Python Keras Convolutional Network

by

Ronald Cook

www.linkedin.com/in/ronald-cook-programmer

October 2025

Python Keras Convolutional Network

Introduction

The purpose of this project is to demonstrate the implementation details of a convolutional network application programmed with Python class structures, rather than using a Jupyter Notebook.

To illustrate how to design real applications in an object oriented way, a one page sample application is divided into several Python classes. Instead of the interactive Notebook, a desktop GUI is constructed using PySide6.

The sample application is a convolutional neural network (CNN) based on the well known MNIST dataset. The network objective is to find the set of model parameters that produces the best output prediction. In the convolutional network, the gradient descent approach is used to gradually update the model weights and biases to achieve an optimal solution.

The goal here is to understand how to implement the desktop application with multiple Python classes. Where do we begin? How do we separate the input, training, testing, and output into a Python class structure? How do we implement a user interface?

Background

It is assumed you have already read some tutorials on convolutional networks, and maybe used a Jupyter Notebook application. Understanding these networks requires knowledge of matrix algebra, partial derivatives, and statistics. Since we are focusing on Python class structure, this document will not repeat network background information. The references below are for refreshing your memory.

References:

Matrix algebra:

https://www.quantstart.com/articles/scalars-vectors-matrices-and-tensors-linear-algebra-for-deep-learning-part-1/

https://www.quantstart.com/articles/matrix-algebra-linear-algebra-for-deep-learning-part-2/

https://www.quantstart.com/articles/matrix-inversion-linear-algebra-for-deep-learning-part-3/

Matrix calculus:

https://towardsdatascience.com/matrix-calculus-for-data-scientists-6f0990b9c222

Neural network:

https://medium.com/machine-learning-algorithms-from-scratch/digit-recognition-from-0-9-using-deep-neural-network-from-scratch-8e6bcf1dbd3

http://neuralnetworksanddeeplearning.com/chap1.html

Gradient Descent:

https://builtin.com/data-science/gradient-descent

Hyper-parameters:

https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a

Learning rate decay:

https://www.geeksforgeeks.org/machine-learning/learning-rate-decay/

https://www.jeremyjordan.me/nn-learning-rate/

Convolution network:

https://victorzhou.com/blog/intro-to-cnns-part-1/

https://medium.com/secure-and-private-ai-math-blogging-competition/cnn-maths-behind-cnn-910eab425b5d

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Back propagation – softmax:

https://www.mldawn.com/back-propagation-with-cross-entropy-and-softmax/

Back propagation – convolution:

https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

Back prop – convolution padding

https://bishwarup307.github.io/deep%20learning/convbackprop/

Momentum:

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

Grid Search:

https://www.geeksforgeeks.org/machine-learning/hyperparameter-tuning-using-gridsearchcv-and-kerasclassifier/

Cross validation:

https://www.datacamp.com/tutorial/k-fold-cross-validation

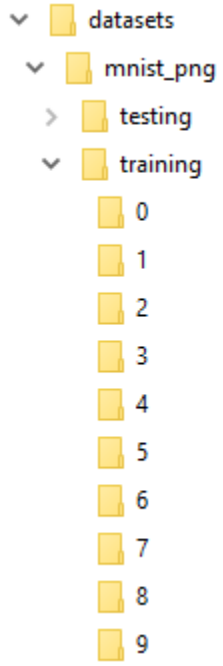
Model file HDF5/Keras format:

https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5

Datasets

You must download the MNIST dataset, which contains thousands of images of handwritten digits 0 to 9. The png images used by this application were downloaded from: https://github.com/myleott/mnist_png.

The file structure will look like the following, where the digit images are separated into individual folders 0, 1, ... 9.

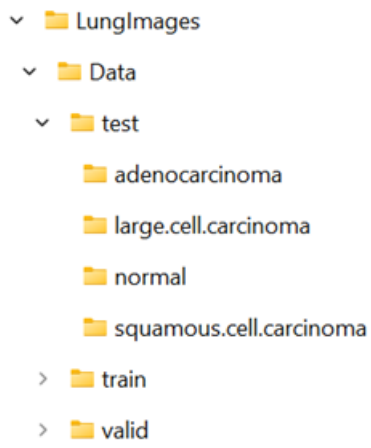


The advantage of having separate folders is to allow the user to examine specific digits. Another advantage of having separate folders is that we can quickly observe how many images there are for each digit.

After loading the separate image files, they are combined and randomly shuffled before the network training or testing task begins.

Note: even though this application was based on processing handwritten digits from the MNIST dataset, other image files could be loaded into the same folder structure. Here, the folder order represents the label indexes, which are used in the network processing. The indexes are merely labels for the images, but the images do not need to be digits. This design makes the code more reusable.

A second dataset containing lung cancer images may be downloaded from: https://www.kaggle.com/datasets/mohamedhanyyy/chest-ctscan-images



There are four classes: adenocarcinoma, large.cell.carcinoma, normal, and squamous.cell.carcinoma divided into four folders. To use this dataset, the folders are internally indexed as 0, 1, 2, 3 in loading order.

Note, the class indexes below:

0 - adenocarcinoma

1 - large.cell.carcinoma

2 - normal

3 – squamous.cell.carcinoma

Application Layout

The application code has the following structure:

datasets

LungImages

mnist_png

results

app_results (files: mnist.keras, train_val_plots.png, hyper_params.png)

convonet

data

app_input (files: mnist_data_params.json,
mnist_hyper_params-5.json,
lung-data-params.json,
lung-hyper-params-2.json)

dist

docs

src

app (main class: AppWin)

appcrtl (classes: AboutDialog, Controller, DataTask, Plotter,
TrainTask, TuneTask, Util)

thenet (classes: NetModel, NetTraining, NetTesting)

theprep (classes: DataParams, DataPrep, HyperParams)

test

test_cnnpy

test_appctrl

test_thenet

test_theprep

The VSCode IDE was used to build the Python application, on a Windows 11 PC, with 16GB ram, and 8GB Nvidia GPU. See uv-instructions.txt to clone this application from GitHub. After installing the software, at the command prompt, go to clone subdirectory, convonet.

Activate the virtual environment: call .venv\Scripts\activate.

Note: app.py will not run properly without the virtual environment.

Run the application, with the Python command: python src/cnnpy/app.py.

It may take a couple minutes to open.

If the windows exe file has been created, run the application directly: “dist/app.exe” (quotes required).

Python references:

UV dependency tool:

https://realpython.com/python-uv/

Python application layouts:

https://medium.com/@adityaghadge99/python-project-structure-why-the-src-layout-beats-flat-folders-and-how-to-use-my-free-template-808844d16f35

https://realpython.com/python-application-layouts/

https://python.land/project-structure/python-packages

PyTest:

https://realpython.com/pytest-python-testing/

PySide6:

https://www.pythonguis.com/tutorials/pyside6-creating-your-first-window/

https://www.datacamp.com/tutorial/introduction-to-pyside6-for-building-gui-applications-with-python

GPU CUDA:

https://medium.com/@chandrakundu/uv-tutorial-part-3-installing-pytorch-with-cuda-1157bf0d655f

Pyproject.toml:

https://realpython.com/python-pyproject-toml/

https://packaging.python.org/en/latest/guides/writing-pyproject-toml/

PyInstaller:

https://realpython.com/pyinstaller-python/

Class Framework

This application is fairly simple, because it relies on the Python and Keras libraries to do the complex work. Think of this code as an application programming interface (API) to existing library code. Note: in this application Python class names use CapWords naming, while class methods use camelCase naming. Some imported classes, such as keras, implement class methods with underscore format, while others, such as PySide6, use camelCase.

