Predicting handwritten digits using the MNIST dataset

Bjarki Thor Norddahl
Kristiania University College
Oslo, Norway
bjno005@student.kristiania.no

***Abstract*—The study this paper is written for is a submission to a mandatory assignment in a course on Deep Learning. It is written as if this was a real study in order to get better in writing these kinds of papers. The experiments in the results and discussions section are set up for each of the tasks in the assignment.**

In this study, I conducted a series of experiments to improve the accuracy of a Multilayer Perceptron (MLP) model on the MNIST dataset. I compared MLP with 1-D and 2-D Convolutional Neural Networks (CNNs), and found that while the 2-D CNN performed best with an accuracy of 98.1%, the MLP model showed promise with an accuracy of 94.2%. I then altered the MLP model structure, experimented with different data augmentation techniques, and tried different optimizers. The best performing model, which included all the changes in previous experiments, achieved an accuracy of 97%. However, I identified that testing accuracy only on the model from the last epoch could have overlooked potentially better-performing models. Future work will focus on modifying the code to test all models for a more comprehensive performance assessment. This study demonstrates that with further tuning and experimentation, MLP models could potentially rival state-of-the-art models in recognizing handwritten digits from the MNIST dataset.

I. Introduction

Deep Learning (DL) has revolutionized many fields including image recognition. A staple benchmark dataset in this field is the MNIST [3] dataset. The dataset is considered to be "The Hello World of Deep learning" [4] methods within computer vision. The dataset consists of around 70.000 images of handwritten digits from 0 to 9. In this study I aim to train different DL models to classify images of these digits. This paper presents an exploration of various DL models and techniques which can be used to solve this problem. Most standard implementations of neural networks achieve 98-99% accuracy in classifying these digits [2], this study aims to explore what variables and conditions affect the outcome of the models' accuracy.

I will investigate the impact of using three different data augmentation techniques, the effect of varying the number of hidden layers in the models, changing the activation functions, the performance of three different DL model architectures (MLP, 1-D and 2-D convolution) and what effects different optimization methods have on the results. When investigating the impact of these changes the model with the highest accuracy is picked as the starting point for the next set of changes in subsequent models. This is done to maximize the likelihood of creating a model with the highest accuracy while trying to glean some insights into which parts of the neural network have the biggest effect on the outcome of the model.

The rest of the paper is organized as follows: Section II provides a description of the methods I use in this study. Section III presents and discusses the results. Finally section IV concludes the paper.

II. Methodology

This section describes the different DL models and techniques used in this study. The models are trained and evaluated using the MNIST dataset. I use the train and test splits provided by the PyTorch library [3] to split up the dataset. This is done the same way on all models throughout this study. To keep track of the different models run in this experiment I used the Weights and Biases library (wandb). This library allows one to evaluate the results and track the performance of the models in real-time. It allows for tracking of the accuracy and loss of the models while training while makes it easier to see the changes the different conditions described in the introduction have on the models' performance.

A. *Multilayer Perceptron (MLP)*

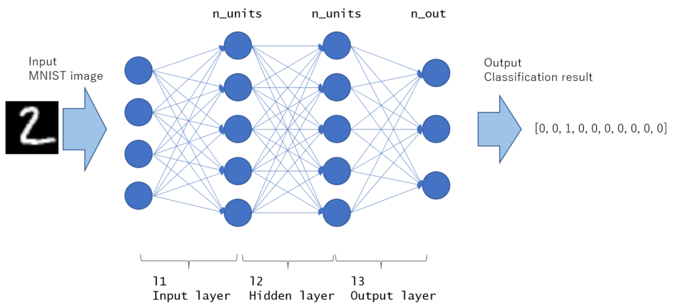


Fig. 1: MLP (Multi Layer Perceptron). From Corochann. (2021, September 28). MNIST training with Multi Layer Perceptron. corochannNote - Deep learning, Machine learning, Android etc. https://corochann.com/mnist-training-with-multi-layer-perceptron-536/

A MultiLayer Perceptron (MLP), seein Figure 1, is a type of feed forward artificial neural network that tries to map sets of input data to sets of appropriate outputs. The network consists of multiple layers of neurons, like the ones seen in Figure 2,

