Final Project Tutorial

Members: LTC Olha Danilova, MAJ Janet Karanja, LT Charlotte Meyer, and LT Kurt Pasque

Agenda:

- Introduction
- Data
- Neural Networks
- GitHub
- Local vs. Hamming CPU
- Output Analysis
- Future Work

Introduction

Research Question

Neural network models are frequently used in classification problems, but their performance on metrics like accuracy and robustness are highly dependent on the hyperparameters that users set when constructing the model. Given this, how can we run multiple tests of different parameters in parallel using the high-performance computing (HPC) cluster here at NPS to find optimal hyperparameter settings of a neural network.

Motivation

"*Right now, there are hundreds of engineers in the bay area being paid lots of money to make tiny tweaks to lambda values [a hyperparameter of regularized neural networks] on their models.*" - Professor Johannes Royset said this a number of times during this cohort's OA4201 - Nonlinear Programming. We are hoping to show a way we can test and tweak hyperparameters we care about using Hamming!

Additionally, there are a number of operations research applications on the models we encounter and use frequently, from simulation models, to optimization models, to data analysis algorithms. We hope this project shows a non-trivial example of experimenting with hyperparameters that could be utilized in many aspects of operations research, specically our thesis efforts here at NPS!

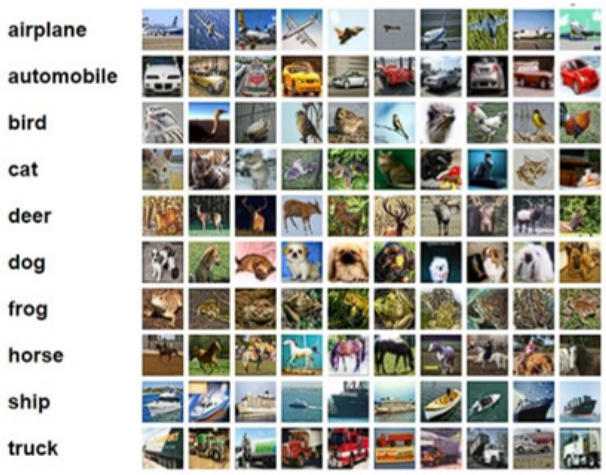
Extending from NPS, this mass testing of model hyperparameters might be needed for a number of models this cohort will build and run for the DoD and this method (*coupled with some computing resources*) can help ground our hyperparameter decisions through testing.

The Experiment

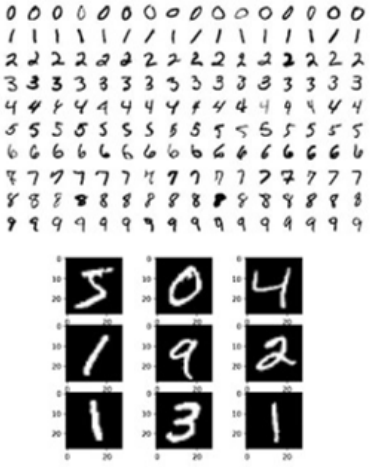
Using 2 open-source image datasets and using 1 neural network model structure, we will test model accuracy by adjusting 1 hyperparameter that affects the model structure and perform some simple analysis on its performance. We hope to display how this methodology enables us to speed up experimentation, enabling us to test many facets of our models at a large scale.

Data

The below images show a small sample of the datasets we are using for our experiments:



CIFAR



MNIST

Overview of Each

- CIFAR:
  - http://www.cs.toronto.edu/~kriz/cifar.html
  - The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class.
  - The 10 classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are mutually exclusive.
  - There are 50000 training images and 10000 test images.
  - The dataset is divided into five training batches and one test batch, each with 10000 images.
- MNIST:
  - https://www.kaggle.com/datasets/hojjatk/mnist-dataset
  - The MNIST database of handwritten digits
  - Training set of 60,000 examples, and a test set of 10,000 examples.

Pro's about these datasets

- Each dataset is publicly available and we obtained them via simply downloading from the source website defined above to our local machine.
- Given how many images are contained within each of these datasets (well over 200,000 images), they are already stored in very compact file types.

Challenges with these datasets

- Even though basic data within each dataset is the same, they are each stored in different and obscure file types at the point of download. Custom methods for each dataset had to be constructed that could load the data into memory and adjust them to be in the appropiate input data structure we needed. All these methods needed are contained within our `functions/load_data.py` file (more on our experiment file structure later).

