Modified MNIST Classification

Group 22: Brendan Furtado (260737867), Nikhil Krishna (260862308), Hongshuo Zhou(260792817)

I. ABSTRACT

In this project, we aim to classify and output the digit with the highest numeric value in images from a modified version of the MNIST dataset. We tackled the problem by developing convolutional neural networks from three different approaches, with the best one achieving a 96.1% accuracy on the test set on Kaggle. Analysis of our implementation process such as the convolutional network architectures, cross-entropy loss and length of training are discussed in detail.

II. INTRODUCTION

A. Preliminaries

Convolutional Neural Networks (CNNs) are a class of deep neural networks that alternate between using convolutional layers, pooling layers, and fully connected layers, typically applied to computer vision problems. [9] Unlike a standard neural network, each neuron has a 3-dimensional vector associated with it representing the height, width, and depth of the image data. The convolutional layers are typically used for computing the output of each neuron in a locally connected region of the input. Max-pooling is typically applied after convolutions take place and reduce the dimensionality of the feature space by using only the maximum value of each local cluster of neurons in the previous layer. Typically, the fully connected layers are placed at the end of the CNN after all convolutions and pooling processes have been performed. This streamlines the data, and allows for a solid prediction of the classes. [10] Figure 1 shows an example of the architecture of VGGNet. [9] One can see that typically max pooling is applied after each of the convolutional layers and that the model ends with a fully connected layer.

Fig. 1. VGGNet architecture: a well-known CNN [9]

B. Task Description

The task at hand was to classify greyscale images into one of 10 classes. Each image has 3 single-digit numbers and our task was to formulate a model that would receive an image and classify it into class 0 to class 9, corresponding to which digit had the highest numerical value in the image. The two requirements were that the code be written in Python and that the training algorithm had to use supervised learning.

C. Important Findings

Between all three CNN models we implemented, the Keras model ran the training loop the fastest and achieved the highest accuracy score in the Kaggle comptetition. The Nadam optimizer and early stopping with default parameters were used in this model's execution. In general, the model seemed to improve at a faster rate while also showing higher efficiency. While three different optimizers and parameters were used between the models, the number of epochs stayed relatively the same. The Keras model ran approximately 30 epochs within three runs that stacked the training of the model (includes early stopping occurrences). The ResNet18 and PyTorch implementations both ran a total of 25 epochs, running anything higher resulted in memory overloads. The Keras model ran for an average of 108 seconds per epoch, wheres the ResNet18 model ran for an average of 171 seconds per epoch. The PyTorch CNN's runtime ended up being between the two other models. The ResNet18 model gave the second best results in the Kaggle competition, with an accuracy of 93.1%, whereas the PyTorch model fell below the 90 percent benchmark.

III. RELATED WORK

Two main papers were used throughout the project for reference. The first was an publication by Ciresan et al. that presented a new architecture for deep neural network's by combining multiple convolutional layers to create a Multi-Column CNN. [5] The publication's results show that the performance of a multi-column CNN is better than that of a single-column CNN. Additionally, they used the MNIST dataset for classification. This relates to our project as we used this paper to see how the data needs to be structured before inputting it into a CNN.

The second paper was the paper on ResNet18. The authors present a residual learning framework to help ease the training of deep neural networks. This made deep learning much easier to optimize. [6] Since we implement ResNet18 as one of our models, we used this paper's work substantially.

IV. DATASET AND SETUP

For this project we have been provided with a training set of 45,000 images and a validation set of 5,000 images. Every image has an associated label representing the digit in the image with the highest numeric value. Each image has a resolution of 128×128 pixels where each pixel has a greyscale value between 0 and 255. The original MNIST

dataset was modified and given to us for training. The images in the original MNIST dataset have resolutions of 28×28 pixels, whereas our images are 128×128 pixels. Further, the new MNIST dataset consists of 3 numbers, while the original only had one number per image.

For the pre-processing methods, we normalized the pixel values to get better prediction results compared to raw data. All models except the Keras implementation, which used 48000 points for training and 2000 for validation, were ran with a training set of size 45,000 and validation set of size 5,000.

