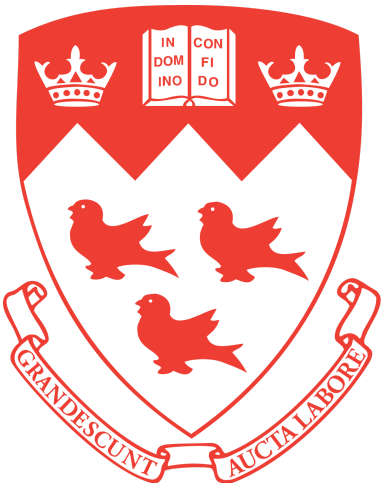
Project 3 Report
COMP 551 - Applied Machine Learning



By
Group 71
Aldo Abou Chedid - 26079749
Alain Daccache - 260714615
Dylan Havelock - 260721075

December 14, 2020
McGill University

Abstract

In this project, we classify image data using a modified MNIST dataset, where each image consists of 1 to 5 digits, with a label 10 representing the no-digit class, for a total of 11 classes. This is a multi-label classification problem, where we have to predict a set of target variables instead of one. Our proposed model consists of a two step approach, first separating the digits using computer vision, then training a deep neural network on the single digit images.

Data Preprocessing

We transform our multi-label problem into a single-label problem. Each sample is a 64 x 64 grayscale image. We split our training, validation, and testing data. To pre-process our data, we first use computer vision to separate the digits by specifying a threshold and finding contours. We discard the images for which the algorithm detects a different number count than the associated label. We then extract each digit from the instance into its own image, by converting each instance of shape (64, 64, 1) to (k, size=12, size=12), where k is the number of digits in the instance, then padding with zeros to fill the image. To convert to a PyTorch appropriate data structure, we make it iterable with a n x 2 array, where n is the total number of extracted images, and the 2 entries consisting of the vectorized image, and the label.

Model Training

For our methodology, we train and validate the network for a certain number of epochs, and early stop once the validation loss has not improved for a specified number of iterations. In training mode, we first clear the gradients (since they accumulate). We then make a forward pass and get the probabilities for each class (first getting log-probabilities, then exponentiate). We capture the class with highest likelihood, then verify the prediction with the labels provided. We use this to calculate the accuracy, and our criterion (specified in the architecture) applied to the log-likelihoods found and the labels, to calculate the loss, and add it to our running loss. We then get the gradients with respect to the parameters by backpropagating (`loss.backward()`), in other words fine-tuning the weights based on the loss obtained in the epoch. Then, we update the parameters (`optimizer.step()`).

We test our model using single digit images extracted from the provided dataset. This means we are training our model on single digit instances rather than the full 1-5 digit images provided in the original dataset. To test on the full training instances, that is the 1-5 digit images, we first run our separation algorithm to separate digits, run each digit in our model individually, then combine the results of each digit together to generate the full label.

Deep Neural Network Architecture

Our network's architecture consists of input, hidden, and output layers. All layers will use the ReLU activation function (`nn.ReLU`). The 'frontend' (for feature extraction) consists of convolutional and pooling layers. We started with a single **convolutional layer** (`nn.Conv2d`) with a small filter size (3,3) and a modest number of filters (32). We then used **dropout** (`nn.Dropout`) to randomly deactivate neurons at each training step in order to avoid the problem of overfitting. This helped us reach 99% accuracy in 30 epochs. By adding a **max pooling layer** (`nn.MaxPool2d`), we were able to reach that accuracy in 15. The filter maps can then be flattened to provide features to the classifier.

We repeat the same process for another **convolutional layer**. We avoid using Dropout on this layer to avoid erasing part of our predictions during training. We stopped adding hidden layers once the input size became too small. Then, we added a dense layer, and finished with softmax. We used **batch normalization** (`nn.BatchNorm2d`) after **convolutional** (`nn.Conv2D`) and **fully connected** (`nn.Linear`) layers, to change the distribution of the output of the layer, specifically by standardizing the outputs, thus stabilizing and accelerating the learning process. The original paper [1] talks about applying batch normalization just before the activation function (`nn.ReLU`), so we have followed that practice.

We added these layers one by one, testing the relative improvement each has on the architecture (Fig. 1→5). We also experimented with *ResNet*, a residual network architecture available in PyTorch, typically used in image classification tasks [2]. The paper describes a method of making CNNs with a depth of up to 152 layers trainable. We noticed the similarities in terms of the layering of the architecture we were converging to, thus decided to compare both, as shown in Fig. 6 and Table 1.

The 'backend' (for the classifier) is a one-hot encoded vector from the labels; the ten labels correspond to the ten nodes in this layer, used to predict the probability distribution of an image belonging to each of the 10 classes. It also uses a special activation function called softmax. This normalizes the values from the ten output nodes such that all the values are between 0 and 1, and the sum of all ten values is 1. This allows us to treat those ten output values as probabilities, and the largest one is selected as the prediction for the one-hot vector.