Neural Networks

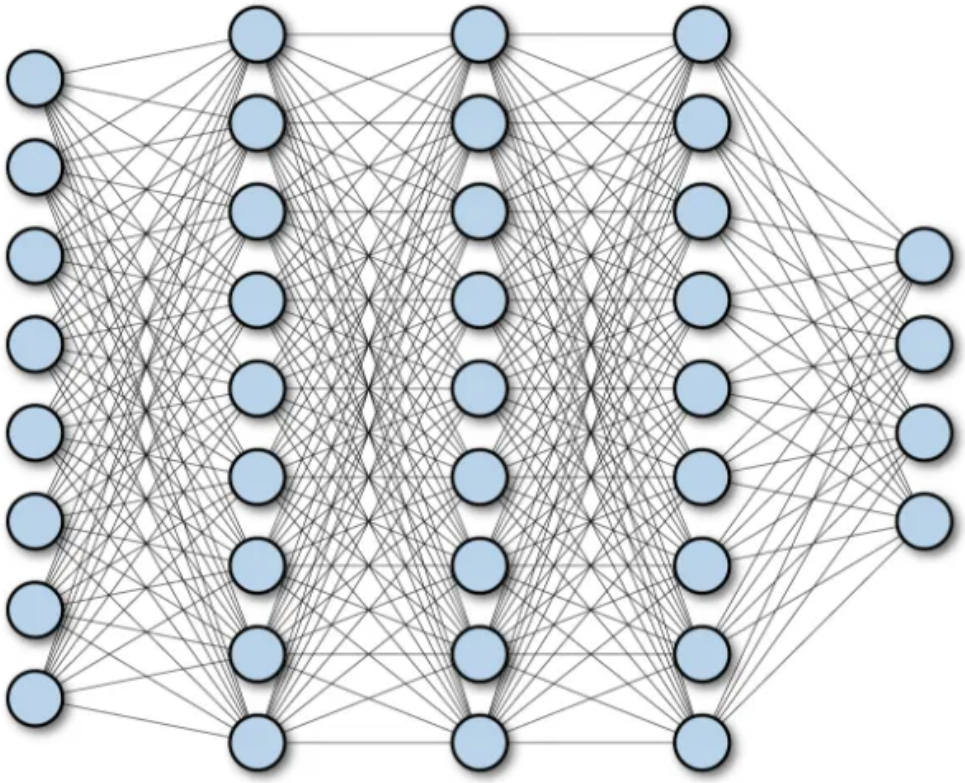
The model we will employ is a neural network in Python using the Tensorflow package. Below is an intro to neural networks, intro to the type we use, and then finally an overview of our specific neural network model structure.

Brief Overview of Neural Networks

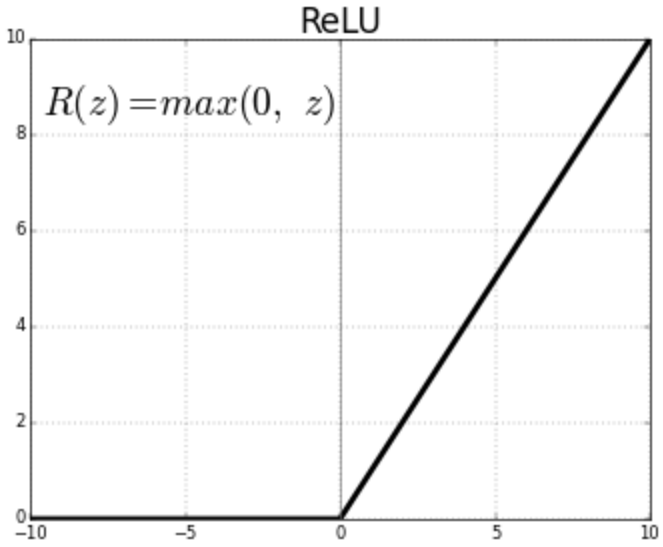
- *High-level overview*: Neural networks are mathematical models inspired by the human brain, composed of interconnected nodes (neurons) that process and transform input data through layers to learn patterns and relationships, iteratively adjusting weights and biases, usually with the goal to optimize a predictive task.

- *How neural networks "learn"*: Neural networks utilize weights and biases assigned to connections between neurons, initially set randomly, then adjusted iteratively through techniques like backpropagation, where gradients of an error function are computed with respect to these parameters using techniques such as gradient descent, enabling the network to learn by minimizing prediction errors and updating these weights to better represent the relationships in the data.