V. Proposed approach

To solve the task, we create 3 different models. These models will be compared to each other to determine which best fits the data and best generalizes to new data.

A. Keras

The Keras model used had 5 general sections, the first 3 of which used two convolutions, while the fourth used only one. Batch normalization and dropout layers help reduce overfitting; max pooling was utilized to increase training accuracy. The model ends with a standard flatten layer and a fully connected network to classify the images. All activations used for the Keras model were with the ReLU activation aside for the output layer. This used the softmax activation function. For Keras model specifics, see the corresponding code and Figure 8 in the Appendix.

B. ResNet18

The pre-built ResNet18 architecture was implemented through PyTorch. The popularity of this model came from the structure of "identity shortcut connections" that skip one or more layers. These skip connections between layers and outputs from previous layers to the layers ahead (stacked layers). This key component in the layer structure is now used in many modern CNN architectures such as DenseNet. With ResNet introduced in 2015, the ability to train deeper neural networks became more robust. [4][7] The ResNet models are used a lot in the research community, so we decided to adapt it for this classification task.

The structure of our PyTorch implementation of the ResNet18 architecture consists of the following. There is one convolutional layer, added ontop of the ResNet, with input and output channels of 1 and 64 respectively, and kernel, stride, and padding of 7, 2 and 3. These are standard parameters for this type of architecture. We also use the softmax function in the forward pass to map output values to a probability value that is easy to interpret. These probability values (visualized as a 1-d vector) give the probabilities that the largest number in the image belongs to a certain class (classes being 0 to 9).[8] Data for this model is later resized, normalized and loaded into different sets within a function called get_data_loaders.

C. PyTorch

The PyTorch implementation uses a sequential container for a total of three layers. Each layer applies two convolutions of different in and out channels and kernel parameters. The nonlinear function ReLU and a max pooling are also called within each layer to characterize the CNN. The structure of this model is shown below.

1) *Section 1*: 2 convolutions, the first taking in 1 input channel and the second outputting 32 channels. A batch normalization layer was added here along before the ReLU activation was passed in. Then max-pooling was applied with a kernel size of 2. Finally, a dropout layer was included with 0.3 as the parameter.
2) *Section 2*: 2 convolutions, the first taking in 32 input channels and the second outputting 64 channels. A batch normalization layer was added here along before the ReLU activation was passed in. Then max-pooling was applied with a kernel size of 2. Finally, a dropout layer was included with 0.3 as the parameter.
3) *Section 3*: 2 convolutions, the first taking in 64 input channels and the second outputting 128 channels. A batch normalization layer was added here along before the ReLU activation was passed in. Then max-pooling was applied with a kernel size of 3 and a stride of 2. Finally, a dropout layer was included with 0.3 as the parameter.
4) *Section 4*: Takes in the flattened output of section 3 (18432 inputs) and outputs a predicted class. This is a fully connected layer.

For further analysis of the PyTorch model, see the corresponding code.

VI. RESULTS

A. Keras

With the Keras CNN, we fit the model a total of 3 times to the data employing early stopping each time. We achieved a 93.1% accuracy on the training data and a 95.2% accuracy on the validation data after the first fit. After the final run, our training accuracy and validation accuracy both increased to 97.2% and 97.6%, respectively. This was the model that produced the best accuracy on the test set. Our Kaggle submission of 96.1% accuracy was produced with this model.

Keras' nadam optimizer with a learning rate of 0.002 was used during training. Additionally, to calculate cost we used the standard cross-entropy loss for categorical labels. The model was set to run through 20 epochs for each run. These hyperparameters led to very favorable results, and due to the computational power needed to test further models, we decided to use these. Figure 2 shows how the training set accuracy and validation set accuracy change with the epoch for the first training run. For both data sets, as the epoch number increased, generally the accuracy increased. Interestingly, the validation accuracy was typically higher than the training accuracy for each epoch.