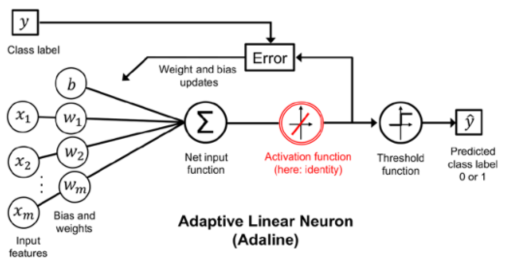


Fig. 2: A single neuron. From Machine Learning with Pytorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python. (p. 339), by Raschka, S., Liu, Y., & Dzhulgakov, D. (2022), Packt Publishing.

in a directed graph, where each layer is fully connected to the next one.

In this study I experimented mostly with an MLP neural network because it is relatively trivial to get a high accuracy with convolutional neural networks (CNN). Changes were made to the MLP model include adding batch normalization, dropout layers, changing the depth of the neural network, using different activation functions and optimization methods to try to achieve the a similar accuracy as a standard CNN. The effects of these changes are discussed in the results and discussions sections below.

B. 1-D Convolution

Convolutional Neural Networks (CNNs) are a type of DL models that are typically used in image recognition tasks. A 1-D convolutional layer uses a mathematical operation to analyze a sequence, usually a temporal or time-step sequence. When using 1-D convolution on images we first have to flatten the input image matrix to a tensor which the kernel or filter, w, 'slides' over. The elements, x^i, are then multiplied with the weights, w^r, sequentially before being summed to produce a single new value, y^i, for each element as shown in Figure 3. The example below shows a convolution with padding size of 0 and kernel size of 4, which changes the size of the output tensor. The padding can be tweaked to control the size of the output feature maps [4].

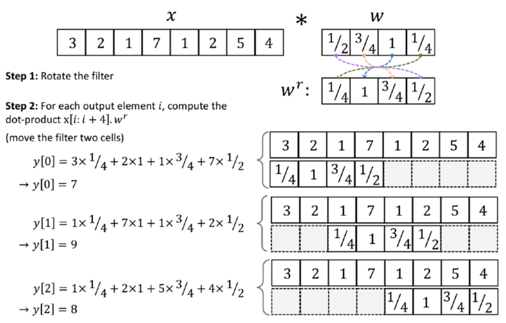

x
*
w
3 2 1 7 1 2 5 4
1/2 3/4 1 1/4
Step 1: Rotate the filter
w^r: 1/4 1 3/4 1/2
Step 2: For each output element i, compute the dot-product x[i: i + 4].w^r
(move the filter two cells)
y[0] = 3×1/4 + 2×1 + 1×3/4 + 7×1/2
→ y[0] = 7
y[1] = 1×1/4 + 7×1 + 1×3/4 + 2×1/2
→ y[1] = 9
y[2] = 1×1/4 + 2×1 + 5×3/4 + 4×1/2
→ y[2] = 8

Fig. 3: Discrete 1-D Convolution. From Raschka, S., Liu, Y., & Dzhulgakov, D. (2022). Machine Learning with Pytorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python. (p. 456) Packt Publishing.

C. 2-D Convolution

2-D Convolutional layers in CNNs work in a similar way as 1-D convolutional layers but they operate on 2-D spatial data such as images. Unlike 1-D convolution which operated on a tensor (flattened input), 2-D convolution preserves the spatial dimensions of the input data (height and width of the digit image). This makes it particularly suitable for tasks such as image recognition, where spatial relationships between the pixels in the images are important.

In 2-D convolution, the kernel (filter) slides of the the 2-D input data, performing an element-wise multiplication with the part of the data that the kernel is currently on, and then summing up the results into a single output (pixel). This operation is repeated for every location on the input volume resulting in a 2-d map out outputs called a feature map [4]. The kernels height and width are parameters that can be tuned.

The process is visualized as shown in Figure 4.

