Comparative Performance Analysis of Multi-Layer Perceptrons and Convolutional Neural Networks on K-MNIST Handwritten Character Classification

James Dooley
Department of Electrical & Computer Engineering
University of Florida
Gainesville, FL, USA
james.dooley@ufl.edu

Tre' R. Jeter
Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL, USA
t.jeter@ufl.edu

***Abstract* – Handwritten character recognition is a fundamental problem in the field of computer vision, with applications ranging from optical character recognition to autonomous navigation systems. In this work, we conduct an extensive and methodologically rigorous comparison between Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) in the context of classifying the Kuzushiji-MNIST (K-MNIST) dataset. Our goal is to shed light on the strengths and weaknesses of these two distinct neural network architectures in handling this unique dataset. We present a comprehensive framework for training, evaluation, and analysis, encompassing preprocessing, model design, training strategies, and performance metrics. Our results provide critical insights into the trade-offs between CNNs and MLPs. We analyze performance metrics such as accuracy and loss. To offer a comprehensive perspective on this comparative study, we also discuss the interpretability of the models, examining the ability of each architecture to provide insights into the classification decisions.**

***Index Terms*—Neural Networks, Multi-Layer Perceptron, Convolutional Neural Network.**

I. Introduction

In the ever-evolving field of computer vision, the ability to classify and recognize patterns in visual data is a paramount challenge. Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) are two of the most pivotal neural network architectures in this regard. CNNs, specialized in processing grid-like data, have revolutionized image classification tasks, while MLPs, more general-purpose, have found extensive utility in various domains. In this study, we focus on the critical role of these architectures in the context of classification tasks.

Specifically, the problem at hand centers on the classification of handwritten Japanese characters from the K-MNIST dataset. This dataset comprises 70,000 grayscale 28×28-pixel images, separated into 10 distinct classes, each representing a unique character. To tackle this classification challenge, we employ Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs). These two prominent deep learning architectures are leveraged to build classification models. The objective is to develop a robust classification model that can accurately assign these characters to their respective classes based on their visual patterns.

Addressing the challenge of classifying the handwritten Japanese characters in the K-MNIST dataset, we propose a comparative study. Specifically, we explore the efficacy of CNNs and MLPs. We design and implement classification models using these architectures to scrutinize their performance in classifying the dataset's characters. Through a meticulous comparative analysis of results, we aim to provide insights into the strengths and weaknesses of both approaches, shedding light on their respective suitability for this classification task.

Note: James handled the MLP experiments while Tre' handled the CNN experiments.

Organization. The rest of the report is organized as follows: Sections II and IV describe our methodology to carry out our experiments. Sections III and V explain our experimental results. Section VI is an open discussion about expectations and our comparative analysis outcomes of CNNs and MLPs in regards to our classification task. Lastly, Section VII concludes the report. The model architectures used for each experiment can also be seen in the Appendix.

II. Methodology for MLP

Phases and Objectives. The goal here is to find an architecture and a set of hyperparameters that can result in a network that can achieve the highest accuracy on the pre-divided training and test splits within the K-MNIST dataset. To accomplish this the MLP design will be broken into three distinct phases:

1) Manual Tweaking - The goal of this phase is to manually explore the hyperparameter space and get a general feel for the bounds of what makes a good performing network.
2) Hyperparameter Space Exploration - Using features derived from the first phase, a more formal exploration of the hyperparameter space will be conducted testing a variety of model depths and widths.
3) Reaching for the Stars - After phase two a brief analysis will be conducted to determine which model configurations and hyperparameters are not working and prune them from the feature space, after which a final set of trials will be conducted to try and reach peak accuracy on the provided test set.

Manual Tweaking. Brevity is the name of the game in phase one. This phase is manually exploring the feature space to find good candidates for hyper-parameters to test in subsequent phases, and get a feel for which hyper-parameters can be safely left alone. The MLP used here utilized:

1) A (28×28) input layer which is flattened.
2) 3 hidden layers composed of 512, 256, and 128 neurons respectively, with each neuron using the ReLU activation function.
3) A final softmax layer with 10 neurons.

In between each dense layer, was a dropout layer, with 50 percent dropout, and batch normalization layer. This model was trained for only 10 epochs, using the Adam optimizer, and categorical cross entropy for the loss function. After toying with this model over several iterations it became clear that dropout was necessary to have a model that was capable of generalization at all, and so it became standard in all future trials.

