Convolutional Neural Network for MNIST classification

*

Rahul Mishra
200267922
Department of ECE
North Carolina State University
rmishra4@ncsu.edu

Varsha Nagarajan
200261357
Department of CSE
North Carolina State University
vnagara2@ncsu.edu

Rajaram Sivaramakrishnan
200253665
Department of ECE
North Carolina State University
rthiruv@ncsu.edu

***Abstract*—The following report details our design and implementation of a convolutional neural network (CNN) for MNIST digit classification. The network architecture involves multiple layers including convolution, maxpooling, and fully connected layers. Optimal values for hyperparameters are critical in obtaining good results. Hyperparameter tuning is discussed in detail and associated results have been visualized through various plots. The final tuned model gives an accuracy of over 99% when evaluated on the test dataset. The detailed findings and results are presented in the form of tables and plots.**

***Index Terms*—Convolutional Neural Network, MNIST, deep learning, image classification**

I. Introduction

The goal of this project is to implement a CNN architecture using a deep learning framework. A convolutional neural network for MNIST classification was designed and implemented using TensorFlow. Hyperparameter tuning was then performed to decide the effective learning rate, kernel size, optimizer functions, dropout rate, epochs and batch sizes. After finding and fixing on the best set of hyperparameters, different activation functions like ReLU, sigmoid and tanh were used to introduce non-linearity and their performance was compared by evaluating the validation set on the tuned model. Corresponding results and learning curves accompany the report. The trained model was then used to make predictions on the test set.

II. Dataset

The dataset used is *MNIST dataset*. It is a database of handwritten digits with each image being a gray scale image of dimension 28×28. This dataset consists of 60000 training images and 10000 test images. For hyperparameter tuning, we evaluate the various parameters on a 25% validation split.

III. Convolutional Neural Network Architecture

1) **Input**: The input data is a grayscale image of size 28×28, consisting of raw pixels of image containing digit (0-9) data.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	832
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18496
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 64)	0
flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 1024)	3212288
dropout_1 (Dropout)	(None, 1024)	0
dense_2 (Dense)	(None, 10)	10250

Total params: 3,241,866
Trainable params: 3,241,866
Non-trainable params: 0

Fig. 1. CNN Model Architecture

2) **Conv2D Layer**: Consists of 32 filters/kernals of size 5x5 convolving over the input image. ReLU activation is used to introduce non-linearity.
3) **Maxpooling2D Layer**: 2x2 max-pooling is used to reduce the dimension of the output of first convolution layer.
4) **Conv2DLayer**: An additional convolutional layer consisting of 64 filters of size 3x3 is introduced followed by a ReLU activation function.
5) **Maxpooling2D Layer**: Another max-pooling layer is introduced to further reduce the dimension.
6) **Fully Connected Layer 1**: This layer consists of 64 hidden units and is connected to to all activations in the previous previous max-pooling layer. This layer works similar to a regular neural network with many parameters to be learnt. ReLU activation is used at the end of fully connected layer.
7) **Fully Connected Layer 2:** This layer consists of 10 hidden units and a softmax activation is applied to predict the digit in the input image. The softmax activation outputs the models predictions in the form of probabilities for each of the 10 labels ranging from 0 to

9 where all probabilities sum up to 1.

IV. Range of Hyper-Parameters

The below table summarizes the range of various hyperparameters that we experimented with.

TABLE I
HYPERPARAMETER TUNING RANGES

Hyperparameters	*Range*
Learning Rate	1, 0.1, 0.01, 0.001, 0.0001, 0.00001
Dropout Rate	0.25, 0.5, 0.75
Optimizer	RMSProp,SGD, Adam
Epochs	15, 20, 30, 50
Kernel Size	(3,3), (5,5)
Batch Size	50, 100, 128, 256

V. Visualization and Analysis of Cross-Validation for Hyper-Parameters

For the purposes of hyperparameter tuning, we use a 25% validation split and assess the model's performance on a single validation set. Due to time and computational constraints, we did not perform a k-fold CV.

