Modified MNIST Character Recognition

COMP-551: Applied Machine Learning, Section 001
Kaggle Project: Find the "Largest" Digit
22 March 2018

Amir Abushanab
260624419
Department of Computer Science
McGill University
Montreal, Canada
amir.abushanab@mail.mcgill.ca

Vince Porporino
260681922
Department of Computer Science
McGill University
Montreal, Canada
vincenzo.porporino@mail.mcgill.ca

Lili Zeng
260429344
Department of Physics
McGill University
Montreal, Canada
lili.zeng@mail.mcgill.ca

I. Introduction

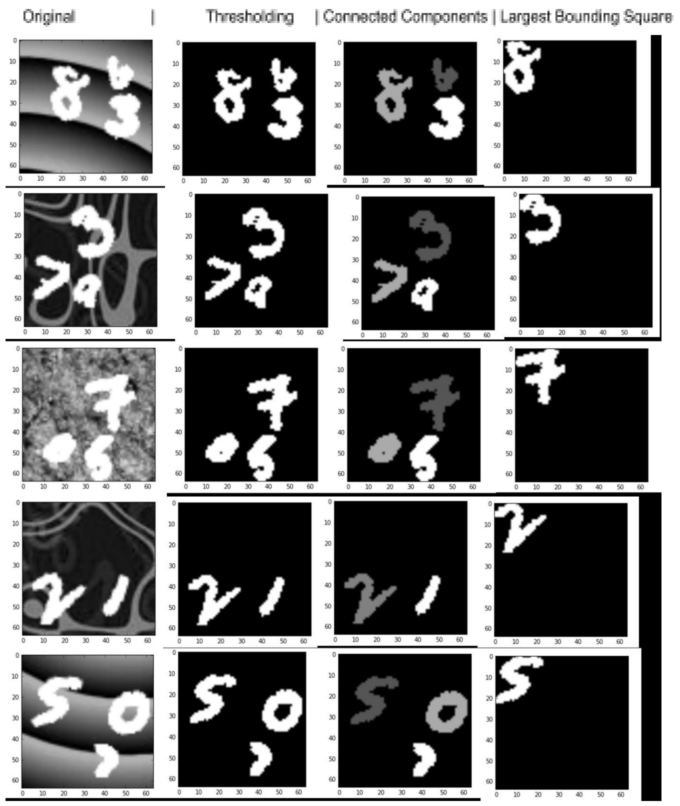
The task of this Kaggle project is to design a machine learning classifier algorithm to identify handwritten digits in a given image. The dataset for this project is a variation of the MNIST dataset, where each datapoint consists of a random grayscale image in the background and 2 or 3 MNIST digits randomly rotated, scaled, and placed in the foreground, represented as a 64x64 integer matrix. The objective is to identify the single digit that has been scaled the most from its original size in each of the 10k test set images, using a 50k labelled set as training.

We began by researching the best known learning algorithms for classifying the original MNIST digits and found that convolutional neural networks offer the best performance with accuracy as high or higher than 99% (see reference 1). We concluded that training a CNN and preprocessing the dataset to resemble as close as possible to the MNIST data would produce the best performance for our task. This assumption proved correct but for completeness sake and comparison, we also classified the data using other learning algorithms including a feed forward neural network implemented from scratch. Our most optimized CNN architecture classified the test set with 94.733% accuracy, while the rest of the learning algorithms produced results ranging from 20-80%.

II. Feature Design

To get the dataset to resemble as close as possible to the MNIST data, for each datapoint, we want to remove the background image, determine the most scaled digit i.e. the labelled digit, align that digit in the corner of the image, and remove the other digits entirely. Since we don't know the original size of the digits before scaling, there is no way to definitively isolate the labelled digit. As a heuristic to guess the labelled digit, we compute the smallest squares that completely bound each digit; their bounding squares, and take the digit with the largest bounding square by area. This heuristic is justified because the MNIST digits have roughly the same dimensions, therefore the most scaled up digit almost always has the largest bounding square. As this is a heuristic, there is a small margin of error in every dataset (see discussion section for more detail).