Evaluation of Hyperparameters & Other Factors

Other factors contribute to the performance of our neural network: the loss function, the optimization function, and the number of training epochs. As for **training epochs**, we early stop the training based on the validation loss. Since results converged between only 15 and 25 epochs, we chose 5 iterations for the early stop, as 10 would likely overfit. In order to save time, the training for the optimizer parameters was capped at 10 epochs.

For the **optimization function**, we considered the **Adam** (adaptive learning rate) and **SGD** (stochastic gradient descent) optimizers. While Adam always gave a higher accuracy on the validation dataset compared to SGD, the former was very sensitive to certain values of learning rate, sometimes leading to failure of the accuracy to converge (a choppy validation accuracy over training epochs). Although it would be better practice to settle for SGD without momentum, we opted for Adam and kept the model training instance that gave a stable (log like) accuracy curve for our model.

At the last step of our evaluation, we compared the Adam optimizer under two settings: constant learning rate ($1 \cdot 10^{-4}$) and variable learning rate (starting at 0.01 and decreasing by a factor of 10 every 10 epochs). We settled on the optimizer with the variable learning displayed more accurate and more stable results.

The categorical **cross-entropy loss function** will be optimized, suitable for multi-class classification, and we will monitor the classification accuracy metric, which is appropriate given we have the same number of examples in each of the 10 classes.

Table 1 shows a summary of all hyper parameter configuration results tested. We opted not to do a simple grid search over the parameters since it would take too long to train each combination of parameters, considering the number of parameters and training time for each configuration. Figures 6 to 11 show the performance of additional configurations.

References

[1] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

[2] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." ArXiv:1512.03385 [Cs], December. http://arxiv.org/abs/1512.03385 (2015).

Appendix

Table 1. Performance of runs with different architectures

Epochs	ReLU	Batch Normalization	Max Pooling	Dropout	Optimizer	Learning rate	Momentum	Validation Accuracy (rounded)	Train Accuracy
25	False	False	False	False	Adam	0.0001	-	0.878	0.876
16	True	False	False	False	Adam	0.0001	-	0.997	0.998
25	True	True	False	False	Adam	0.0001	-	0.997	0.999
19	True	True	True	False	Adam	0.0001	-	0.996	0.998
25	True	True	True	True	Adam	0.0001	-	0.992	0.994
35	ResNet				Adam	0.0001	-	0.997	0.999
20	ResNet				Adam	Variable	-	0.998	1.000
10	True	True	False	True	Adam	0.00005	-	0.995	0.998
10	True	True	False	True	Adam	0.0001	-	0.996	0.998
10	True	True	False	True	Adam	0.0005	-	0.994	0.995
10	True	True	False	True	SGD	0.005	-	0.991	0.995
10	True	True	False	True	SGD	0.01	-	0.994	0.995
10	True	True	False	True	SGD	0.05	-	0.990	0.994
10	True	True	False	True	SGD	0.01	0.8	0.992	0.994
10	True	True	False	True	SGD	0.01	0.85	0.990	0.992
9	True	True	False	True	SGD	0.01	0.9	0.974	0.974