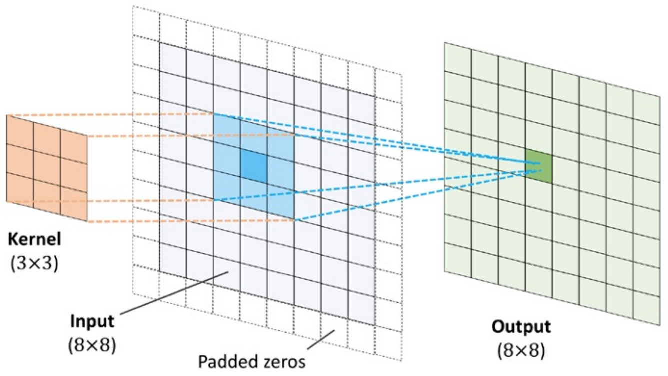


Fig. 4: Visualization of the output of a 2-D convolution. From From Raschka, S., Liu, Y., & Dzhulgakov, D. (2022). Machine Learning with Pytorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python. (p. 460) Packt Publishing.

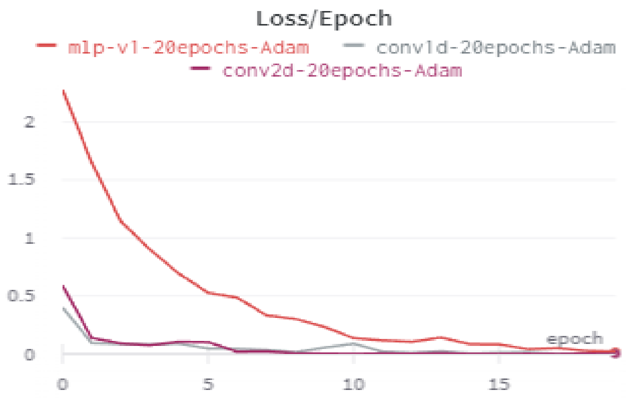
Just like in the 1-D convolution mentioned above, padding of various size can be added to the input to control the size of the output feature maps. Stride, which determines how much the filter moves between convolutions, is another parameter that affects the size of the output feature map.

III. Results and Discussion

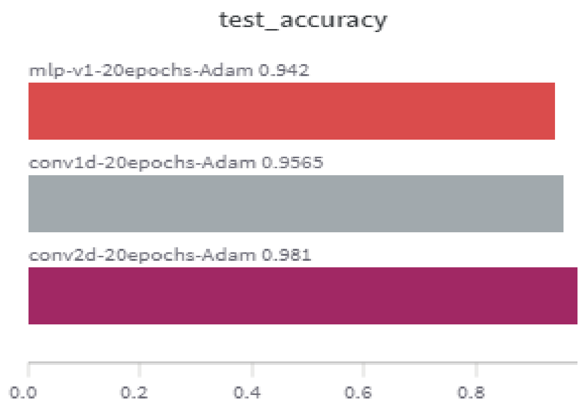
In this section I will go over the different results of the experiments done in this study. All models are initially run with batch size of 128, using the Adam optimize for 5 epochs to make sure there was no fault in the code and that the models were functional, after which the number of epochs per experiments were increased to 20. The aim for this study is to explore and analyze the impact of changes made to the models' structure and the data it takes as input, has on the test accuracy of the models.

A. *Experiment 1: Trying different DL models*

In this experiment I compared the difference between MLP, 1-D and 2-D CNNs. As seen in Figure 5 the 2-D CNN was the clear winner of this experiment, having the accuracy of 98.1%. The 1-D CNN was surprisingly accurate as well, achieving 95.65%. There might be a pattern in tensors that is hard for humans to see but trivial for the CNN. Another potential explanation to this high accuracy is that the network could be memorizing the dataset. I did not investigate this further as that was not the focus of this study. Creating a more accurate 2-D CNN model is trivial as state of the art models achieve anywhere from 99 to 99.8% accuracy. Therefore the MLP model, which achieved 94.2% accuracy, is chosen for the next set of experiments.



(a) Loss per epoch comparison between MLP, 1-D and 2-D CNN models.



