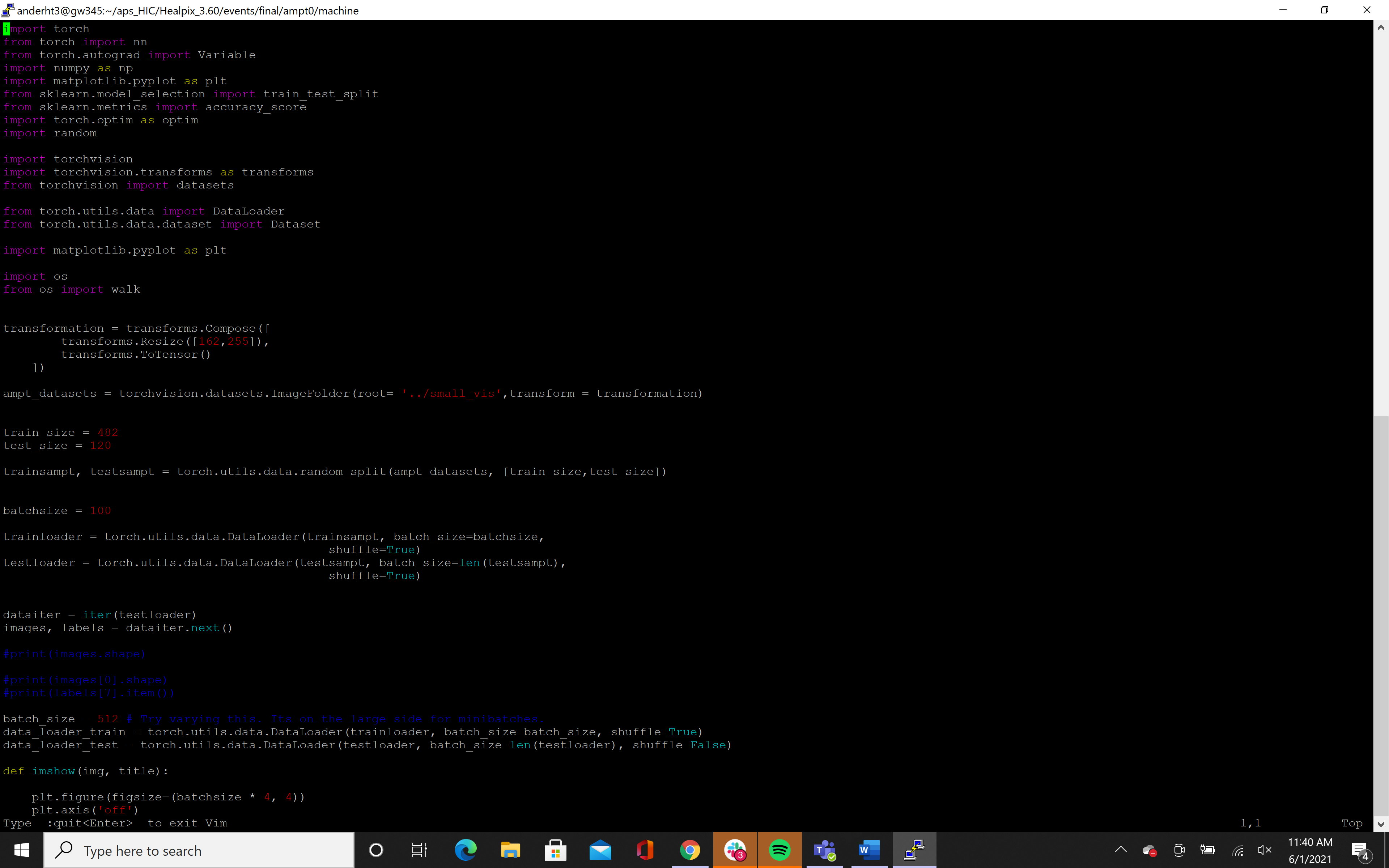
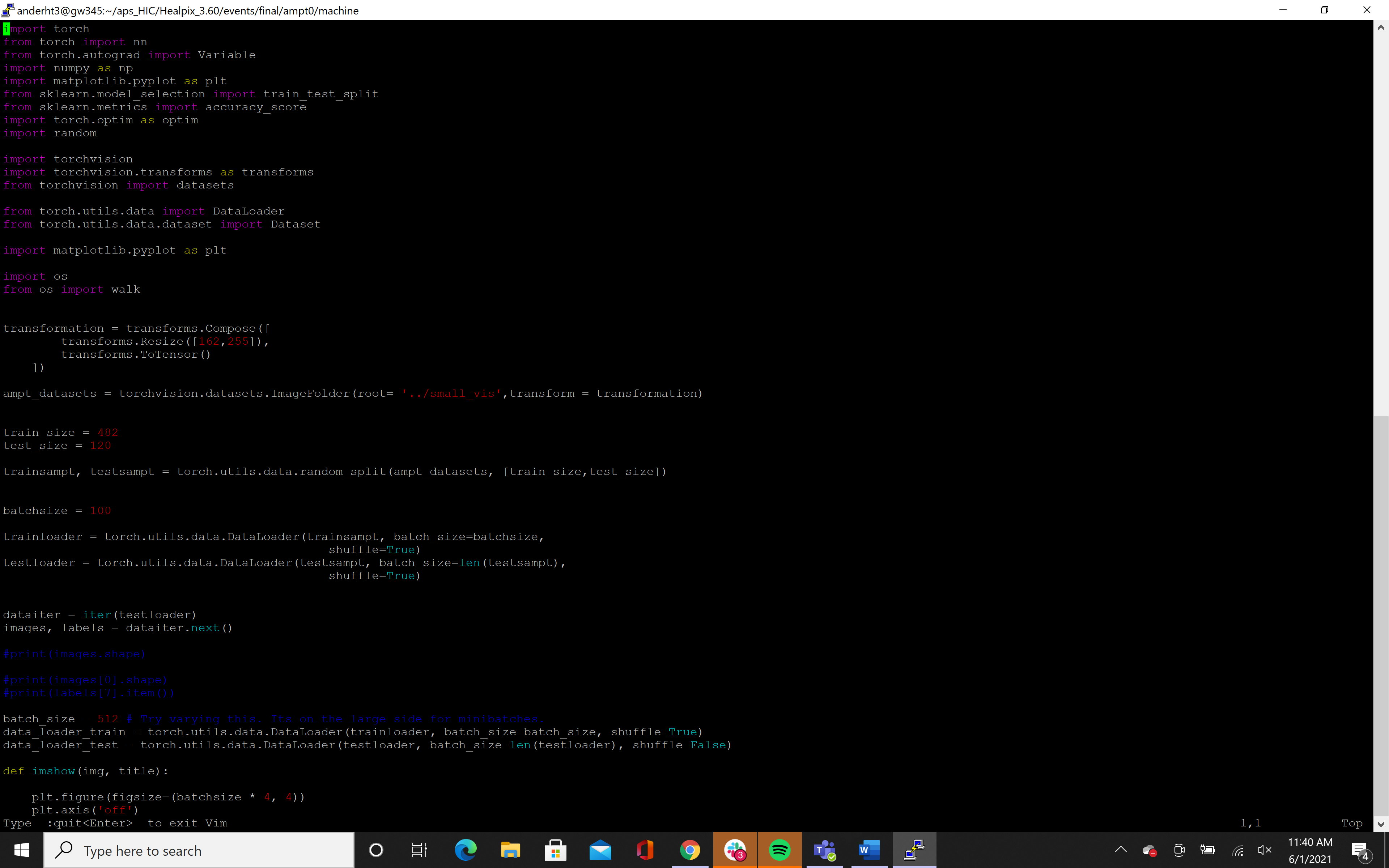
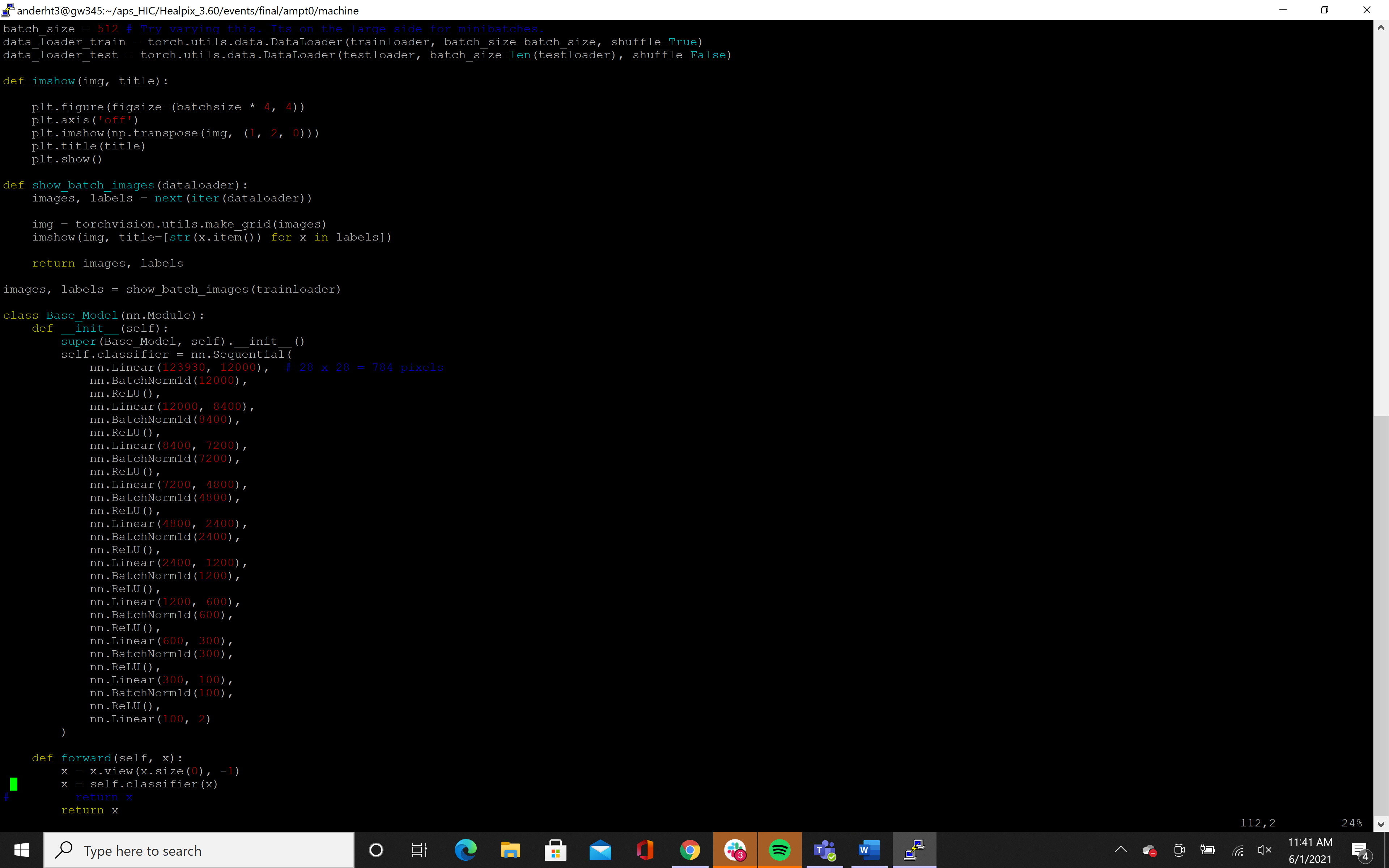
**Documentation for the file machine\_small.py (and machinelearning.py with different parameters)**



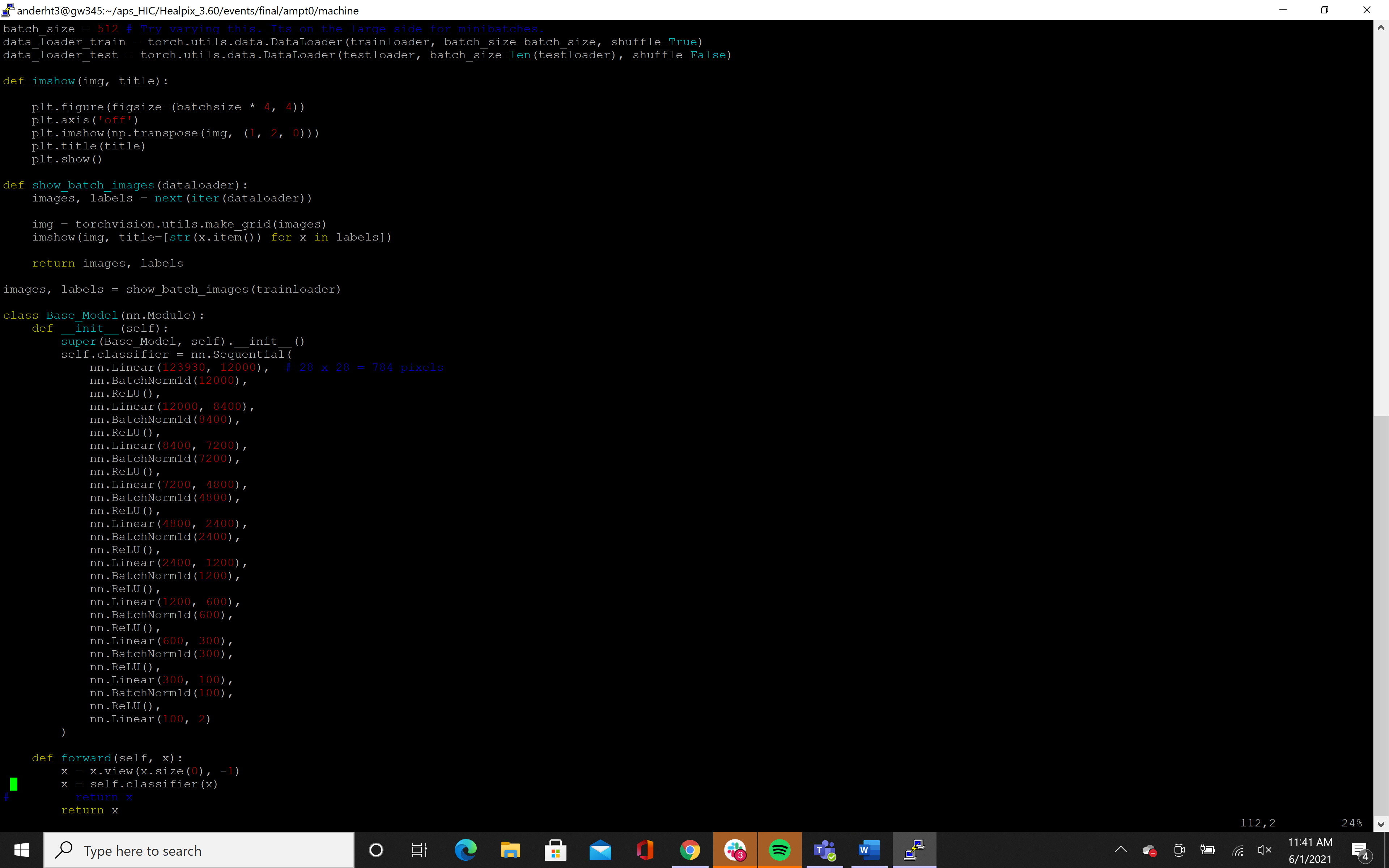
The lines seen above are meant to load in the files of pictures and place them into test and train data structures. The first command of transformation creates a call to a transformation for images. This specific transformation only resizes the image and translates it into a tensor structure instead of a list or array. The following line creates