A. (Accuracy and Cost) vs. Learning Rate

Fig. 2 and 3 shows the plot of accuracy and cost vs the learning rate. As we can observe, the changes in validation accuracy/loss across different learning rates is not that prominent with learning rates of 10^{-3} through 10^{-1}, giving a similar accuracy of about 99.2% and loss being very close to zero. For very low learning rates, the loss may decrease, but at a very shallow rate, resulting in increase in the convergence time. When entering the optimal learning rate zone, we generally observe a quick drop in the loss. Increasing the learning rate beyond this will cause an increase in the loss, as the parameter updates cause the loss to "bounce around" and even diverge from the minima. This can result in delayed convergence or worse still, there is a good chance of not reaching the minima. We know that the best learning rate is associated with the steepest drop in loss, so we will mainly analyze the slope of the plot, specially the span between 10^{-5} and 10^{-4} where we see the steepest drop in loss. Other readings suggest that the ideal scoping zone for efficient learning rates is the region that lies to the left of the lowest point in the graph, which in our case is 10^{-1}. Hence, following the above explanation, we choose our ideal learning rate from the region of the steepest drop in loss and fix on 10^{-4}. We went a step further to assess the validation and cost variations across various epochs for different learning rates and found the curve to be a little bumpy for potential learning rate candidates of 10^{-3} and 10^{-2} while that of 10^{-4} was comparatively smooth. Thus, to avoid the problem of delayed or no convergence and divergence, we start with an initial learning rate of 10^{-4} which adapts over time by the use of Adam Optimizer.

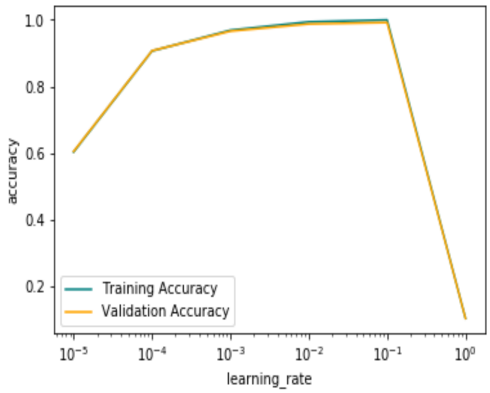

Fig. 2. Accuracy vs. Learning Rate

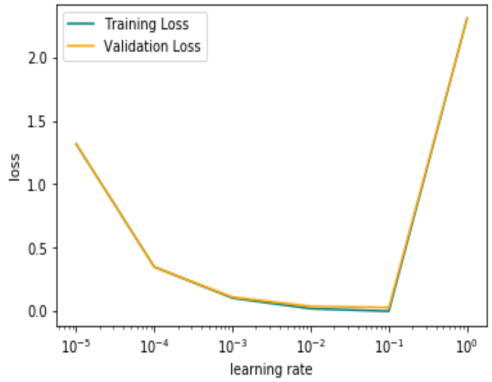

Fig. 3. Loss vs. Learning Rate

B. (Accuracy and Cost) vs. Dropout Rate

Table II and III shows the variation of training and validation accuracy for different dropout rates. The values are very close and there is no clear winner. In this case, we fall back to the choice made by popular researches in this field and stick with the dropout regularization rate of **0.5**.

TABLE II
ACCURACY VS DROPOUT

Dropout	**Accuracy**	
	Training	***Validation***
0.25	0.9991	0.9909
0.50	0.9995	0.991
0.75	0.9994	0.9913

TABLE III
COST VS DROPOUT

Dropout	**Cost**	
	Training	***Validation***
0.25	6.834053e-05	0.046428785
0.50	0.006503931	0.038569666
0.75	0.020871224	0.04305145

C. (Accuracy and Cost) vs. Optimizer

Figures 4, 5, 6 and 7 show the training and validation accuracy and losses across different epochs when working with different choice of optimization methods viz., Stochastic Gradient Descent (SGD), Adam and RMSProp. While both training and validation accuracy saturates at about 90% when using SGD, both Adam and RMSProp perform better and equally well giving more than 99% accuracy. We see a similar result when analyzing the respective losses as well. Loss in case of SGD appears to saturate at around 0.35 while RMSProp and ADAM, both give decent results. In this case, we decide to go with Adam Optimizer mainly because it combines the benefits of Momentum and RMSProp to give a best of both worlds method.