We use python's openCV library to do the image processing. openCV's threshold function takes an image and an integer threshold as parameters and sets all pixels below the threshold to 0 and all above the threshold to 255. Since the data points have very light grayscale backgrounds and very dark foreground digits, the threshold function is perfect with a threshold close to 255. openCV's ConnectedComponentsWithStats function takes as parameters the thresholded image and a connectivity of 4 or 8 pixels and returns the labelled components in the image i.e. the digits. Each pixel in a given component is connected to at least 4 or 8 other dark pixels in that component depending on the inputted parameter. The function also outputs the stats of each component, namely the leftmost and topmost coordinate, and the length and width of the bounding rectangle. We use these stats to compute the bounding squares of the digits and from there we isolate the labelled digit in the top left corner. This is the extent of the preprocessing, we use the newly processed image matrices as features in all of the learning algorithms. See Figure 1 for examples of the step by step process.



Original | Thresholding | Connected Components | Largest Bounding Square

Figure 1: Steps of data preprocessing with original data (first column), binarized data using a threshold (second column), connected components of data ranked by size (third column), and resulting image with the largest connected component (fourth column).

III. Algorithms

We used the following machine learning algorithms to classify the dataset:

A. Linear SVC

This algorithm uses support vector machine classification with a linear kernel so that the time to train the model scales well with large dataset sizes, and produces a linear decision boundary. With multi-class data, the algorithm uses one-vs-the-rest scheme to do the identification.

B. Naive Bayes

This algorithm assumes every pair of features is independent and uses maximum a posteriori (MAP) estimation to classify examples. The conditional probabilities of the examples are assumed to be Gaussian and Multinomial.

C. Stochastic Gradient Descent

SGD implements regularized linear models with a penalty that shrinks model parameters and a decreasing learning rate based on the loss gradient of each example.

D. Adaptive Boosting

This is a meta classifier that fits an initial classifier on the data, then fits more copies of the classifier on the same data focusing on more and more difficult examples.

E. Random Forests

This is another meta classifier that fits many decision tree classifiers on different subsections of the data, then uses averaging to improve the accuracy and mitigate overfitting.

F. Feed-Forward Neural Network

A neural network combines layers of linear models modified by an activation function, to produce a complex nonlinear model that can approximate any decision boundary, given enough units inside each layer (neurons). The model works by forward propagating the data points through the network to obtain prediction probabilities, and then backpropagating the loss on the prediction probabilities through the network to update the parameters of each neuron. Hyperparameters of the model include the choice of activation function, the number of layers, the number of neurons in each layer, the number of training epoch, and the size of the batch and the step size if using batch stochastic gradient descent.

G. Convolutional Neural Network

A convolutional neural network (CNN) is similar to a neural network in that it combines layers of linear models to produce non-linear models, but differs in how neurons are arranged, its depth, and the different type of layers it contains. The CNN is able to make use of the position of objects and extract "features" because neurons are arranged in a 3-dimensional shape; during the forward pass, the convolutional layer convolves a filter over the input (takes the matrix multiplication) and as the filter is moved along the width and height of the input it produces a 2-d activation map, and so each filter will have its own 2-d activation map. Next, the pooling layer reduces the spatial size of the representation, and hence reduces the amount of parameters and overfitting. Finally, the fully-connected layers perform the classification.

IV. Methodology

To create the models for the linearSVC, naive bayes, SGD, adaptive boosting, and random forests learning algorithms, we used python's sklearn library. Furthermore, all of these models and the feed forward NN and CNN models were trained using the preprocessed training dataset of 50k data points. We chose in most cases to do a 75/25 training and validation set split using the supplied 50k training data as it is industry standard. For the first CNN model, we used a 90/10 split because it produced slightly better accuracy on the kaggle test set.

We started by training the first 5 models with the sklearn default parameters to gauge their classification accuracy, if any model had 85% accuracy or above we would look at optimization techniques and hyper parameter tuning. None of these models produced satisfactory results, thus we focused our attention on neural network classifiers.

