2023

EEL6814: Project 1

BY ZAIN NASRULLAH AND JOSEPH MADDEN

IEEE

Abstract—**This paper discusses two deep learning classifiers implemented on the Kuzushiji-MNIST dataset. The first is a multilayered perceptron classifier and the second is a convolutional neural network classifier. Techniques regarding hyperparameter tuning are discussed and further areas of improvement.**

Keywords—machine learning, deep learning, Multilayered Perceptron, Convolutional Neural Network, image recognition

I. Introduction

Deep learning classifiers are the forefront of image classification technologies. Their ability to generalize to a wide range of different problem sets with relatively high accuracy makes them unparalleled [1-3]. In this paper, two deep learning classifiers will be trained on the Kuzushiji-MNIST dataset. The first model will be an MLP classifier, and the second model will be a convolutional neural network.

A. *Dataset Provided*

The provided source data for this project was the Kuzushiji-MNIST dataset (hereby referred to as KMNIST). The KMNIST dataset, similar to the MNIST dataset, contains 70,000 images, each in 28x28 grayscale. There are ten classes, each corresponding to one hiragana character.

The dataset is split into 60,000 training images and 10,000 test images. The GitHub page provides the dataset in both MNIST format and NumPy format. The NumPy format will be utilized for both models. The dataset is available for review at the following link: https://github.com/rois-codh/kmnist.

The samples from both sets are normalized to values between zero and 255. In the case of the CNN, the samples are additionally reshaped into 28x28 matrices before being passed in as input.

II. MLP Classifier

The first network used was a regular Multi-Layer Perceptron, or an MLP. This consists of first flattening out the images into long vectors, which can then be passed into the first Densely Connected Layer. I used different MLP architectures (such as using Dropout for some MLP's, and for other MLP's adding an extra hidden layer to see if this would make a difference in the learning of the MLP.

A. *Methods*

1) *Design*

As mentioned above, the design of the MLP included an initial Flattening layer, Densely connected layers (which was varied as a form of hyperparameter tuning), and then a final output layer that had the same number of outputs as number of classes (10 classes in general for the K-MNIST dataset), using a "softmax" activation function which normalized the outputs of the MLP network, making sure they summed up to 1, converting them into probabilities of the input belonging to 1 of those 10 classes.

Different hyper-parameters considered for the MLP were:

· Number of Hidden Layers (either one hidden layer of 128 neurons, or two hidden layers, the first one being 128 neurons, and the second one being 64 neurons), both ending in an output layer of 10 neurons and a SoftMax activation function

· Different Learning Rates using a Nadam (Nesterov's ADAM) optimizer

· Different Batch Sizes, or among the thousands of inputs, in what size groups are they passed in to train the Network in intervals.

One consideration taken into account for the training of the MLP was when to stop training it- for if it kept on training for too long, without any improvements in accuracy or loss, then it would end up tuning its weights to perfectly learn and memorize the input data, overfitting it, without the ability to learn and generalize to new, unseen data. For this, I defined a call back function, which made it so that if the model accuracy was greater than 0.995 (99.5 % accuracy), than the network would stop training.

Another way in which over-fitting was mitigated was adding DROPOUT layers, with a probability of 0.2 for the Dropout- this means that there's a 20% chance that the inputs to the neurons of that layer are set to 20%- this varies the training of the layer and makes it so that the model doesn't over-generalize, by shutting off parts of the network at certain, random times.

The Deep Learning architecture for MLP's that worked best for me was to choose a single hidden layer of about 400 neurons, giving my a final training accuracy of 97% and a Validation accuracy of 96%. Because the training and validation accuracies are so close to each other, this lets us know that the model generalizes well on new, unseen data (the validation dataset), and doesn't overfit.

The reason 400 neurons was chosen, is because, according to Sheela, K. G., & Deepa, one popular methodology for choosing the number of neurons in a layer, as found by Trenn, which emphasizes scalability, simplicity, and adaptivity, is to use the following formula for a single hidden layer: Nh = n + no / 2, where n is the number of inputs and no is the number of outputs [6]. Thus, the number of inputs are 28*28 = 784, the number of outputs is 10, to 784 + 10 / 2 = 397, which I rounded up to 400. This proved to give excellent numbers of, as repeated above, a training accuracy of 97% and a validation accuracy of 96%, after just 5 epochs of training. On the test data set, this yielded a test accuracy of 90%, which meant that the model somewhat overfitted on the training and validation data-set, but not by a lot, proving that it did learn significantly well.