Where to begin? Most programs have this high level structure: input → computation → output.

Input could be hard coded, from a database, from user dialogs, or from files. Let's use files for our input. We have different types of input, each of which can be a separate Python class. So, let the input classes be DataParams, DataPrep, DataTask, HyperParams.

DataParams loads or saves a json file containing the locations of the training and testing datasets. It also needs to allow the user to create or modify this data. Thus, the basic methods in this class will be loadParams, saveParams, editParams, and newParams. Note the use of camel case naming of class methods.

HyperParams loads a json file of hyper parameters required to specify the layers in a network model. The methods will also be named loadParams, saveParams, editParams, newParams. However, they are completely different than the methods in DataParams.

DataPrep loads the datasets, which in this case are thousands of images files. There is no need for a save, edit, or create method. However, due to the complexity of images, several methods are used to input the images. The primary method will be prepData which calls loadImages and transform. The method 'loadImages' calls extractImage. And the method 'transform' calls shuffleArrays. Because thousands of images will be loaded, we need a separate thread to avoid

the GUI being blocked while the loading progresses. The class DataTask runs the DataPrep functions in a separate thread.

The computation tasks are quite complex, but the Keras libraries will be called to perform the difficult functions. We need to train the network model, and test the final model. Selecting a good combination of hyper parameters can be done with Keras grid search. Thus we need a tuning task. For tuning, training, and testing, the Python classes are NetModel, NetTraining, NetTesting, TrainTask, and TuneTask.

NetModel applies the hyper parameters in the layer specifications of a neural network model, which is based on the Keras Sequential class. After the model is trained by the NetTraining class, it may be saved with the saveModel method. Later, the loadModel method can be called to retrieve the already trained model, which can be used by the NetTesting class.

TrainTask runs NetTraining in a separate thread. NetTraining has two basic methods. The 'train' method performs 'training only' by calling the Keras model.fit method. The 'trainVal' method performs 'training plus validation' by calling the Keras model.fit method, but using split datasets, one for training and one for validation.

The NetTesting functions are very fast and do not need a separate thread. They depend on an existing trained network model created by NetTraining. Thus, NetTesting can be run immediately after running NetTraining, or later by loading a network model from a file. There are two methods: evaluate and predict. The evaluate method uses an already trained model to process an independent test dataset. Each sample goes through the model layers, is checked for correctness, and the result recorded. But there is no gradient descent or backpropagation, which allows for very fast processing. The predict method is similar to the evaluate method, except it records both the correct and incorrect results, to form what is called the confusion matrix.

TuneTask runs the tuning function in a separate thread. It calls createTuningModel to specify its own network model designed for compatibility with the KerasClassifier and GridSearchCV classes. A subset of the hyper parameters are passed to GridSearchCV to create all combinations defining a 'grid'. For example, if there are two values for init_rate, and three values for epochs, the grid consists of 6 combinations. For each combination, the model fit method is called using a subset of training data, and results are stored. Thus, the processing time will depend on the total number of grid combinations and the dataset size. A progress bar is updated periodically via the Controller class.

The output in this application consists of plots, tables, and files. The Plotter and Controller manage plots and tables, while NetModel handles the trained model output file. The file handlers: DataParams, HyperParams, and NetModel actually handle both input and output files.

Controller has many methods to invoke tasks and produce output displays. It also holds most of the data that is passed between classes. For handling input data, AppWin calls three Controller methods: setDataParams, setHyperParams, and setNetModel, all of which store an instance of each class. This allows Controller to retrieve the data stored in these classes. Other Controller methods are briefly described as follows:

trainPrep: retrieves the data parameters and hyper parameters from DataParams and HyperParams respectively, initializes the neural network model, and displays a model summary table.

showSummary: calls extractSummary

prepTrainingData: calls DataPrep and DataTask to load the training dataset.

prepTestingData: calls the same methods to load the test dataset.

dataTaskFinished: is called when the dataset has completed loading. It calls the samples method.

samples: plots a few images after loading each dataset, to verify proper internal format.

train2: creates an instance of NetTraining, which is passed to TrainTask, and starts TrainTask in a separate thread. TrainTask calls trainVal or train on the instance of NetTraining.

trainTaskFinshed: is called when training has completed. It calls plotHistoryVal to display a plot of training accuracy and validation accuracy. It also plots training loss and validation loss.

evaluate: invokes the NetTesting evaluate method, and creates the result table of accuracy and loss.

predict: invokes the NetTesting predict method, and creates the confusion matrix plot.

runTuner: runs the TuneTask in a separate thread, and calls extractTuneResults and tuneTable to display the grid search results.

stopTuner: allows the user to interrupt the TuneTask processing.

As mentioned above, the GUI is based on PySide6. The AppWin class implements all menus, and delegates all menu actions to other classes using ‘signal’ connections. Some classes also send return signals to AppWin. The Controller class attempts to isolate other classes from AppWin, but for input files, it is simpler to let AppWin connect directly to DataParams, HyperParams, and NetModel.

The next section reviews some test classes.

Class Test Overview

The test folder layout mirrors the src folder layout. A variety of test cases are included to demonstrate how they work in a nested class structure. It is questionable if these tests are worthwhile, because the tester can see graphically what works by running the app.py directly.

To invoke the test cases, cd to project directory at convonet.

Activate the virtual environment: call .venv\Scripts\activate.

Note: pytest will not run properly without the virtual environment.

At the command line prompt, enter: pytest. All tests will be executed, and a printed summary shows the number of cases that passed or failed.

First, in the test_cnnpy folder, the test_app.py file contains the TestAppWin class to test a selection of menu options. In particular, the “Load Data Params” and “Load Hyper Params” menu actions are tested. Actually triggering the menu items is commented out, as it would require the tester to perform an exact sequence of mouse clicks and loading input, which would overcomplicate the testing scenario.

Second, in the test_appctrl folder, the test_controller.py file contains the TestController class which performs some basic tests on the updateStatus signal.

Next, in the same folder, the test_plotter.py file contains the TestPlotter class to test the plotHistoryVal method, without actually displaying the plot. Again, since the plot is not actually displayed, this test may be superfluous.

In the test_theprep folder, the test_dataprep.py file contains the TestDataPrep class which tests a method called shuffleArrays, a very important function that randomly shuffles the input datasets so that during training the samples do not have the same label in consecutive order.

The test_nettrain.py file contains the TestNetTraining class, which tests the sizing method that computes the total number of batches required for updating the training progress bar.

This document does not walk through the code line by line. But the next section reviews the GUI actions of AppWin.

Project Overview

As seen in the application code under the convonet project directory, there is a main class, AppWin, which calls the other classes via menus. To run the application from the convonet directory:

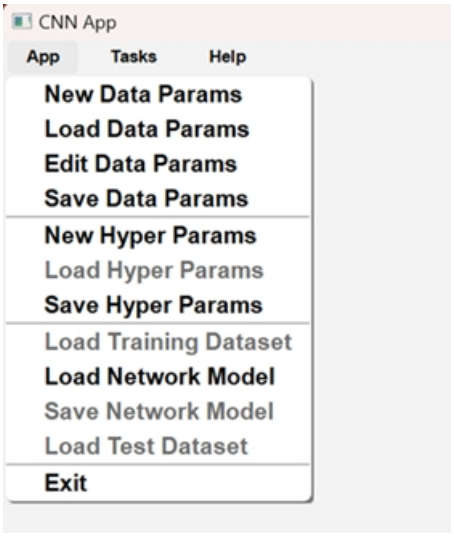
Activate the virtual environment: call .venv\Scripts\activate.