Hyperparameter Space Exploration. The goal of this phase is to investigate features looked at in phase one, and build out a more thorough hyperparameter search. Additionally, in an effort to reduce overfitting the model to the training data, these models will have the input data augmented, by having image rotation and Gaussian noise as possible model attributes. For this experiment six factors were chosen:

1) Learning Rate - The rate at which the weights are moved down the gradient of the error surface. Tested Values .001, .002, .003, .005, .006, .007, .008, .009, .01.
2) Hidden layer size - This is the size of the last hidden layer in the MLP. Every layer preceding the final layer doubles the number of neurons. There could have been another hyperparameter for controlling the width of all layers of the network, but it seemed like the experiment already had a lot of variables. Chosen levels - 128, 256, 512, 1024 Neurons.
3) Model Depth - How many hidden layers to be included in the model. Deeper networks have the capability of learning more complex features, but can take longer to train. chosen levels - 1, 2, 3, 4 Hidden layers
4) Batch Normalization - This feature determines whether there will be a batch normalization layer between every dense layer in the network. Levels - True, False.
5) Rotation Augment - This experiment will feature two types of data augmentation. Through manual tweaking it was discovered that much more than the bare minimum of rotation augmentation caused models to diverge, or train poorly. Levels in degrees of rotation - 0, .5, 1, 5
6) Gaussian Noise - A Gaussian noise that is applied to the input image. The level represents the sigma value for Gaussian distribution. Levels are - 0, .015, .03, .06

A full factorial run of this test design would require 5120 runs; too many for the scope of this project. Instead, 59 runs were executed, with randomly chosen levels for each feature. The idea being that in the aggregate, better performing hyperparameters would be over represented among the best performing models. This will allow for removing those levels which under perform, and adding additional levels to further explore the hyperparameter space. In this first round of exploratory testing, each model was given 20 epochs, and the results for highest accuracy on the training and test set were recorded for each model. Additionally, using a validation accuracy checkpoint, only the model weights with the highest accuracy on the test set are saved, in the event that over-fitting occurs and the model starts to perform worse on the test set.

batch_normalization	rotation_augment	gaussian_noise	Acc	Acc_test
True	0.000	0.015	0.973150	0.9428
True	0.000	0.030	0.964367	0.9415
True	0.000	0.060	0.970850	0.9390
True	0.005	0.000	0.945083	0.9369

Figure 1: Maximum Accuracy for each combination of Hyperparameters

Reaching for the Stars. The final phase of MLP model building uses a pared down set of hyperparameters from the previous phase in search of a model with the highest accuracy on the test set. Key take-aways from the previous phase were that deeper models with more neurons on the last layer performed better and Gaussian noise is always good. Because of that, in this phase model depth allowed is increased to 5 hidden layers, and the last hidden layer of the MLP is allowed to have up to 2048 neurons. Finally, training in this phase is allowed to continue for up to 55 epochs, allowing for much more time for a larger and deeper model to train.

III. Results for MLP

Manual Tweaking. In phase one the goal was to find good hyperparameters to test, and develop a testing pipeline to be used later in the more rigorous rounds of experimentation. That being said, the best model found during this phase is highlighted in Table II. 95% and 92.9% are respectable results for simply manually tweaking values. It became immediately clear that model depth and breadth were good targets for hyperparameter tuning, while dropout layers seemed to be a must-have in order to get a model that could generalize to the test set.

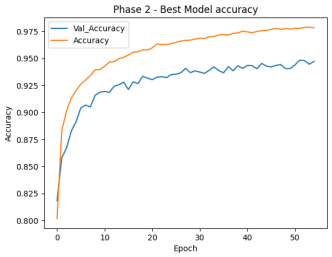
Hyperparameter Space Exploration. The factors and levels

hidden_layer_size	model_depth	Acc	Acc_test
512	4	0.973150	0.9428
1024	3	0.964367	0.9415
512	2	0.945083	0.9369
256	3	0.962767	0.9366

(a) Max Accuracy for Hidden Layer Size and Model Depth

model_depth	Acc	Acc_test
4	0.973150	0.9428
3	0.964367	0.9415
2	0.952017	0.9369
1	0.943100	0.9270

(b) Max Accuracy for each Model Depth.



(c) Test and Train Accuracy vs. Epochs.

```
[[974   0   1   0  10   0   3   4   7   1]
 [  2 920   3   0  13   1  28   9  18   6]
 [  9   3 875  51   8   5  15  11  14   9]
 [  4   0   7 979   0   1   3   4   2   0]
 [ 19   7   0   1 934   1  12   6  15   5]
 [  4   8  29   7   3 922  17   3   6   1]
 [  2   0  12   1   6   1 974   1   2   1]
 [  2   1   2   0   1   0   8 982   1   3]
 [ 10   5   3   3   0   0   3   1 975   0]
 [ 10   4   2   2  12   0  12  12   8 938]]
```

(d) Test Set Confusion Matrix.

Figure 2: Results for Phase 2.