For the feed-forward neural network, we attempted to create the simplest neural network possible, following the architecture of Britz's post on his blog WildML (reference 2). We used one hidden layer for speed, with either 5, 10, or 50 neurons, and tried activation functions of either hyperbolic tangent or sigmoid. The activation function of the outer layer is softmax and we used cross-entropy loss as our loss function. We trained the data by batches of either 100 or 200 data points, with step size 0.01 or 0.001 and number of epochs 10 or 20. The architecture of this neural net was designed to prioritize efficiency and simplicity over performance. The implementation that was coded from scratch in python takes only a minute to train the whole dataset.

For the convolution neural network, we attempted to recreate the architecture shown in Figure 2b of Ciregan's publication as it performed with an accuracy of 99.7% on the original MNIST (reference 3). We used it as the foundation to create 3 CNN distinct architectures, where the general structure was an input layer corresponding to the size of the 4D images, followed by a series of convolutional and pooling layers, then a flattening layer followed by a series of fully-connected/dense layers with varying degrees of dropout. We made iterative improvements by adding layers one at a time and cross referencing it with other models made for MNIST and chose to tune the hyper parameters: number of epochs, batch size, dropout rate, and optimizer type because these parameters did not have consistent, well established values in the CNN MNIST publications. Finally, we found that binarizing the preprocessed data points gave a slight accuracy performance for all sets of hyper parameters.

V. Results

A. *First 5 ML classifiers*

As was expected, the first 5 ML classifiers did not produce satisfactory accuracy scores, therefore we did no further work on them and focused on neural networks. Random Forest classifier did perform better than expected (82%+), however even an untuned, unoptimized CNN produced 90%+ accuracy so we decided CNNs gave us the best chance in the Kaggle competition.

Table 1: Accuracy of various classifiers.

Classifier	**Accuracy**
LinearSVC	0.3464
Naive Bayes (Gaussian)	0.14912
Naive Bayes (Multinomial)	0.46088
SGD	0.46816
Adaptive Boosting	0.47672
Random Forest	0.82936

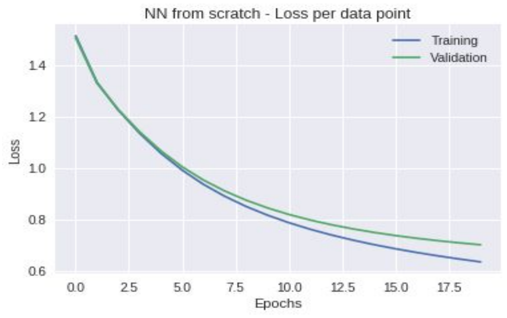
See the submitted python notebook for the above algorithms' confusion matrices.

B. *Feed-Forward Neural Network*

We achieved a maximum validation accuracy of 79.24% by using sigmoid as the hidden layer activation function, 50 neurons in the hidden layer, a batch size of 200, a step size of 0.001, and 20 epochs. See Table A in Appendix for full results of hyper parameter tuning.

The most important hyper parameters were the number of neurons and the step size, changing them produced dramatically different results. Because we prioritized efficiency and simplicity over performance, the neural network from scratch did not compete with the CNN performance. A more complex multilayered architecture would produce better results but the time to train the whole dataset would get impractically long. Figure 2a and b show the best tuned neural net error loss and accuracy progression over 20 epochs.

(a)



(b)

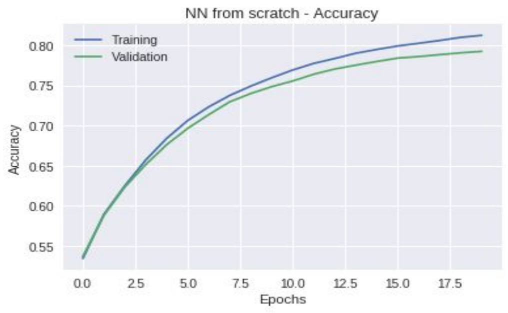


Figure 2: (a) Training and validation loss per data point per epoch. (b) Training and validation accuracy per epoch.

C. Convolutional Neural Network

We achieved a maximum kaggle test accuracy of 94.733% using CNN model 1, batch size of 256, 20 epochs, adam type optimizer, 0.25 dropout rate and 256 dense value. See Table 2 in Appendix for full results of hyper parameter tuning.

