CNN's will outperform FFNN's in Standard Image classification
github: `https://github.com/wlucla/GP_F25.git`

Williston Luo

Abstract

While it is well known that CNN's outperform FFNN's for numerous applications beyond standard image Classification, many literary research articles generally provide a convoluted underlying research goal such as physiological data analysis[1]that makes it difficult for beginners such as our group to learn exactly how and why CNN's are better. Thus, our group chose to study and prove that, using the theories taught in class including number of layers, epochs, number of neurons in every layer, learning rate, activation functions, random hyperparameter searches, optimization schemes such as ADAM or mini-batch gradient descent, etc, the top performing FFNN we can fabricate, with our limited compute on CPU, will always under perform a CNN which utilizes spatial information within images.

1 Introduction

Even among the scientific community not too familiar with Neural Networks, most understand Neural network functioning in terms of the FFNN. This can easily be seen on sources such as YouTube, News Articles, and Tech products/ads [2]. However, for beginners such as our group as well as aforementioned community members, it was not immediately obvious that there exists many different types of neural networks beyond the FFNN (until we were shown the neural network zoo in lecture), with each performing best at one particular task. This project introduced to us the CNN, which in the paper provided for this project [3], is cited to be proficient at image classification tasks. As mentioned before, though well-established journals do indeed note the higher performance of CNNs relative to FFNNs for certain tasks including image classification, they treat this information as obvious and never present it in a simple quantitative way (probably because the researchers are much more educated than us in terms of Neural Networks). Thus, our project will attempt to bridge this gap by obtaining a baseline CNN model provided on Github by Amrit Khera [4] and then implementing and tuning our own FFNN to classify images. The result should quantitatively/empirically prove the higher performance of CNNs relative to the FFNN for image classifications.

Before explaining our implementations in technical detail, we would like to justify all techniques that we will be using. First, given that the FFNN inherently expects a vector input while our images are matrices, the only logical conclusion is to flatten the images completely into a column/row vectors. Since our images are 28x28 matrices, this gives rise to a new problem which is that input features are now 784x1 vectors. Our group noticed that this causes training to be intolerably slow on our CPUs (¿3 hours for 100-200 epochs in a hyperparameter random search testing 200 sets of hyperparameters). Since we want to maximize the efficiency of the time we spend together when working on the project, we decided add a Dimensionality reduction module to reduce our input space to 20 dimensions only, which reduced our training on CPUs to about 2 minutes for 100-200 epochs for each set of hyperparameters. Next, because manual searching by intuition for the best hyperparameters is nearly impossible for beginners,

we decided to implement a random searching algorithm where, for each training run, hyperparameters were randomly (with uniform probability) chosen out of a preset range for each hyperparameter. Lastly, our group conversed about the best evaluation metric to choose for our validation and test performance. We came to the conclusion that, because the labels of the MNIST digits dataset [5] are uniformly distributed with no class-imbalance, we would choose Accuracy as our performance evaluation metric.

2 Background

Our group has read and understood the mathematical formulation of the CNN through a great rigorous derivation of the CNN functionality provided online in a textbook and we will provide the basic intuitions of a CNN in terms of image classification, derived from the Deeplearning Book. [6] This book provides a much more in depth explanation of CNN's compared to course textbook. [7] First of all, contrary to the FFNN where the network learns weights, a CNN should be intuitively thought of as learning *Kernels*. For example, an input image (dimensions are **x, y, colors**) will be subjected to convolutions with multiple different kernels in parallel (let there be **n** kernels in total in the first convolution layer) within the first layer to output **n** different feature representations of the image after the first convolutions. A question that arose in our group was: *what is the significance of the convolution operation? Why are you even convolving the image with a smaller kernel as if you were scanning the image?* We came to the conclusion that, images inherently have spatial correlations. As an extreme example, if you were looking at a picture of a centered red ball, the pixels next to the center red pixel are likely also red. Thus, the convolution operation with a kernel is thought of as extracting abstract features from the image. In one convolutional layer, you can extract as many features as you want by defining as many kernels as you need. Furthermore, an important note to make is that the convolution of kernels with image can be written in form of matrix multiplication (likely a huge matrix multiplication). Thus, to prevent a degeneracy to linear regression, we will need to pass the results of the convolutions to nonlinear activating functions. The fact that convolution is functionally equivalent to matrix operations seems very counterintuitive but can be proved with a small example. Let us convolve a *3x3* image $\mathbf{X}$ with a *2x2* kernel $\mathbf{K}$ and see that it could be formulated as matrix multiplication to obtain the output $\mathbf{Y}$.