Learning Rate	Hidden Layer Size	Model Depth	Batch Norm.	Rotation	Gaussian Noise
$[.001, .002, .003, .005, .006, .007, .008, .009, .01]$	$[128, 256, 512, 1024]$	$[1, 2, 3, 4]$	$[True, False]$	$[0, .5, 1, 5]$	$[0, .015, .03, .06]$

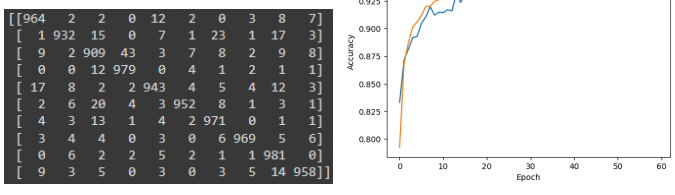
TABLE I: Tested Hyper-parameters for MLP - Phase 2.

Layer 1	Layer 2	Layer 3	Dropout	Train Acc	Test Acc
512	256	128	40%	95.48%	92.91%

TABLE II: Model design and accuracies for phase 1

used for this experiment are shown in Table I. 60 runs were executed with the hyperparameters used in each run being randomly selected. Figure 1 and Figure 2 both show the results for this experiment, aggregated across the various hyperparameters, but only displaying the 4 best performing experiment levels.Aggregate data for learning rate is not displayed here, as performance was nearly uniform across all learning rates. In Figure 1 it is noticeable that both batch normalization and Gaussian noise are heavily present in the best performing models, with only 1 model in the top 4 having no Gaussian noise. The inverse is true for using rotation as a way to augment the data, with the top models having no rotation augmentation. Figure 2a and Figure 2b both show how the model performance is affected by the depth and breadth of the MLP. Larger and deeper hidden layers showed better performance. With these results the hyperparameters utilized were altered for the final phase of testing. Mainly that there were to be at least 3 hidden layers in the MLP, up to 5, and 512 was the floor for number of neurons in the final hidden layer, and going up to 2048. Additionally, smaller values of rotation augmentation were introduced as selectable levels, and the option for 0 Gaussian noise was removed.

Reaching for the Stars. Using the modified hyperparameter



(a) Validation Set.

(b) Test Set.

Figure 3: Confusion Matrix and Accuracy Curves for Best Performing MLP.

set described above, a final set of runs was conducted. Table III shows the hyperparameters and accuracy performance of the best MLP model found over the course or this final experiment, while Figure 3 shows the final confusion matrix and accuracy curves. The model definition is shown in the Appendix. It is interesting to note the model with the best performance was ultimately one that was relatively shallow, but wider than those considered in the first and second phases of the experiment.

IV. Methodology for CNN

Setup. K-MNIST is already known to be split into a training set and a testing set. In experiments with validation metrics, the training set was further split to create a validation set. The training set undergoes an 80/20 split meaning 80% of the original training set remains for training, and the other 20% makes up the new validation set. Next each set of data (training, validation, and testing) is normalized by dividing each pixel by 255. This places each pixel in a 0-1 range.

Out-of-the-Box CNN[1]**.** An initial experiment employed an arbitrary benchmark CNN for the K-MNIST dataset. The first convolutional layer has 32 filters with a kernel size of 3×3 and a ReLU activation. The input size was the size of the images themselves: $(28 \times 28 \times 1)$. The second convolutional layer has 64 filters with a kernel size of 3×3 along with a ReLU activation function. These layers are followed by a max pooling layer, a dropout layer (0.25), a flattening layer, a hidden layer with 128 neurons and a ReLU activation. The last two layers are a dropout layer (0.5) and a hidden layer with 10 neurons and a softmax activation function. This model was trained with batch size of 128, a categorical cross entropy loss function, and an Adadelta optimizer for 12 epochs.

Modifications. To improve this model, we changed three key attributes. We changed the loss function to a sparse categorical cross entropy because our labels are already integers and it is inefficient to one-hot encode them using the categorical cross entropy loss function. Secondly, we change the optimizer from Adadelta to Adam. Adam is more widely used and takes into account bias correction early in the training process that leads to more stable convergences. Lastly, we added an early stopping criteria on the validation loss with a patience of 5.

Basic CNN. Moving on from the Out-of-the-Box CNN, we deploy a basic CNN with 5 layers to see how it would fair with the K-MNIST dataset. The first consists of a convolutional layer with 32 filters, kernel size of 3×3, ReLU activation function, and input the size of the images $28 \times 28 \times 1$. The second consists of a max pooling layer (2×2) followed by a flattening layer. The last two layers are dense layers: 128 neurons/ReLU activation function and 10 neurons/softmax activation function. Included is the Adam optimizer and sparse categorical cross entropy loss function. This basic CNN model was not expected to perform well because of its simplicity.