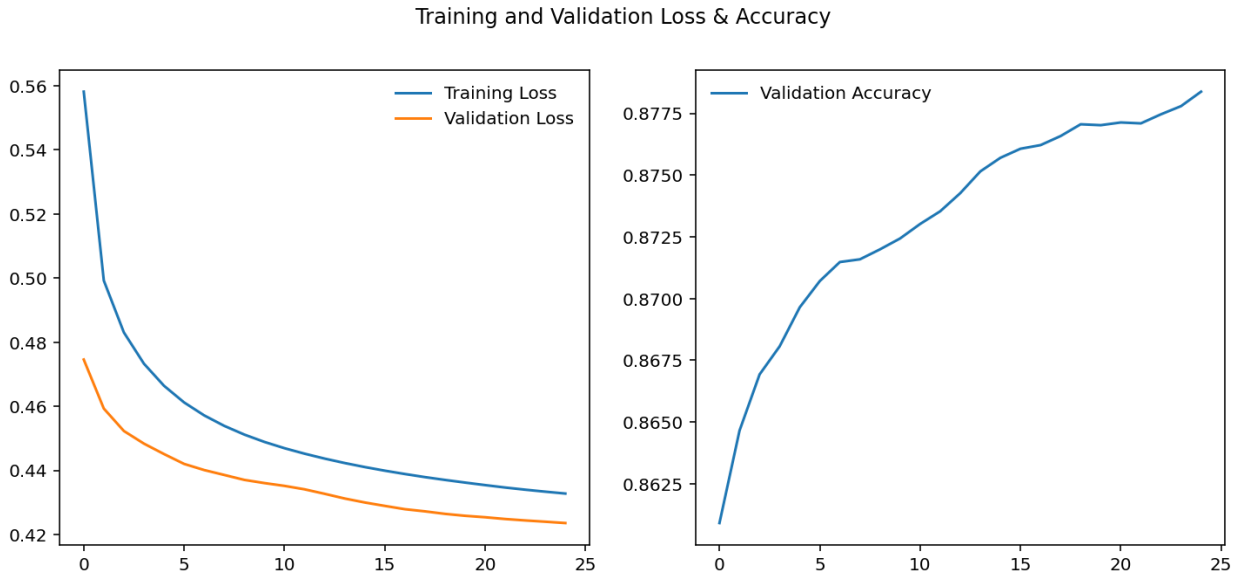


Figure 1． ReLU = False, BatchNorm = False, MaxPool = False, Dropout = False

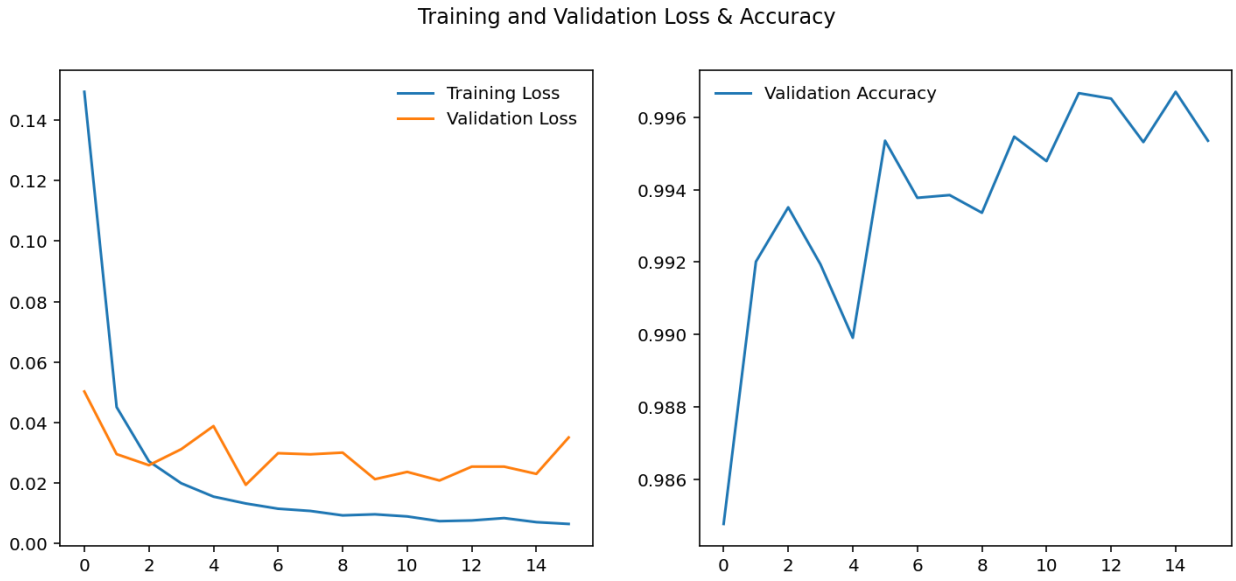


Figure 2． ReLU = True, BatchNorm = False, MaxPool = False, Dropout = False

