to Adam, SGD lacks adaptive learning rates, therefore, a higher initial learning rate is necessary to achieve comparable performance).
- The 0.001 learning rate represents the standard default for Adam in computer vision tasks. Preliminary experiments validated this choice for the Adam optimizer test: higher rates (0.01) caused training instability, while lower rates (0.0001) converged too slowly. The selected rate achieves a smooth loss decrease ($1.58 \rightarrow 0.313$) without oscillations, with test performance plateauing around epoch 12, indicating appropriate convergence behavior.
- The 15-epoch training duration was determined empirically.
- He initialization was used throughout, as it is specifically designed for ReLU activations and prevents vanishing/exploding gradients by maintaining variance across layers.

1.3 Convolutional Neural Network Model

Our CNN architecture consists of three convolutional blocks followed by fully-connected layers. The design follows the principle of progressively increasing feature channels while reducing spatial dimensions:

Algorithm 2 CNN Architecture

Input: $x \in \mathbb{R}^{B\times 3\times 28\times 28}$
Conv$_1$: Conv2d($3 \rightarrow 32$, kernel=3, padding=1) + ReLU + MaxPool(2)
  Output: $\mathbb{R}^{B\times 32\times 14\times 14}$
Conv$_2$: Conv2d($32 \rightarrow 64$, kernel=3, padding=1) + ReLU + MaxPool(2)
  Output: $\mathbb{R}^{B\times 64\times 7\times 7}$
Conv$_3$: Conv2d($64 \rightarrow 128$, kernel=3, padding=1) + ReLU + MaxPool(2)
  Output: $\mathbb{R}^{B\times 128\times 3\times 3}$
Flatten: $\mathbb{R}^{B\times 1152}$
FC$_1$: Linear($1152 \rightarrow 256$) + ReLU
FC$_2$: Linear($256 \rightarrow 128$) + ReLU
FC$_3$: Linear($128 \rightarrow 8$)
Output: Logits $\in \mathbb{R}^{B\times 8}$

The architecture has 422,344 total parameters, and its rationale is as follows:

- **Progressive channel expansion** (32→64→128): Captures increasingly complex features
- **Spatial dimension reduction** (28→14→7→3): Achieved through max-pooling with stride 2
- **Unitary padding**: Preserves spatial dimensions within each convolutional layer
- **ReLU activation**: Introduces non-linearity and mitigates vanishing gradients
- **Multi-stage FC layers**: Provides smooth transition from feature maps to class predictions

The number of convolutional layers resulted from balancing model complexity and computational efficiency. Given the input, we recognized that three convolutional layers would provide enough feature extraction while not over-reducing the spatial dimensions. The channel progression and kernel sizes were selected based on established (state-of-the-art) CNN design principles proven and discussed in theoretical classes. Additionally, the fully-connected layers were sized to gradually reduce the feature representation to the final class logits (preventing abrupt bottlenecks). Though the design could certainly be further optimized through hyperparameter tuning, this architecture serves as a solid baseline (experimentally validated in the results section).

Training Configuration Table 2 presents the complete hyperparameter configuration used in our experiments with the CNN architecture:

Table 2: Experimental Hyperparameters

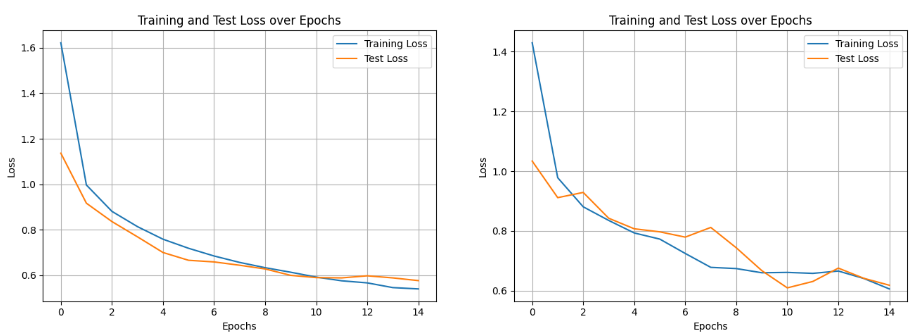
Parameter	**CNN-Test1**	**CNN-Test2**	**CNN-Test3**
Learning Rate	0.001	0.01	0.001
Optimizer	Adam	SGD w/ Momentum (0.9)	Adam
Loss Function	CrossEntropyLoss	CrossEntropyLoss	MeanSquaredError
Epochs	15	15	15
Activation Function	ReLU	ReLU	ReLU
Initialization	He	He	He

Many of the optimal hyperparameters found in the preliminary CNN experiments align with those identified for MLNNs. In order to make our comparison of the two the most relevant possible.