Increasing Depth of CNN. Developing the Basic CNN further, we increase it's depth with a goal of allowing the model to learn more about the data. The newer model includes 10 layers, double that of the original model. Specifically, the first layer is a convolutional layer with 32 filters, kernel size of 3×3, ReLU activation function, and an input shape of $28 \times 28 \times 1$. The second is a batch normalization layer followed

[1] https://github.com/rois-codh/kmnist/blob/master/benchmarks/kuzushiji_mnist_cnn.py

Learning Rate	Hidden Layers	Model Depth	Batch Norm.	Rotation	Gaussian Noise	Train Acc	Test Acc
[.03]	[2048]	[3]	[*True*]	[.5]	[.015]	[99.8%]	[95.57%]

TABLE III: Best MLP Model Parameters and Performance - Phase 3.

by a max pooling layer with pool size 2×2. These layers are repeated in the same order except the convolutional layer now has 64 filters. The second max pooling layer is followed by a flattening layer and hidden layer with 128 neurons with a ReLU activation function. The hidden layer is followed by a dropout layer (0.5) and an output layer with 10 neurons with a softmax activation function. To further improve the training process, a learning rate scheduler is implemented. For every 10 epochs, the learning rate is reduced from 0.001 to 0.0001 to 0.0001 for 30 epochs. An early stopping criteria is also in place with patience of 5 monitoring the validation loss.

Deeper CNN with K-Fold Cross Validation. Seeing how well the model improved, we adapted the same model architecture to a *K*-fold cross validation scenario. We implemented a 5-Fold cross validation and observed performance and loss of each fold.

Best Hyperparameters with K-Fold Cross Validation. The final experiment searches for the best hyperparameters while also implementing *K*-Fold Cross Validation. The hyperparameters in question are the learning rate and the dropout rate for the model. This experiment uses the same model and architecture of the deeper CNN described above. The only difference is fine-tuning the learning rate and dropout to get the optimal results. Table IV shows the combination of values tested during training.

Learning Rate	Dropout
$[0.1, 0.01, 0.001, 0.0001]$	$[0.3, 0.5, 0.7]$

TABLE IV: Tested Hyperparameters.

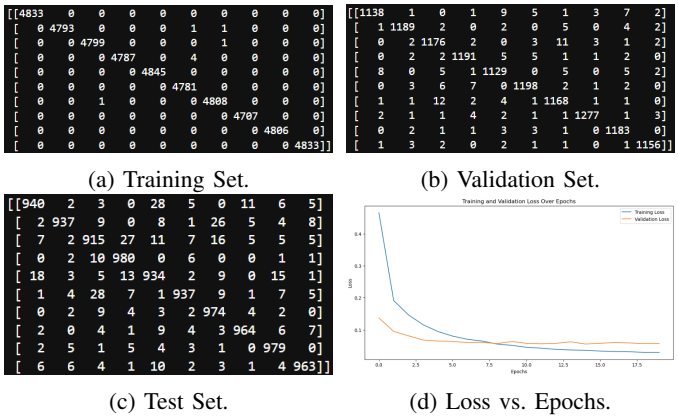
Each combination of hyperparameters was taken through the cross-validation phase meaning for every possible combination, a 5-Fold cross-validation was tested. Table VI in Section V shows the performance of each combination at each fold.

V. Results for CNN

Out-of-the-Box CNN. The benchmark reported 94.63% on the test set, however when verifying this ourselves with these same parameters, the test accuracy was around 55%. With our modifications, the final testing accuracy was 95.23%. Training and validation accuracies were 99.0% and 98.54%, respectively with early stopping occurring at epoch 20 out of 30. The confusion matrices for training, validation, and test sets can be seen below in Figure 4 along with the loss curves.

Basic CNN. After 5 epochs, training and validation accuracies were reported at 99.45% and 97.43%, respectively. The final testing accuracy was 92.96%. The resulting confusion matrices and loss curves can be viewed in Figure 5.

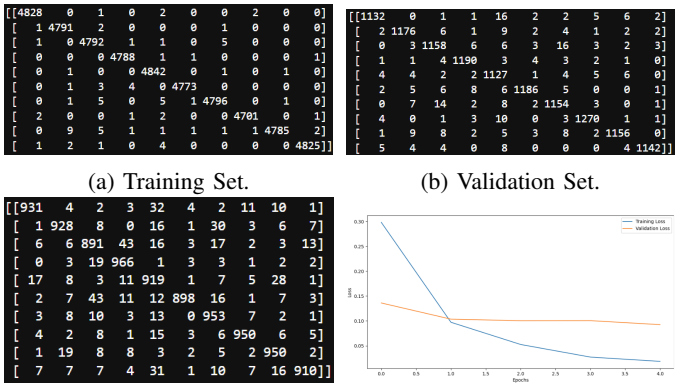
Increased Depth of CNN. With the improved model, training and validation accuracies increased to 99.5% and 98.9%,



(a) Training Set. (b) Validation Set.