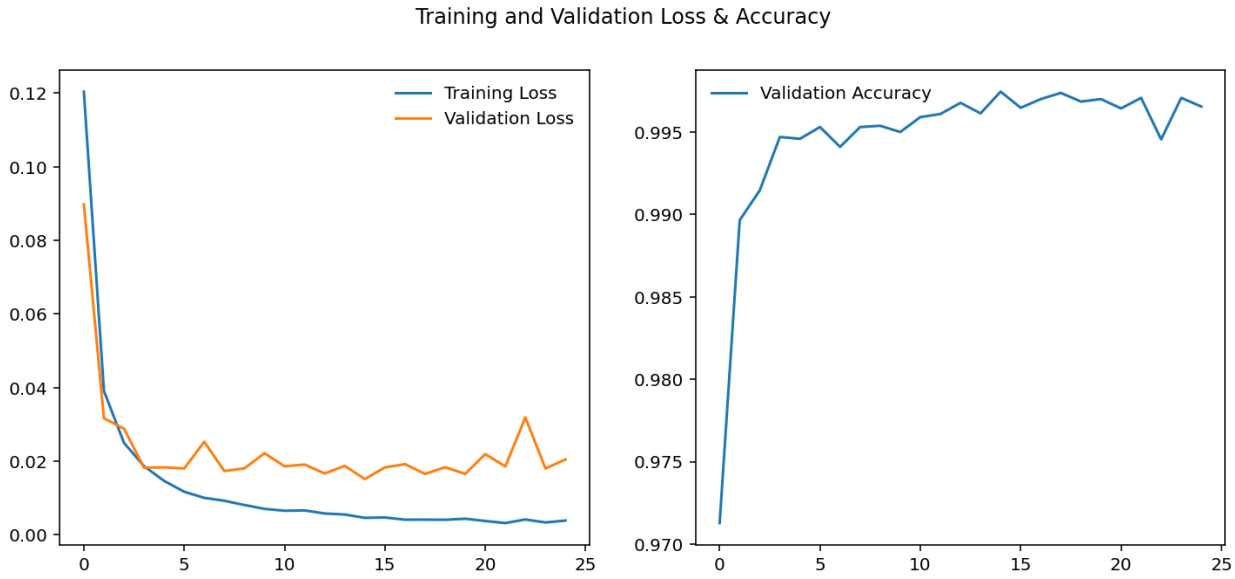


Figure 3． ReLU = True, BatchNorm = True, MaxPool = False, Dropout = False

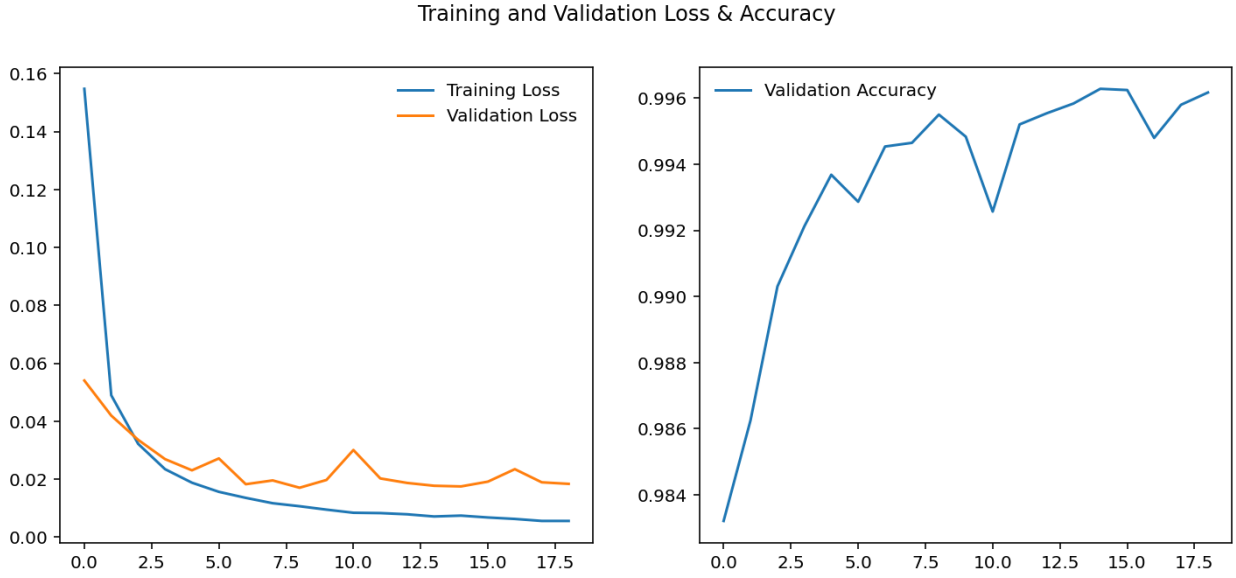


Figure 4． ReLU = True, BatchNorm = True, MaxPool = True, Dropout = False