After doing hyper parameter tuning on each of the three CNN models, their accuracy performance all converged to around 95%.

Table 2: Accuracy of CNN

CNN model	**Validation Accuracy**	**Error Loss**
1	0.9472	0.2243
2	0.9506	0.2067
3	0.9408	0.2479

The most important hyper parameters were dropout rate and number of epochs, however there weren't any parameters that produced dramatically different results when varied. The effect of tuning hyperparameters contributed at most 2% of the accuracy measure.

Figure 3 shows the CNN model 1 accuracy prediction, see the Appendix for more plots on the other CNN models.

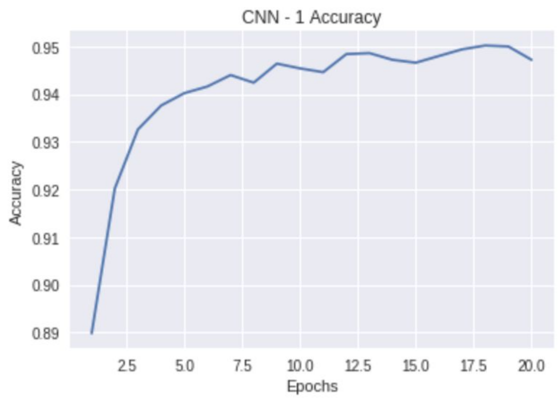
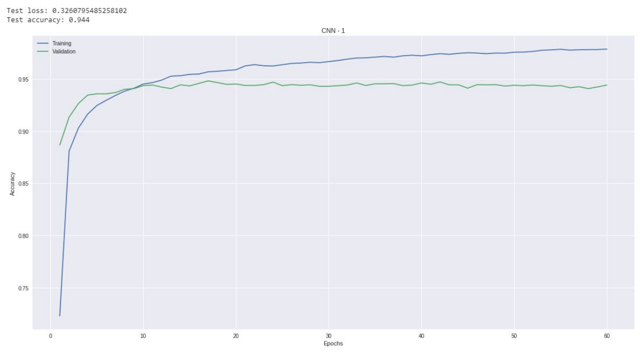


Figure 3: Accuracy vs epoch for CNN model 1.

We observed that increasing the number of epochs passed a certain critical point (25) negatively contributed to the accuracy performance. Figure 4 shows that when the number of epochs is 60, the train accuracy overfits while the valid accuracy slowly worsens.

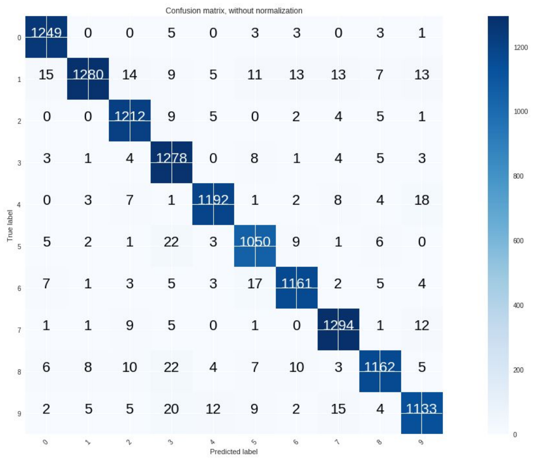


```
Test loss: 0.3260795485258102
Test accuracy: 0.944
```

Figure 3: Training (blue) and validation (green) accuracy vs epoch for CNN model 1.

Figure 4a and b show the heat map of the CNN model 1 confusion matrix. The digits "1" and "8" proved the hardest to classify, but not by very much.

(a)



(b)

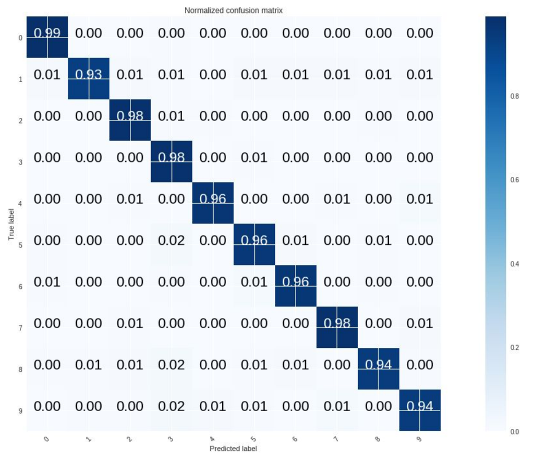


Figure 4: Unnormalized (a) and normalized (b) confusion matrices for model 1.