(c) Test Set. (d) Loss vs. Epochs.

Figure 4: Results on Modified Out-of-the-Box CNN.



(a) Training Set. (b) Validation Set.

(c) Test Set. (d) Loss vs. Epochs.

Figure 5: Results on Basic CNN.

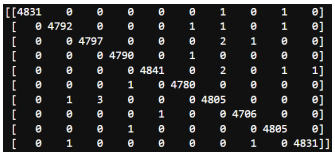
respectively which are improvements compared to both the *out-of-the-box* and basic CNN methods. The resulting testing accuracy improved to 95.92% which is a 0.69% increase and 2.96% increase on the previous explained methods, respectively. The related confusion matrices and loss curves are in Figure 6.

Deeper CNN with K-Fold Cross Validation. The average accuracies for training and testing sets was 99.8% and 96.02%, respectively. Table V depicts the training and testing accuracies in each fold. The resulting confusion matrices can be viewed in Figure 7 and the resulting loss curves can be seen in Figure 8.

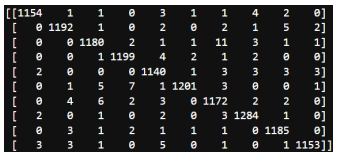
Data	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Training(E)	99.98(20)	99.61(6)	99.76(7)	99.83(7)	99.83(7)
Testing	96.36	95.45	96.02	96.08	96.17

TABLE V: Accuracies with K-Fold Cross Validation. *(E = Early Stopping)*

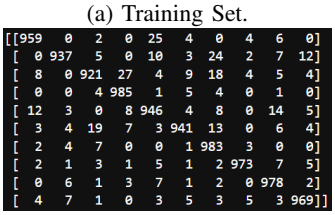
Best Hyperparameters with K-Fold Cross Validation. The hyperparameter combination yielding the highest average test-



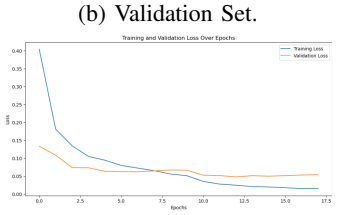
(a) Training Set.



(b) Validation Set.

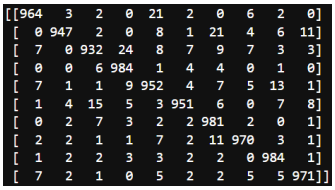


(c) Test Set.

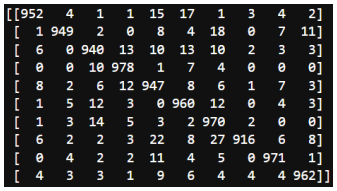


(d) Loss vs. Epochs.

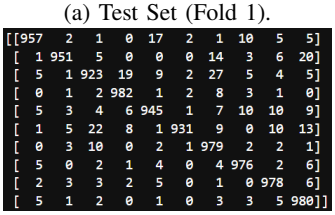
Figure 6: Results on CNN After Increasing Depth.



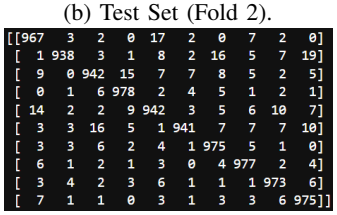
(a) Test Set (Fold 1).



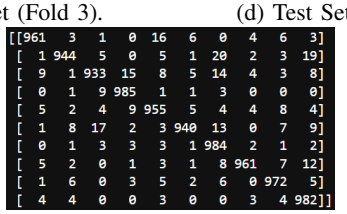
(b) Test Set (Fold 2).



(c) Test Set (Fold 3).



(d) Test Set (Fold 4).



(e) Test Set (Fold 5).

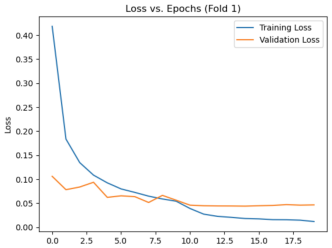
Figure 7: Confusion Matrices of Each Fold for the Test Set.

ing accuracy was when the learning rate = 0.1 and the dropout rate = 0.5. The final model used these hyperparameters to yield a final testing accuracy of ≈96.7% with early stopping around epoch 19. The final confusion matrix for the test set and loss curve plot are viewed in Figure 9.