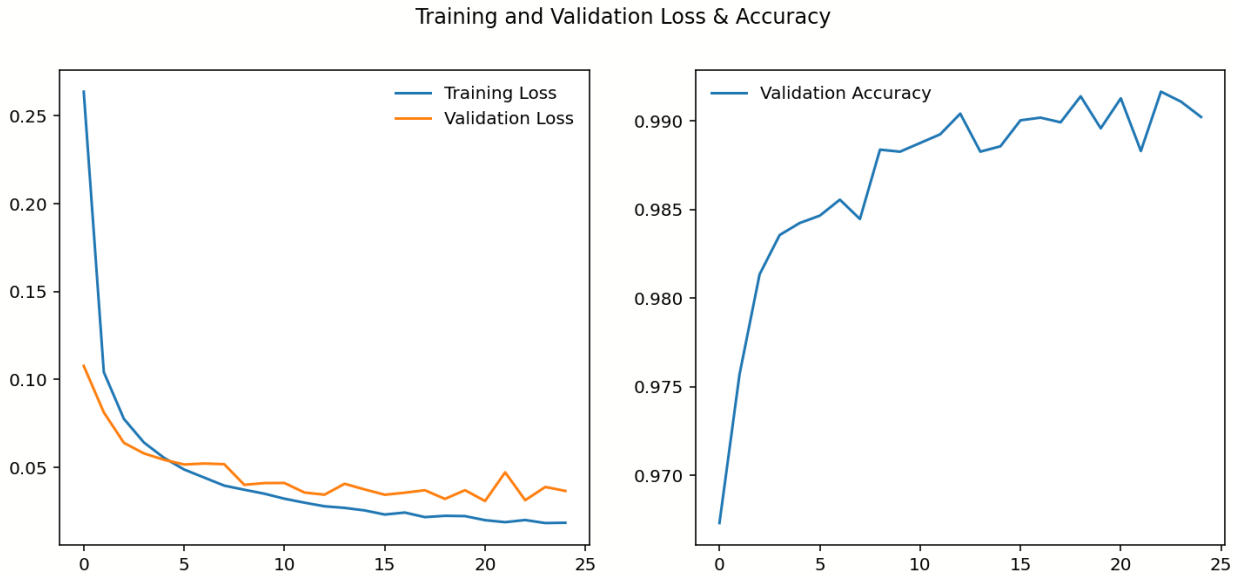


Figure 5. ReLU = True, BatchNorm = True, MaxPool = True, Dropout = True

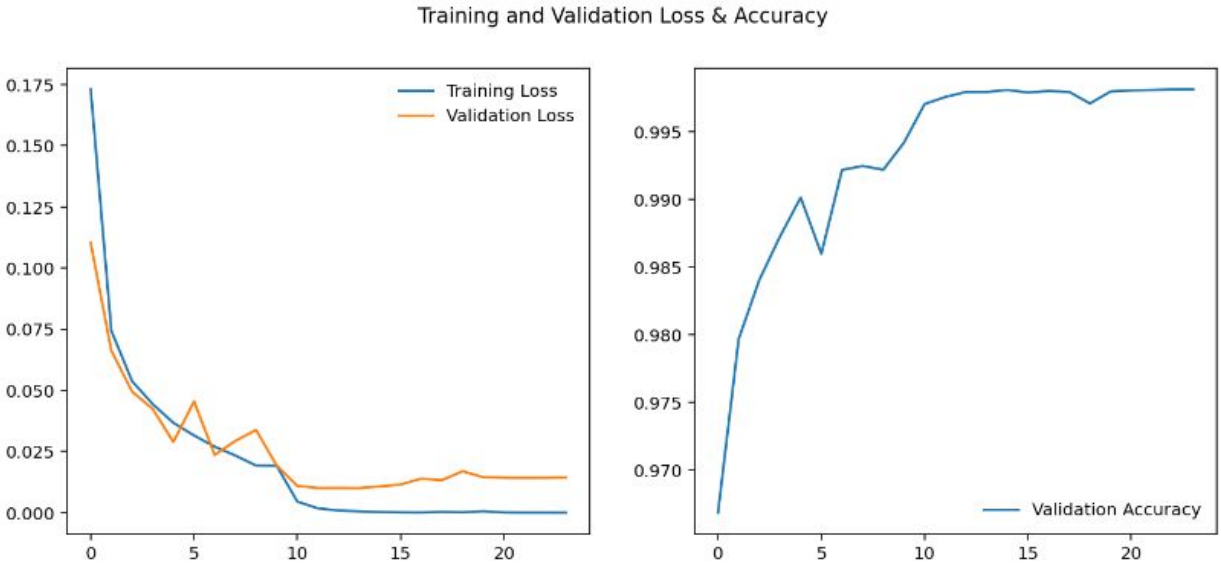


Figure 6. Using ResNet Architecture.