VI. Discussion

In analyzing the performance of our CNN models, we noticed that when fully optimized and with tuned parameters all 3 models produce very similar prediction accuracy of just under 95%. No matter what changed or tuned in the architectures, we were never able to cross the 95% barrier. When we looked at the validation set misclassifications, we found two different types of errors. Error type 1 is when the labelled digit is very ambiguous and the CNN mis-classifies it with a very similar looking digit. This type of error can be reduced with diminishing returns using more and more complex neural network architectures. Error type 2 is when the bounding box heuristic is wrong and the preprocessor chooses the wrong digit to keep, no CNN architecture can correct this error. We concluded that misclassifications of error type 2 was the bottleneck of our Kaggle performance. With perfect preprocessing, we suspect our CNN models would perform above 98%, however with the preprocessing design that we chose, there would always be at least 3-4% misclassification on any dataset.

In sum, the high-point of our methodology is that each of our CNN models gave near optimal accuracy given that they were constrained by type 2 errors on top of being space and time efficient. The low-point is the preprocessing bottleneck that stopped our algorithms from performing at 98-99% accuracy. If we were to to further work on this kaggle project, we would consider different ways to process the data in order to omit type 2 errors. An area of exploration could be to design a neural net to identify the orientation of the digits and correct rotated digits in order for the bounding box heuristic to produce better results.

VII. Statement of Contributions

Amir: coded and trained all algorithms, coded the final submitted notebook, designed the CNN models, did CNN data analysis
Vince: wrote and prepared written report, designed and implemented feature extraction/preprocessing, did CNN error analysis, did CNN parameter tuning
Lili: designed and implemented feed forward neural network from scratch, did data analysis and parameter tuning of scratch neural network

We hereby state that all the work presented in this report is that of the authors.

VIII. References

[1] "What is the class of this image ?" [Blog post] Retrieved from *http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html*

[2] "Implementing a Neural Network from Scratch in Python – An Introduction." [Blog post] Retrieved from *http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/*

[3] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification." Computer vision and pattern recognition (CVPR), 2012 IEEE conference on. IEEE, 2012. Retrieved from *http://ieeexplore.ieee.org/abstract/document/6248110/*

[4] *https://github.com/keras-team/keras/tree/master/examples*

Appendix

Table A: Neural network from scratch hyperparameter tuning using 37.5k train set, 12.5k valid set. Step size of 0.1 not included because validation accuracy is too low.

Activation function	Batch size	Step size	Neurons in hidden layer	N epoch	Validation accuracy (%)	Error loss per data point
tanh	100	0.01	5	10	45.23	1.582
tanh	100	0.01	5	20	47.91	1.553
tanh	100	0.01	10	10	57.16	1.249
tanh	100	0.01	10	20	56.80	1.256
tanh	100	0.01	50	10	68.66	0.9532
tanh	100	0.01	50	20	69.70	0.8772
tanh	100	0.001	5	10	53.10	1.381
tanh	100	0.001	5	20	53.54	1.337
tanh	100	0.001	10	10	63.70	1.057
tanh	100	0.001	10	20	64.02	1.011
tanh	100	0.001	50	10	79.23	0.5424
tanh	100	0.001	50	20	79.02	0.4463
tanh	200	0.01	5	10	35.66	1.740
tanh	200	0.01	5	20	37.26	1.690
tanh	200	0.01	10	10	53.50	1.415
tanh	200	0.01	10	20	55.38	1.328
tanh	200	0.001	5	10	47.15	1.544
tanh	200	0.001	5	20	49.29	1.468
tanh	200	0.001	10	10	63.42	1.061
tanh	200	0.001	10	20	64.63	0.9757
sigmoid	100	0.01	5	10	51.41	1.406
sigmoid	100	0.01	5	20	51.27	1.390
sigmoid	100	0.01	10	10	62.88	1.113
sigmoid	100	0.01	10	20	62.84	1.069