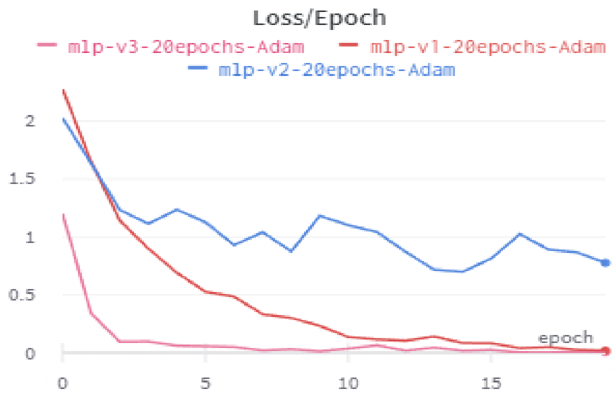
(b) Test accuracy of the three models.

Fig. 5: Accuracy and loss results from experiment 1.

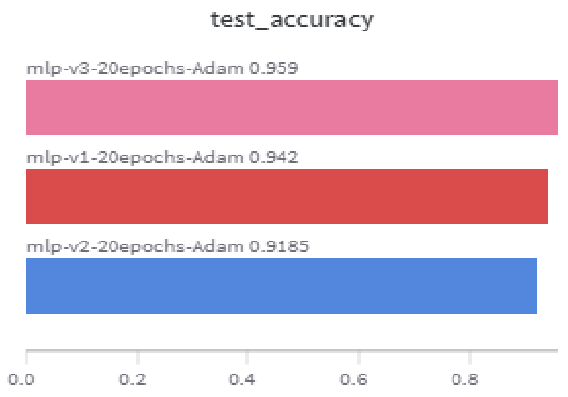
B. *Experiment 2: Changing model structure*

In this experiment I take the structure of the previous experiment and alter it in three different ways by changing the model structures. I adjust the amount of hidden layers, add batch normalization and drop-out layers. In this experiment model with the highest accuracy was the third MLP model. It has batch normalization, no drop-out layers and uses the ReLU activation function in all layers except for the last where it uses a sigmoid function.

As seen in Figure 6, the best model from this experiment improves on the previous accuracy by 1.7% by achieving 95.9% accuracy.



(a) Loss per epoch comparison between three MLP models with different hidden .



(b) Test accuracy for the three models.

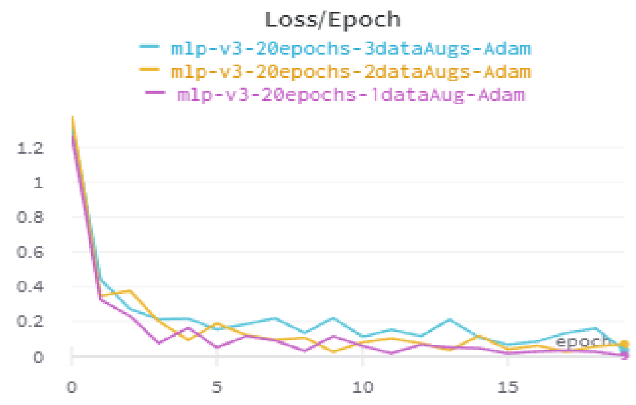
Fig. 6: Accuracy and loss results for experiment 2.

C. *Experiment 3: Data augmentation*

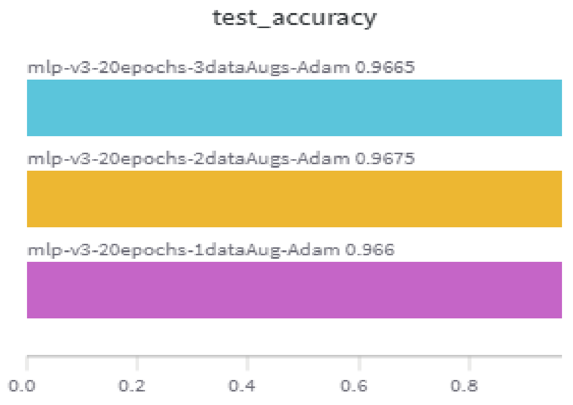
In this experiment the model architecture is not altered, instead the changed variable is the dataset itself. Three different augmentations are made on the dataset. The first data augmentation is rotating the images randomly up to 15 degrees using the "torchvision.transforms" library. My intuition tells me that rotating the images any further could change the meaning of the numbers. For example an eight that was been rotated on it's side can no longer be considered an eight.

