1.0 Abstract

In this mini project, our group implemented and investigated the performance and accuracy of a convolutional neural network (CNN) to classify images containing 1 to 5 numerical digits accessed through an h5py storage file. The investigation was driven by observing the effects of changing various factors on the resultant scoring metrics, including overall accuracy, R2 scoring and root mean squared error (RMSE). Key elements that were evaluated to tune model parameters and hyper-parameters included the number of neural network (NN) layers, kernel/filter size, batch size, regularization techniques and length of training (number of epochs), among others. A number of predicted label sets, derived from the various network architectures that were tested, were submitted to an online Kaggle competition between both public and private (COMP-551) groups to achieve the most accurate framework. Despite a single highest test accuracy of 97.9% obtained using a 5 CNN layer model (with 2 fully connected, FC, layers), this study achieved its most consistent results using a 3 CNN layer model (with 2 FC layers).

2.0 Introduction

This report presents the validation and prediction results from various neural network architectures. This classification task utilized a modified MNIST dataset (44,800 training and 11,200 validation samples), assembling arrays (tensors) to identify the images comprised of 1 to 5 digits with their corresponding integer value, and assigning a class label of 10 to non-digits. Using Pytorch and torch-vision modules, neural networks were constructed with automatic differentiation and optimization capabilities. Pytorch tensors were implemented to help construct the neural network architecture. Utilizing a variation of hyperparameters, our group classified the unseen test dataset (14,000 samples) to 97.99% accuracy. A combination of user-written functions and sklearn modules were used to analyze the various accuracies and error metrics, assisting in improving the overall prediction accuracy of the neural network.

3.0 Results

3.1 Architectures

Various network architectures were implemented in attempts to increase prediction accuracy on the given dataset. Using Pytorch's Sequential network model, layers of nodes and optimizers were inputted in the desired order in the Python file. Torch.nn modules, such as Conv2d and Linear (layer types), ReLU (activation function), BatchNorm2d and 1d (normalization), MaxPool2d (downsizing feature sets) and Dropout (regularization), were applied in various ways to the individual model layers. Minimization was targeted using cross-entropy (negative log-likelihood) as the cost (loss) function with backpropagation. An example of the type of framework used in this study is depicted schematically for a 3 CNN layer model (with 2 FC layers) in Figure 1. The different configurations that were tested, including model depth, filter size, batch size, optimizer hyper-parameters and length of training (number of epochs) are summarized in Table 1 (see footnote for description of baseline comparison model). Corresponding training, validation and testing (where applicable) metrics are also included. Additional tests were carried out among the better model candidates in attempts to further improve model prediction accuracies.

3.2 Optimizers

Pytorch modules for optimization were used to improve prediction results, specifically the Adam optimizer. Adam optimization for neural networks is an adaptive learning rate optimization algorithm[1]. Tuning of optimization hyper-parameters was focused mainly on the initial learning rate (denoted α in Table 1), with lesser emphasis on weight decay (L2 regularization). The baseline model for comparison

[1] https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

across different architectures kept the learning rate at 0.005, as deflections from this value appeared to generally increase the RMSE and reduce prediction accuracy.

3.3 Batch Size

The batch size of a neural network defines the number of training samples that will be propagated through the network at a time. As our training set consisted of 44,800 unique samples, we elected to use mainly batch sizes in the range of 64 to 128. Through changing the batch sizes in various architectures, it was evident that reducing the batch size to 32 produced much worse accuracy predictions, while changing it to 128 and 256 did not produce any meaningful improvements.

3.4 Activation & Regularization

For the hidden layers of the neural network, the ReLU activation function was used throughout model architecture design. Unlike other activation functions like sigmoid and the hyperbolic tangent function (tanh), ReLU allows for exact zero values[2] and therefore helps diminish the effects of vanishing gradient during backpropagation. Each convolutional layer also implemented batch normalization and dropout as techniques to stabilize the learning process and reduce overfitting, respectively. Limited inclusion of weight decay factors (see Table 1) were attempted but did not produce any significant improvements.