2 Results

2.1 MLNN

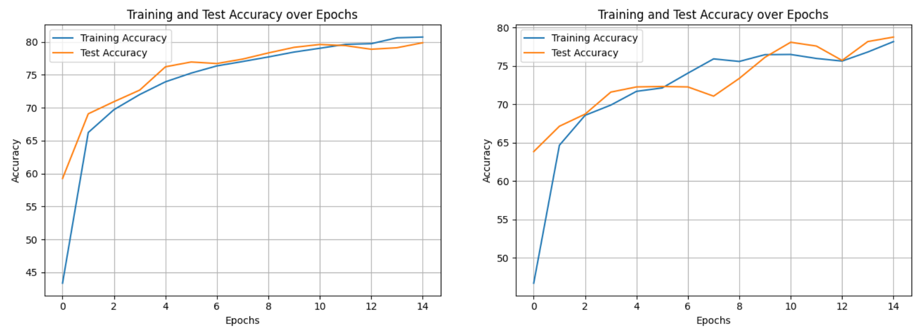
For this section, the focus was put on comparing the performance of the first two MLNN configurations (differing in optimizer and learning rate as detailed in Table 1). However, we also conducted the tests for the third configuration, to evaluate the impact of a different loss function (Mean Squared Error, or MSELoss) on training dynamics and performance. As expected, the results were inferior (average loss of accuracy around 5-10%) to those obtained with CrossEntropyLoss, confirming the latter's superiority over the former in this context, particularly due to solving the vanishing gradient problem. Given the predictability of this outcome, we decided to focus exclusively on CrossEntropyLoss in our experiments given the length constraints and to maintain clarity in our analysis. Let's analyze the results and make some observations.
Looking at the loss curves in Figure 1, we can see that both configurations show a decreasing trend, indicating that the models are learning. However, the Adam optimizer (Figure 1a) demonstrates a more stable and consistent decrease in loss compared to the SGD with Momentum (Figure 1b), which exhibits a more erratic pattern, oscillating significantly during training. This suggests that Adam is better suited because not only does it adapt the learning rate for each parameter, but it also combines the benefits of both Momentum and RMSProp, leading to more efficient convergence. This is a relatively simple task, so both optimizers manage to reduce the loss, but with an increase in complexity we might have seen more pronounced differences.



(a) (Adam, lr=0.001) (b) (SGD+Momentum, lr=0.01)

Fig. 1: Loss curves comparison between MLNN configurations

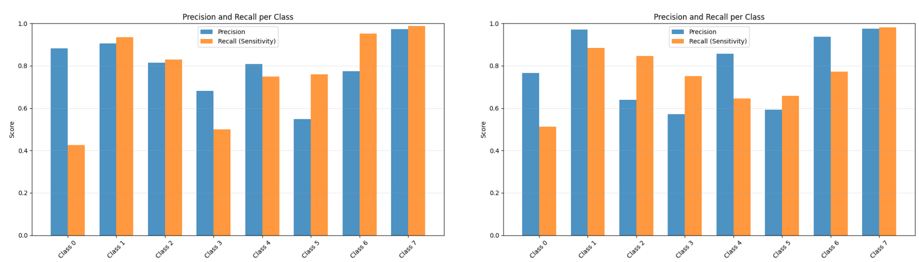
Similarly, the accuracy curves in Figure 2 show that the Adam optimizer achieves higher accuracy more quickly than SGD with Momentum. The Adam configuration reaches approximately 80% accuracy by epoch 10, while the SGD configuration lags behind, only reaching around 75% accuracy by the same epoch. This further supports the conclusion that Adam's adaptive learning rates and momentum mechanisms provide a significant advantage in training efficiency.