sigmoid	100	0.01	50	10	78.46	0.4986
sigmoid	100	0.01	50	20	77.50	0.4170
sigmoid	100	0.001	5	10	51.88	1.450
sigmoid	100	0.001	5	20	53.33	1.396
sigmoid	100	0.001	10	10	60.98	1.202
sigmoid	100	0.001	10	20	63.18	1.104
sigmoid	100	0.001	50	10	74.83	0.8146
sigmoid	100	0.001	50	20	79.14	0.6352
sigmoid	200	0.01	5	10	47.89	1.536
sigmoid	200	0.01	5	20	51.51	1.410
sigmoid	200	0.01	10	10	59.55	1.179
sigmoid	200	0.01	10	20	62.78	1.033
sigmoid	200	0.01	50	10	75.89	0.6123
sigmoid	200	0.01	50	20	75.64	0.5564
sigmoid	200	0.001	5	10	51.87	1.452
sigmoid	200	0.001	5	20	53.38	1.389
sigmoid	200	0.001	10	10	60.91	1.204
sigmoid	200	0.001	10	20	63.06	1.107
sigmoid	200	0.001	50	10	74.85	0.8159
sigmoid	200	0.001	50	20	79.24	0.6343

Table B: CNN hyperparameter tuning using 10k train set, 2k valid set.

Batch size	N epochs	optimizer	Dropout rate	Dense value	Validation accuracy (%)	Error loss
128	20	adadelta	0.2	128	91.6	0.368
128	20	adam	0.2	128	92.5	0.35
128	20	adadelta	0.25	128	91	0.3874

128	20	adam	0.25	128	91.15	0.3777
128	20	adadelta	0.30	128	92.5	0.3339
128	20	adam	0.30	128	91.9	0.3532
128	25	adadelta	0.2	128	90.95	0.413
128	25	adam	0.2	128	91.2	0.4302
128	25	adadelta	0.25	128	91.75	0.3402
128	25	adam	0.25	128	92.75	0.3306
128	25	adadelta	0.30	128	91.95	0.3691
128	25	adam	0.30	128	91.6	0.3887
256	20	adadelta	0.2	256	90.95	0.3572
256	20	adam	0.2	256	91.95	0.4011
256	20	adadelta	0.25	256	92.1	0.3539
256	20	adam	0.25	256	92.6	0.2971
256	20	adadelta	0.30	256	91.7	0.3215
256	20	adam	0.30	256	91.95	0.3759
256	25	adadelta	0.2	256	90.4	0.3479
256	25	adam	0.2	256	91.55	0.3915
256	25	adadelta	0.25	256	92.4	0.3183
256	25	adam	0.25	256	92.2	0.3509
256	25	adadelta	0.30	256	92.65	0.3175
256	25	adam	0.30	256	92.9	0.298

Figure A: Examples of type 1 errors.

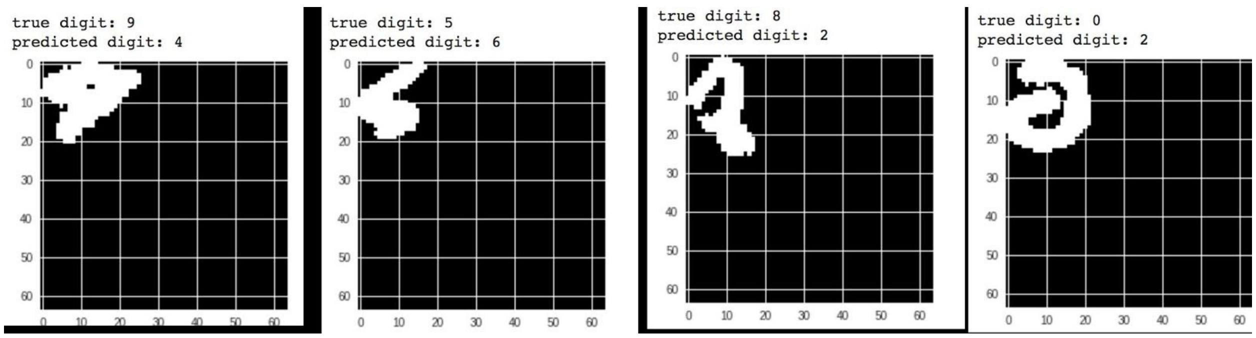


Figure B: Examples of type 2 errors.