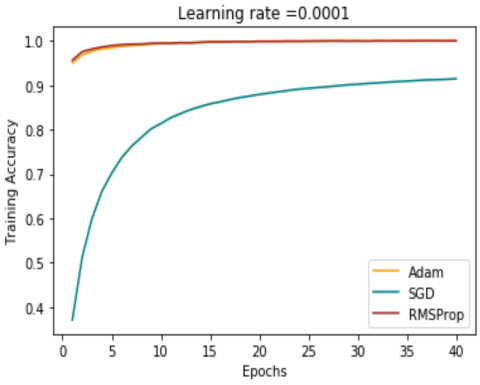

Fig. 4. Training Accuracy vs. Epochs For Different Optimizers

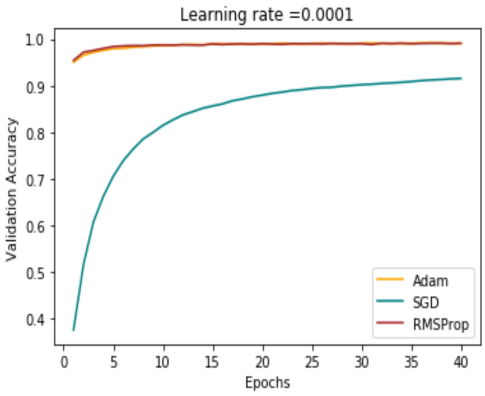

Fig. 5. Validation Accuracy vs. Epochs For Different Optimizers

D. (Accuracy and Cost) vs. Epochs

Figures 8, 9 and 10 show the variation of training and validation accuracy/loss across different epochs when evaluated on an architecture with 2 convolutional layers designed using 5x5 and 3x3 filters with a learning rate of 0.0001, dropout rate of 0.5 and using AdamOptimizer. Without dropout, we notice a pretty bumpy curve with fluctuating accuracies but the same is stabilized when using dropouts. Dropout is one of

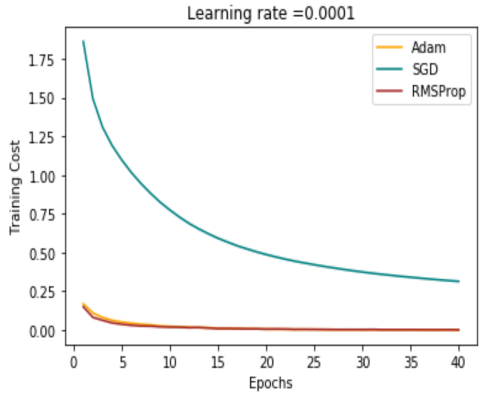

Fig. 6. Training Cost vs. Epochs For Different Optimizers

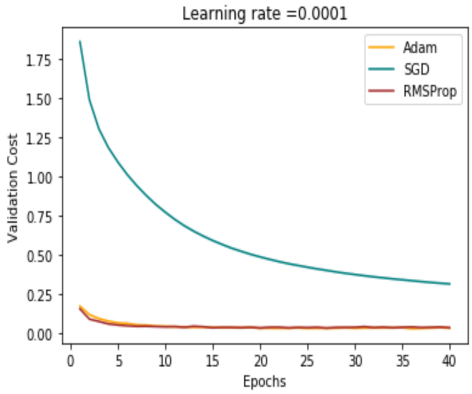

Fig. 7. Validation Cost vs. Epochs For Different Optimizers

the reasons why we do not see overfitting even when trained for more than 40 epochs. However, the curve seems to plateaus post 20 epochs (approx) and we surely will not be achieving a whole lot of increase in accuracy if we train more and hence we fix on this number.

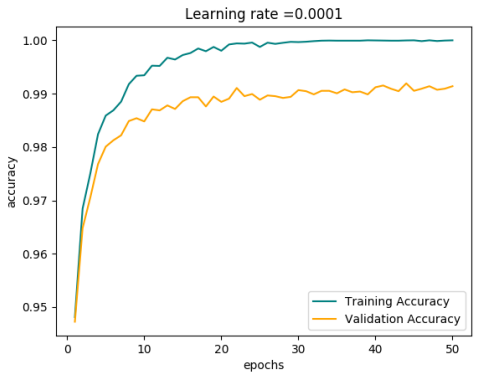

Fig. 8. Accuracy vs. Epoch

E. (Accuracy and Cost) vs. Kernel Size

Table IV and V shows the training and validation accuracies and losses for different filter sizes. The values are extremely