the variable “ampt\_datasets”. This variable houses the two different types of images that were stored in the folder “small\_vis”. Within this folder, there are two separate folders, one with the string melting images and the other with the default images. This call to torchvision.datasets creates a dataset that recognizes that there are two separate sets of data we will be using and labels them as 0 or 1 depending on which folder they came from. The next two lines create the size for the test and train datasets. The train size is about 4 times larger than the test, which means that 482 of the images will be placed into a data set used to train the model. The variables trainsampt and testsampt put an image randomly into either the train or test set. The variables retain the knowledge of which folder the images came from and are shuffled until the size cap has been reached. Next is the variable batchsize. This variable is used to break the training set into smaller batches so that the model does not process all of the images at once. This is a fluctuating variable that can some times cause an overfit of the data if too large or small is used. Therefore the batchsize can be adjusted after looking at the loss from the training data set at the end of the code. If it is too flat with no improvement, the batchsize should be smaller, and if the loss is seeing a lot of jumps, then the batchsize should be made larger. Next are the variables trainloader and testloader. These two variables simply take the original sets for testing and training and put them in a variable that once again randomizes their order to get rid of any bias and also adds in the influence of batchsize if needed. As one can see in the test loader, the batchsize used is the length of the test data set, meaning that multiple batches are not needed.