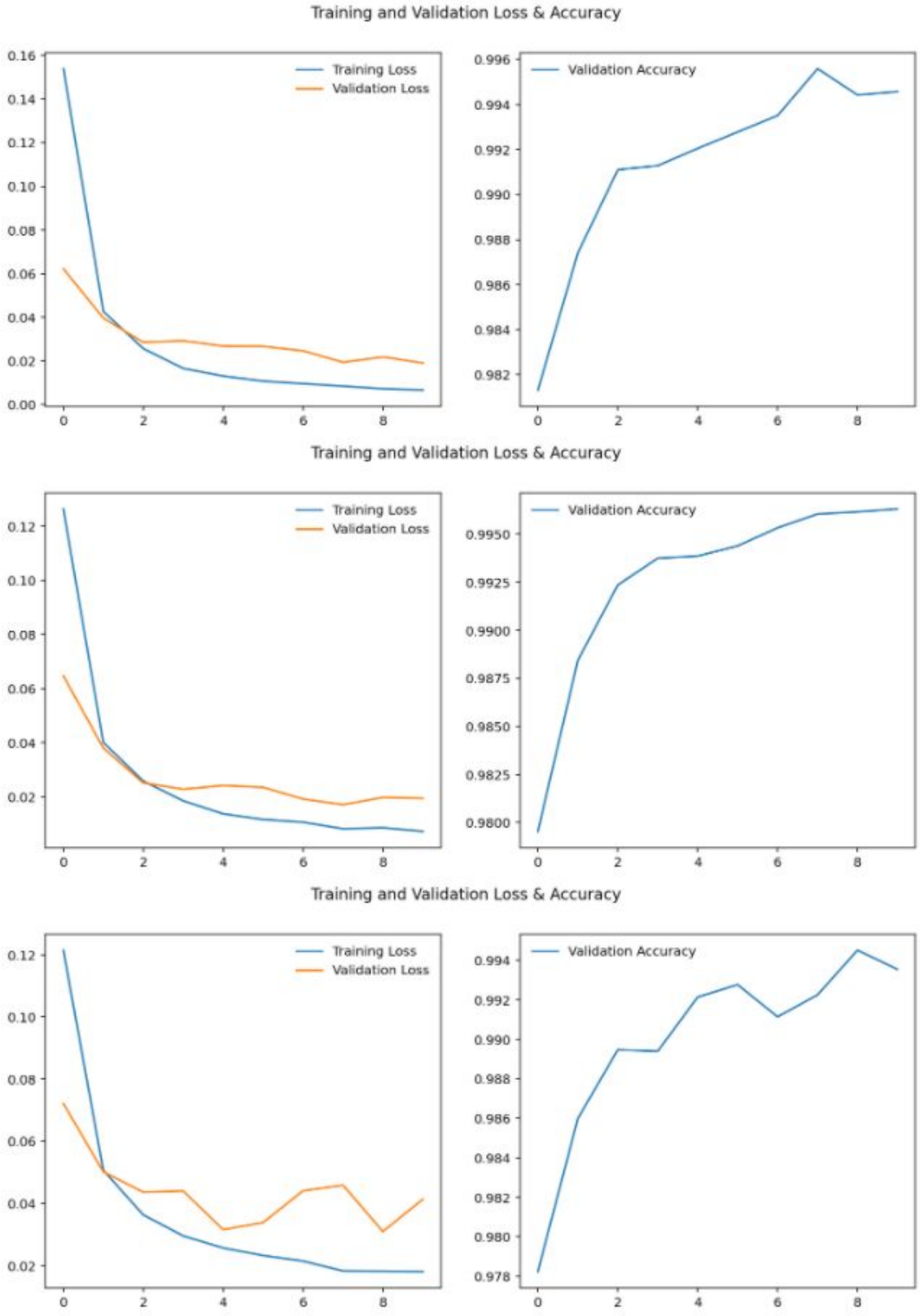


Figure 7. Performance of runs with an Adam optimizer and a learning rate of $5 \cdot 10^{-5}$ (top), $1 \cdot 10^{-4}$ (center), $5 \cdot 10^{-4}$ (bottom).

