**Training Models on the MNIST Dataset**

Matteus DiPietro, Chirag Mittal, Omar Sabbagh

Department of Computer Science, University of Windsor

COMP 4740: Advanced Topics: Artificial Intelligence II

Dr. Robin Gras

March 1, 2022

**Introduction**

**Objective**

Our objective in this implementation of our CNN is to achieve a relatively accurate & precise prediction score (~75%+) that also yields a minimal error rate. We will then compare our results to other architectures trained on the MNIST dataset with other models, in this instance we will be comparing to 3 different models that are unique in each layer.

**CNN Definition**

CNN stands for convolutional neural network. CNNs are a class of Deep Neural Networks that can recognize and classify particular features from images and are widely used for analyzing visual images.

**Training Models on the MNIST Dataset**

Machine learning and image recognition are rapidly growing fields in today’s world and experience in these fields is an asset. Regarding image recognition, there are many state-of-the-art architectures that are being developed on top of the successful convolutional neural network architecture. To better understand these new architectures, it is best to first understand and implement simple convolutional neural networks. To do this we will use the MNIST dataset.

MNIST is a public dataset record of categorized handwritten images ranging from 0-9. This is a widely used dataset for research and experimentation using Machine Learning. We will use this dataset to train and test our model.

The images in the dataset are handwritten digits, each categorized by its first character in the image name, for example if it’s an image of a 7, the image filename will be 7nmistrecordxyz for example.

MNIST is the seen by many as a beginning point for understanding machine learning. The simplicity of the images and the large size allows for algorithms to be effectively tested, while not needing a lot of resources or time to complete the training. The MNIST dataset allows for multiple models to be made and quickly compare so that trends that apply to larger and more complex dataset can be found. Testing multiple models, we will discover multiple useful trends that can be applied to future models and create a model that achieves a high accuracy when trained and tested on the MNIST dataset.

**Tools**

* Python Version 3.7.12
* TensorFlow Version 2.8.0
* Matplotlib Version 3.2.2
* Scikit Learn (SkLearn) Version 1.0.2
* NumPy Version 1.21.5

**Implementation**

**Library choice**

We had multiple open libraries available for python that made this machine learning implementation easier. These ML libraries (Fast.ai, Keras, PyTorch, TensorFlow, Pandas) all serve a similar purpose in reducing boilerplate code as well as unnecessary complexity. We chose Keras due to it being a high-level API with a gentle learning curve that would ease a lot of the difficulties with optimization and implementation compared with lower-level APIs. For the nature of the assignment, it was a good choice to go with Keras over purely low-level Tensorflow APIs despite the trade-offs. We used the Keras backend that was built into Tensorflow 2. Further, beyond the simplicity of the Keras sequential API, the Tensorflow platform greatly helped with its tooling around data preprocessing. As the MNIST dataset is already a part of the dataset API, it was trivial to load the data and get working on the crux of the model as soon as possible.

**General Workflow**

At a high-level, the procedure can be broken down broadly into four steps:

1. Loading the data from tensorflow.keras
2. Preprocessing the data
3. Implementing the model
4. Training and testing the model

First, thanks to the concise and expressive power of the Tensorflow platform, we were able to load the MNIST data directly in just one line of code. Here, behind the scenes, the tensorflow.keras.dataset API takes the original dataset and partitions it into training, validation, and testing sets, without forcing the implementer to clumsily do these things manually.

**Initial Models**

Multiple models were made so that the changes that occurred during the tuning of hyperparameters to be measured across multiple different models to see if the changes are model specific or if these changes can be generalized to any model. For future projects, knowing this information will allow to quicker design and implementation of high-quality models. To this end, three different models were created to show different levels of complexity and use of layers. The models are described in the following sections and can be seen in figures 1-3.

The main deference between each model varies in the lumber of layers, as well each layer could be a different type. Each layer could be selected from the following types: {Convolutional, Max Pool, Flatten, Dense, Batch Normalization, Dropout}

**Model 1**

The first has a very simple architecture, which can be seen in figure 1. This model acts as a baseline for the other two models and uses SGD as its optimizer. Despite its simplicity, it still achieves an accuracy around 98.57%. This goes to show the simplicity of the MNIST dataset, showing that for small image sizes, a large architecture is not needed. This can also be seen when inception modules are used. The inception modules are much more powerful than a simple sequential model, but because of the dataset, they achieve similar results.