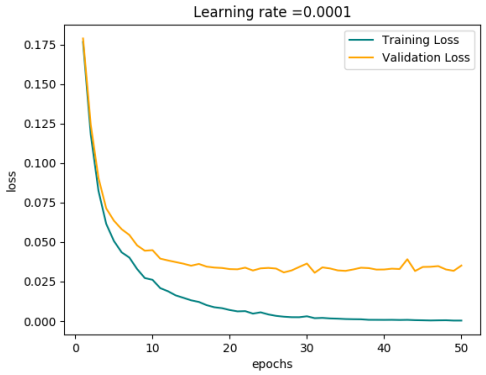

Fig. 9. Cost vs. Epoch

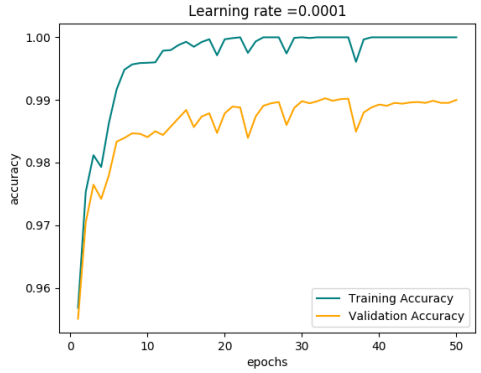

Fig. 10. Accuracy vs. Epoch (Without Dropout)

close, especially for 5x5-5x5 and 5x5-3x3 and choosing any of the combinations would work well in this case. So we pick 5x5-3x3 combination mainly because this combination works equally well and has lesser parameters in comparison to 5x5-5x5 combination.

TABLE IV
ACCURACY VS KERNEL SIZE

Filter Size		Accuracy	
Layer 1	**Layer 2**	***Training***	***Validation***
3x3	3x3	0.9976444	0.9896
3x3	5x5	0.9896	0.98913336
5x5	3x3	0.99822223	0.9898667
5x5	5x5	0.99895555	0.9898667

TABLE V
LOSS VS KERNEL SIZE

Filter Size		Loss	
Layer 1	**Layer 2**	***Training***	***Validation***
3x3	3x3	0.009196762	0.033910025
3x3	5x5	0.007215403	0.032822676
5x5	3x3	0.007461031	0.03325466
5x5	5x5	0.00446453	0.036903277

F. (Accuracy and Cost) vs Batch Size

Table VII shows the training and validation accuracies obtained for different batch sizes. As with most other hyper-parameters discussed above, there is no clear winner in this case too. While smaller batch sizes would be preferable, we decide to not go for too small values as that would just take a bit longer to train completely and the difference in accuracies are not so significantly large to justify for more training time. Hence, we establish middle ground here by choosing our batch size as 100.

TABLE VI
ACCURACY VS BATCH SIZE

Batch Size	Accuracy	
	Training	***Validation***
50	0.99862224	0.9902
100	0.9984222	0.98913336
128	0.9979778	0.98933333
256	0.9978222	0.99006665

TABLE VII
LOSS VS BATCH SIZE

Batch Size	Loss	
	Training	***Validation***
50	0.0055021625	0.0396
100	0.0065	0.038569
128	0.006869	0.0379
256	0.008719	0.035468

VI. LEARNING CURVES FOR DIFFERENT ACTIVATION

After tuning the various hyper-parameters using single validation set, below are the values that we fixed on.

1) **Learning Rate**: 0.0001
2) **Dropout Rate**: 0.5
3) **Optimizer**: AdamOptimizer
4) **Epochs**: 20
5) **Kernel Size**: (5,5) and (3,3) for the two layers respectively
6) **Batch Size**: 100

This tuned model is now evaluated across different activation functions namely, ReLU, Sigmoid and Tanh. The learning curves (loss and accuracy) for training and validation sets can be seen in Fig. 11 and 12. We observe that all 3 activation functions perform satisfactorily with ReLU and Tanh being slightly ahead of Sigmoid. In this case, we break the tie by going with the computationally efficient activation function that is popular among many researches and is actively used today and this would be ReLU.

VII. RESULTS AND ANALYSIS

Test Accuracy: For a 2 layer convolutional neural network with 5x5 and 3x3 filters, learning rate of 0.0001, dropout rate of 0.5, batch size of 100, trained over 20 epochs using an AdamOptimizer, we obtain a test accuracy of 99.32%.