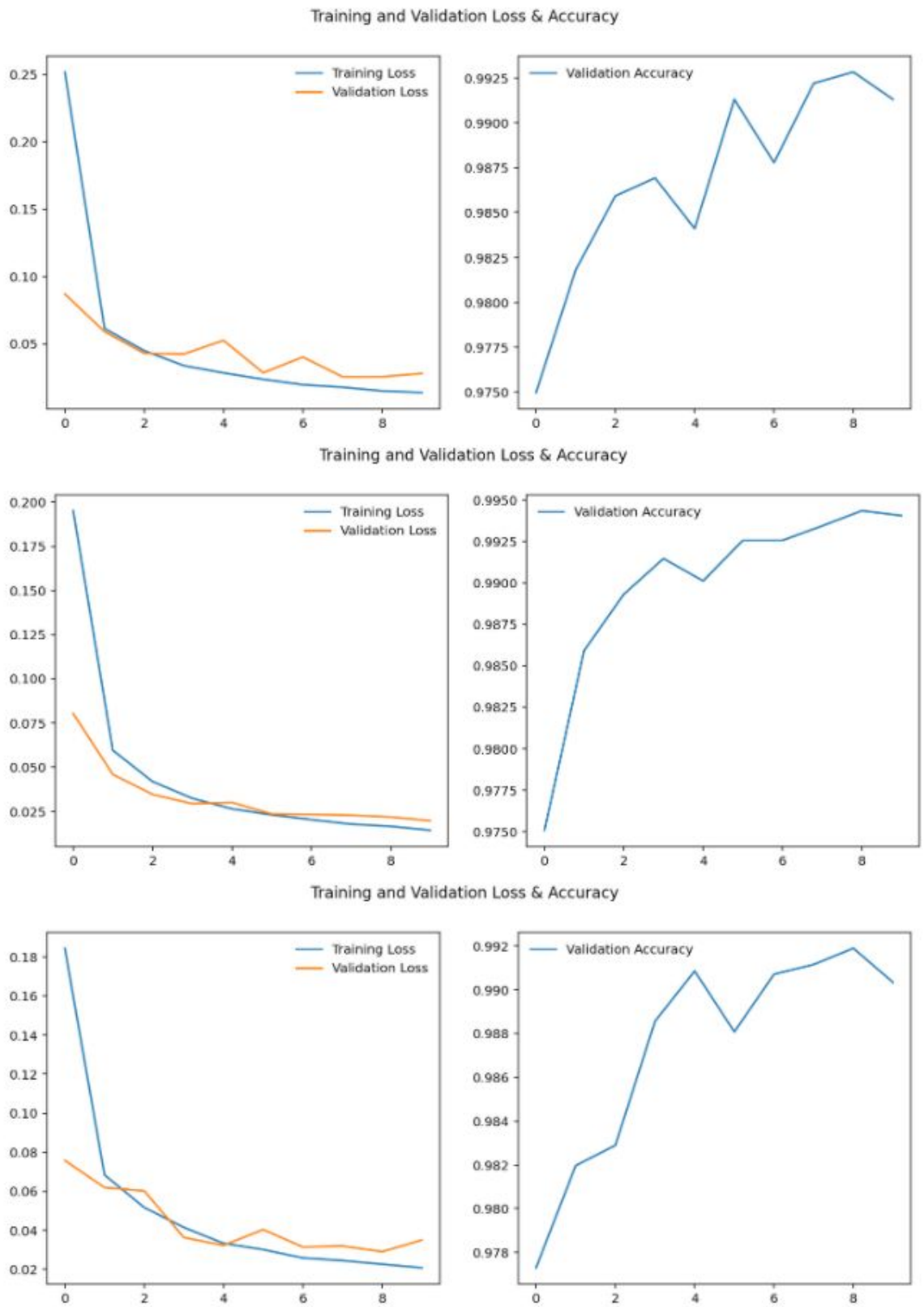


Figure 8. Performance of runs with an SGD optimizer and a learning rate of $5 \cdot 10^{-3}$ (top), $1 \cdot 10^{-2}$ (center), $5 \cdot 10^{-2}$ (bottom).