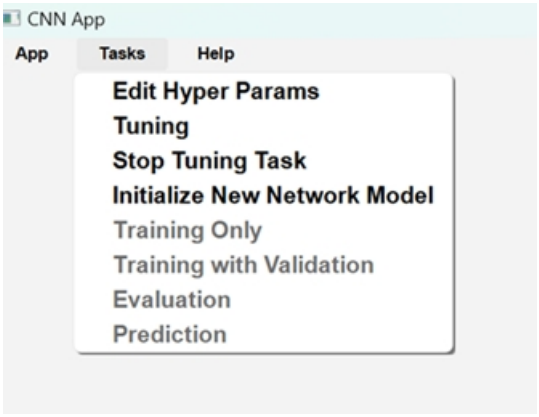
Note: app.py will not run properly without the virtual environment.

Run the application, with the Python command: python src/cnnpy/app.py.

It may take a couple minutes to open.

Menu Structure





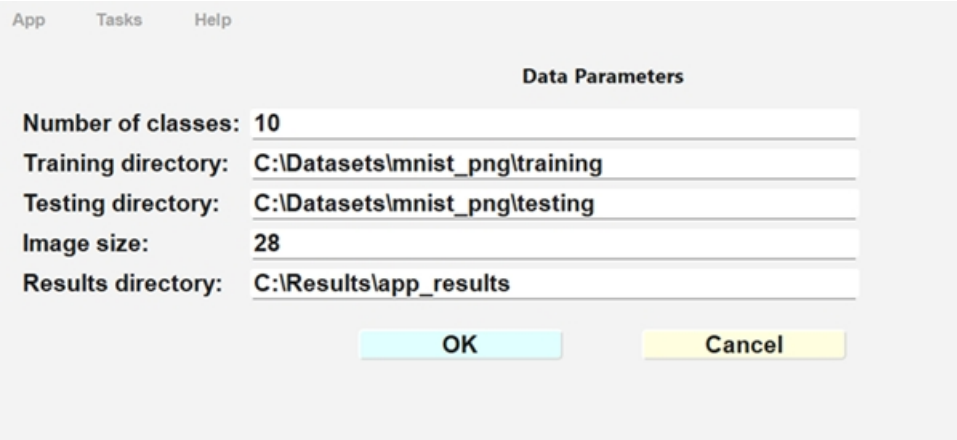
The menu structure is divided into App, Tasks, and Help. The App menu handles input files, while the Task menu manages the tuning, training, and testing operations.

The Help menu item, Snap, not shown here, allows saving a screen image.

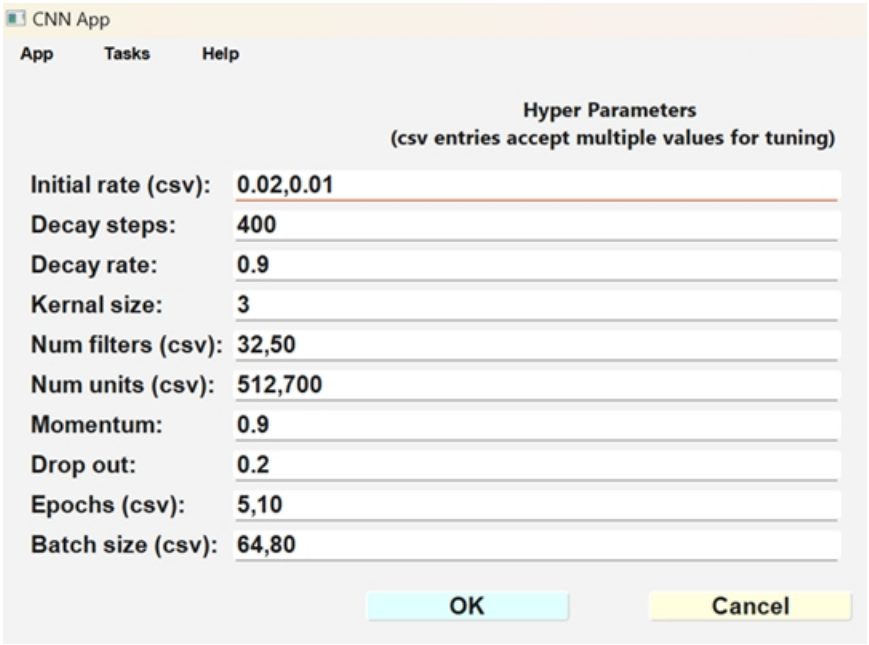
App Menu

The App menu consists of input items, which may be created, loaded, edited or saved.

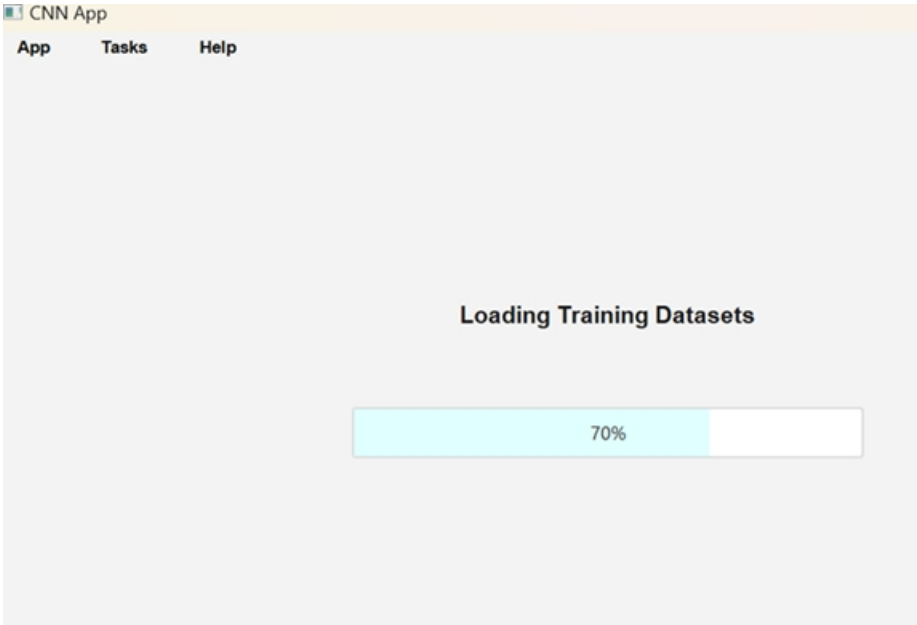
Load Data Params loads a json file containing the folders where the training and testing datasets are located, as well as a folder where the network results may be stored.



Load Hyper Params loads a json file containing a list of all the hyper parameters required to create a network model. Note: the 'csv' entries allow multiple values used by the grid search function in the tuning task. For regular training, only the first value of each parameter is used.



As the name implies, Load Training Dataset loads the training image files, shuffles the data, and normalizes each image pixel to a value 0 to 1.



Load Network Model loads a trained model for testing. This file has a special format with a ".keras" extension.