3.5 Training Length (# of Epochs)

In the context of neural networks, the number of epochs refers to the number of training cycles conducted on the entire training dataset through the model. This was a hyper-parameter that our group adjusted frequently; different epoch counts resulted in different accuracy metrics. Figure 2 shows an example of a representative training/validation loss history plot, displaying the rapid decrease over the early epochs as the model learned and stabilized over time. We opted to stop the iterations once the losses stabilized prior to eventual increases or significant fluctuations in the loss function (typically corresponds to increases in RMSE and related decreases in accuracy).

4.0 Discussion and Conclusions

After extensive testing and adjustments of hyper-parameters and neuron layers, the best Kaggle result (97.9%) corresponded to a 5-layer CNN with 2 fully connected layers. Maximum pooling (with filter and stride sizes of 2) was used on the 2nd and 5th layers, in addition to an initial learning rate of 0.005 (with no weight decay) for the Adam optimizer. This architecture resulted in a 99.42% training accuracy, an R2 score of 0.9963, and a RMSE value of 0.1929 (Table 1). Additional tests were ongoing to adjust learning rates, training length (epochs), batch sizes and weight decay factors, but no significant improvements to test accuracy were observed.

Although our single highest test accuracy corresponded to a 5 CNN layer model (with 2 FC layers), experimental analysis showed that a 3 CNN layer architecture produced more consistent and robust prediction accuracies among hyper-parameter variations, particularly training length (Table 1). A possible explanation for this robustness is that more complex (deeper) models with higher numbers of layers would overfit the data, yielding poorer accuracy metrics.

[2] https://machinelearningmastery.com/activation-regularization-for-reducing-generalization-error-in-deep-learning-neural-networks/

5.0 Appendix

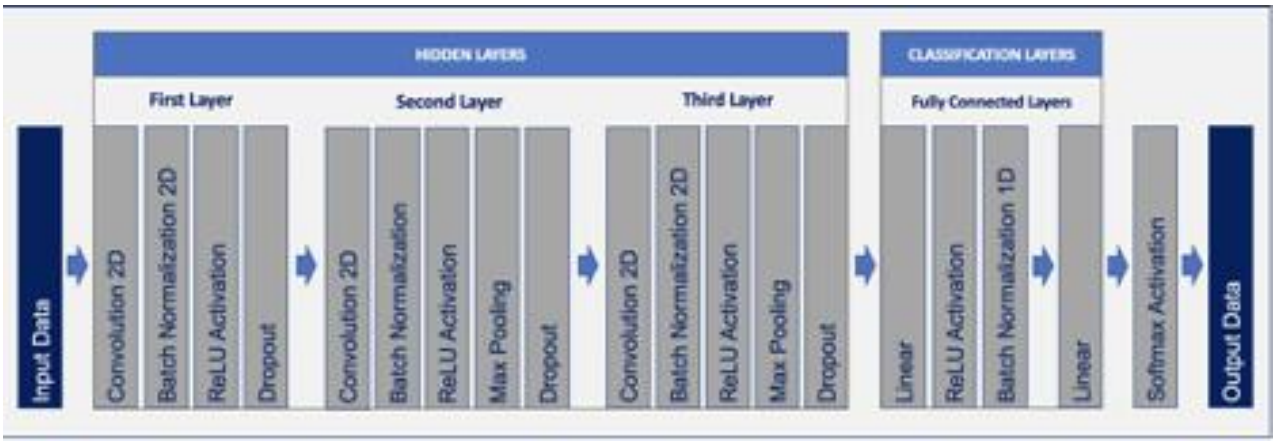

Figure 1. Schematic diagram for a 3 convolutional neural network (CNN) layer model (with 2 fully connected, FC, layers), as used in this study. Note that ReLU activation was used for the hidden layers, with softmax activation applied to the outputs.

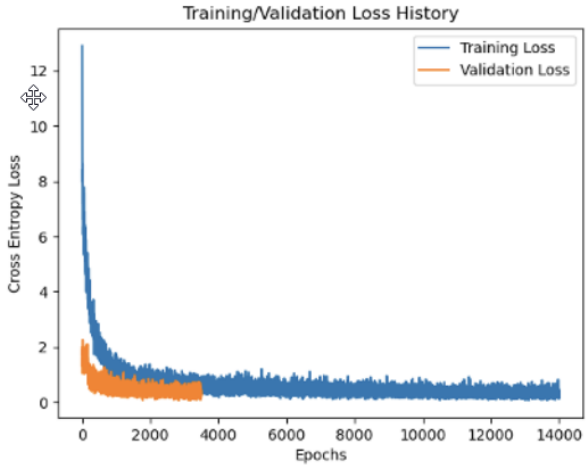

Figure 2. Representative training/validation loss history plot showing a rapid decrease over the early epochs as the model learned and stabilized over time. Training length was adjusted to stop once losses stabilized prior to eventual increases or significant fluctuations in the loss function (typically corresponds to increases in RMSE and related decreases in accuracy).