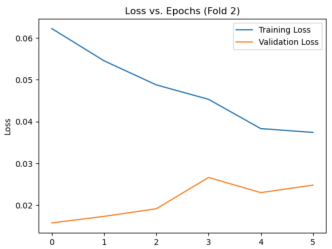
We can see that in this final model, the loss for training and validation both decrease and level off to where the early stopping criteria ends the training. Compared to the other experiments, this combination of hyperparameters proves to reach optimal performance.

VI. Discussion

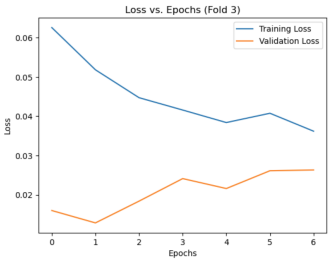
Expectations. In the realm of image classification, it is widely anticipated that Convolutional Neural Networks (CNNs) will outperform Multi-Layer Perceptrons (MLPs) due to several critical factors. CNNs excel in handling spatial information, leveraging parameter sharing, and learning hierarchical features. They use local receptive fields and shared weights



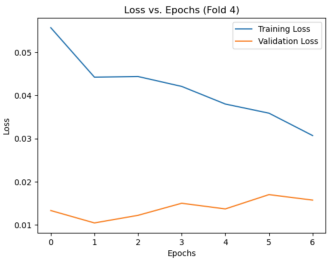
(a) Loss vs. Epochs (Fold 1).



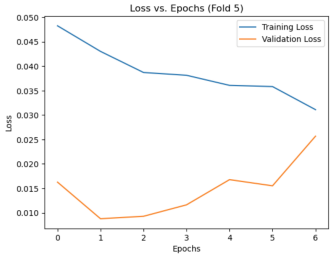
(b) Loss vs. Epochs (Fold 2).



(c) Loss vs. Epochs (Fold 3).



(d) Loss vs. Epochs (Fold 4).

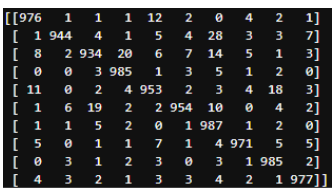


(e) Loss vs. Epochs (Fold 5).

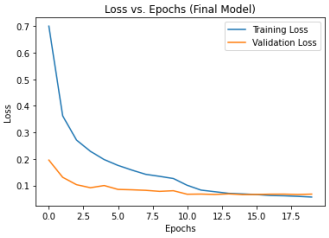
Figure 8: Loss Curves of Each Fold.

Hyperparameters	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
LR: 0.1, D: 0.3	94.78	94.57	96.30	96.40	96.20
LR: 0.1, D: 0.5	**96.29**	**95.99**	**96.16**	**95.85**	**96.49**
LR: 0.1, D: 0.7	95.63	95.30	95.45	94.88	95.35
LR: 0.01, D: 0.3	96.29	94.82	95.55	96.35	95.12
LR: 0.01, D: 0.5	96.19	96.15	96.04	95.81	95.77
LR: 0.01, D: 0.7	95.70	95.16	95.65	95.13	95.26
LR: 0.001, D: 0.3	95.60	94.42	95.05	94.60	95.45
LR: 0.001, D: 0.5	96.26	96.05	96.10	95.65	95.86
LR: 0.001, D: 0.7	95.14	95.49	95.53	95.10	95.26
LR: 0.0001, D: 0.3	95.97	94.63	94.52	96.24	94.13
LR: 0.0001, D: 0.5	95.87	95.96	96.21	96.19	95.80
LR: 0.0001, D: 0.7	95.46	95.34	95.56	95.39	95.68

TABLE VI: Accuracies Over Each Hyperparameter Combination and at Each Fold.



(a) Confusion Matrix on Test Set (Final Model).



(b) Loss vs Epochs (Final Model).

Figure 9: Results for the Final Model.

to capture spatial hierarchies, enabling them to detect local patterns, edges, and textures while remaining translation-invariant. With significantly fewer parameters than MLPs, CNNs efficiently generalize and mitigate overfitting. The architecture of CNNs, comprising multiple layers, progressively learns abstract and complex features, starting from basic patterns like edges and corners and progressing to shapes and objects. The inclusion of convolutional and pooling layers enhances their ability to recognize intricate image patterns. Additionally, CNNs exhibit invariance to small translations and deformations, rendering them robust to variations within an image. Unlike MLPs, CNNs can directly handle raw pixel values without extensive preprocessing, making them inherently suitable for image data feature extraction. For these reasons, we confidently expected the CNN models to outperform the MLP models.