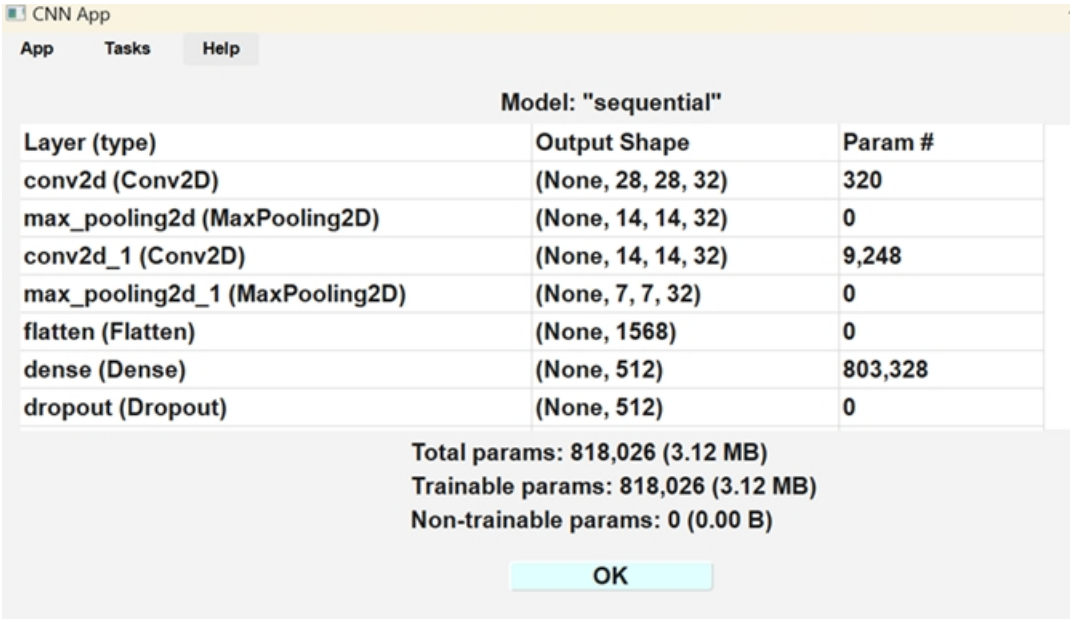
Load Test Dataset reads the test dataset image files, shuffle the data, and normalizes each image pixel to a value 0 to 1.

Tasks Menu

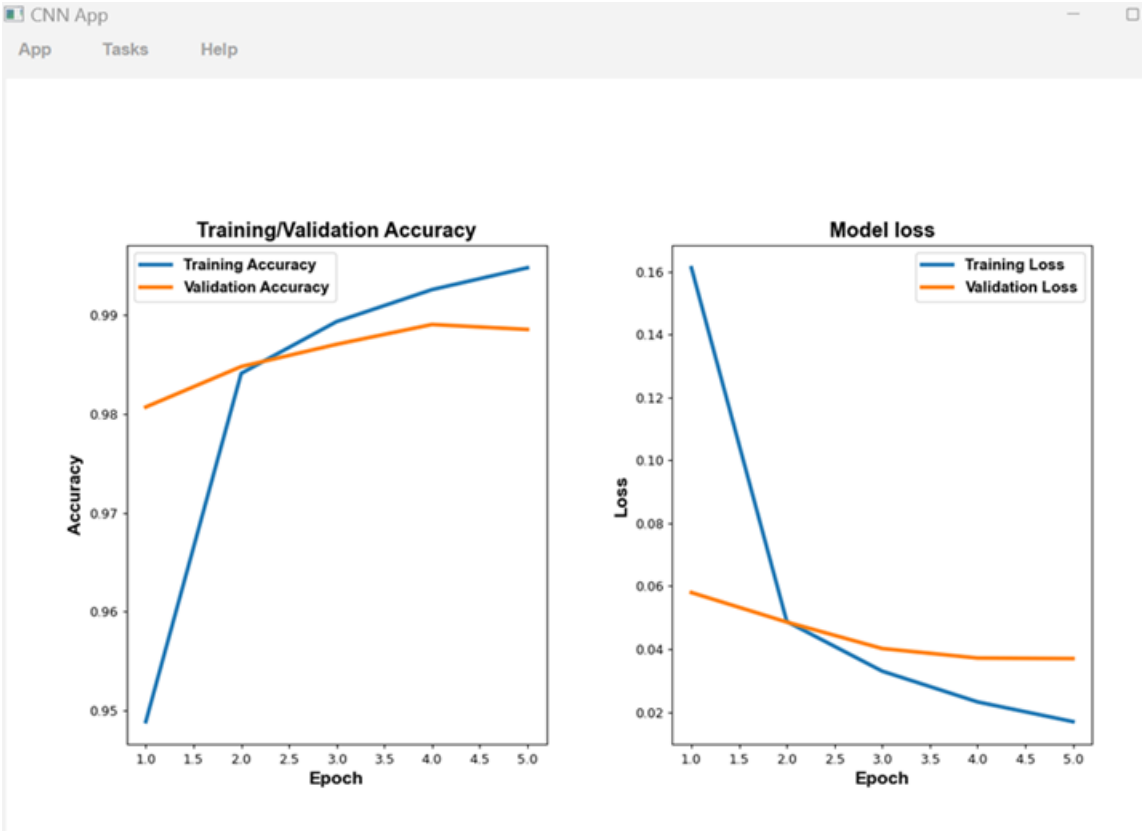
The Tasks menu controls the tuning, training, and testing of a new neural network model.

The Edit Hyper Params gives the user a chance to override the values loaded from the input file.

Initialize New Network Model must be invoked before training can begin. The model code sets up network layers based on the hyper parameters specified by the user. The model is summarized in a table of layers and shape info.

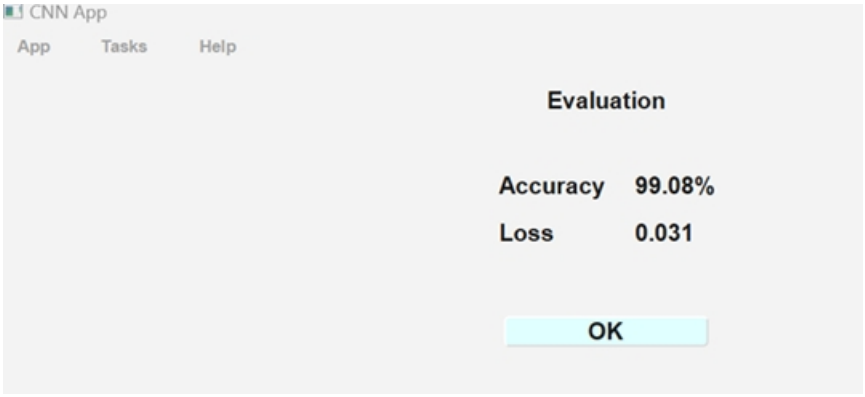


There are two training options. Training Only trains the model without a validation dataset, while Training With Validation trains the model and compares the results with a subset of the training dataset. The results are displayed graphically.