Table 1. Summary of test results from varied convolutional neural network (CNN) model architectures, parameters and hyper-parameters with corresponding training and validation metrics.

#	Layers	Feature Mapping	k	s	p	maxPool (k, s)	Dropout	Batch Size	Epochs	Optimizer	Hyper-parameters (α, β_1, β_2)	Test Accuracy	*Training*			*Validation*		
													Accuracy	R2_Score	RMSE	Accuracy	R2_Score	RMSE
1	7 CNN 2 FC	32-64-128-256-512-512-512	3, ..., 3	1, ..., 1	1, ..., 1	0, (2,2), 0, (2,2), (2,2), (2,2), (2,2)	0.5, 0.2, ..., 0.2	64	20	Adam	0.005, 0.9, 0.999*	-	0.9422	0.9735	0.5089	0.9469	0.973	0.5116
		20-40-80-120-160-200-240	3, ..., 3	1, ..., 1	1, ..., 1	0, (2,2), 0, (2,2), (2,2), (2,2), (2,2)	0.5, 0.2, ..., 0.2	64	50	Adam	0.005, 0.9, 0.999*	-	0.9967	0.9971	0.1627	0.9977	0.9969	0.1673
		20-40-80-120-160-200-240	3, ..., 3	1, ..., 1	1, ..., 1	0, (2,2), 0, (2,2), (2,2), (2,2), (2,2)	0.5, 0.2, ..., 0.2	64	60	Adam	0.005, 0.9, 0.999*	0.77	0.9972	0.9964	0.1738	0.9979	0.9964	0.1686
		20-40-80-120-160-200-240	3, ..., 3	1, ..., 1	1, ..., 1	0, (2,2), 0, (2,2), (2,2), (2,2), (2,2)	0.5, 0.2, ..., 0.2	64	75	Adam	0.005, 0.9, 0.999*	-	0.9978	0.9969	0.1654	0.9987	0.9973	0.1523
2	2 CNN 2 FC	32-64	3, 3	1, 1	1, 1	(2,2), (2,2)	0.5, 0.2	64	25	Adam	0.005, 0.9, 0.999*	-	0.9569	0.9720	0.5049	0.9673	0.9772	0.4519
		32-64	3, 3	1, 1	1, 1	(2,2), (2,2)	0.5, 0.2	64	50	Adam	0.005, 0.9, 0.999*	0.9183	0.9862	0.9951	0.2220	0.9904	0.9969	0.1752
		32-64	3, 3	1, 1	1, 1	(2,2), (2,2)	0.5, 0.2	64	60	Adam	0.005, 0.9, 0.999*	0.9279	0.9350	0.9645	0.5777	0.9442	0.9692	0.5409
		32-64	3, 3	1, 1	1, 1	(2,2), (2,2)	0.5, 0.2	64	75	Adam	0.005, 0.9, 0.999*	0.9362	0.9882	0.9955	0.2167	0.9896	0.9960	0.2097
		32-64	3, 3	1, 1	1, 1	(2,2), (2,2)	0.5, 0.2	64	90	Adam	0.005, 0.9, 0.999*	-	0.9700	0.9844	0.3976	0.9739	0.9861	0.3795
3	CNN	32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	5	Adam	0.005, 0.9, 0.999*	-	0.9456	0.9810	0.4327	0.9623	0.9851	0.3773
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	5	Adam	0.005, 0.9, 0.999* w_decay=0.5	-	0.1853	0.1610	2.8681	0.1959	0.1576	2.8581
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	10	Adam	0.005, 0.9, 0.999*	-	0.9621	0.9836	0.3973	0.9728	0.9888	0.3270
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	10	Adam	0.005, 0.9, 0.999* w_decay=0.0005	-	0.9859	0.9947	0.2273	0.9929	0.9975	0.1616
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	10	Adam	0.005, 0.9, 0.999* w_decay=0.005	-	0.9231	0.9690	0.5863	0.9355	0.9717	0.5655
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	10	Adam	0.005, 0.9, 0.999* w_decay=0.05	-	0.8188	0.8833	1.1475	0.8364	0.8936	1.1041
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	10	Adam	0.005, 0.9, 0.999* w_decay=0.5	-	0.9059	0.9654	0.5827	0.9369	0.9773	0.4743
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	15	Adam	0.005, 0.9, 0.999*	-	0.9532	0.9826	0.4112	0.9585	0.9843	0.3851
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	17	Adam	0.005, 0.9, 0.999*	-	0.9936	0.9972	0.1660	0.9970	0.9990	0.1027
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	20	Adam	0.005, 0.9, 0.999*	-	0.9932	0.9970	0.1743	0.9962	0.9979	0.1391