To investigate the behavior of the Keras implementation further, we plotted a graph of cross-entropy loss versus epoch

for both the training and validation sets for the first training run (Figure 3). One can see that as the loss decreases with epoch, as expected. The validation set also had a generally smaller loss than the training set. Note that Figures 2 and 3 both stop at 14 epochs because early stopping occurred during training. The first training run on the 48,000 data points took an average of 108 seconds per each of the 14 epochs. Similar times were recorded for the second and third runs.

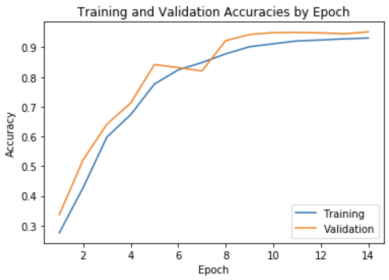

Fig. 2. Accuracy vs. Epoch of Training and Validation Sets for Keras CNN for run 1

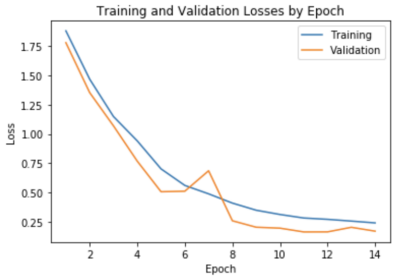

Fig. 3. Loss vs. Epoch of Training and Validation Sets for Keras CNN for run 1

B. ResNet18

The ResNet18 model, implemented through PyTorch, went through 25 total epochs of training. This achieved an accuracy of 94.5% on the validation data (5,000 data points) and a Kaggle submission accuracy of 93.1% This model performed the second best out of all three models, giving slightly lower accuracies than the Keras model. We attempted to increase the total number of epochs to run, however, we ran into memory & RAM problems when running 30 or more epochs.

The optimizer used for this model was the adadelta optimizer with default parameters. Compared to other models like stochastic gradient descent (SGD), the adadelta algorithm uses first order information and because of that has less overhead than SGD. On top of that, the default parameters and learning rate don't need to be tweaked too much, as it runs robustly.[11] We also use the cross-entropy loss function for this model. The training under the default hyperparameters led towards positive results, in terms of an increasing validation accuracy and a decreasing cost function. Figure 4 shows the upward trend of the accuracy on the validation set. Compared to the accuracies from the Keras model, there are more sharp decreases in accuracy going across the graph, most notably at epoch 18 where the accuracy goes down to around 50%. Epochs 19-25 are when accuracies go up again and remain consistent, plateauing to around 93%-94%.

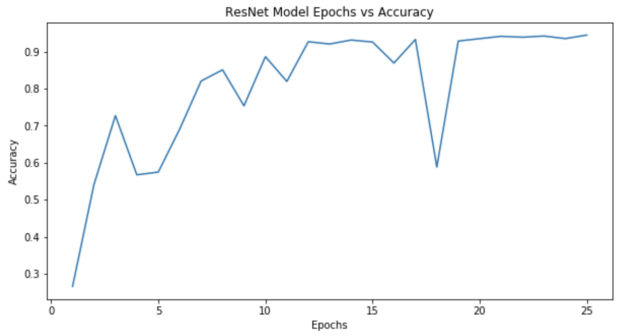

Fig. 4. Epochs vs. Accuracy of Validation Set for ResNet CNN for 25 epochs

To further analyze how the model performs, we also calculated the cross-entropy loss over the number of epochs. Figure 5 shows the downward trend of the loss values as the number of iterations go on. The trend is very similar to the Keras model, however, it never goes below a loss value of 1.5 which is easily achieved by the Keras model.

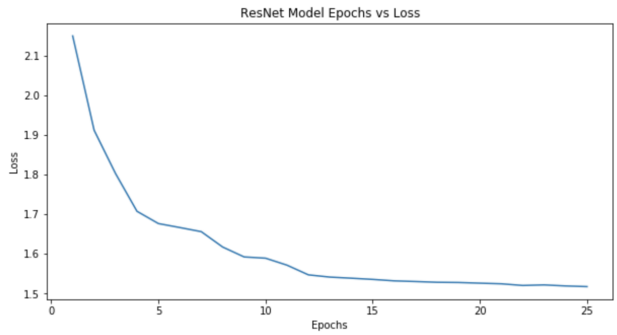

Fig. 5. Epochs vs. Loss of Validation Set for ResNet CNN for 25 epochs

C. PyTorch

The standard convolutional neural network implemented in PyTorch resulted in the worst accuracy of the three models. The training accuracy produced was 82.1% and the validation accuracy was 88.9%. This model was not used to generate predictions on the test data for this reason; it was used as a baseline model.