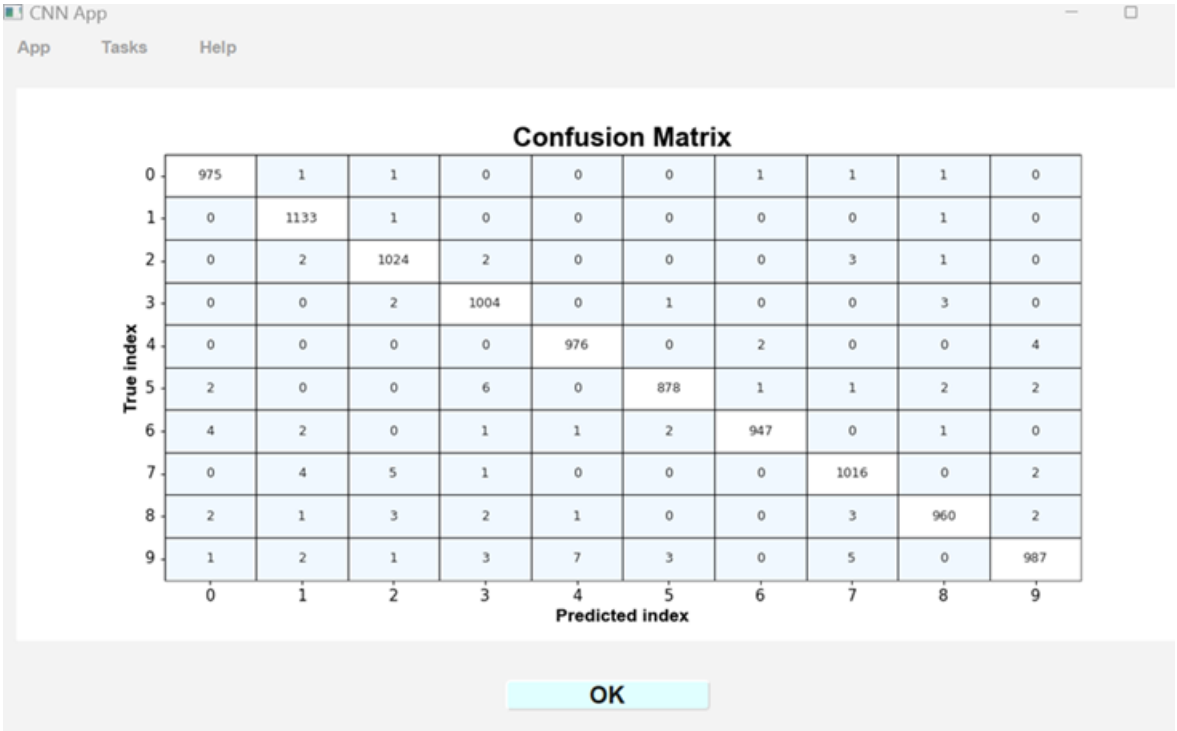


Evaluation and Prediction check the quality of the trained model based on an independent test dataset, which must be loaded before invoking the two options.

Evaluation runs the test dataset through the network using the trained model, and calculates the overall accuracy.

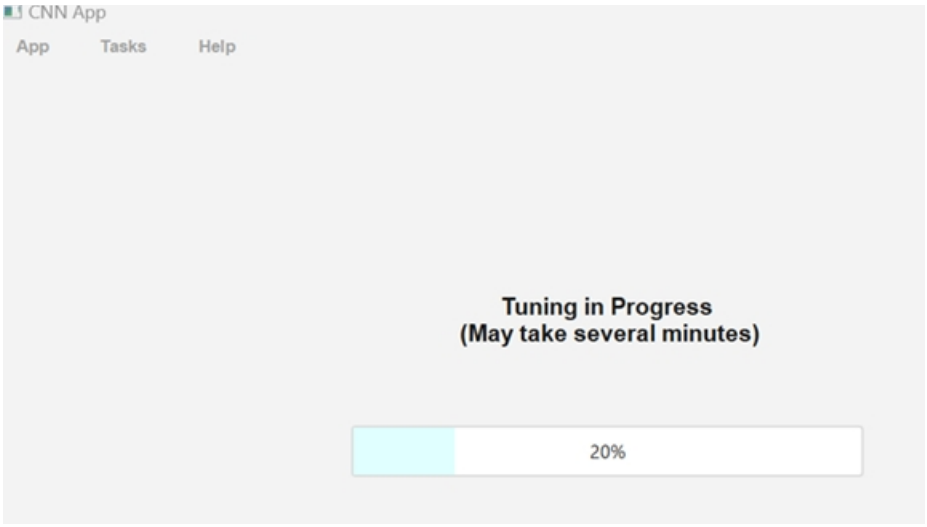


Prediction runs the test dataset through the network using the trained model, and records the count of predicted label indexes versus true label indexes for each network class index. The results are displayed in the “confusion matrix”.



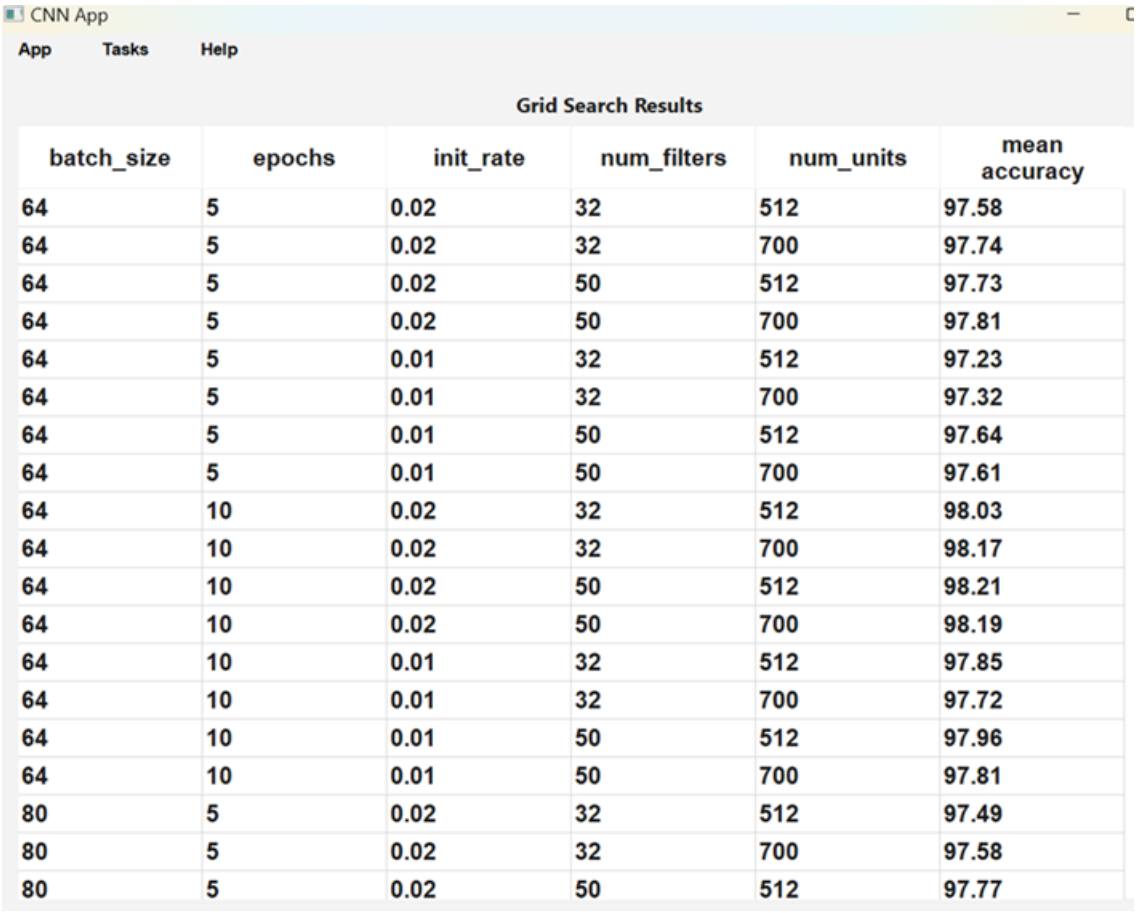
Tuning Task Scenario

Tuning invokes a task that performs a grid search on specified combinations of hyper parameters to determine the best subset to maximize model accuracy. The hyper params, and training dataset must be loaded. The user may want to edit the hyper params before running the tuning task. It is important to consider the total number of combinations of hyper params, because a subset of the training dataset will be run through the network for each combination using a grid search, which may lead to a very long processing time. The progress bar shows that something is happening in the background.