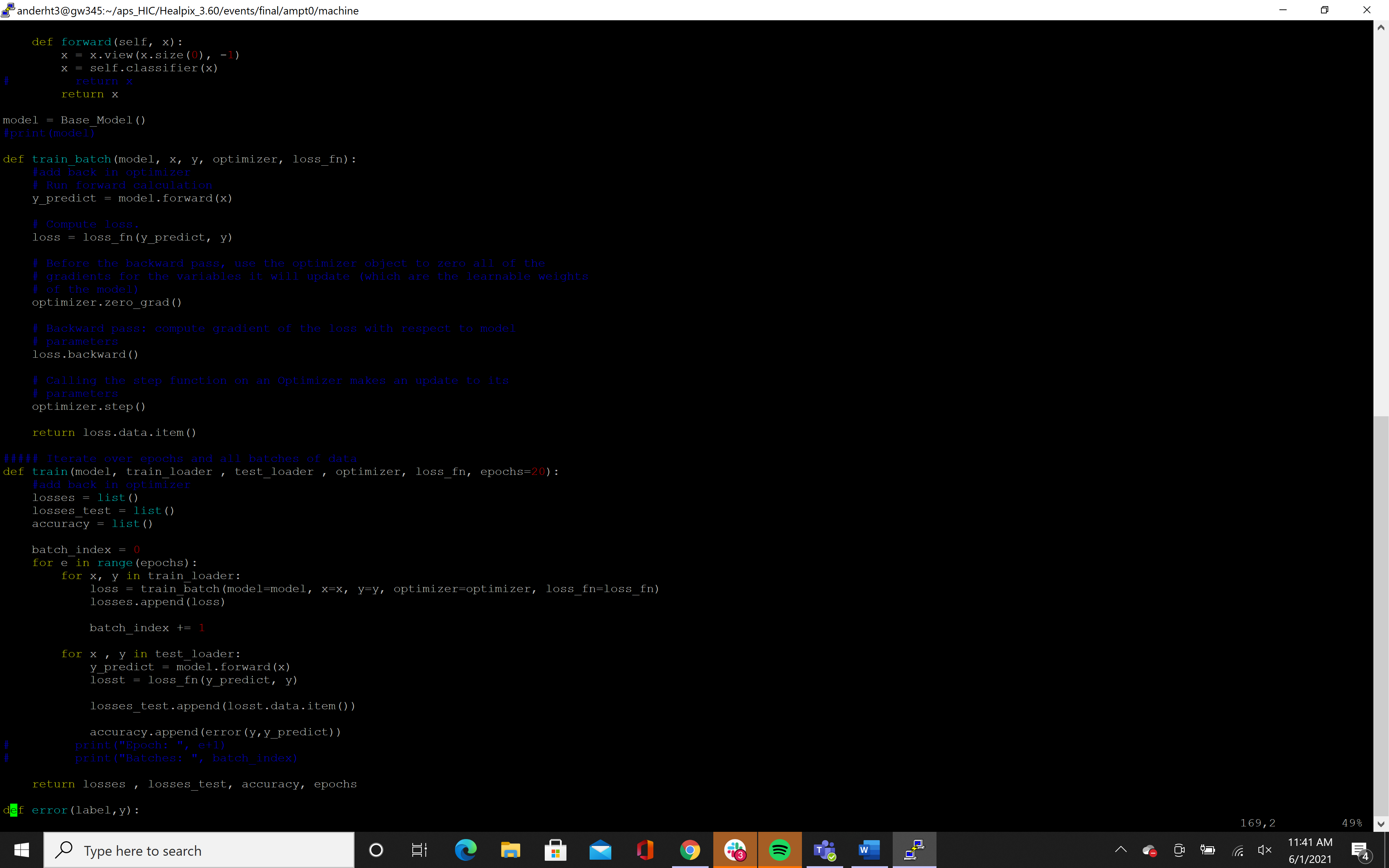
The variable dataiter is simply used along with the code that is commented out to just make sure that the images are stored properly and are paired with the proper label of 0 or 1. The last three lines are just a repeat of the lines for train and test loader but with a different batch size. The train set is shuffled one more time, but the test data is not.