2) *Search Methodology*

To choose the hyper-parameters, the following methodology was used:

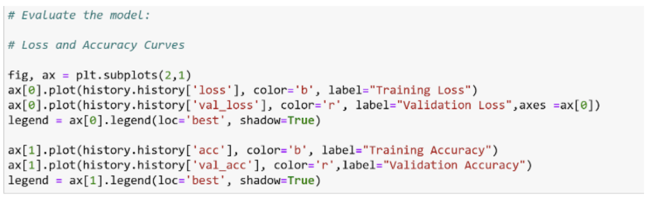
1. Make different models, of different architectures- for an MLP, different architectures can be made by varying up the number of neurons in a layer, or varying the number of layers
2. For each model, train the model off of different learning rates, and different batch sizes.
    a. The learning rates considered were: [0.0001, 0.001, 0.01, 0.1, 1]
    b. And the different batch sizes considered were: [32, 64, 128, 256]
3. The best combinations of every model will be pulled, based on the following metric: Whichever

combination results in a model, that when trained, the validation accuracy and the training accuracies are very close to each other.

Among the models chosen for the 4 main architectures, it is seen that a MLP of a single hidden layer, with 400 neurons in the single hidden layer, trains the best.

B. *Results*

The results of the training of this particular network, called "model_working" in the code, are:



```
# Evaluate the model:

# Loss and Accuracy Curves

fig, ax = plt.subplots(2,1)
ax[0].plot(history.history['loss'], color='b', label="Training Loss")
ax[0].plot(history.history['val_loss'], color='r', label="Validation Loss",axes =ax[0])
legend = ax[0].legend(loc='best', shadow=True)

ax[1].plot(history.history['acc'], color='b', label="Training Accuracy")
ax[1].plot(history.history['val_acc'], color='r',label="Validation Accuracy")
legend = ax[1].legend(loc='best', shadow=True)
```

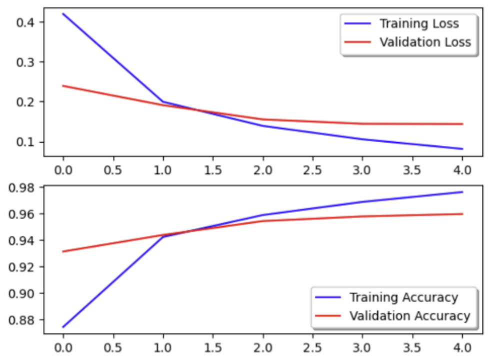
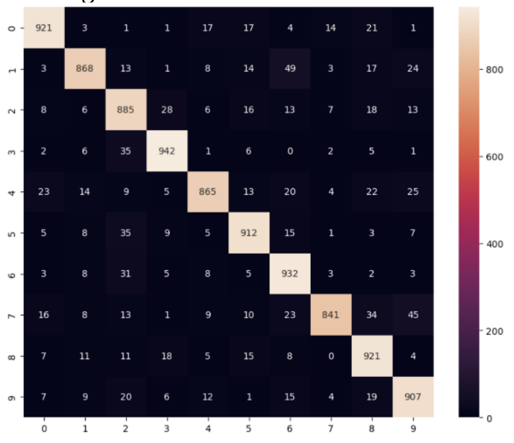


Figure 1: Graphs of Accuracy for Training and Validation sets over the 5 epochs of training

As seen from *Figure 1*, the training and validation loss and accuracies are very close to each other, giving us evidence this MLP will not overfit and generalizes well to new, unseen data.

This also shows that learning stopped at the best point to optimize generalization, as in these graphs, we can see that the training and validation accuracies reach their highest point, and they are just about to diverge away from each other even further before learning is stopped at the 5th epoch.

The resulting confusion matrix of this model is as following:



	0	1	2	3	4	5	6	7	8	9
0	921	3	1	1	17	17	4	14	21	1
1	3	868	13	1	8	14	49	3	17	24
2	8	6	885	28	6	16	13	7	18	13
3	2	6	35	942	1	6	0	2	5	1
4	23	14	9	5	865	13	20	4	22	25
5	5	8	35	9	5	912	15	1	3	7
6	3	8	31	5	8	5	932	3	2	3
7	16	8	13	1	9	10	23	841	34	45
8	7	11	11	18	5	15	8	0	921	4
9	7	9	20	6	12	1	15	4	19	907

Figure 2: Confusion Matrix for the model chosen off the best results from the Hyperparameter search

As seen above, the confusion matrix shows remarkable accuracy, meaning that the majority of images were correctly predicted, with the prediction of a class the model made about an image indeed matched the label of that image.