- *How layers are connected*: Below is a visual depiction that is illustrative of how for a fully connected neural network, each neuron from one layer feeds into the next layer (image source: https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5):



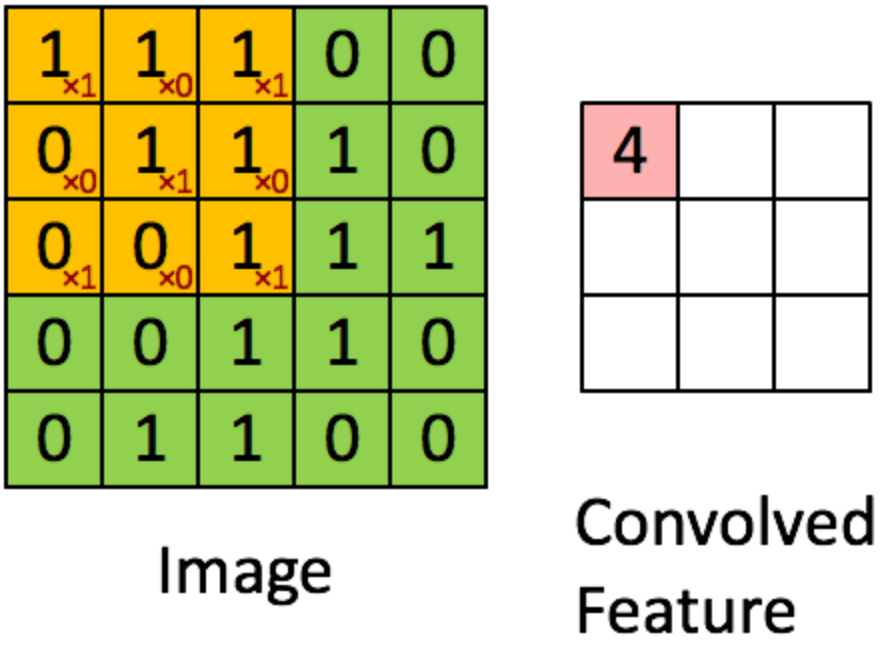
- *Activation Functions*: A key element of neural networks is the activation function. Activation functions in neural networks introduce non-linearities to the output of each neuron, allowing the network to model and learn complex relationships within data by enabling it to capture non-linear patterns. Without activation functions, neural networks would reduce to a series of linear transformations, limiting their capacity to learn and represent intricate patterns in data. Below is a common activation function that we will use called a rectified linear unit (ReLU). The function is simply the max between 0 and a given number. This means any negative number will pass a 0 into the next layer of the neural network, causing our desired non-linearity (image source: https://www.analyticsvidhya.com/blog/2022/03/a-basic-introduction-to-activation-function-in-deep-learning/):



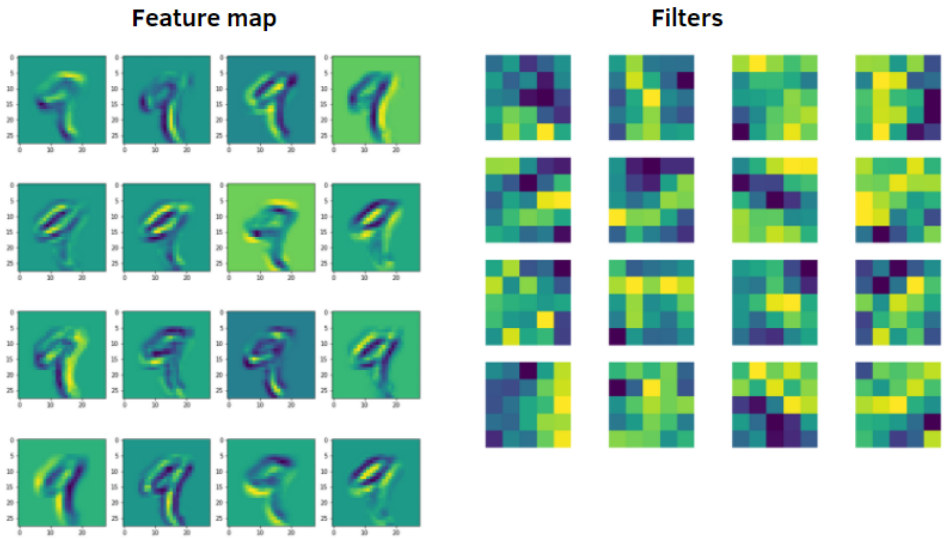
Brief Overview of Convolutional neural networks (CNN)

- *Abstract idea behind CNN's*: Different, small sets of parameters (referred to either as filters or kernels) are optimized and "learn" different patterns and spatial features of input data. For images, this could mean a filter "learns" where vertical edges are and another might "learn" a certain texture pattern and so on. These learned filters are then able to provide "maps" of where their feature occurs within an image, which is highly useful to classifying many types of data like images.