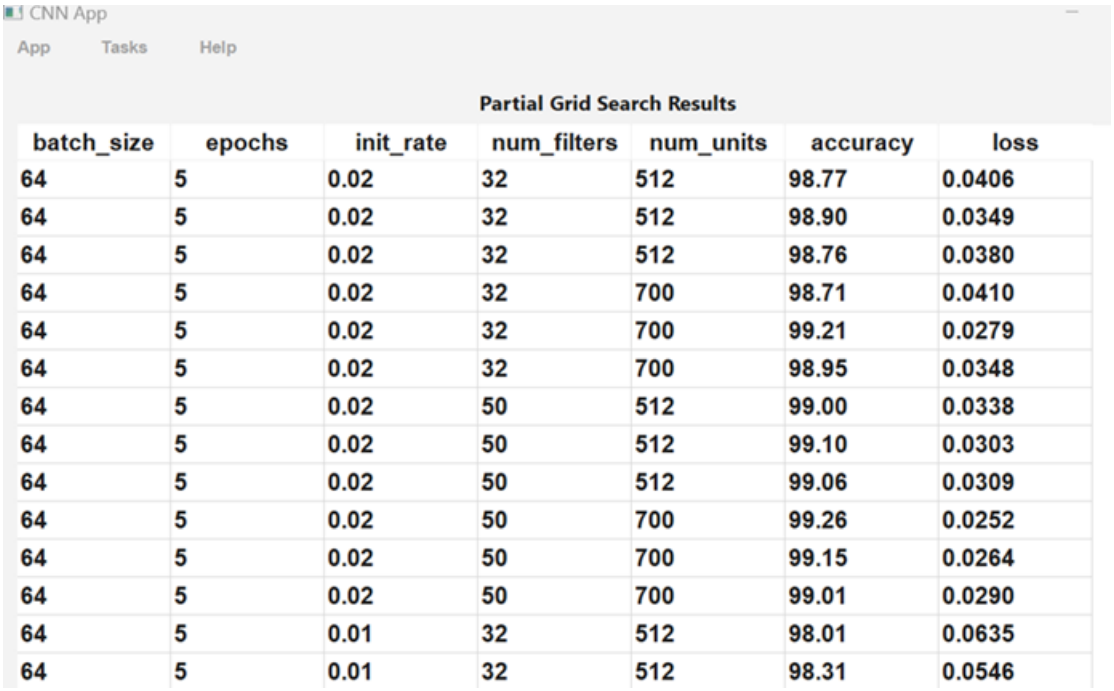
The results will be displayed in a table. The columns are labeled with the names of the hyper parameters, with the last column being the mean accuracy. In the example shown below, note the high accuracy 98.21 for batch size 64, epochs 10, init_rate 0.02, num_filters 50, and num_units 512. These may not be the optimum set of hyper parameters. Note in the next row where num_units is 700, the accuracy changes to 98.19. Also note in the previous 2 rows where the num_units is 512 and the num_filters is 32, the accuracy changes to 98.03. These two cases demonstrate that num_units and the num_filters may be only minor factors in determining the accuracy. Comparing other rows reveals that epochs and init_rate are probably the major factors affecting accuracy.

However, note that the differences are within 1%. The other hyper parameters are not listed here, but can be seen in a previous figure describing loading hyper params. The user could edit those other hyper parameters and rerun the tuning task. For example, the effect of changing the kernal size from 3 to 5 could be compared with the results below.



batch_size	epochs	init_rate	num_filters	num_units	mean accuracy
64	5	0.02	32	512	97.58
64	5	0.02	32	700	97.74
64	5	0.02	50	512	97.73
64	5	0.02	50	700	97.81
64	5	0.01	32	512	97.23
64	5	0.01	32	700	97.32
64	5	0.01	50	512	97.64
64	5	0.01	50	700	97.61
64	10	0.02	32	512	98.03
64	10	0.02	32	700	98.17
64	10	0.02	50	512	98.21
64	10	0.02	50	700	98.19
64	10	0.01	32	512	97.85
64	10	0.01	32	700	97.72
64	10	0.01	50	512	97.96
64	10	0.01	50	700	97.81
80	5	0.02	32	512	97.49
80	5	0.02	32	700	97.58
80	5	0.02	50	512	97.77

If necessary, the user may terminate the tuning process by selecting the Stop Tuning menu item. A table of partial grid search results will be displayed as shown below. The size of the table will depend on how long the tuning process has run before being interrupted.



batch_size	epochs	init_rate	num_filters	num_units	accuracy	loss
64	5	0.02	32	512	98.77	0.0406
64	5	0.02	32	512	98.90	0.0349
64	5	0.02	32	512	98.76	0.0380
64	5	0.02	32	700	98.71	0.0410
64	5	0.02	32	700	99.21	0.0279
64	5	0.02	32	700	98.95	0.0348
64	5	0.02	50	512	99.00	0.0338
64	5	0.02	50	512	99.10	0.0303
64	5	0.02	50	512	99.06	0.0309
64	5	0.02	50	700	99.26	0.0252
64	5	0.02	50	700	99.15	0.0264
64	5	0.02	50	700	99.01	0.0290
64	5	0.01	32	512	98.01	0.0635
64	5	0.01	32	512	98.31	0.0546

Training Task Scenarios

The data parameters and model hyper parameters must be loaded before running the training task.

Training Scenario 1: Invoke Edit Hyper Params, Initialize New Network Model, Training with Validaton. Examine the results, and repeat the sequential process until satisfactory performance is achieved. This scenario may require several iterations because there are many combinations of hyper parameters to consider. However, unlike the grid search in the tuning task, one training iteration should be relatively quick. But if several iterations are performed, the total processing time could be the same as tuning.

Training Scenario 2: Note the best set of hyper parameters from the Tuning task. Invoke Edit Hyper Params, Initialize New Network Model, Training with Validaton. Examine the results, and repeat the sequential process until satisfactory performance is achieved. This scenario should be quicker than scenario 1 because the tuning task has already compared several combinations of hyper parameters.

In both scenarios, the results are displayed graphically. The Snap option under the Help menu may be used to record an image of the graphical window.

Testing Scenarios

The testing dataset should include a different set of images than the training dataset.

Therefore, there are two scenarios to consider:

Test Scenario 1. After a successful training run, Load Test Dataset and run the Evaluation and Prediction tasks.

Test Scenario 2. After a training run, save the Network Model to a file. Close the app for later testing. When ready, launch the application again. Load Data Params and Hyper Params, Network Model, and Test Dataset. Then run the Evaluation and Prediction tasks.

In both scenarios, the results are displayed in the evaluation table and the "confusion matrix" plot. The Snap option under the Help menu may be used to record an image of the graphical window.