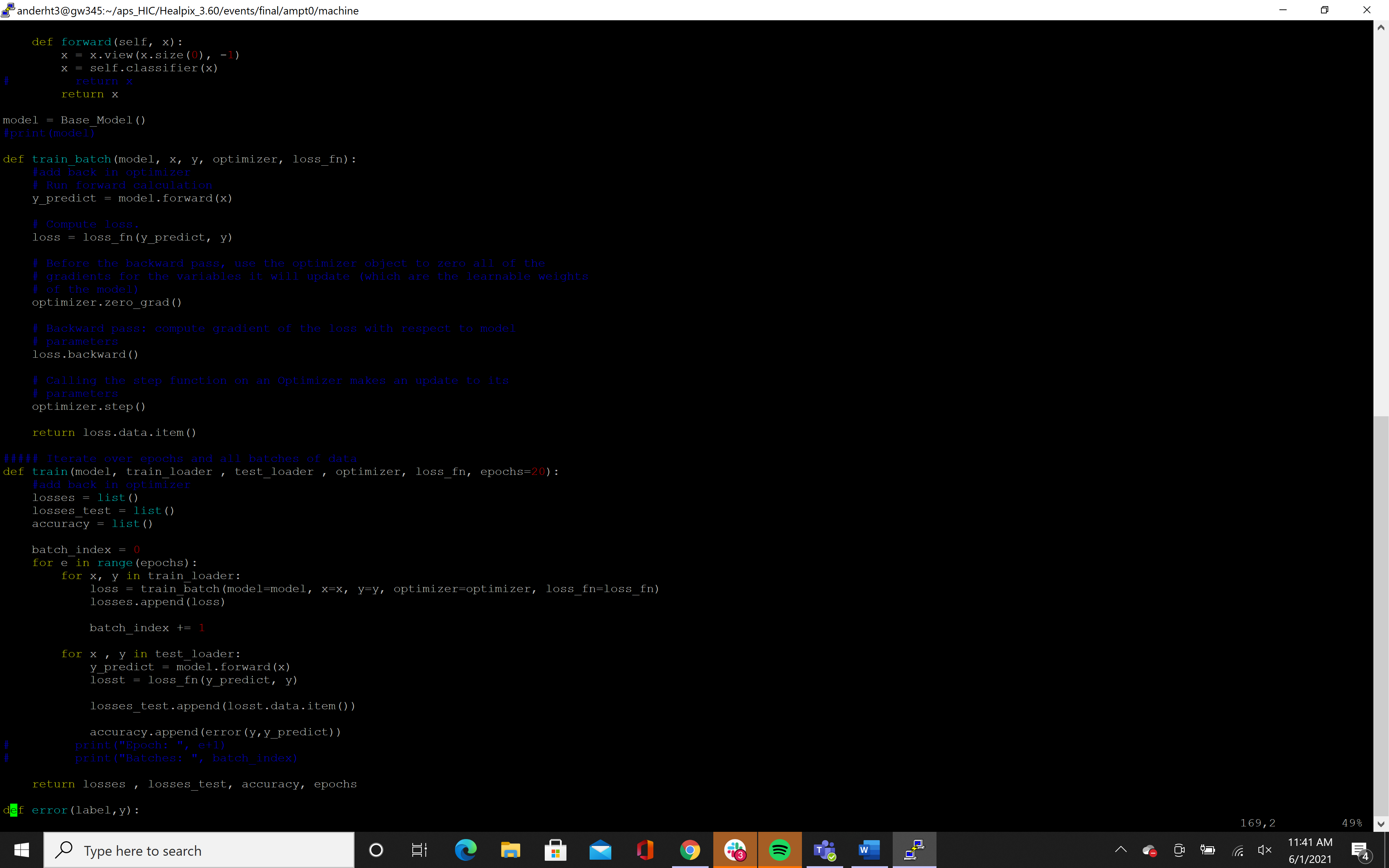
The two methods above are simply for testing to make sure that the images are still clear enough to be able to distinguish their features. The first “imshow” just shows a single image where the second “show\_batch\_images” shows a row of images and the labels that are paired to them. This is also a good way to check if the labels are paired correctly and if they are represented by only 0 and 1.