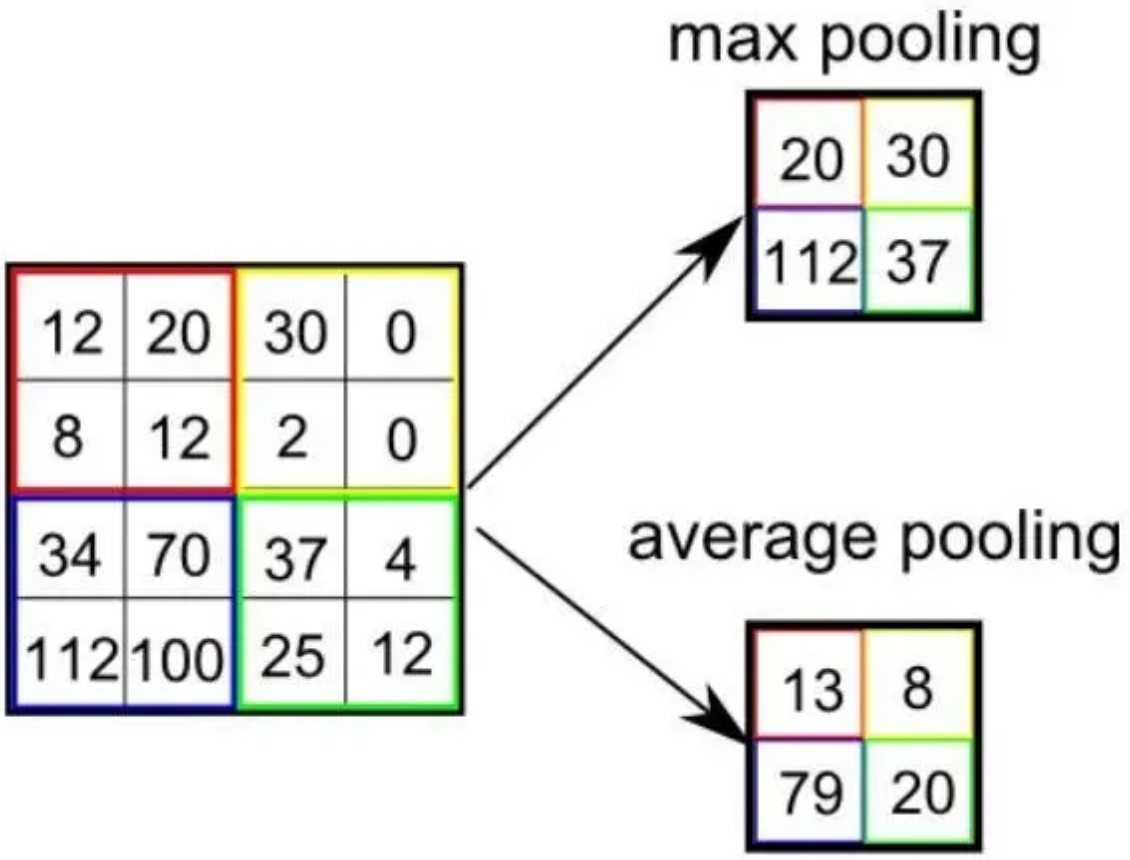
- *The convolution operations*: The convolution operation itself effectively "slides" or "convolves" the fitlers across the input data, extracting features through localized information aggregation and placing the results into a "map". Below is a nice 2-D visualization of how the convolution operation works and generates a feature map. Please not that the filter does not change while it slides (image source: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/):



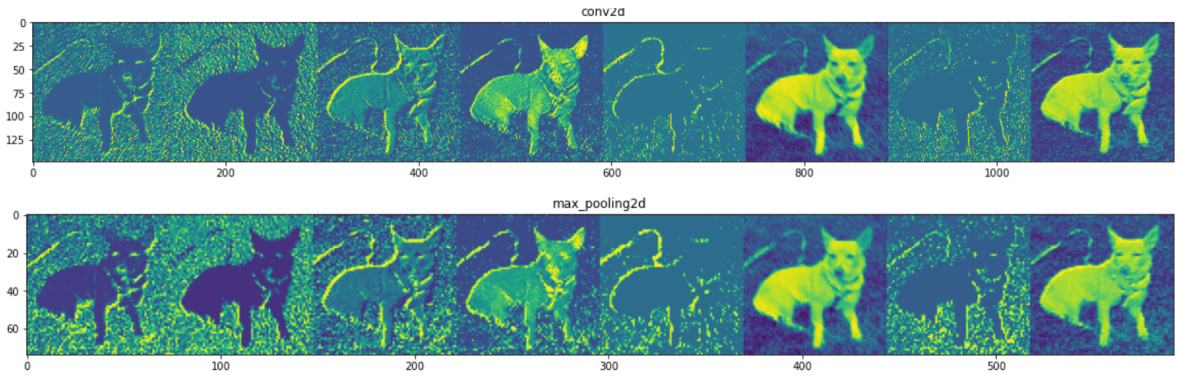
- *Visualizing Feature Maps*: Below is a visual depiction of a feature map after having 16 fitlers applied to 1 MNIST image (image source: https://medium.com/dataseries/visualizing-the-feature-maps-and-filters-by-convolutional-neural-networks-e1462340518e):