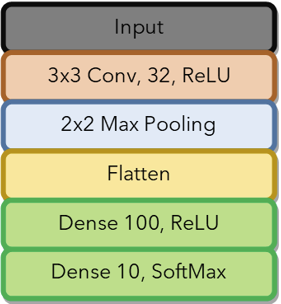


Figure Model 1

**Model 2**

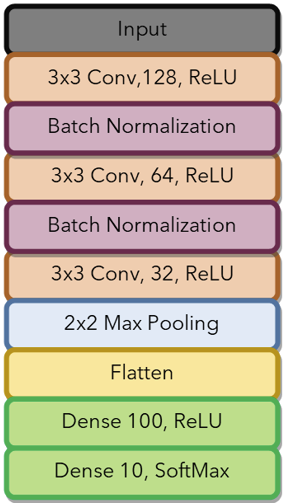


Figure Model 2

Expanding from model 1, model two adds more convolutional layers and introduces Batch Normalization layers that will normalize the activations of the layers so that the mean is close to 0 and the standard deviation is close to 1. Doing so allows for higher accuracy as well improving training time, which allows for more layers to be added onto model 1 without added too much additional time to training. Model 2 continues to use SGD as its optimizer and achieved a base accuracy of 98.79 before the optimization of hyperparameters.

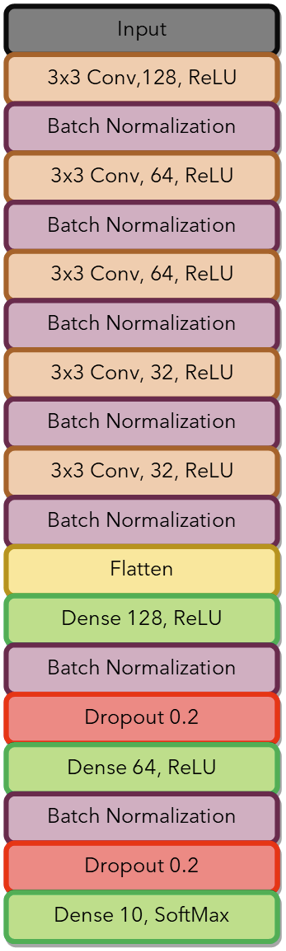


Figure 3 Model 3

**Model 3**

The final model expands even further on Model 2, adding more convolutional layers, batch normalization layers, dense layers, and dropout layers. In this model, batch normalization is also added after the dense layers. Dropout layers help reduce overfitting by randomly dropping neuron activations and adjusting the other neurons so that the sum of all the neurons is the same before and after dropout occurs. By dropping a random number of neurons, it helps prevent the model from overfitting to the training data which would lead to high training accuracies, but low testing accuracies in comparison. Model 3 switches to using Adamax as its optimizer and achieves a base accuracy of 99.38% before the optimization of hyperparameters.

**Experimentation**

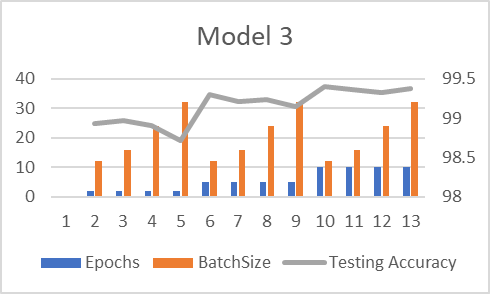
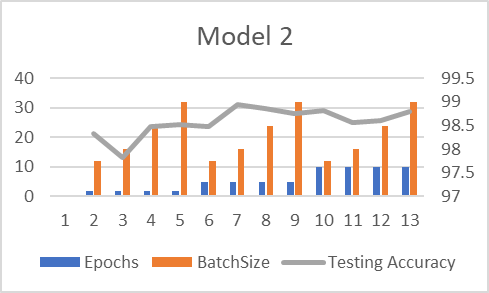
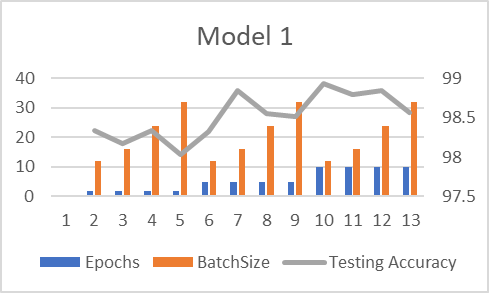


Figure 4 Results of Experimentation

With the three models made, they needed to be tested with some different hyperparameters. The two hyperparameters that we decided to test were the batch size and the number of epochs.