$$\begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix} * \begin{bmatrix} k_{00} & k_{01} \\ k_{10} & k_{11} \end{bmatrix} = \begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix}$$

$$\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} k_{00}x_{00} + k_{01}x_{01} + k_{10}x_{10} + k_{11}x_{11} & k_{00}x_{01} + k_{01}x_{02} + k_{10}x_{11} + k_{11}x_{12} \\ k_{00}x_{10} + k_{01}x_{11} + k_{10}x_{20} + k_{11}x_{21} & k_{00}x_{11} + k_{01}x_{12} + k_{10}x_{21} + k_{11}x_{22} \end{bmatrix}$$

$$\begin{bmatrix} k_{00} & k_{01} & 0 & k_{10} & k_{11} & 0 & 0 & 0 & 0 \\ 0 & k_{00} & k_{01} & 0 & k_{10} & k_{11} & 0 & 0 & 0 \\ 0 & 0 & k_{00} & k_{01} & 0 & k_{10} & k_{11} & 0 & 0 \\ 0 & 0 & 0 & k_{00} & k_{01} & 0 & k_{10} & k_{11} & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{00} \\ x_{01} \\ x_{02} \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{20} \\ x_{21} \\ x_{22} \end{bmatrix} = \begin{bmatrix} k_{00}x_{00} + k_{01}x_{01} + k_{10}x_{10} + k_{11}x_{11} \\ k_{00}x_{01} + k_{01}x_{02} + k_{10}x_{11} + k_{11}x_{12} \\ k_{00}x_{10} + k_{01}x_{11} + k_{10}x_{20} + k_{11}x_{21} \\ k_{00}x_{11} + k_{01}x_{12} + k_{10}x_{21} + k_{11}x_{22} \end{bmatrix} = \begin{bmatrix} y_{00} \\ y_{01} \\ y_{10} \\ y_{11} \end{bmatrix}$$

Clearly, convolution is a matrix operation, and without nonlinear activation of $\mathbf{Y}$, as said earlier, our model collapses to linear regression. Our activation functions can be applied element-wise

to $\mathbf{Y}$ to introduce nonlinearity. let the nonlinear activating function be denoted *h(...)*.

$$h(\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix}) = \begin{bmatrix} h(y_{00}) & h(y_{01}) \\ h(y_{10}) & h(y_{11}) \end{bmatrix}$$

We may even consider an instance where there are 2 kernels at one convolutional layer. This is quite simple and STILL could be expressed as matrix operations. **Here we can explicitly add a bias too; a bias for each kernel.**

$$\begin{bmatrix} k1_{00} & k1_{01} & 0 & k1_{10} & k1_{11} & 0 & 0 & 0 & 0 \\ 0 & k1_{00} & k1_{01} & 0 & k1_{10} & k1_{11} & 0 & 0 & 0 \\ 0 & 0 & k1_{00} & k1_{01} & 0 & k1_{10} & k1_{11} & 0 & 0 \\ 0 & 0 & 0 & k1_{00} & k1_{01} & 0 & k1_{10} & k1_{11} & 0 \\ k2_{00} & k2_{01} & 0 & k2_{10} & k2_{11} & 0 & 0 & 0 & 0 \\ 0 & k2_{00} & k2_{01} & 0 & k2_{10} & k2_{11} & 0 & 0 & 0 \\ 0 & 0 & k2_{00} & k2_{01} & 0 & k2_{10} & k2_{11} & 0 & 0 \\ 0 & 0 & 0 & k2_{00} & k2_{01} & 0 & k2_{10} & k2_{11} & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{00} \\ x_{01} \\ x_{02} \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{20} \\ x_{21} \\ x_{22} \end{bmatrix} + \begin{bmatrix} b1 \\ b1 \\ b1 \\ b1 \\ b2 \\ b2 \\ b2 \\ b2 \end{bmatrix}$$