- *Pooling layers*: Another key aspect of many CNN's is pooling layers. There are no parameters in these layers as they simply try to reduce the feature maps into only the most important features. This primarily serves to reduce the dimensions the network is processing (enabling fewer trained parameters) as well as make the model more invariant to effects of translation (where important objects in the image appear in different parts). Below is an easy visual of how pooling works in the 2-D settings (image source: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/):



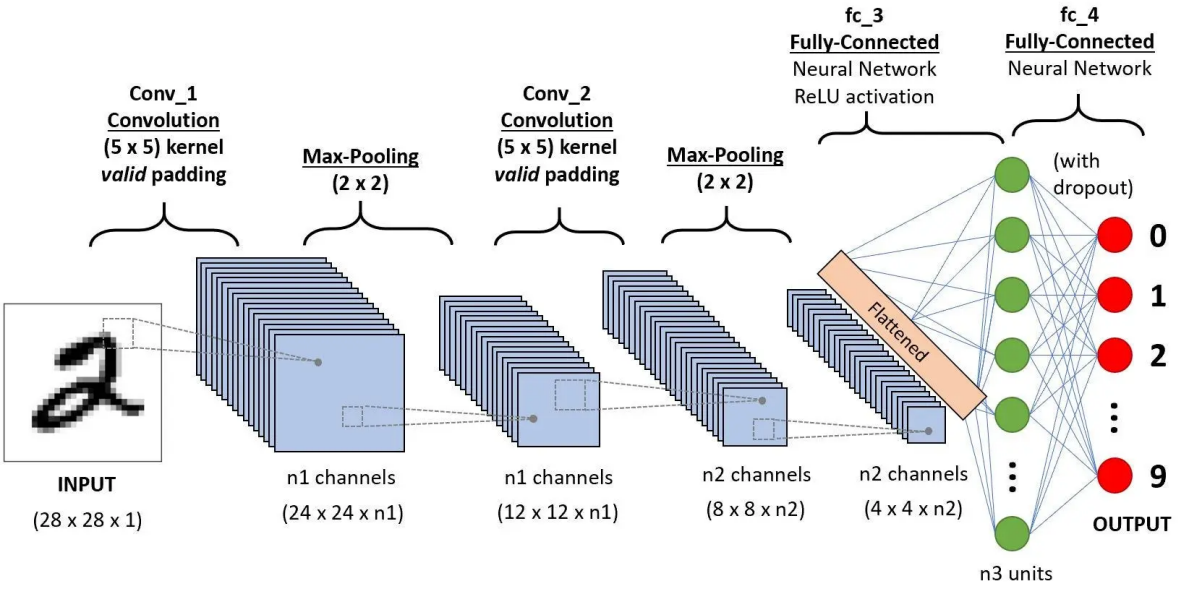
- *Convoluation then Max Pooling Visual*: To cap off the dicussion on how convolutions and pooling layers work, the below visualization shows what each "feature map" looks like for a an image of a dog from a trained neural network (image source: https://www.analyticsvidhya.com/blog/2020/11/tutorial-how-to-visualize-feature-maps-directly-from-cnn-layers/)



- *Further reading/background*: this article provides a great summary of CNN strucutures and various operations used within: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

Brief Overview of the Neural Network Model we Used

If you ask ChatGPT to write you a good, simple-ish neural network model in Tensorflow for classifying images in our 2 datasets, you will like get the model that we went with. The model has 3 convolution layers, 2 max-pooling layers, and 2 fully connected layers. The below image is the best visual for our model we could find. The main differences are the "window" sizes for our layers are different in our model, and we have 1 more convolution layer after the 2nd max-pooling layer (image source: https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/):



- *Tensorflow implementation*: Below is a screenshot of our code for the model used as written in Tensorflow.