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	20	Adam	0.01, 0.9, 0.999*	-	0.9935	0.9969	0.1691	0.9963	0.9988	0.1090
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	20	Adam	0.005, 0.9, 0.999* w_decay=0.5	-	0.0316	-0.0191	3.1108	0.0350	0.0076	3.0669
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	32	25	Adam	0.005, 0.9, 0.999*	-	0.0289	-0.4297	3.7545	0.0299	-0.4420	3.7787
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	25	Adam	0.005, 0.9, 0.999*	0.9692 0.9633	0.9965	0.9988	0.1057	0.9975	0.9993	0.0866
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	25	Adam	0.0001, 0.9, 0.999*	-	0.2141	-0.1119	3.4678	0.2153	-0.1331	3.4868
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	25	Adam	0.01, 0.9, 0.999*	-	0.9914	0.9956	0.2048	0.9951	0.9985	0.1236
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	128	25	Adam	0.005, 0.9, 0.999*	0.9670	0.9975	0.9989	0.1044	0.9992	0.9996	0.0609
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	256	25	Adam	0.005, 0.9, 0.999*	0.9594	0.9928	0.9972	0.1674	0.9956	0.9984	0.1206
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	30	Adam	0.01, 0.9, 0.999*	-	0.9252	0.9699	0.5096	0.9272	0.9705	0.5009
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	30	Adam	0.005, 0.9, 0.999*	-	0.9978	0.9991	0.0905	0.9987	0.9996	0.0632
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	35	Adam	0.005, 0.9, 0.999*	0.9670 0.9690	0.9989	0.9995	0.0683	0.9996	0.9998	0.0408
		32-64-128	3, 3, 3	1, 1, 1	1, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	40	Adam	0.005, 0.9, 0.999*	-	0.9973	0.9981	0.1321	0.9980	0.9989	0.0985
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	20	Adam	0.005, 0.9, 0.999*	-	0.9819	0.9934	0.2549	0.9899	0.9965	0.1875
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	30	Adam	0.005, 0.9, 0.999*	-	0.9871	0.9951	0.2179	0.9922	0.9974	0.1606
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	40	Adam	0.005, 0.9, 0.999*	-	0.9927	0.9978	0.1546	0.9950	0.9986	0.1182
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	50	Adam	0.005, 0.9, 0.999*	-	0.9947	0.9979	0.1459	0.9968	0.9987	0.1082
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	60	Adam	0.005, 0.9, 0.999*	0.9623	0.9964	0.9986	0.1153	0.9978	0.9988	0.1007
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	128	60	Adam	0.005, 0.9, 0.999*	-	0.9961	0.9981	0.1347	0.9970	0.9988	0.1100
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	256	60	Adam	0.005, 0.9, 0.999*	0.9588	0.9961	0.9986	0.1147	0.9980	0.9991	0.0911
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	65	Adam	0.005, 0.9, 0.999*	-	0.9957	0.9974	0.1558	0.9967	0.9981	0.1280
		32-64-128	5, 3, 3	2, 1, 1	2, 1, 1	0, (2,2), (2,2)	0.5, 0.2, 0.2	64	75	Adam	0.005, 0.9, 0.999*	-	0.9963	0.9956	0.2020	0.9970	0.9941	0.2365
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	10	Adam	0.005, 0.9, 0.999*	-	0.5606	0.3914	2.5017	0.5762	0.4034	2.4755
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	15	Adam	0.005, 0.9, 0.999*	-	0.9720	0.9731	0.4978	0.9750	0.9732	0.4923
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	16	Adam	0.005, 0.9, 0.999*	0.9624	0.9641	0.9113	0.9207	0.9661	0.9082	0.9336