$$= \begin{bmatrix} k1_{00}x_{00} + k1_{01}x_{01} + k1_{10}x_{10} + k1_{11}x_{11} + b1 \\ k1_{00}x_{01} + k1_{01}x_{02} + k1_{10}x_{11} + k1_{11}x_{12} + b1 \\ k1_{00}x_{10} + k1_{01}x_{11} + k1_{10}x_{20} + k1_{11}x_{21} + b1 \\ k1_{00}x_{11} + k1_{01}x_{12} + k1_{10}x_{21} + k1_{11}x_{22} + b1 \\ k2_{00}x_{00} + k2_{01}x_{01} + k2_{10}x_{10} + k2_{11}x_{11} + b2 \\ k2_{00}x_{01} + k2_{01}x_{02} + k2_{10}x_{11} + k2_{11}x_{12} + b2 \\ k2_{00}x_{10} + k2_{01}x_{11} + k2_{10}x_{20} + k2_{11}x_{21} + b2 \\ k2_{00}x_{11} + k2_{01}x_{12} + k2_{10}x_{21} + k2_{11}x_{22} + b2 \end{bmatrix} = \begin{bmatrix} y1_{00} \\ y1_{01} \\ y1_{10} \\ y1_{11} \\ y1_{00} \\ y2_{01} \\ y2_{10} \\ y2_{11} \end{bmatrix}$$

Now, consider the fact that a proper image classifier should be relatively invariant to up/down/left/right translations of the image; as an example with the original idea of the image of a red ball, the image classifier should be able to tell whether or not it is a Red Ball even if I present it an image of a red ball slightly shifted by an arbitrary number of pixels. Clearly, an extra step after nonlinear activations is required to introduce this 'translational invariance'. This extra step is known as 'pooling' and happens immediately after the nonlinear activations. There are may forms of pooling, one example would to be take, for example, 2x1 segments of the features (outputted by the first convolutional layer and nonlinear activation function), and retain only the maximum value. Like below.

$$Pool(\begin{bmatrix} h(y1_{00}) & h(y1_{01}) \\ h(y1_{10}) & h(y1_{11}) \end{bmatrix}) = \begin{bmatrix} max[h(y1_{00}), h(y1_{10})] & max[h(y1_{01}), h(y1_{11})] \end{bmatrix}$$

Now, consider the fact that a proper image classifier should be relatively invariant to up/down/left/right translations of the image; as an example with the original idea of the image of a red ball, the image classifier should be able to tell whether or not it is a Red Ball even if I present it an image of a red ball slightly shifted by an arbitrary number of pixels. Clearly, an extra step after nonlinear activations is required to introduce this 'translational invariance'. This extra step is known as 'pooling' and happens immediately after the nonlinear activations. There are may forms of pooling, one example would to be take, for example, 2x1 segments of the features (outputted by the first convolutional layer and nonlinear activation function), and retain only the maximum value. Like below.

$$Pool(\begin{bmatrix} h(y_{00}) & h(y_{01}) \\ h(y_{10}) & h(y_{11}) \end{bmatrix}) = \begin{bmatrix} max[h(y_{00}), h(y_{10})] & max[h(y_{01}), h(y_{11})] \end{bmatrix}$$

Thus, even if your input image shifts a little by a few pixels (convolution with a kernel is invariant to shifting, so your result of the convolution is just a shifted version of the result of convolving