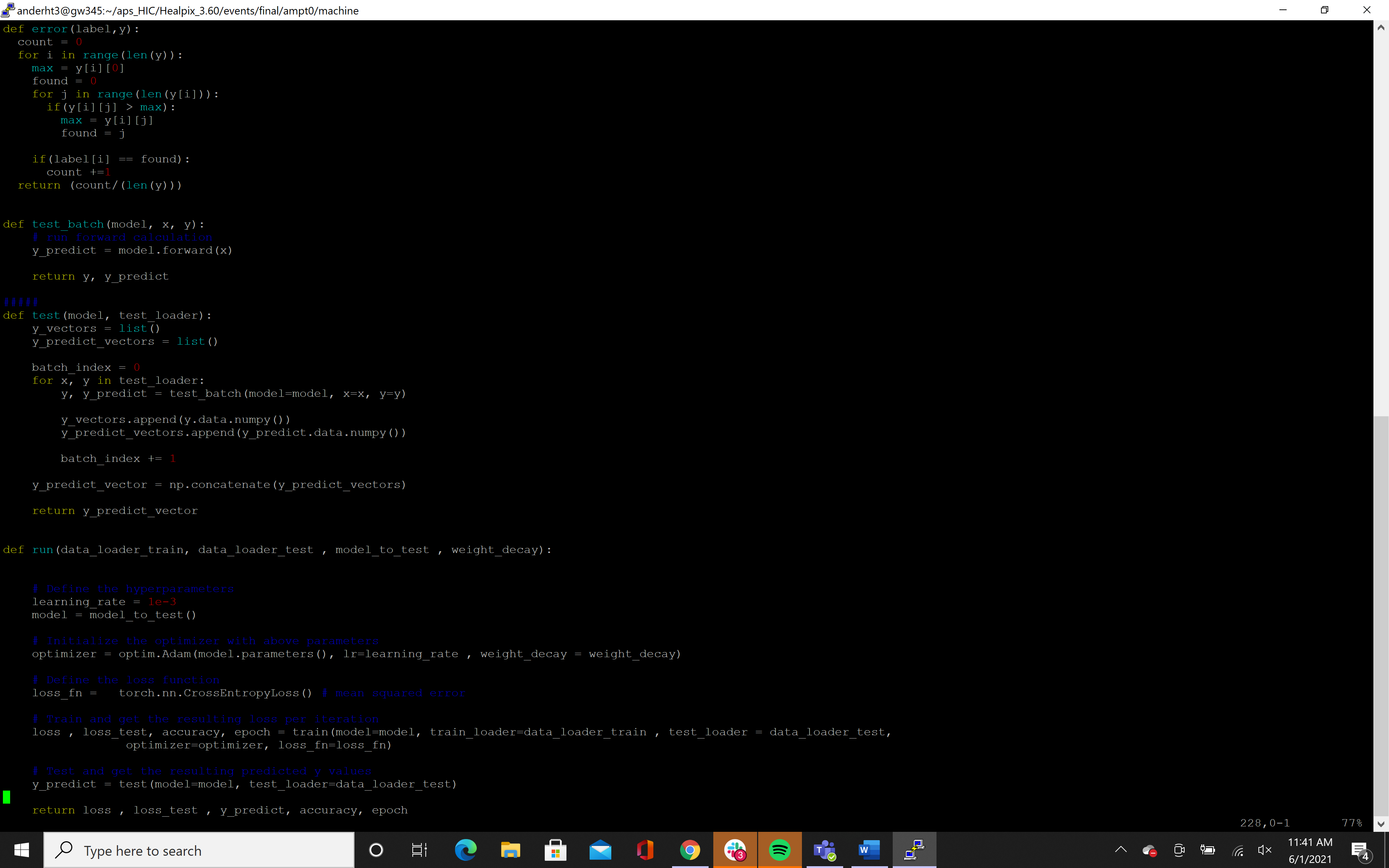
Finally we come to the class where our model is defined. In this model, I am using both linear operators and two types of batch normalization. This is done to improve the quality of the model. The neural network operation “ReLU” adds a rectified linear unit function element wise to the model. This makes the model non-linear after the linear translations through the pixels. The batch normalization simply normalizes each layer. In this block of code, there are currently 9 layers that are implemented. The number of layers, like the batch size, can be adjusted if the model over or under fits the data. The starting number on the first layer is the number of pixels in each image, and the final number in the last linear layer is the number of features. In this case the number of features looked at are two since we have two types of images (default and string melting). The forward pass is simply to move the model forward over the data and implement the model.