The second data augmentation was achieved using the ColorJitter() function. It randomly changes the brightness, contrast, saturation and hue of the image it is used on. In this case the brightness and contrast were changed up to 25%.

The third data augmentation was resizing and cropping the images in a random manner. This was achieved using the RandomResizedCrop() function. The best result was achieved with two data augmentations, random rotation and random adjustments to brightness and contrast. The model achieved 96.75% accuracy improving the accuracy by 0.85% compared to the previous experiments' best model. These two data augmentations are used in all subsequent experiments.



(a) Loss per epoch comparison between the same MLP model run on data with different data augmentations.



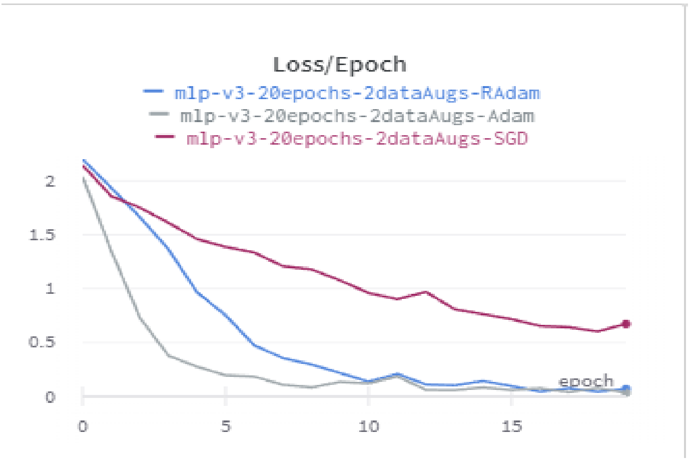
(b) Test accuracy for the three models.

Fig. 7: Accuracy and loss results for experiment 3.

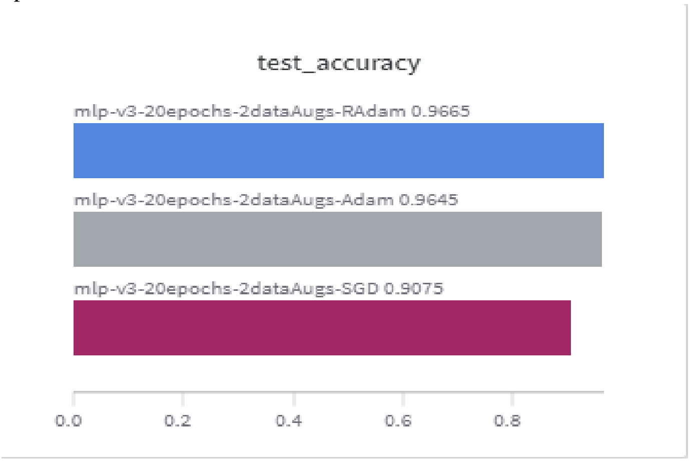
D. *Experiment 4: Using different optimizers*

In this experiment the optimizer method was changed to see if it had any significant impact on test accuracy. As seen in Figure 8, using Stochastic Gradient Descent (SGD) reduced the accuracy significantly from 96.65% to 90.75%. According to scikit-learn [1] SGD requires regularization which could be the reason why it performs so bad on this model.

The difference between the two other models running Adam and RAdam is very small. RAdam performs slightly better with 96.65% accuracy, which is still 0.1% worse than the best model in the previous experiment.



(a) Loss per epoch comparison between MLP models using different optimizer methods.