the kernel with the unshifted image: aka the nonlinear activations will just be a little shifted too), after pooling, the results will be relatively the same. Intuitively, one can think of this as invariance to translation. Next, the logical question our group thought of was, *"But what about deformations of the image that aren't necessarily simple translations? What if I squish the image or rotate it? Can the CNN still classify the image?"* A necessary thing for us to understand is; a network can only classify squashed and rotated images if you've presented examples of these images to the network during training (and if you have enough kernels). If you've done this, there will be learned kernels that account for the detection of such variations of the image. Thus, you must allow for sufficient channels (of kernels) in each convolutional layer to learn all these different possible variations of the input image. One thing to note that, because we have pooling too, when combined with the features learned by the kernels, we can have, for example, network invariance to translations and rotations simultaneously. Next, after many convolutional layers, we end up with many pooled vectors; this is only part of the story, since the ultimate goal is to classify the input image. What we can thus do is to stack these vectors on top of each other to form a large feature vector, which we can then feed into a FFNN that classifies these features via a softmax output. The way a CNN updates its weights is via backpropagation, just like for the FFNN. The derivation is not complicated, but we are appreciative that the Professor has spared us from typing out the derivation in latex.

3 Dataset

As previously mentioned, we will be using the MNIST dataset [5]. The dataset contains hand drawn digits without class imbalance, and thus allowed us to use Accuracy as our metric of model performance evaluation. There were a couple of steps for data pre-processing. First of all, a train_val_test_splitter function was written to divide the MNIST's csv formatted dataset into a train_data.csv, test_data.csv, and validation_data.csv. Based on our group's work, we decided to reduce the number of features from 784 dimensions to 20 dimensions to speed up training. We wrote a function pca_training to write a new csv file called DR_train_data.csv, which contains the dimensionally reduced dataset. To discourage data leakage, we wrote a separate function pca_test_val to perform PCA on the test_data.csv and validation_data.csv, *(USING THE MEAN AND COVARIANCE MATRICES FROM TRAINING DATA FOR NO DATA LEAKAGE)*. For verification, we implemented a dimensionality_reduced_visualizer that can visualize our Dimensionally reduced datasets. After visually confirming that the dimensional reduction was successful, we move on to building our own FFNN model. Figure 1 shows the handwritten ground truth integer, and Figure 2 validates that our PCA was successful by visualizing the same integer through our dimensionally reduced orthogonal eigenbases.

4 Model And Methodology

Our group will be implementing our FFNN modules from scratch (not pure scratch, we still will use PyTorch and sklearn). The CNN we will be implementing is in a Jupyter Notebook provided by Amrit Khera [4]. In the FFNN_BLACKBOX.py script, we fully implement a toolkit necessary to build a FFNN with accuracy evaluation, data loaders, model initializations, training functions, and evaluation functions. Load_Data will take the DR_train_data.csv, DR_test_data.csv, DR_val_data.csv (refer to Dataset section for more about what these CSVs are) and return the features and labels in separate variables. Next, create_dataloader will take the outputs of Load_Data and return a train_loader, val_loader, and test_loader. The point of creating loaders is because during our iterative optimization, PyTorch can easily iterate between features and labels of different batches without us implementing it. Our FFNN class implementation inherits from torch.nn.Module, where we bestow upon it fully connected Linear layers and a choice of

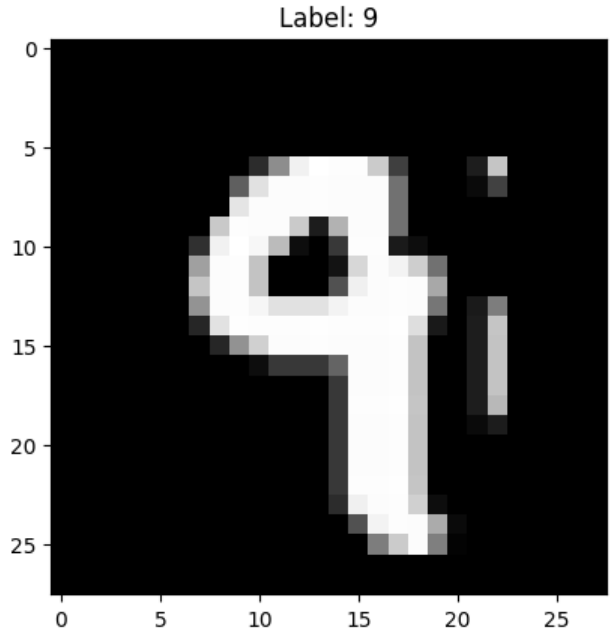

Figure 1: Here is the ground truth handwritten '9'