Outcomes. The CNN out-performs the MLP in serveral ways, boasting a higher accuracy rate on both test and training datasets, while also being much more light-weight. The CNN has a 19% lower error rate on the test data, something critical for any system deployed in a real-world application, but even more impressive is the size of the model required to do so. For any practical real-world system the size of the model and speed of execution will also be factors included in choosing which model to deploy. So all-in-all the CNN is more accurate, has a lighter hardware requirement, and better generalization ability, as shown by its greater performance on the test dataset.

Data	Total Parameters	Error Rate (Test)	Error Rate (Train)
CNN	255,226	3.3%	.02%
MLP	48,457,738	4.3%	.2%

TABLE VII: Head-to-Head Comparison

VII. Conclusion

The comparison between Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) for the task of classifying the K-MNIST dataset yields compelling evidence in favor of CNNs. CNNs exhibit a superior ability to handle spatial information, thanks to their use of local receptive fields and shared weights, enabling them to capture local patterns and maintain translation invariance. The hierarchical feature learning within CNNs, with progressively more complex abstractions at each layer, allows them to excel in recognizing intricate image patterns, from basic edges to complex objects. Furthermore, CNNs demonstrate impressive generalization capabilities due to their significantly reduced parameter count, making them more robust against overfitting. In contrast, MLPs, being densely connected, struggle to grasp the spatial hierarchies and patterns that are prevalent in image data. The outcomes of the comparison reflect the widely expected result, with CNNs achieving superior accuracy compared to MLPs while maintaining a lighter model size. This is not only essential for real-world applications but also highlights the efficiency of CNNs in handling image data. Therefore, the evidence strongly supports the preference for CNNs over MLPs when tackling image classification tasks, reaffirming their effectiveness and practicality in this domain.

VIII. References

[1] https://medium.com/@harunijaz/a-step-by-step-guide-to-installing-cuda-with-pytorch-in-conda-on-windows-verifying-via-console-9ba4cd5ccbef

[2] https://www.tensorflow.org/install/pip

[3] https://anaconda.org/conda-forge/tensorflow-gpu,https://gist.github.com/anirban94chakraborty/0364e37ec1ddd57fe935fcf1e7fabd36,https://medium.com/analytics-vidhya/installing-tensorflow-gpu-on-anaconda-3f49c59c122b

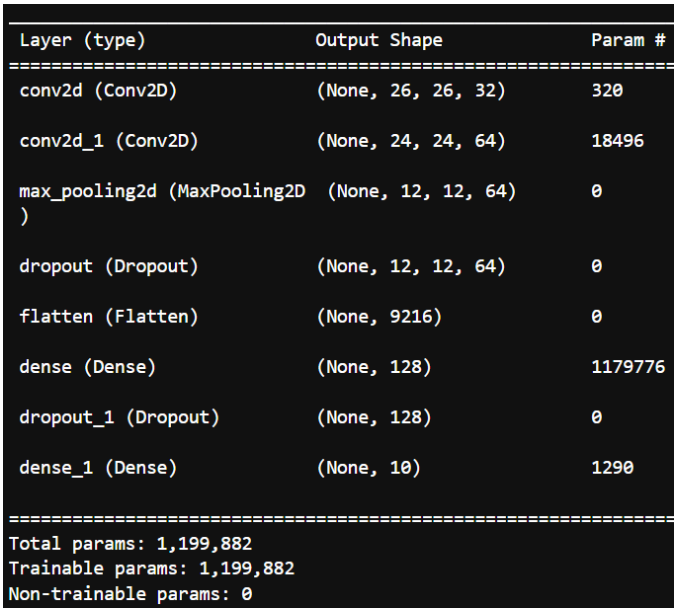
[4] https://github.com/rois-codh/kmnist/blob/master/benchmarks/kuzushiji_mnist_cnn.py

[5] https://github.com/mattburtnz07/PyTorch-Convolution-Neural-Network/blob/master/beginners-guide-to-pytorch-cnn-s-by-a-beginner.ipynb

Appendix A
Architectures

```
Layer (type)                 Output Shape              Param #
=================================================================
random_rotation (RandomRot   (None, 28, 28)            0
ation)

gaussian_noise (GaussianNo   (None, 28, 28)            0
ise)

flatten (Flatten)            (None, 784)               0

dropout (Dropout)            (None, 784)               0

dense (Dense)                (None, 8192)              6430720

batch_normalization (Batch   (None, 8192)              32768
Normalization)

dropout_1 (Dropout)          (None, 8192)              0

dense_1 (Dense)              (None, 4096)              33558528

batch_normalization_1 (Bat   (None, 4096)              16384
chNormalization)

dropout_2 (Dropout)          (None, 4096)              0

dense_2 (Dense)              (None, 2048)              8390656

batch_normalization_2 (Bat   (None, 2048)              8192
chNormalization)

dropout_3 (Dropout)          (None, 2048)              0

dense_3 (Dense)              (None, 10)                20490

=================================================================
Total params: 48457738 (184.85 MB)
Trainable params: 48429066 (184.74 MB)
Non-trainable params: 28672 (112.00 KB)
```