The cost function used here was, of course, cross-entropy loss. To optimize this cost function, PyTorch's adam optimizer was used with default parameters. These seemed to reduce the value of the training and validation costs simultaneously with the epoch number, so we did not alter them further. The model was trained on 25 total epochs.

Figure 6 shows how the validation accuracy of the model changed with epoch. After the second epoch, a large jump

in accuracy was seen, but was quickly taken away during training in epoch 3. After this, the model increased in validation accuracy as epoch number increased. The cross-entropy loss was also tracked with each epoch. Figure 7 shows the corresponding plot. Of the three validation loss graphs produced, this one is the most unstable. As epoch increased though, it does have a downward trend when considering all 25 epochs. However, after epoch 10, the loss fluctuated and showed no consistent downward trend.

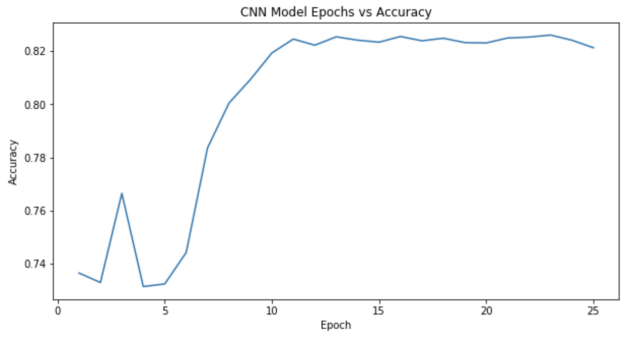

Fig. 6. Epochs vs. Accuracy of Validation Set for PyTorch CNN for 25 epochs

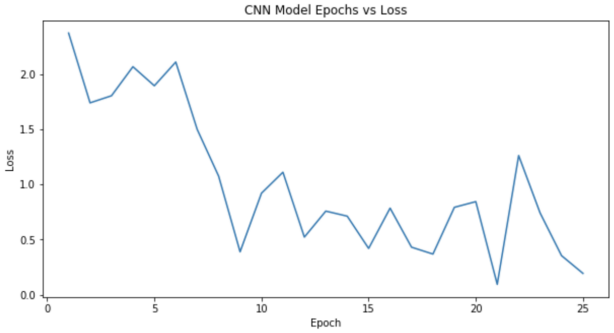

Fig. 7. Epochs vs. Loss of Validation Set for PyTorch CNN for 25 epochs

TABLE I
VALIDATION ACCURACIES & LOSS OF KERAS, RESNET18 (PYTORCH IMPLEMENTATION), AND STANDALONE PYTORCH CNN MODELS

Value	Keras	ResNet18	PyTorch
Val-Accuracy (%)	97.6	94.5	88.9
Loss	.101	1.52	.194

VII. DISCUSSION & CONCLUSIONS

The convolutional neural network implemented through Keras produced the best results for this training data. It trained the fastest and also yielded the highest accuracy. The ResNet18 model produced an accuracy of 94.5% on the validation set, while the PyTorch implementation of the CNN gave a relatively poor validation accuracy of 88.9%. We can see that a simple CNN will do the job better at times than a well-known model such as ResNet18 for certain data. Furthermore, we also noticed that the Keras framework allows for much quicker training and implementation than the relatively complicated PyTorch framework. The Keras framework also is higher-level than the PyTorch framework it seems.

In the future, we could combine all three models with an ensemble approach and generate predictions through that. This would most likely give a higher accuracy than the 96.1% produced by the Keras model alone. Finally, we could further tune the hyperparameters in the model (number of epochs, learning rates, early stopping parameters, etc.).