III. Convolutional Neural Network

CNNs are the preferred deep learning technique for image classification. The reason for this is their ability to utilize locality in image data. CNNs perform very well on the original MNIST dataset, so it only makes sense to apply one to the KMNIST dataset. The following sections will detail the design, implementation, and results of using a CNN classifier.

A. *Methods*

1) *Design*

As with any machine learning model, understanding the hyperparameters of the model is essential. The following are the standard hyperparameters to be considered for any CNN.

- The kernel size for convolutional layers
- The number of kernels
- The number of convolutional layers
- The length of strides
- The pooling size

In addition, there are the hyperparameters that exist for all neural networks. These hyperparameters are:

- Learning Rate
- Number of epochs / stopping criteria
- Batch size
- Activation function
- The number of hidden layers
- The width of hidden layers
- Weight initialization

The momentum of the stochastic gradient descent optimizer will also be considered for the model.

Out of consideration for the time and scale of this project, it would not be possible to conduct searches for all these hyperparameters. Therefore, some hyperparameters will be held constant and not considered for experimentation. These hyperparameters will instead be selected based on results from prior academic reading and best practices.

The activation function for the hidden layers will be the rectified linear unit function. The rectified linear unit is the most common choice for a wide range of deep learning applications. It has demonstrably good results for function approximation while also improving training speedup [4-5].

The kernel initialization for the convolutional layers and the hidden layers will be a uniform distribution with a range between -α and α where:

$$\alpha = \sqrt{(6/n)}$$

The *n* in this case is the number of input units to the weight tensor. This is to avoid making any assumptions on the dataset and provide a fair starting point for the training models to improve on.

2) *Search Methodology*

The hyperparameter tuning for this model employed randomized searches. Random values for the hyperparameters were used to train ten models. The model with the highest accuracy on the test set was the one saved into memory. This approach was repeated twenty times to create a tournament style search that revealed the best results.

There are two reasons that the search was conducted this way. The first was to prevent overfitting. The stopping criteria for the models was a minimum improvement of -0.5 for the cross-entropy loss each epoch or a maximum of five epochs (whichever comes first). This in conjunction with tournament style elimination ensures that the final twenty models quickly learned the data but did not overfit the dataset.

The second reason is that there were simply too many hyperparameters to train in an exhaustive manner. This search method made it possible to get a profile of what the best model for the hyperparameters is, while consuming less time and resources. Assuming a unique model each time, over 200 models were tested.

a) Learning Parameters

The learning parameters: learning rate and momentum, were restricted to discrete sets with uniform distributions over those sets. The set for the learning parameter was: [0.0001, 0.001, 0.01, 0.1, 1]. The set for the momentum parameters was: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9].

b) Hidden Layer Parameters

The hidden layer parameters for the model were the number of hidden layers and the number of perceptrons in each layer. For ease of testing, the number of perceptrons in each layer were kept uniform and not varied. The number of layers was a discrete uniform distribution between one and three inclusive.

The number of perceptrons was a discrete uniform distribution with the integers between five and 50 inclusive. The assumption in this case is that the minimum number of perceptrons should be greater than four to prevent underfitting.

c) Convolutional Parameters

The three convolutional hyperparameters considered were the number of filters learned per layer, the number of convolutional layers, and the batch size of the model. For ease of training, the number of filters learned per convolutional layer were kept uniform between layers. The number of filters was a discrete uniform distribution consisting of [16, 32, 64, 128, 256]. The number of convolutional layers was a discrete uniform distribution consisting of the integers one, two, and three inclusive. The batch size for the model was a discrete uniform distribution over the set [16, 32, 64, 128, 256].

d) Implementation

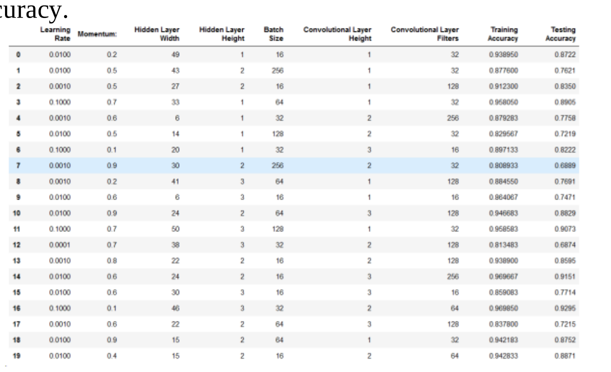
The various CNN models designed for the experimentation phase were all implemented in Python using the Keras packages. To avoid over-automating the hyperparameter tuning, for-loops were introduced and 20 models were stored in memory each round.