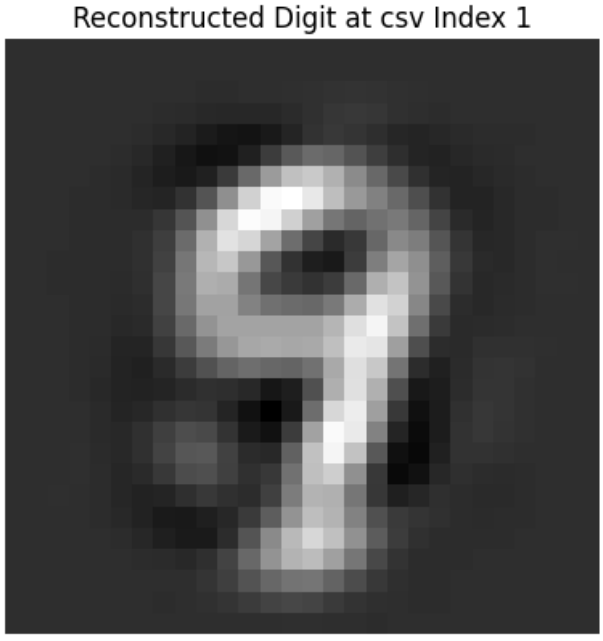

Figure 2: Here is the Same data reconstructed from 20 Principle Components derived from the Training Data's covariance matrix and mean.

activation function between ReLU, tanh, and sigmoid. It took a significant time to figure out, but FFNN_BLACKBOX successfully granted us the ability to initialize FFNN's of arbitrary hyperparameters. Since our ultimate goal is digit classification, our error function can only be

Cross Entropy loss. Furthermore, we discovered that the standard mini-batch gradient descent is very unstable, and sometimes returns a validation accuracy of less than 10 percent which is worse than random guessing. Thus, we chose a gradient with momentum and standardized weight updates: the Adam optimizer. Our training function takes the training and validation loaders, model, optimizer, loss function, and number of epochs as parameters and outputs a plot of training accuracy and validation accuracy as a function of epochs. Lastly, we wrote a separate function that evaluates our model performance on the test data set called test_set_eval.py which will evaluate our model performance using the DR_test_data.csv. Please refer to the README for the exact steps to reproduce our results presented below.

5 Results

A few results during the Hyperparameter Random Search are shown here for our own FFNN. It is evident that some hyperparameter perform better than others. In Figure 3, the best hyperparameter achieved by the random search were: Learning Rate=0.013494565585173276; Number of Hidden Neurons=201; Number of Hidden Layers=1; Activation Function =ReLU; Batch Size=184. **Though we did not test sub-optimal hyperparameter on the Test Data set, we provide Validation Accuracy plots for FFNN's initialized with suboptimal hyperparameter sets.** In general, these are prone to very poor performance, and in some instances perform as if the Network were randomly guessing the output class for input digits with an accuracy of $\tilde{1}0\%$(Figure 4). In other bad-hyperparameter initiations of the FFNN, the Validation Accuracy did steadily increase but only to around 90% (Figure 5). Clearly, we cannot show all figures from our hyperparameter random search, but even with our limited compute, this relatively constrained hyperparameter random search was able to yield a test accuracy of $\tilde{9}4.5\%$.

The CNN provided by an MIT license [4]performed at a higher test and validation accuracy IN ADDITION to faster training compared to the FFNN hyperparameter random search. On the same dataset, a batch size of 100 for the CNN trained only for 1 epoch already significantly outperforms our Hyperparameter-tuned FFNN, able to achieve 98.7% validation accuracy and 96% test accuracy. Furthermore, this 1 epoch of the CNN, without any dimensionality reduction techniques, took only 5 minutes to run on a laptop CPU. In comparison, the hyperparameter random search of the FFNN took 50 minutes (even with the data reduced from 784D to 20D) to run and the "optimal hyperparameters" found still underperformed this single epoch of the CNN.