(a) (Adam, lr=0.001)   (b) (SGD+Momentum, lr=0.01)

Fig. 2: Accuracy curves comparison between MLNN configurations

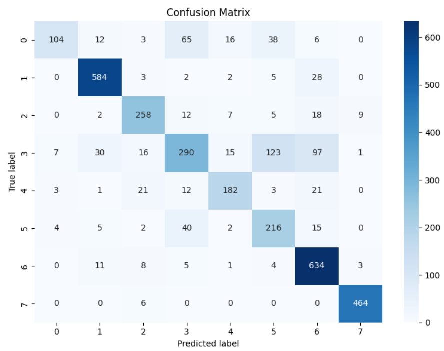
The bar charts in Figure 3 illustrate the precision and recall for each class in both MLNN configurations. We observe that the Adam optimizer (Figure 3a) generally outperforms the SGD with Momentum (Figure 3b) across most classes. Notably, classes with fewer samples (such as class 0) exhibit lower precision and recall in both configurations, highlighting the challenges posed by class imbalance. Despite in most classes the recall and precision values are relatively high, indicating that the models thresholds are effective at identifying the majority of instances correctly, there is still room for improvement, especially for underrepresented classes.



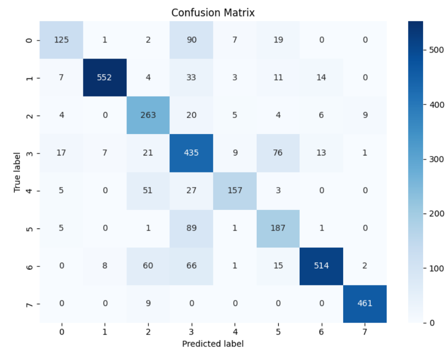
(a) MLNN-Test1   (b) MLNN-Test2

Fig. 3: Precision and Recall comparison across classes

The confusion matrix in Figure 4b shows that the momentum-based SGD optimizer is a bit less stable in comparison to Adam because, for example, it tends to classify more samples as class 3. This could be due to the optimizer's inability to adapt learning rates for individual parameters, leading to suboptimal updates. In contrast, the Adam optimizer (Figure 4a) demonstrates a more balanced classification across classes, with fewer misclassifications. This further emphasizes Adam's effectiveness across several metrics. The F1 score is a little under 0.80 for both configurations, with Adam slightly outperforming SGD with Momentum. This indicates that while both models are effective, and for simple tasks like this one the differences are not very pronounced, Adam provides a marginal advantage in balancing precision and recall (the thresholds are better optimized).



(a) MLNN-Test1

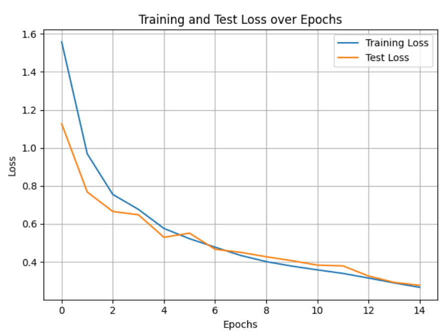


(b) MLNN-Test2

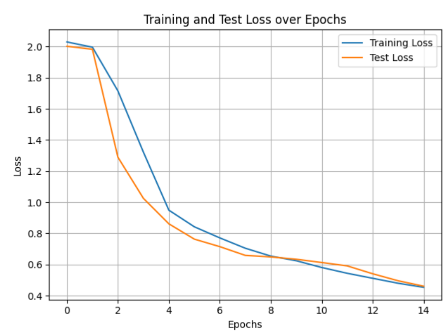
Fig. 4: Confusion matrices for both MLNN configurations

2.2 CNN

Many of the conclusions from the last section are applicable to the CNN test results as well, with some differences.
The Adam optimizer test achieves the lowest loss values across all epochs again, but in a more pronounced way than in the MLNN (seen in Figure 5a). Adam works especially well for CNNs because these networks are deep, have many different parameters, and their gradients can change a lot. For MLNNs, which are simpler and have more consistent gradients, Adam's benefits aren't as noticeable. Another noticeable change is how the second test's erratic nature in the first batch is not replicated here, which may be due to the fact that this architecture is much more robust than the previous, performing more complex operations with more layers.