This results in a large memory footprint for the training phase at the end of each round. Therefore, the discarded models are deleted from memory at the end of each round. It is also recommended that the training code only be utilized on hardware with a sufficient amount of memory and computational resources. In future implementations, it is recommended that users employ the provided tuning libraries like KerasTuner.

The structure of each CNN will consist of, at minimum, one convolutional layer, one 2D pooling layer, a flattening layer, a dense MLP layer, and a SoftMax regression classification layer on the output.

B. *Results*

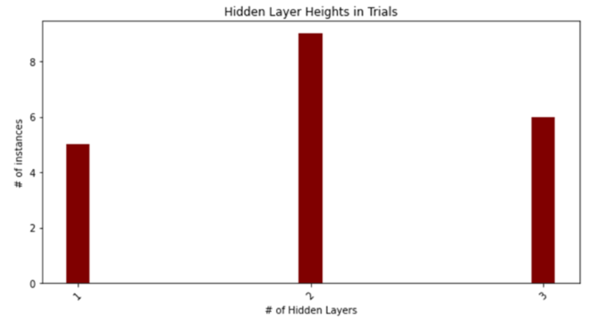
The table below shows the top twenty models from the training phase of the program. Their hyperparameter values are listed along with their training accuracy and their testing accuracy.



	Learning Rate	Momentum:	Hidden Layer Width	Hidden Layer Height	Batch Size	Convolutional Layer Height	Convolutional Layer Filters	Training Accuracy	Testing Accuracy
0	0.0100	0.2	49	1	16	1	32	0.938950	0.8722
1	0.0100	0.5	43	2	256	1	32	0.877600	0.7621
2	0.0010	0.5	27	2	16	1	128	0.912300	0.8350
3	0.1000	0.7	33	1	64	1	32	0.958050	0.8905
4	0.0010	0.6	6	1	32	2	256	0.879283	0.7758
5	0.0100	0.5	14	1	128	2	32	0.829567	0.7219
6	0.1000	0.1	20	1	32	3	16	0.897133	0.8222
7	0.0010	0.9	30	2	256	2	32	0.808933	0.6889
8	0.0010	0.2	41	3	64	1	128	0.884550	0.7691
9	0.0100	0.6	6	3	16	1	16	0.864067	0.7471
10	0.0100	0.9	24	2	64	3	128	0.946683	0.8829
11	0.1000	0.7	50	3	128	1	32	0.958583	0.9073
12	0.0001	0.7	38	3	32	2	128	0.813483	0.6874
13	0.0010	0.8	22	2	16	2	128	0.938900	0.8595
14	0.0100	0.6	24	2	16	3	256	0.969667	0.9151
15	0.0100	0.6	30	3	16	3	16	0.859083	0.7714
16	0.1000	0.1	46	3	32	2	64	0.969850	0.9295
17	0.0010	0.6	22	2	64	3	128	0.837800	0.7215
18	0.0100	0.9	15	2	64	1	32	0.942183	0.8752
19	0.0100	0.4	15	2	16	2	64	0.942833	0.8871

Table 1: Results of tournament style elimination for the best hyperparameter results.

Using the tabulated results, the final parameters for convolutional neural network classifiers were taken from the statistical mode of each parameter. For example, the number of hidden layers in the final model was selected as two based on the data from graph X. The mode was chosen as the results are discrete and prone to outliers.



Graph X: Bar chart representing the winning parameters from the tournament style elimination.

A) Analysis

With the results from the previous section, it became clear what the ideal hyperparameters for the model should be. In the case of multiple modes, the value that created the less complex model (typically the smallest value) was preferred. This was another measure to prevent overfitting.

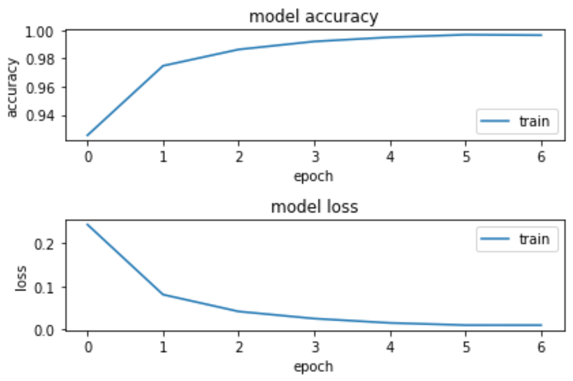
For the learning parameters, the learning rate should be approximately 0.01 with a momentum of 0.6.