6 Conclusions and Discussion

After an extensive random hyperparameter search, we have quantitatively proved that CNN's not only outperform FFNN for image classification; CNN's also use significantly less compute than the hyperparameter random searches a FFNN requires to achieve $\tilde{9}4.5\%$ test accuracy. Thus, we have proved that different types of neural networks are good at a particular set of tasks. Here even if we used PCA to reduce the dimensionality of the features that we train our FFNN network on in hopes it would reduce the training time, due to the fact that CNN's are inherently better at learning image features, our FFNN will always underperform the CNN. Hence, we accomplished our goal of providing a empirical and quantitaive justifciation of the primary fact most Machine Literature gloss over–that CNN's outperform FFNN's in tasks of image classification because CNN's are mathematically formulated to recognize image features (kernels) while FFNN's inherently take only vectorized inputs, not images.

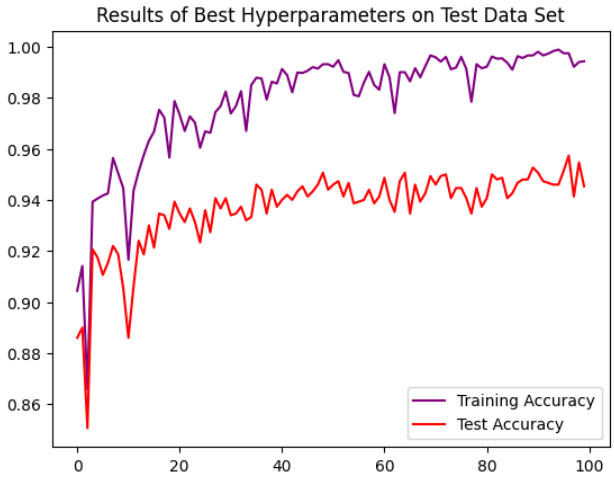

Figure 3: Our FFNN results. Final Test Accuracy=94.5%. Final Training Accuracy=99.4%. X-Axis=Epochs. Y-Axis=Accuracy

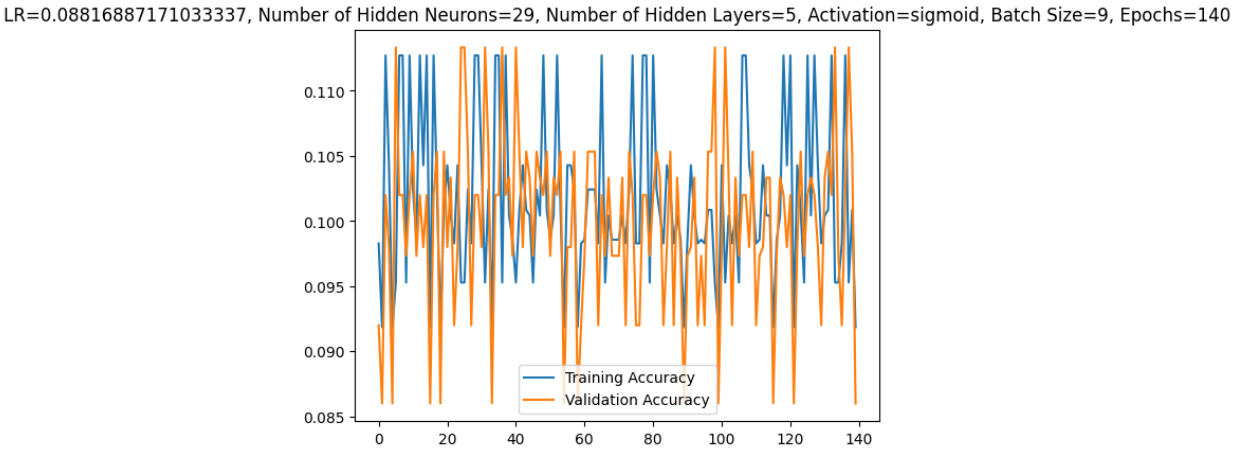

Figure 4: Poor Hyperparameter choices reduce performance in our FFNN. Validation accuracy of these hyperparameter are so poor it was as if the network were randomly guessing one of ten digits as the output. X-Axis=Epochs. Y-Axis=Accuracy