Case of Lung Cancer Dataset

Here are a few selected results when the lung cancer dataset is trained or tuned. For tuning, running out of memory is very likely, even for small batch sizes. However, in this case, a partial grid search will be captured.

Note, the class indexes below:

0 - adenocarcinoma

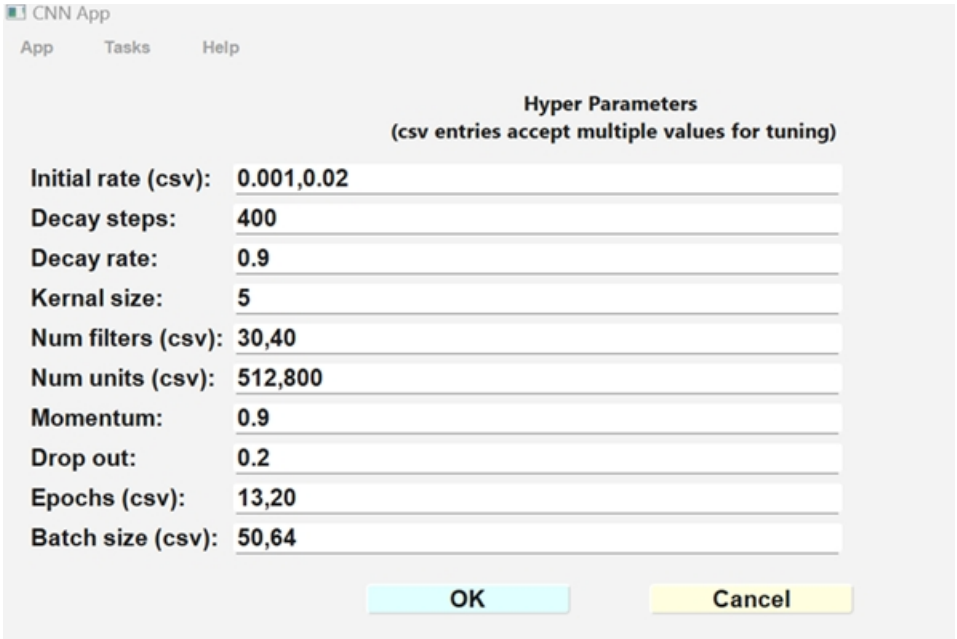
1 - large.cell.carcinoma

2 - normal

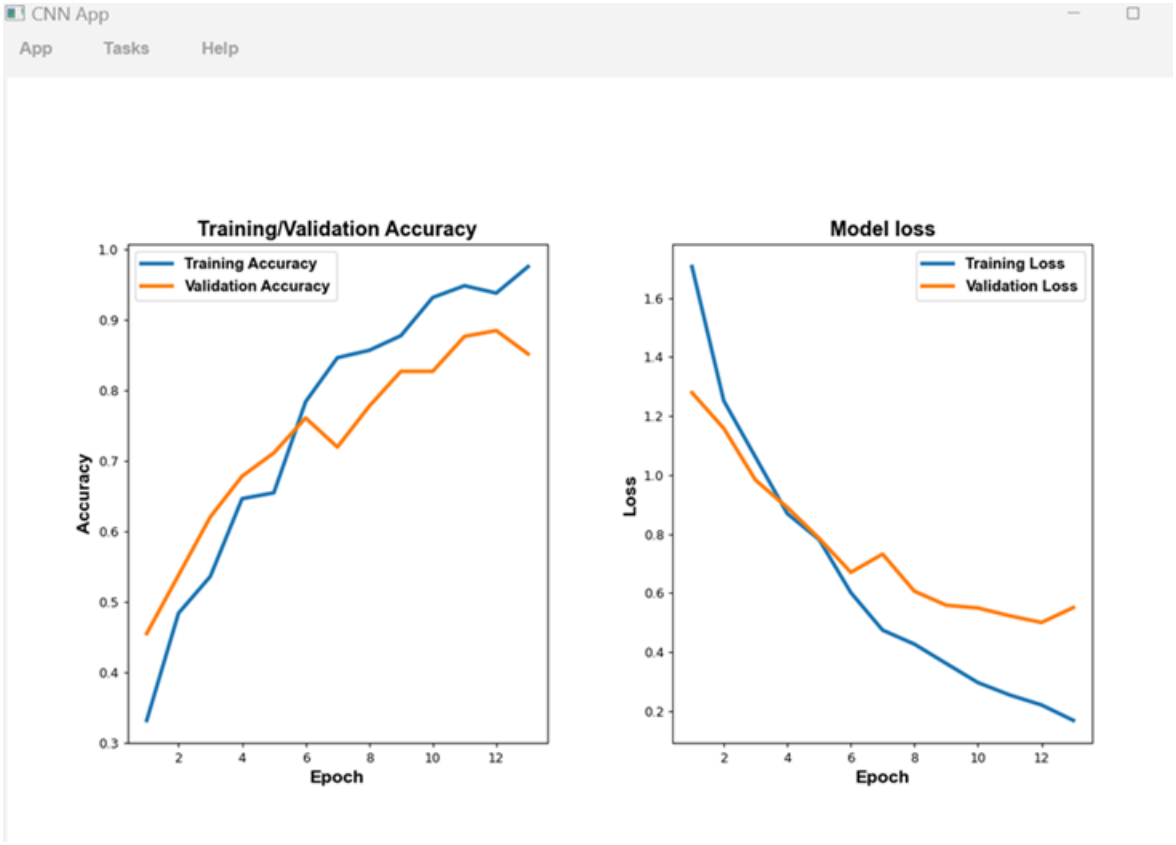
3 – squamous.cell.carcinoma

Image Samples (Lung Cancer)

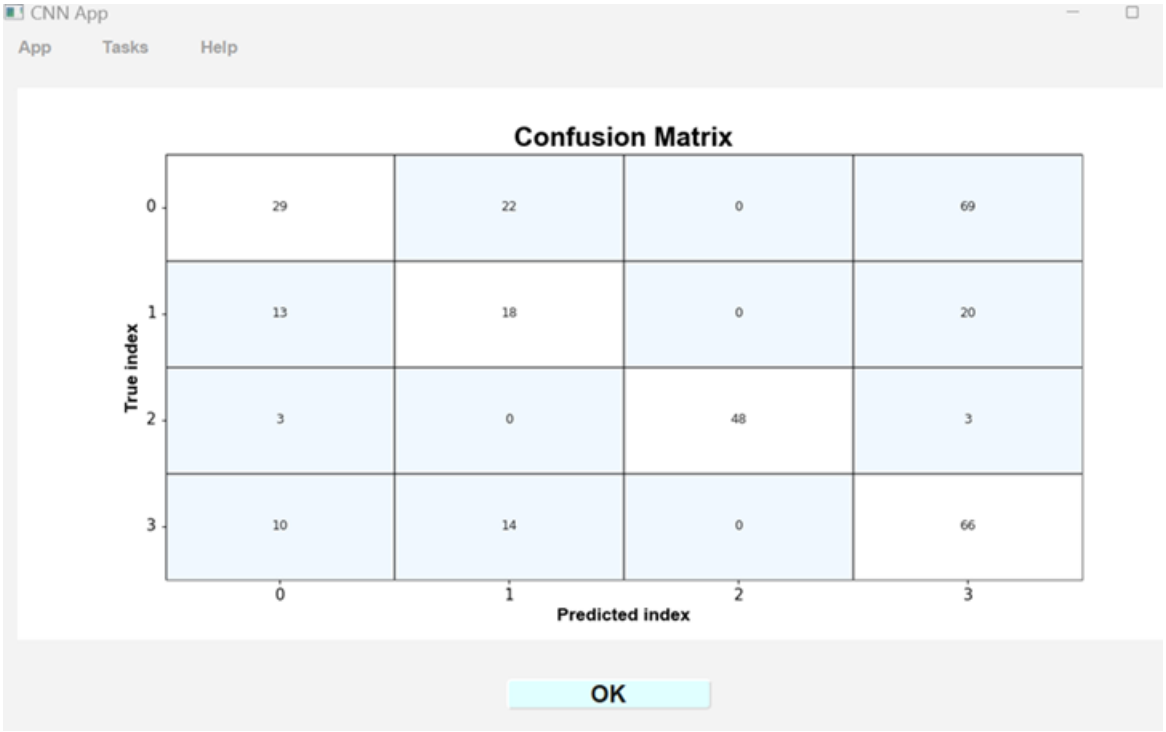
Hyper Params (Lung Cancer)