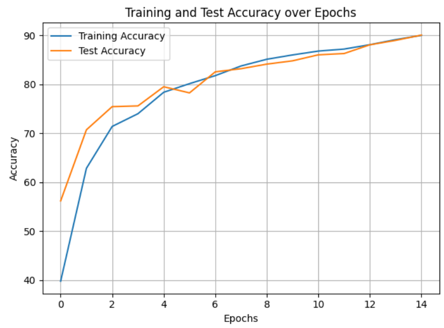
(a) (Adam, lr=0.001)



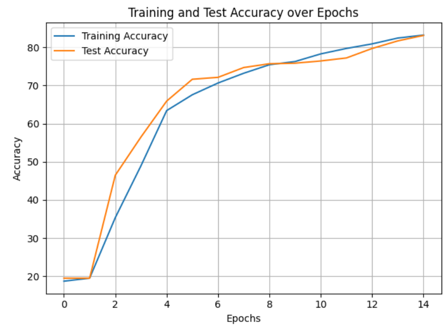
(b) (SGD+Momentum, lr=0.01)

Fig. 5: Loss curves comparison between CNN configurations

The verified trend of a steeper performance difference between the two continues in Figure 6, as the Adam test reaches a Test accuracy of about 90% to Momentum's 82%.



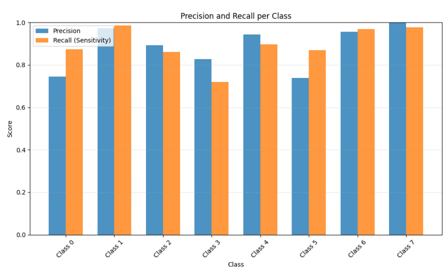
(a) (Adam, lr=0.001)



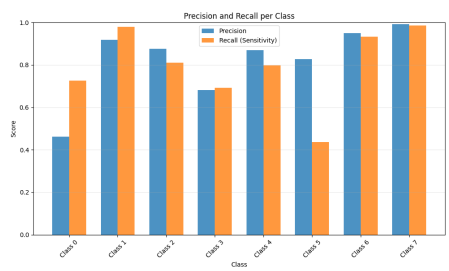
(b) (SGD+Momentum, lr=0.01)

Fig. 6: Accuracy curves comparison between CNN configurations

This phenomenon can also be seen by how visually clear it is in Figure 7 that the Adam test outperformed its counterpart in terms of the Precision and Recall metrics, evidenced by how high every bar is in Figure 7a.



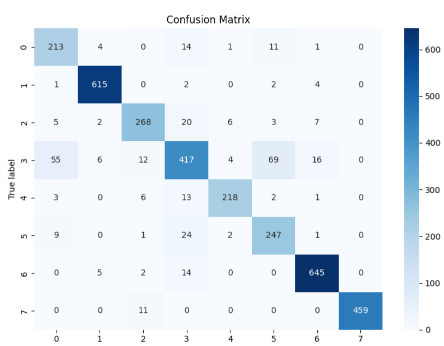
(a) CNN-Test1



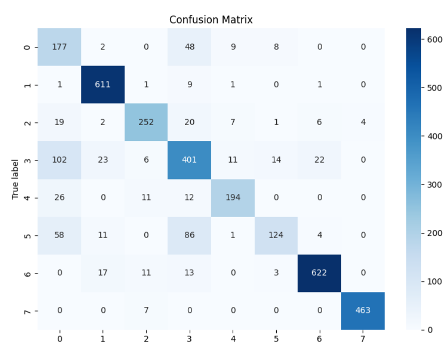
(b) CNN-Test2

Fig. 7: Precision and Recall comparison across classes

Finally, Figure 8 shows how this translates into the model's classification performance, bolstering a F1-score of .



(a) CNN-Test1



(b) CNN-Test2

Fig. 8: Confusion matrices for both CNN configurations

Similarly to the previous section, a test was also ran using the Adam optimizer with a MSELoss loss function, with even worse results. The accuracy after 15 epochs was of about 55%, corresponding to about a 35% drop in performance compared to Test 1 and about 15% compared to the same configuration for the MLNN. These massive differences may be due to the higher complexity of the CNN's architecture, which causes the choice of the loss function to be more impactful in its performance. When choosing MSELoss, a function suitable for regression, not classification, the model can struggle to learn the appropriate decision boundaries for the classification task. Since the MLNN model is more shallow, it is less sensitive to the choice of loss function, resulting in more stable performance in this specific scenario.