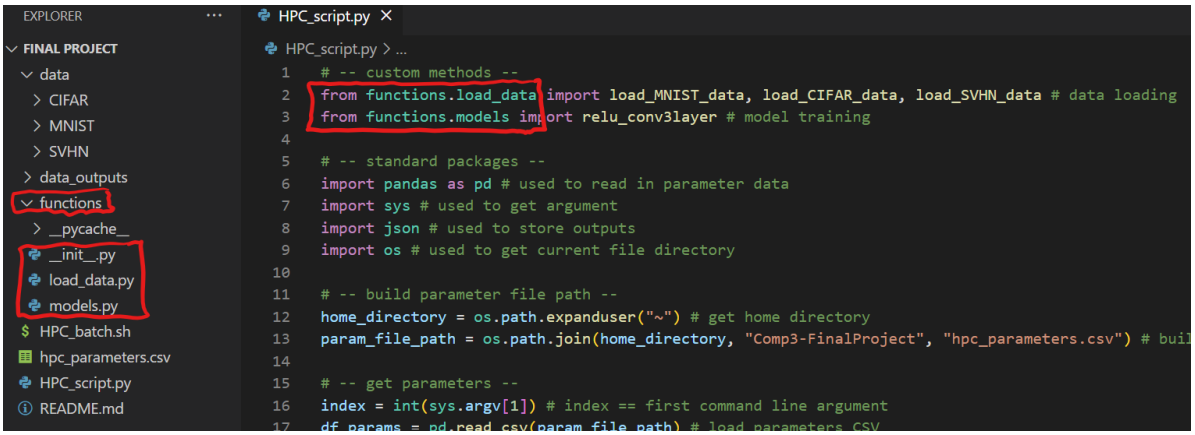
```
model = Sequential([Conv2D(num_first_filters, (3,3), activation='relu'),
                    MaxPooling2D((2, 2)),
                    Conv2D(64, (3, 3), activation='relu'),
                    MaxPooling2D((2, 2)),
                    Conv2D(64, (3, 3), activation='relu'),
                    Flatten(),
                    Dense(64, activation='relu'),
                    Dense(10, activation='softmax', kernel_initializer = initializers.RandomNormal(mean=0.5, stddev=1., seed=0))])
```

- *What are we testing exactly?* As you can see in our code, the number of filters we are using in our first layer is a variable. This is the parameter we will be adjusting in our experiment from 1 to 65 in steps of 2. Using less filters in a model reduces the number of parameters to be trained which directly affects the size of the follow-on layers, training time, and overall complexity. Thus, this is an important hyperparameter to optimize so we get enough predictability, while trying to minimize our number of filters.

GitHub

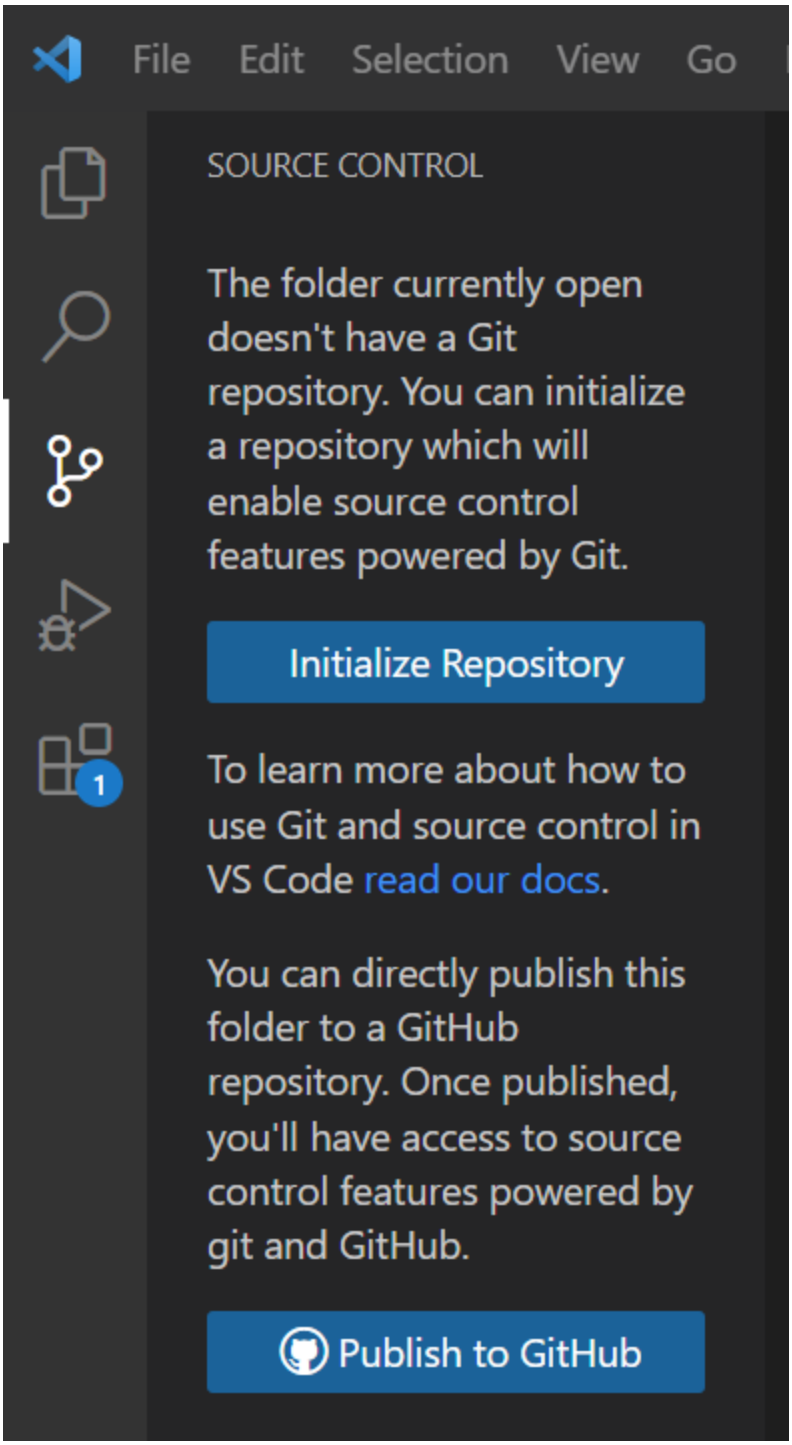
We have created a GitHub repository (repo) that contains the data, functions, and scripts needed to run our final project files in our HPC environment. The goal of the repo is to hold our scripts and data in a way that can be used by any user. Here is a link to our GitHub repo, with instructions for how to use the project in Hamming on the README.md: https://github.com/KurtPask/Comp3-FinalProject