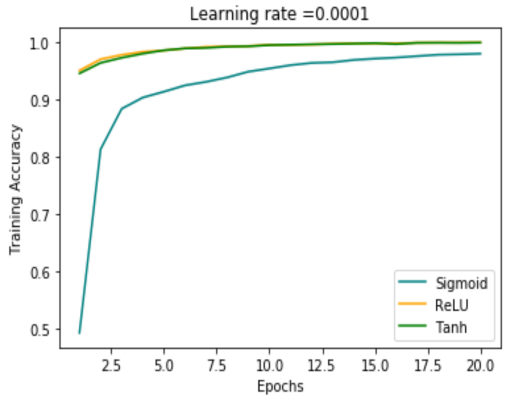

Fig. 11. Training Accuracy for Different Activation Functions

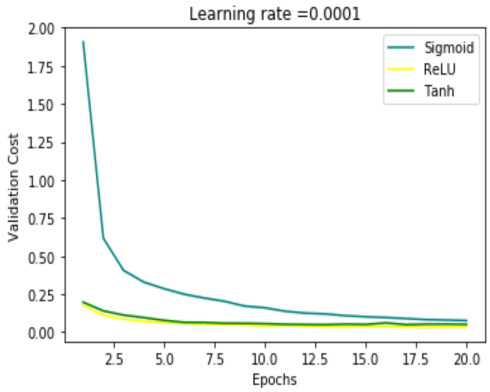

Fig. 14. Validation Cost for Different Activation Functions

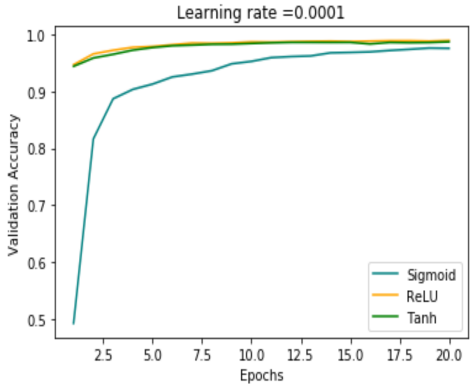

Fig. 12. Validation Accuracy for Different Activation Functions

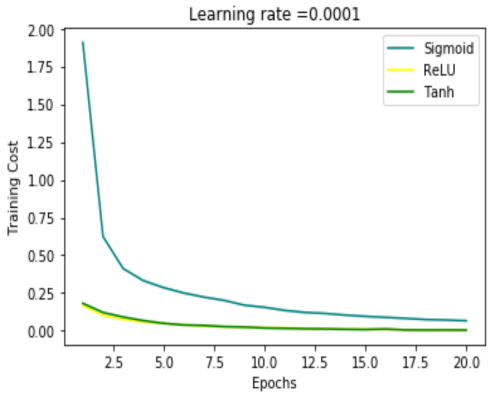

Fig. 13. Training Cost for Different Activation Functions

In the previous section, we have provided a detailed analysis of the various hyperparameters explored. We tried out different ranges and performed a number of combinations of experiments to arrive at the above values. For learning rate, our prime objective was to use a value that would avoid the problem of divergence and delayed or no convergence. For optimizers, we wanted to use AdamOptimizer mainly because it combines Momentum and RMSProp to give us the best of both worlds. As evident from the plots, we do observe AdamOptimizer performing really well for the given dataset. The choices for dropout regularization rate and batch size were particularly difficult to make owing to very little variation across different ranges. Due to this, we fall back to intuitions and theories derived from various research papers that have reported good performance for a dropout rate of 0.5. And as for batch size, we try to choose a value not too small or large and hence we fix on 100. Since the images we are working with are clutter-free and black and white, we do not need a very complicated architecture to model the images. A simple one-layer network itself gives a 92% accuarcy which was further improved by adding another convolutional layer followed by max-pooling. The values we have arrived at are backed by either intuitions derived from previous researches or evidence offered by plots. Since most of the hyperparameters have very similar values in the ranges explored, a slight deviation from the values we fixed on will not result in very bad results either. We can still achieve 98-99% accuracy with nearby values.

VIII. Discussion

The code for the above analysis can be found in two files namely *Digit.py* and *Digit_Tuning.py*. The *Digit.py* contains commented section for various hyperparameters tuned and was executed manually for different parameters. *Digit_Tuning.py* contains the consolidated code for various hyperparameter tuning which were tested out in loops. The reason we have two versions is because the consolidated version could not even run on systems with GPU memory of about 12 GB as the requirement shoots up to a value even more than that. So, we used *Digit.py* to evaluate in parts and present our results. *Digit.py* is much more cleaner and toned down version of *Digit_Tuning.py*. So, if you wish to run our codes, we suggest you use *Digit.py* for that.