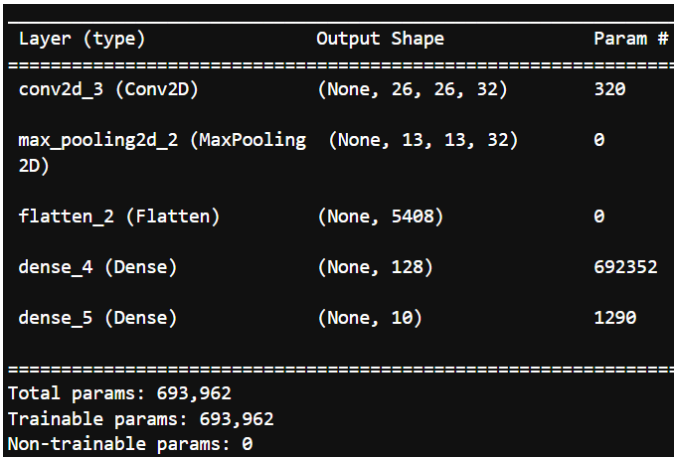
Figure 10: Architecture for Best Phase-3 MLP.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
conv2d_1 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d (MaxPooling2D)	(None, 12, 12, 64)	0
dropout (Dropout)	(None, 12, 12, 64)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 128)	1179776
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1290

Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0

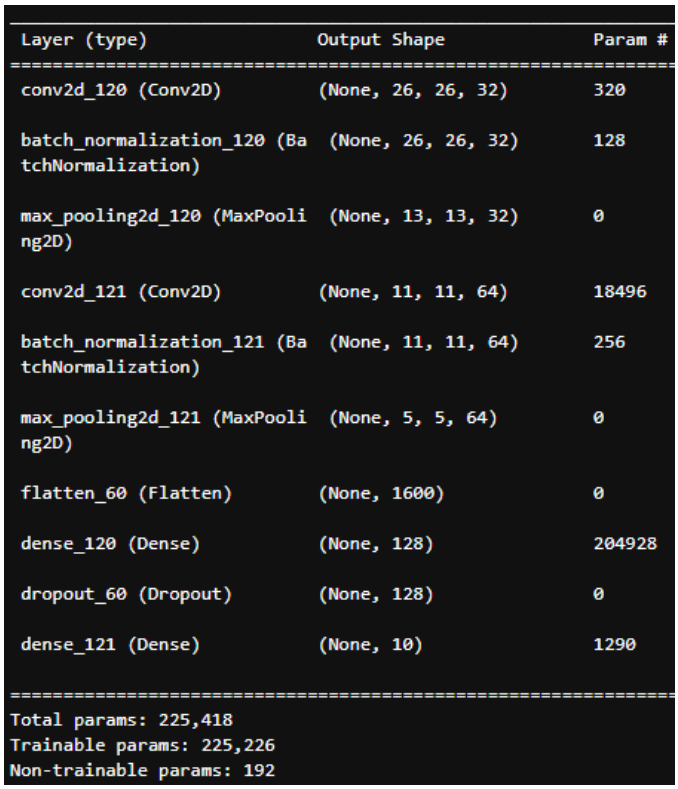
Figure 11: Architecture for Out-of-the-Box CNN Model.



Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 32)	0
flatten_2 (Flatten)	(None, 5408)	0
dense_4 (Dense)	(None, 128)	692352
dense_5 (Dense)	(None, 10)	1290

Total params: 693,962
Trainable params: 693,962
Non-trainable params: 0

Figure 12: Architecture for the Basic CNN Model.



Layer (type)	Output Shape	Param #
conv2d_120 (Conv2D)	(None, 26, 26, 32)	320
batch_normalization_120 (BatchNormalization)	(None, 26, 26, 32)	128
max_pooling2d_120 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_121 (Conv2D)	(None, 11, 11, 64)	18496
batch_normalization_121 (BatchNormalization)	(None, 11, 11, 64)	256
max_pooling2d_121 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_60 (Flatten)	(None, 1600)	0
dense_120 (Dense)	(None, 128)	204928
dropout_60 (Dropout)	(None, 128)	0
dense_121 (Dense)	(None, 10)	1290

Total params: 225,418
Trainable params: 225,226
Non-trainable params: 192

Figure 13: Architecture for Final CNN Model.