How the Repo is Structured: We tried to use best practices in building our final experiment script, by storing some of our more complex and repeated operations as functions within sub-folders. This enables us to read in each function we need as an import and simplifying the readability of our code and enabling to scale up into more experiments down the road, since we have already built our foundational functions for processing data and building models. Below is a screenshot of our file structure as it appears in Visual Studio Code as well as how we actually import our functions in our main script. Its worth pointing out that if you want to set up your files like this, you must include the `__init__.py` file in your subfolders (ours are empty).



How we Put the Repo on GitHub: The Visual Studio Code IDE has a feature we used that can help you (1) wrap a folder as a git repository and (2) publish it to your GitHub account. If follow-on

work is done in Visual Studio, there are buttons that can simplify adding, committing, and pushing changes to GitHub. You will have to tweak your setup to ensure your username and other information is store in your .gitconfig file to make this work. Visual Studio has some instructions to help you do this if you try to publish with the wrong or no information in your .gitconfig. Below is a screenshot of how we first setup the repo from our machine:



For further study on how to use git and github for the work we do here, we highly encourage watching this video put together by a biomedical researcher: https://www.ryan-alcantara.com/projects/p90_Github_Tutorial_for_researchers/

Running our Experiment in Hamming

To run the experiment, we only needed to run 3 bash commands (its a public repo, so anyone can do it!):

- git clone https://github.com/KurtPask/Comp3-FinalProject
- cd Comp3-FinalProject
- sbatch HPC_batch.sh

After running these commands, the monitoring of the run and the subsequent analysis on the results were more manual, but it just goes to show the benefit of utilizing git repos when working on research teams here at NPS.

Local vs. Hamming CPU

Local

We did not do a full test of all 100 hyperparameter tests on a local machine, but we ran the first 2 and the last 2 parameter settings in our `hpc_parameters.csv` file. After making some broad assumptions on how long the runs would take in between, our estimated local run time would be **~21 hours**, ouch!

Hamming CPU

Implementing our script by running each hyperparameter test in parallel on Hamming resulted in a runtime of **~84 minutes**, 15 times faster than our local implementation!