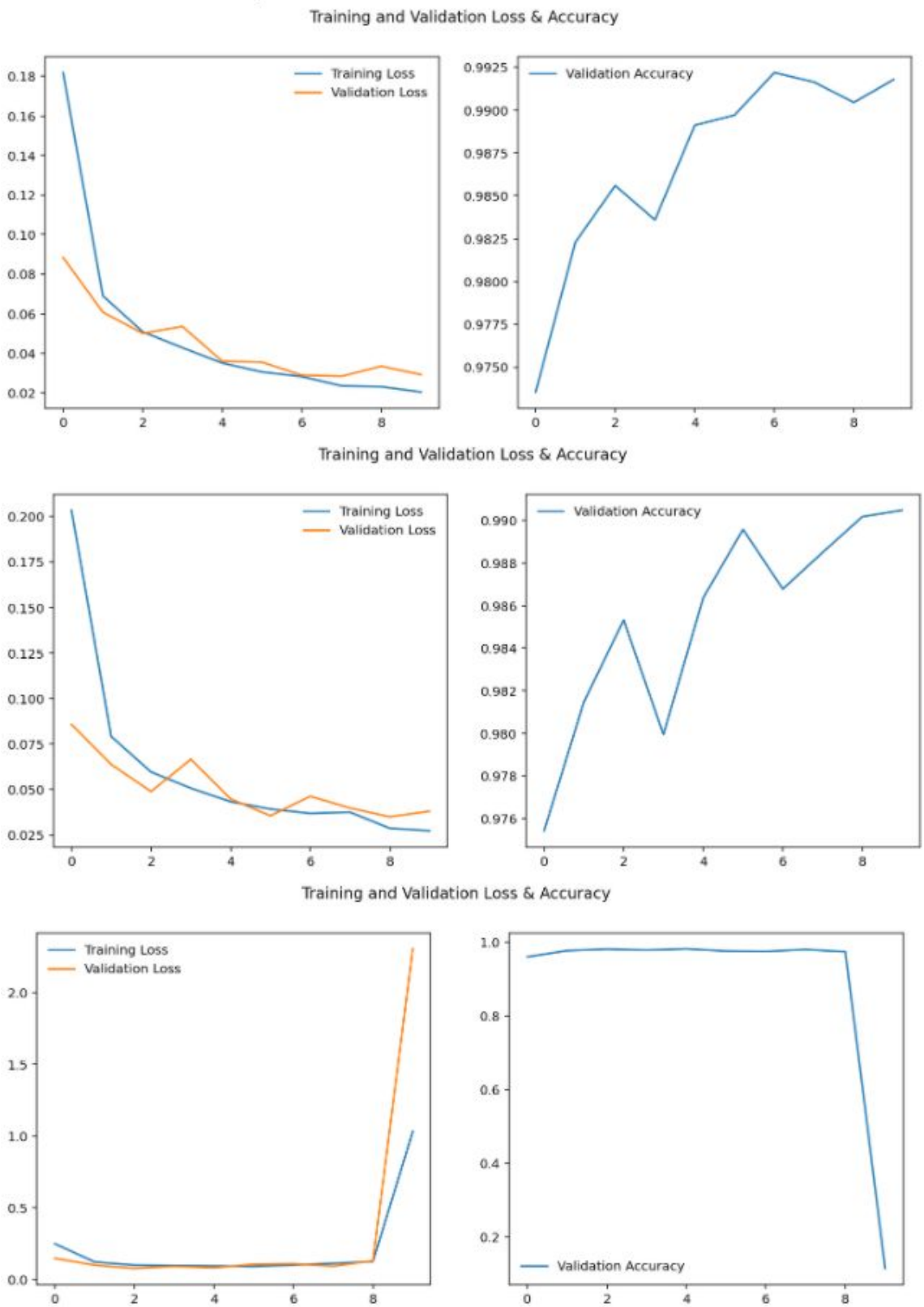


Figure 9. Performance of runs with an SGD optimizer with a learning rate of 0.01 and a momentum of 0.8 (top), 0.85 (center), 0.9(bottom).

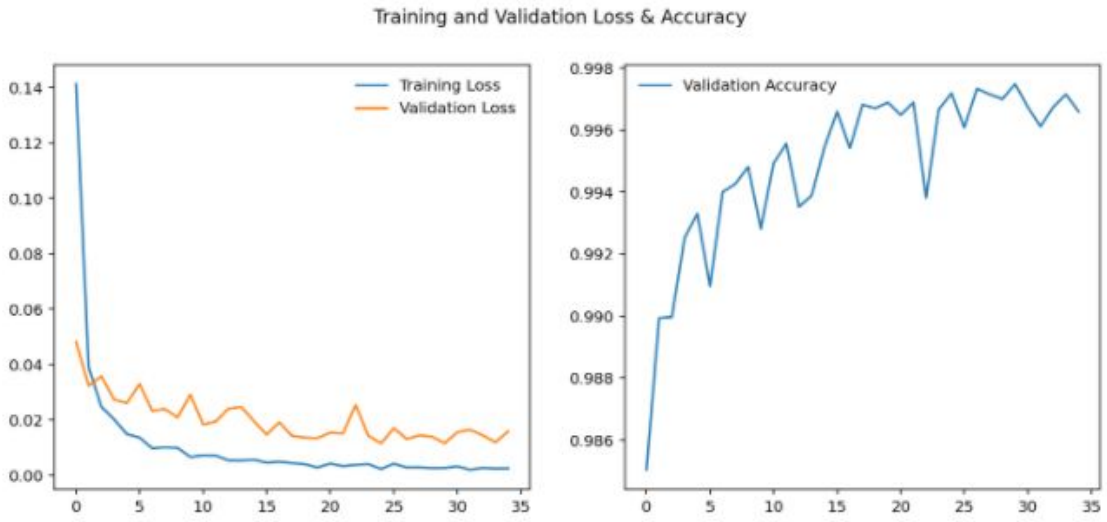


Figure 10 ResNet with an Adam optimizer at a constant learning rate of 0.0001

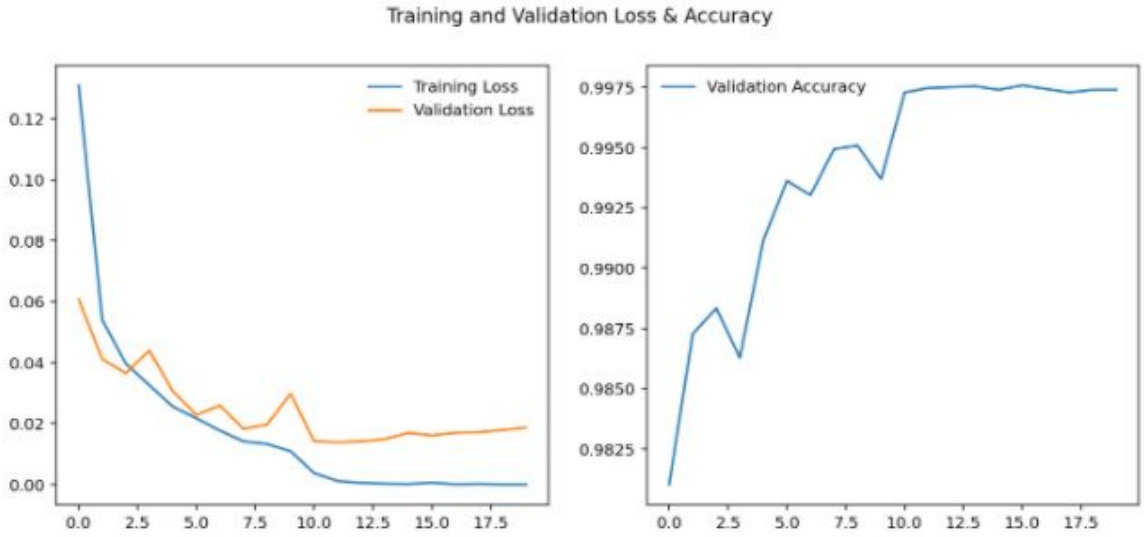


Figure 11. ResNet with an Adam optimizer at a varying learning rate