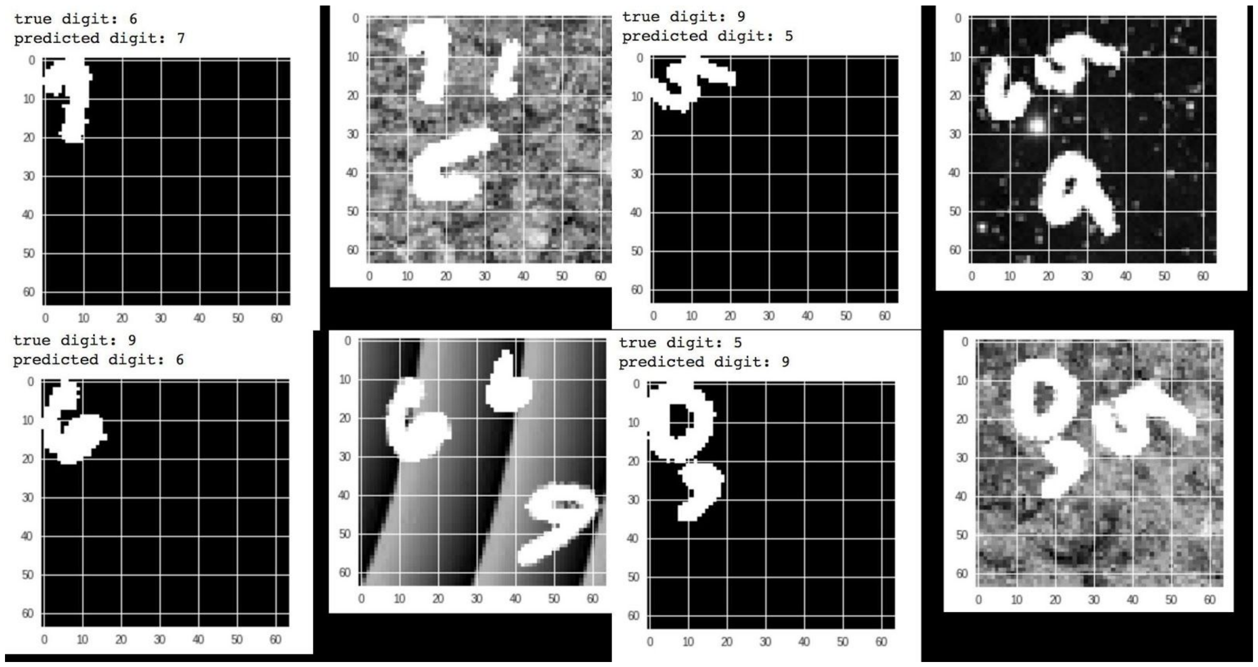


Figure C: CNN model 2 accuracy progression

```
Test loss: 0.20674980012893676
Test accuracy: 0.9506399999809265
```