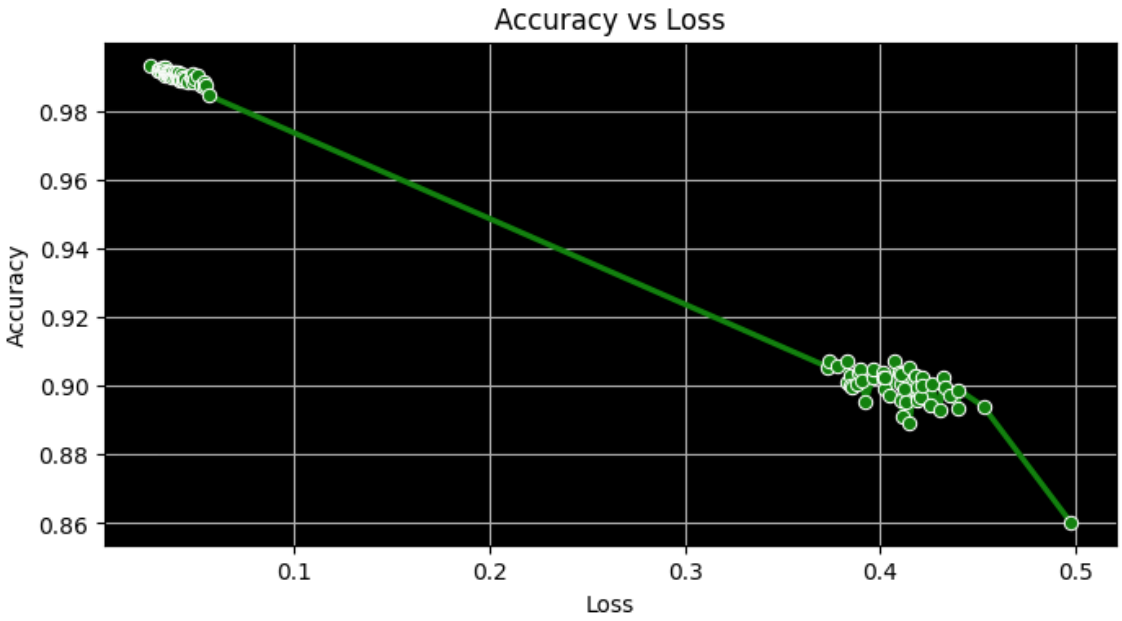
Output Analysis

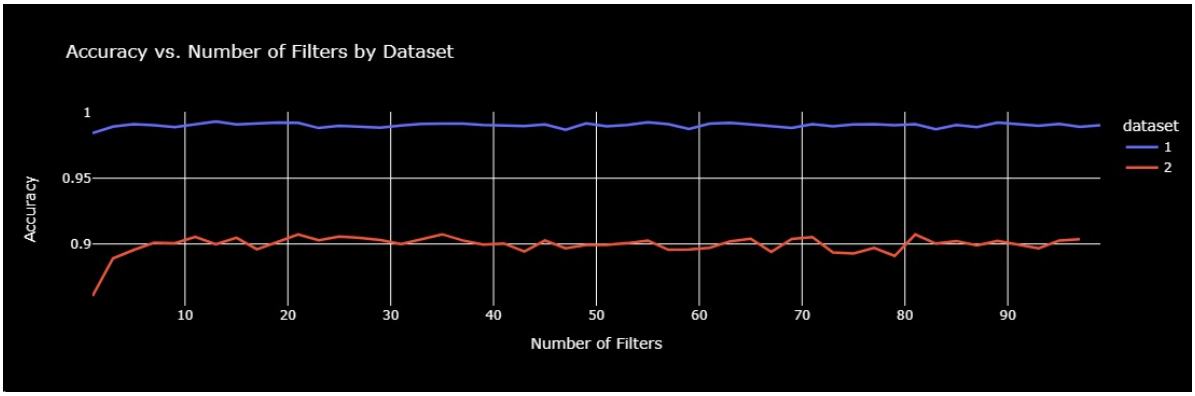
After the run was complete, we were able to run some analysis on accuracy vs loss to verify the model runs performed as they should, which they did. We observe a negative relationship between accuracy and loss across the datasets. Additional analysis involved accuracy levels vs the number of filters across the datasets. It is evident that dataset 1 performed exceptionally well across the board.

Total results : 100

Average loss: 22.3%

Average accuracy: 94.53%





Challenges, Conclusions, and Future Work

Challenges

- Using Hamming for this experiment was not trivial! Understanding all the adjustments we needed to make to the batch script to run on Hamming was difficult, such as knowing the amount of memory we needed to request to actually run the code on Hamming and how to setup our output files. We are still unsure if the optimal setup was used for our final run, so there is more to learn here.
- Debugging can be challenging in Hamming. It took a good bit of trial and error to adjust the code that worked well on our local machines for use in Hamming. Most of the obstacles came because of version control issues with the packages we were using.

Conclusions

Overall, this project showed that mass experimentation like this can be done using the Hamming resources we have available to us! The basic methodology used where we feed index values to a python script that then reads in the hyperparameters to test, can be applied to many use cases relevant to the students an thesis work in our class.

Future Work

A few things we did not get to, but could be interesting extensions of this work:

- Make this methodology more robust, such as how to deal with failed batches or how to more effeciently implement the scripts so that the failure rate and time to complete are reduced.
- Evaluating other metrics important to neural networks, like robustness against attacks on input data. This would involve building the "attack", attacking the test set data, and evaluating a pre/post loss and accuracy on the test data.
- Utilizing GPU's. LTC Smith introduced us to the Beards partition on Hamming, but we did not get around to it as we had a lot to debug with the CPU implementation.
- Improving the user experience with this by building an app with some simple buttons that could 1) automate the git clone, 2) running of the batch file, and 3) providing a status of the run as its executing.