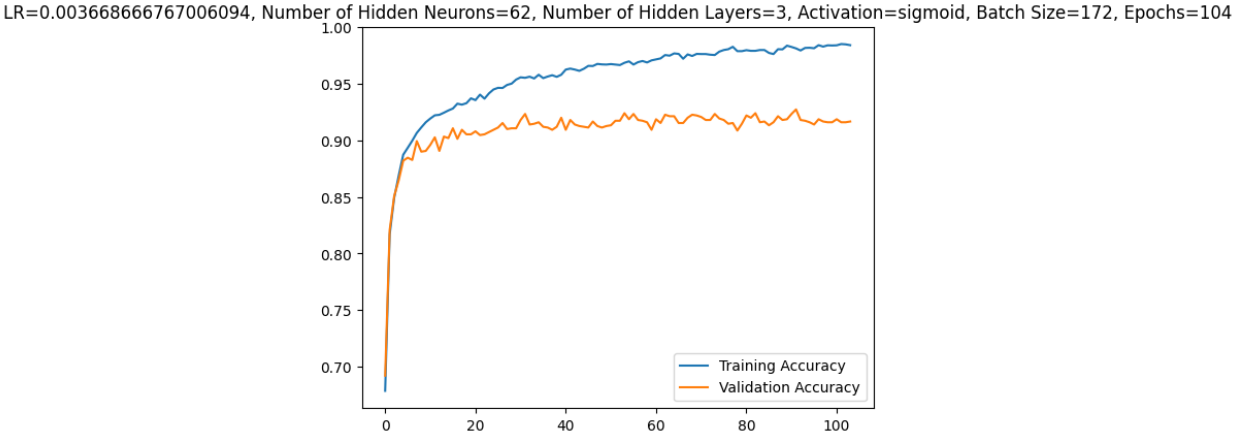

Figure 5: These Suboptimal Hyperparameters for our FFNN generate a smooth Accuracy curve but plateaus around 90% validation accuracy. X-Axis=Epochs. Y-Axis=Accuracy

7 Acknowledgments

We'd like to thank Dr. Diepeveen for discussing with us about the contents of our proposal so we could get started on this project. Also, we are appreciative of many of the clarifications he's provided, such as how we don't need to type out how backpropagation in CNN's work and it is sufficient to understand the basics of image convolution and how it fits into a CNN's convolutional layer.

8 Very Important

In this project, we've accomplished our primary goal exactly as stated in our proposal, however, because we've never rigorously derived the functionality of CNN's in class, what we believe we can do better is heavily derived from this fact: our group members read online-textbooks that rigorously prove the functionality of the CNN, but at the end we never had time to play around with as many mechanisms of the CNN as we'd have liked. Other than this, our project accomplishes exactly the proposal, and we even included dimensionality reduction as an extra step in our implementation.

References

[1] Mao, S., & Sejdic, E. (2023). A Review of Recurrent Neural Network-Based Methods in Computational Physiology. IEEE transactions on neural networks and learning systems, 34(10), 6983–7003. https://doi.org/10.1109/TNNLS.2022.3145365

[2] Emergent Garden (Author). (2023, August 17). Watching neural networks learn [Video]. YouTube. https://www.youtube.com/watch?v=TkwXa7Cvfr8

[3] O'Shea, K., & Nash, R. (2015, December 2). An Introduction to Convolutional Neural Networks (arXiv:1511.08458v2). arXiv. https://arxiv.org/pdf/1511.08458

[4] AmritK10. (2018, Decemeber 13). MNIST-CNN [Computer software]. GitHub. https://github.com/AmritK10/MNIST-CNN.git

[5] Kim, Oh Seok., Pascal., Dato-on, Dariel. (2017. MNIST in CSV [MNIST Dataset]. Kaggle. https://www.kaggle.com/datasets/oddrationale/mnist-in-csv/data

[6] Goodfellow, I., Bengio, Y., & Courville, A. (2016). In Deep Learning (Convolutional neural networks, Chapter 9). MIT Press. https://www.deeplearningbook.org/

[7] Bishop, C. M. (2006). Pattern recognition and machine learning (Chapter 5.5). Springer.