The three models where testing with epochs of 2,5,10 and with batch sizes of 12,16,24,32. The base values for epochs and batch size were 10 and 32 respectively.

**Batch Size**

All three models followed a similar trend where the accuracy would decrease as we increased the batch size. However, the tradeoff was that training took considerably longer with the lower batch sizes. This is expected as the lower the batch size is, the more time the model updates its parameters.

**Epoch**

Changing the epochs behaved very similar to changing the batch size, except for epochs, increasing resulted in higher accuracies and longer runtimes. The effects of increasing the number of epochs had a greater impact on the accuracy of the model than the batch size did.

**Final Model**

The testing gave the following results for the highest accuracy achieved by each model.

|  |  |
| --- | --- |
| Highest Accuracy | |
| Model 1 | 98.93% |
| Model 2 | 98.94% |
| Model 3 | 99.4% |

While all the models gave impressive results with the lowest accuracies during training being around 98% and the highest being achieved by Model 3 with epochs set to 10 and a batch size of 12.

**Comparisons**

Yue Pan from the School of Computer Science at the University of Nottingham experimented using Image Super-Resolution and a residual network on the MNIST database. The main goal was to show that Image Super-Resolution, which is achieved by putting the original image through three convolutional processes to upscale the resolution of the image. This process creates an output resolution that is clearer than that which would be created by bicubic interpolation, another popular method of upscaling images. The need for Image Super-Resolution and bicubic interpolation is that it is beneficial for Convolutional Neural Networks to have high resolution images to train on to allow for features to be found. This can be a problem for datasets like MNIST where the images are only 28\*28\*1. Using a residual neural network, Yue Pan was able to achieve an accuracy of 99.43%. Our model achieves an accuracy very close to this while using a much simpler architecture. However, for larger datasets with higher resolutions, it is very likely that Yue Pan’s model would outperform ours, but in the context of the MNIST dataset, the two models perform similarly.

**Conclusion**

Our model achieves a very high accuracy on the MNIST accuracy, but more important are the trends that were discovered during the testing of the hyperparameters. It was found that a lower batch size generally achieves higher accuracies while increasing training time, while increased epochs raised accuracy and training time. These two trends suggest a general trend that given any model, the longer the model trains, the greater the accuracy it will be able to achieve. Using this knowledge, we could strive to create a more effective model. Further focusing on the third model could result in even higher accuracies. There are many more hyperparameters that were not tested, some being the dropout rate, different optimizers, learning rate of the optimizers and activation functions to name a few. Creating an ensemble of models could also lead to even higher accuracies. There are still many ways that the model could be improved.

**Code**

The implementation of these models can be found in the google colab at the following link: [ADV AI 2 P1 - Colaboratory (google.com)](https://colab.research.google.com/drive/1QiR3i4O1nk9FljcZl3-QTM3PIgeDVUu6?usp=sharing#scrollTo=mp3gYOKbwjbB) A extra sample output can be found in the files tab of the colab.

**References**

*Basic CNN architecture: Explaining 5 layers of Convolutional Neural Network*. upGrad blog. (2021, December 9). Retrieved March 2, 2022, from <https://www.upgrad.com/blog/basic-cnn-architecture/>

*CNN for deep learning: Convolutional Neural Networks*. Analytics Vidhya. (2021, July 23). Retrieved March 2, 2022, from https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/#:~:text=In%20deep%20learning%2C%20a%20convolutional,a%20special%20technique%20called%20Convolution.

Prabhu. (2019, November 21). *Understanding of convolutional neural network (CNN) - deep learning*. Medium. Retrieved March 2, 2022, from https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

Yue Pan 2022 *J. Phys.: Conf. Ser.* **2181** 012027