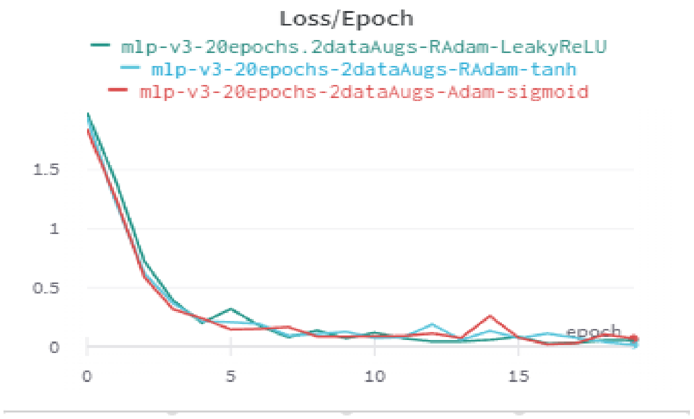


(b) Test accuracy for the three models.

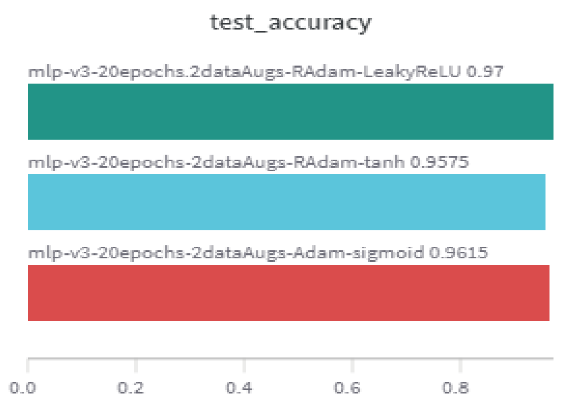
Fig. 8: Accuracy and loss results for experiment 4.

E. *Experiment 5: Changing the activation function.*

In this experiment the activation function in the last layer of the MLP model is changed. The three models use Tanh, Sigmoid and LeakyReLU activation functions in the last output layer. Tanh is said to improve learning [4] but in this case it performs mediocre when compared to the other models. The best performing model achieved test accuracy of 97%. It uses LeakyReLU which permits non-zero gradients for negative input, which makes the network more expressive overall. [4]



(a) Loss per epoch comparison between MLP models with different activation funcitons in the models' last layer.



(b) Test accuracy for the three models.

Fig. 9: Accuracy and loss results for experiment 5.

IV. Conclusion

In conclusion, this study has demonstrated the potential of MLP neural networks in achieving high accuracy rates on the MNIST dataset. Despite the initial accuracy of 94.2%, through a series of structured experiments involving alterations to the model structure, data augmentation techniques, and optimization methods, I was able to enhance the MLP model's performance to achieve an accuracy of 97%. This is a significant improvement, although it still falls short of the 98.1% accuracy achieved by the first 2-D CNN model which had not been improved upon.

One of the key learnings was the realization that testing accuracy solely on the model from the last epoch in the training cycle may not always yield the best results. This approach potentially overlooks models that could have performed better and were not saved or tested.

Moving forward, future work will involve modifying the code to test the best models throughout the training cycle, providing a more comprehensive understanding of model performance. With further tuning and experimentation, I am optimistic that MLP models could potentially rival state-of-the-art models in terms of its accuracy.

References

[1] 1.5. Stochastic gradient descent. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/sgd.html

[2] Gupta, J. (2021, December 14). Going beyond 99% — MNIST Handwritten Digits Recognition. Medium. https://towardsdatascience.com/going-beyond-99-mnist-handwritten-digits-recognition-cfff96337392

[3] MNIST — Torchvision 0.15 documentation. (n.d.). https://pytorch.org/vision/0.15/generated/torchvision.datasets.MNIST.html

[4] Raschka, S., Liu, Y., & Dzhulgakov, D. (2022). Machine Learning with Pytorch and Scikit-Learn: Develop Machine Learning and Deep Learning Models with Python. (p. 339, p. 451-498) Packt Publishing.

[5] Yang, D. (2021, December 9). Tutorial 1: MNIST, The Hello world of Deep Learning. Medium. https://medium.com/fenwicks/tutorial-1-mnist-the-hello-world-of-deep-learning-abd252c47709

[6] Corochann. (2021, September 28). MNIST training with Multi Layer Perceptron. corochannNote - Deep learning, Machine learning, Android etc. https://corochann.com/mnist-training-with-multi-layer-perceptron-536/