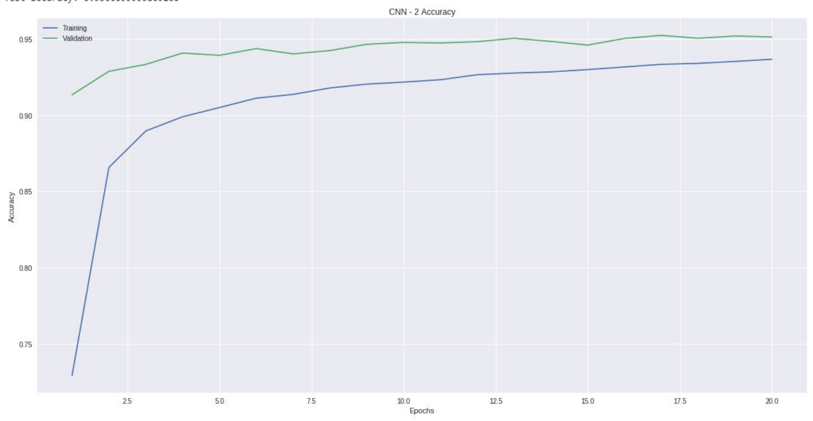


Figure D: CNN model 2 error loss progression

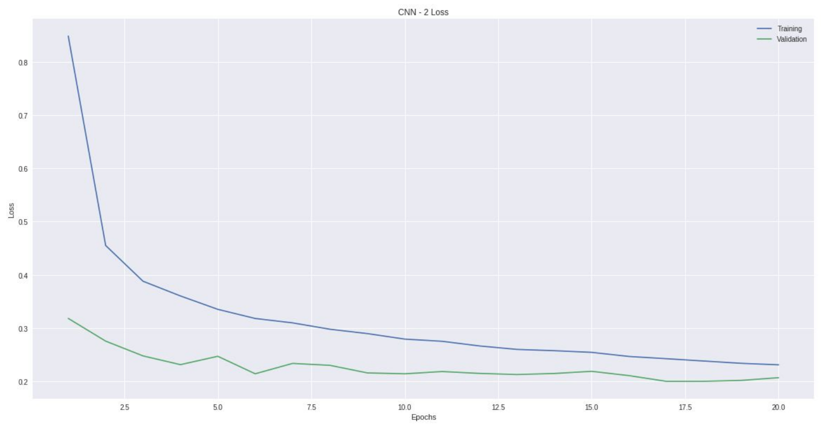


Figure E: CNN model 3 accuracy progression

```
Test loss: 0.247911856443882
Test accuracy: 0.940879999961853
```

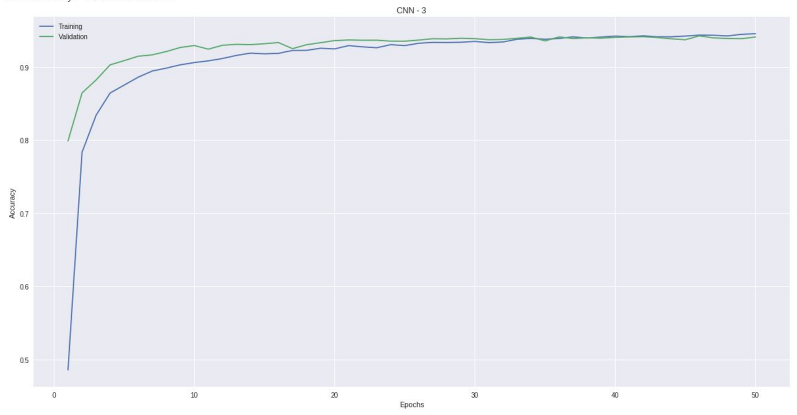
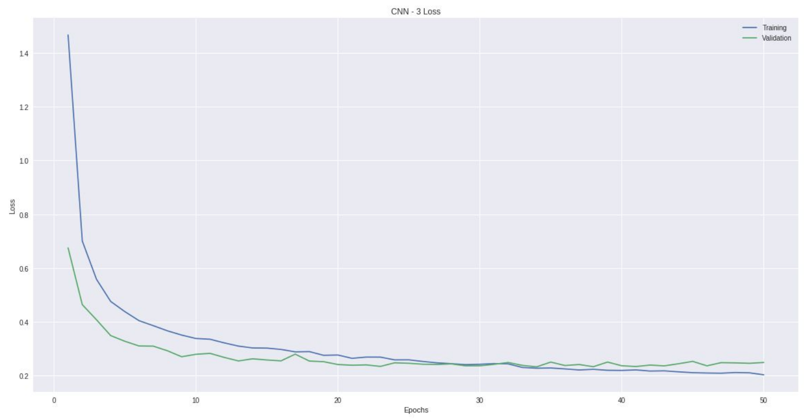


Figure F: CNN model 3 error loss progression


CNN - 3 Loss
Training
Validation
Loss
Epochs