VIII. STATEMENTS OF CONTRIBUTION

Brendan Furtado: Help create the ResNet implementation in PyTorch. Helped a bit with the standalone CNN in PyTorch. Trained both PyTorch models and created accuracy and loss graphs for both of them. Helped write the paper.

Nikhil Krishna: Helped with implementing Keras CNN model and PyTorch CNN model. Trained the Keras CNN model and created appropriate graphs. Wrote certain sections of the paper.

Hongshuo Zhou: Report writing: Abstract, Related Work and Dataset and setup

REFERENCES

[1] Scikit-learn: Machine Learning in Python, Authors: Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E.. Journal of Machine Learning Research, Volume: 12, Year: 2011

[2] Hastie, Trevor "An Introduction to Statistical Learning with Applications in R," 8th ed. New York: Springer, 2013

[3] "Documentation on ResNet." Pretrained ResNet-18 Convolutional Neural Network - MATLAB, www.mathworks.com/help/deeplearning/ref/resnet18.htmlmw_591a2746-7267-4890-8390-87ae4dc7204c.mw_6dc28e13-2f10-44a4-9632-9b8d43b376fe.

[4] Fung, Vincent. "An Overview of ResNet and Its Variants." Medium, Towards Data Science, 17 July 2017, towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035.

[5] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," CoRR, vol. abs/1202.2745, 2012. [Online]. Available: http://arxiv.org/abs/1202.2745

[6] He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[7] Shorten, Connor. "Introduction to ResNets." Medium, Towards Data Science, 15 May 2019, towardsdatascience.com/introduction-to-resnets-c0a830a288a4.

[8] Lan, Haihan. "The Softmax Function, Neural Net Outputs as Probabilities, and Ensemble Classifiers." Medium, Towards Data Science, 4 Oct. 2018, towardsdatascience.com/the-softmax-function-neural-net-outputs-as-probabilities-and-ensemble-classifiers-9bd94d75932.

[9] Hamilton, W 2019, Lecture 15 — Convolutional Neural Nets, lecture slides, Applied Machine Learning COMP551, McGill University, delivered October 30, 2019.

[10] Karpathy, A *Convolutional Neural Networks*. Retrieved from http://cs231n.github.io/convolutional-networks/.

[11] Zeiler, Matthew D. ADADELTA: AN ADAPTIVE LEARNING RATE METHOD. Google Inc/New York University, Dec. 2012, arxiv.org/pdf/1212.5701.pdf.

IX. APPENDIX

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 126, 126, 64)	640
conv2d_9 (Conv2D)	(None, 124, 124, 64)	36928
batch_normalization_4 (Batch	(None, 124, 124, 64)	256
max_pooling2d_5 (MaxPooling2	(None, 62, 62, 64)	0
dropout_4 (Dropout)	(None, 62, 62, 64)	0
conv2d_10 (Conv2D)	(None, 58, 58, 128)	204928
conv2d_11 (Conv2D)	(None, 54, 54, 128)	409728
batch_normalization_5 (Batch	(None, 54, 54, 128)	512
max_pooling2d_6 (MaxPooling2	(None, 27, 27, 128)	0
dropout_5 (Dropout)	(None, 27, 27, 128)	0
conv2d_12 (Conv2D)	(None, 25, 25, 256)	295168
conv2d_13 (Conv2D)	(None, 23, 23, 256)	590080
batch_normalization_6 (Batch	(None, 23, 23, 256)	1024
max_pooling2d_7 (MaxPooling2	(None, 11, 11, 256)	0
dropout_6 (Dropout)	(None, 11, 11, 256)	0
conv2d_14 (Conv2D)	(None, 9, 9, 128)	295040
max_pooling2d_8 (MaxPooling2	(None, 3, 3, 128)	0
flatten_2 (Flatten)	(None, 1152)	0
dense_4 (Dense)	(None, 64)	73792
dense_5 (Dense)	(None, 64)	4160
dense_6 (Dense)	(None, 10)	650

Total params: 1,912,906
Trainable params: 1,912,010
Non-trainable params: 896

Fig. 8. Keras CNN model summary