COMP 551 - Mini Project Three
Group 50 - Dean Meluban, Javier Ordenes Alegria, Ryan Wilson

#	Layers	Feature Mapping	k	s	p	maxPool (k, s)	Dropout	Batch Size	Epochs	Optimizer	Hyper-parameters (α, β_1, β_2)	Test Accuracy	*Training*			*Validation*		
													Accuracy	R2_Score	RMSE	Accuracy	R2_Score	RMSE
4	4 CNN 2 FC	32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	20	Adam	0.005, 0.9, 0.999*	-	0.9929	0.9915	0.2725	0.9936	0.9906	0.2897
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	25	Adam	0.005, 0.9, 0.999*	0.9723	0.9950	0.9913	0.2724	0.9960	0.9934	0.2406
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	27	Adam	0.005, 0.9, 0.999*	0.9744	0.9976	0.9984	0.1198	0.9982	0.9972	0.1652
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	27	Adam	0.0001, 0.9, 0.999*	-	0.5321	0.4653	2.3648	0.5430	0.4740	2.3367
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	128	27	Adam	0.005, 0.9, 0.999*	-	0.9814	0.9582	0.6169	0.9846	0.9685	0.5346
		32-64-128-256	3, 3, 3, 3	1, 1, 1, 1	1, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	30	Adam	0.005, 0.9, 0.999*	-	0.9934	0.9925	0.2600	0.9941	0.9903	0.2979
		32-64-128-256	5, 3, 3,3	2, 1, 1, 1	2, 1, 1, 1	0, (2,2), 0, (2,2)	0.5, 0.2, ..., 0.2	64	27	Adam	0.005, 0.9, 0.999*	-	0.9917	0.9960	0.1954	0.9941	0.9961	0.1916
5	5 CNN 2 FC	32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	10	Adam	0.005, 0.9, 0.999*	-	0.9413	0.9794	0.4393	0.9471	0.9819	0.4094
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	15	Adam	0.005, 0.9, 0.999*	-	0.9469	0.9535	0.676	0.9518	0.955	0.6627
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	18	Adam	0.005, 0.9, 0.999*	-	0.9775	0.9890	0.3283	0.9828	0.9919	0.286
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	20	Adam	0.005, 0.9, 0.999*	-	0.9897	0.9818	0.4015	0.9918	0.9833	0.3791
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	21	Adam	0.005, 0.9, 0.999*	-	0.9898	0.9823	0.3956	0.9921	0.9835	0.3833
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	32	22	Adam	0.005, 0.9, 0.999*	-	0.9593	0.8866	1.0205	0.9620	0.8939	0.9832
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	22	Adam	0.005, 0.9, 0.999*	0.9799	0.9942	0.9963	0.1929	0.9963	0.9965	0.1767
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	96	22	Adam	0.005, 0.9, 0.999*	0.9763	0.9925	0.9923	0.2548	0.9968	0.9966	0.1721
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	23	Adam	0.005, 0.9, 0.999*	-	0.9807	0.9699	0.5155	0.9850	0.9742	0.4728
		32-64-128-256-512	3, 3, 3, 3, 3	1, 1, 1, 1, 1	1, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	25	Adam	0.005, 0.9, 0.999*	-	0.7386	0.8127	1.3638	0.7467	0.8105	1.3645
		32-64-128-256-512	5, 3, 3, 3, 3	2, 1, 1, 1, 1	2, 1, 1, 1, 1	0, (2,2), 0, 0, (2,2)	0.5, 0.2, ..., 0.2	64	22	Adam	0.005, 0.9, 0.999*	-	0.9508	0.8811	1.0673	0.9557	0.8845	1.0541

* Denotes initial hyper-parameter values (adaptive optimizer).

** k = kernel size, s = stride length, p = padding size, maxPool = maximum pooling.

*** Baseline model for architecture comparison: Feature mapping = 32-64(-128)(-256)(-512), k = 3, s=1, p=1, maxPool=(2,2), optimizer=Adam.