2.3 Comparative Analysis

One of the most striking conclusions of these tests is that the CNN consistently outperforms the MLNN in all quality metrics, with the most important result being the fact that the CNN achieves higher test accuracies compared to the MLNN by about 10%. Precision and recall values for each class are also higher for the CNN, particularly for underrepresented classes like 0 and 4, indicating better generalization even when faced with class imbalance. The F1-score, which balances precision and recall, further confirms the CNN's superiority, with a noticeable margin over the MLNN (0.85 vs under 0.80).

Another important point to be made is about what recall and specificity tells us about these models' adequacy for medical image analysis. These metrics are very important, since they represent a classic trade-off between identifying true positives and avoiding false positives, which can be a literal life-or-death matter in this context. The CNN demonstrates higher sensitivity across most classes, meaning it is more effective at correctly identifying true positives (correctly classifying blood cell types), which is the most important aspect of the trade-off in clinical scenarios. Specificity, or the ability to correctly identify negatives, is also improved in the CNN, as evidenced by the more balanced confusion matrices and fewer misclassifications compared to the MLNN. This strongly points towards the CNN being a much more reliable model in such a critical domain, improving upon the MLNN's performance in both metrics.

It is also important to note that, due to having a simpler architecture, the MLNN trains significantly faster per epoch than the CNN (about 4-5$\times$ faster). However, for how much more complex it is in terms of functionality, the CNN has significant parameter optimizations, like using simple 3$\times$3 filters in the convolutional layers instead of fully connected layers, which causes an exponential drop in the amount of parameters that need to be trained. These improvements really shine through during the training process, as even though the CNN takes longer to train, its capabilities ultimately cause it to require fewer epochs to reach a higher level of accuracy. In this case, the process speed for both models is comparable in absolute terms anyway due to the small input size, but the CNN's remains a powerful tool for larger inputs if computational power and time are not the bottleneck.

Overall, this report allowed us to conclude that the CNN architecture demonstrates clear advantages over the MLNN for blood cell classification on the BloodMNIST dataset. Its ability to use spatial information and hierarchical feature extraction leads to better performance across all evaluation metrics, validating the theoretical and practical preference for convolutional architectures in tasks of this sort. The MLNN, while useful as a baseline, is limited by its inability to capture spatial dependencies, resulting in an all-around lower performance.

3 Conclusion

This study compared the performance of Multi-Layer Neural Networks (MLNN) and Convolutional Neural Networks (CNN) for blood cell classification using the BloodMNIST dataset. As expected, the CNN architecture outperformed the MLNN across all evaluation metrics, including accuracy, precision, recall, and F1-score. We were able not only to validate the aforementioned theoretical thesis but also do so in a practical, real world scenario with all the complexities that it entails (e.g. class imbalance)

References

1. Esteva, A., Robicquet, A., Ramsundar, B., Kuleshov, V., DePristo, M., Chou, K., Cui, C., Corrado, G., Thrun, S., Dean, J.: A guide to deep learning in healthcare. Nature medicine 25(1), 24–29 (2019)
2. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature 521(7553), 436–444 (2015)
3. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems 25 (2012)
4. Habibzadeh, M., Krzyżak, A., Fevens, T.: Comparative study of shape, intensity and texture features and support vector machine for white blood cell classification. Journal of Theoretical and Applied Computer Science 5(1), 20–35 (2011)
5. Acevedo, A., Merino, A., Alférez, S., Molina, Á., Boldú, L., Rodellar, J.: A dataset of microscopic peripheral blood cell images for development of automatic recognition systems. Data in brief 30, 105474 (2020)
6. Yang, J., Shi, R., Ni, B.: MedMNIST Classification Decathlon: A Lightweight AutoML Benchmark for Medical Image Analysis. IEEE 18th International Symposium on Biomedical Imaging (ISBI), 191–195 (2021)
7. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)