For the hidden layer parameters, two layers are preferred with approximately six perceptron's.

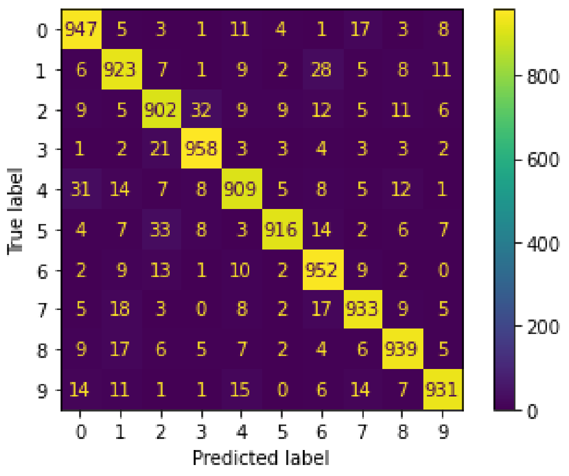
For the convolutional layer parameters, the mode is one layer with 32 filters and a batch size of 16.

B) Final Model

Utilizing the results from the previous section, a new model was created to test the hyperparameters. The new model was given ten epochs with a minimum required improvement of -0.1 in the loss function at least every two epochs. The stopping criteria was reached in seven epochs. The final accuracy reported from the training phase was 98%



Evaluating the test data on the model returned an accuracy score of 93%.



	0	1	2	3	4	5	6	7	8	9
0	947	5	3	1	11	4	1	17	3	8
1	6	923	7	1	9	2	28	5	8	11
2	9	5	902	32	9	9	12	5	11	6
3	1	2	21	958	3	3	4	3	3	2
4	31	14	7	8	909	5	8	5	12	1
5	4	7	33	8	3	916	14	2	6	7
6	2	9	13	1	10	2	952	9	2	0
7	5	18	3	0	8	2	17	933	9	5
8	9	17	6	5	7	2	4	6	939	5
9	14	11	1	1	15	0	6	14	7	931

IV. Discussion

As we can see, the two architectures considered in this paper, a Multi-Layer Perceptron (MLP), and a Convolutional Neural Network (CNN), yield different results when trying to classify the K-MNIST dataset.

This is because the MLP only consists of densely connected layers, while the CNN's input layer consists of convolutional layers, with a kernel (filter) that passes over the inputs, convolving the weights of the kernel with the values of the inputs. This method allows the MLP to extract features corresponding to locality.

In the case of the KMNIST dataset, the structure likely learns the edges between the characters and black space to classify the characters. ~~After several convolutional layers followed by pooling and subsampling, this is passed to regular MLP's, culminating in a SoftMax layer that converts the results to probabilities of the input corresponding to one of the 10 classes.~~

V. Conclusion

Both the MLP model and the CNN model were able to achieve above 90% accuracy scores for the validation and test sets. The CNN manages to slightly outperform the MLP because of its greater complexity and ability utilize features the MLP cannot (locality).

For further studies into convolutional network classifiers, this team recommends either using newer optimizers like the Adam algorithm or creating an ensemble learner using CNNs.

VI. References

[1] Z. -Q. Zhao, P. Zheng, S. -T. Xu and X. Wu, "Object Detection With Deep Learning: A Review," in IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 11, pp. 3212-3232, Nov. 2019, doi: 10.1109/TNNLS.2018.2876865.
[2] Zeebaree, Subhi & Haji, Lailan & Rashid, Imad & Zebari, Rizgar & Ahmed, Omar & Jacksi, Karwan & Shukur, Hanan. (2020). Multicomputer Multicore System Influence on Maximum Multi-Processes Execution Time. Test Engineering and Management. 83. 14921 - 14931.
[3] Obaid, Kavi & Zeebaree, Subhi & Ahmed, Omar. (2020). Deep Learning Models Based on Image Classification: A Review. International Journal of Social Science and Business. 4. 75-81. 10.5281/zenodo.4108433.
[4] A. F. Agarap, Deep Learning using Rectified Linear Units (ReLU). 2019.
[5] K. Hara, D. Saito and H. Shouno, "Analysis of function of rectified linear unit used in deep learning," 2015 International Joint Conference on Neural Networks (IJCNN), Killarney, Ireland, 2015, pp. 1-8, doi: 10.1109/IJCNN.2015.7280578.
[6] Sheela, K. G., & Deepa, S. N. (2013, June 20). Review on methods to fix number of hidden neurons in neural networks. Mathematical Problems in Engineering. https://www.hindawi.com/journals/mpe/2013/425740/