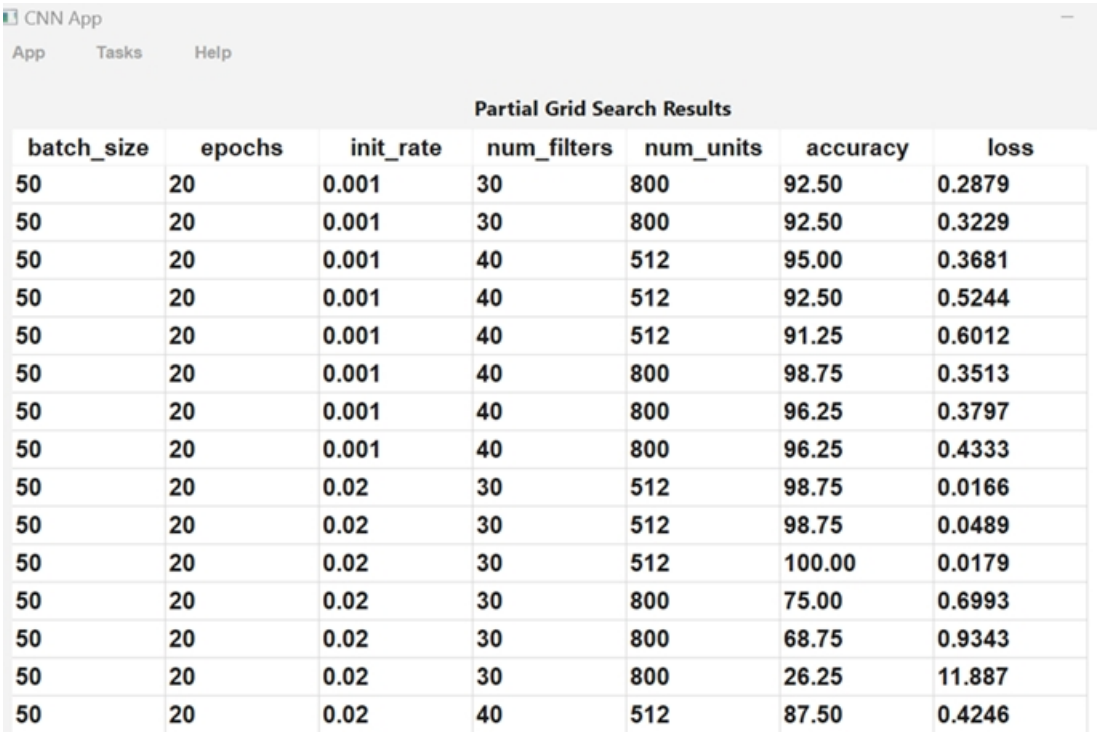
Train Plots (Lung Cancer)



Confusion Matrix (Lung Cancer)



Partial Grid Search (Lung Cancer)



Partial Grid Search Results

batch_size	epochs	init_rate	num_filters	num_units	accuracy	loss
50	20	0.001	30	800	92.50	0.2879
50	20	0.001	30	800	92.50	0.3229
50	20	0.001	40	512	95.00	0.3681
50	20	0.001	40	512	92.50	0.5244
50	20	0.001	40	512	91.25	0.6012
50	20	0.001	40	800	98.75	0.3513
50	20	0.001	40	800	96.25	0.3797
50	20	0.001	40	800	96.25	0.4333
50	20	0.02	30	512	98.75	0.0166
50	20	0.02	30	512	98.75	0.0489
50	20	0.02	30	512	100.00	0.0179
50	20	0.02	30	800	75.00	0.6993
50	20	0.02	30	800	68.75	0.9343
50	20	0.02	30	800	26.25	11.887
50	20	0.02	40	512	87.50	0.4246

Limitations

From the Project Overview and sample scenarios shown above, it may appear that this Python application has a very flexible design that allows editing various parameters, loading different image datasets, and tuning the model to find the optimal solution.

However, a few limitations should be pointed out. First, the folder architecture must store each class of images in a separate folder, to allow the folder order to represent neural network label indexes. Second, the network model is hard-coded with a convolutional layer structure. Only the numerical hyper-parameters can be edited by the user interface. Third, in the tuning task, for the Keras grid search, only a limited number of hyper-parameters are allowed to form the grid. Fourth, because the PySide Signal technique was used to communicate between some classes, those classes may not be directly reusable in a non-PySide environment.

To alter any of these limitations would require modifying the code.

The next section outlines some alternative project architectures.

Discussion

After explaining the Python class architecture and reviewing the GUI actions, alternative approaches can be evaluated. What alternatives exist compared to the current application?

Consider dividing the Python classes into a frontend and backend architecture. The frontend classes would be similar to: AppWin, Controller, DataParams, and HyperParams. The backend classes would be analogous to: NetModel, NetTrain, NetTest, DataTask, TrainTask, and TuneTask.

Implementation alternatives:

1. Frontend and backend combined in a Python desktop program, as in the current application, using internal data exchange.

2. Frontend as a Python desktop program connected to a Python backend server, using HTTP requests and responses.

3. Frontend as a web HTML-JavaScript client connected to a backend Python server, using HTTP requests and responses.

Python Client:

https://www.digitalocean.com/community/tutorials/python-http-client-request-get-post

https://www.qt.io/blog/restful-client-applications-in-qt-6.7-and-forward

MPLD3/D3js (matplotlib->JavaScript):

https://mpld3.github.io/

Python Server:

https://www.datacamp.com/tutorial/python-backend-development

https://pieces.app/blog/the-top-4-python-back-end-frameworks-for-your-next-project

https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Django/Introduction

https://www.geeksforgeeks.org/python/flask-tutorial/

Conclusion

This application illustrates a desktop program based on multiple Python classes, with a GUI constructed using PySide6. Building a convolution neural network with an object oriented design encourages consideration of modular coding practices. This document has described such an application. The Python class architecture has been reviewed, along with sample input and output.

For programmers who have used only Jupyter Notebooks, the code probably appears quite different and more complex than what is found in a typical notebook. Rather than hard coding various parameters, json input files are employed. The output plots should look familiar. And the sequential flow of the code is also similar.

It is hoped that this application design and its Python classes will be reusable. As already observed for digit and lung cancer images, this program can be applied to different classification projects. The principle of reusability supports efficient development practices. By creating Python classes that can be reused across various applications, developers can significantly reduce redundancy. This approach not only streamlines the coding process but also creates a more modular architecture that may make modifications simpler.