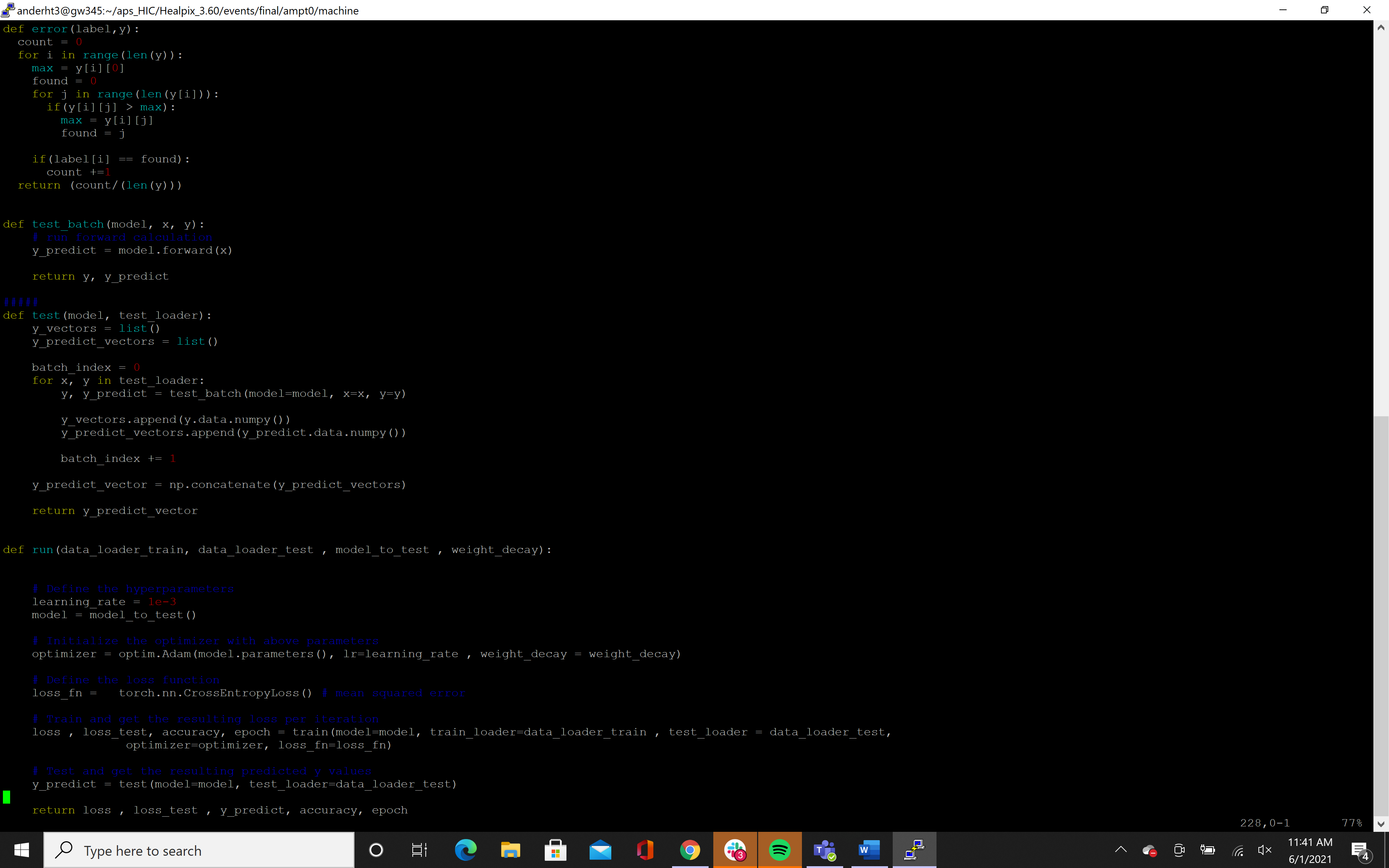
The top line and the line under that is commented out is simply a test to show that the model that was created has all of the correct layers. The function train\_batch is called in a method below called simply “train”. This method train\_batch trains the model on a single batch that is put through from the function train. First, the model moves forward on the input x. X is the images with the specific number of pixels that were specified earlier in the code. The model then processes the images and decides if the output of 0 or 1 is consistent with the label that the images were given through the loss function. The loss function is defined in later code, but this simply gives a score to show how good the model is so far at giving the correct outputs. The optimizer, which is also defined in later code, is made to have zero gradient to make this a linear operator. The loss is then moved backward through the model to show the find the gradient for the optimizer. With this new gradient, the optimizer is updated and the loss is returned.



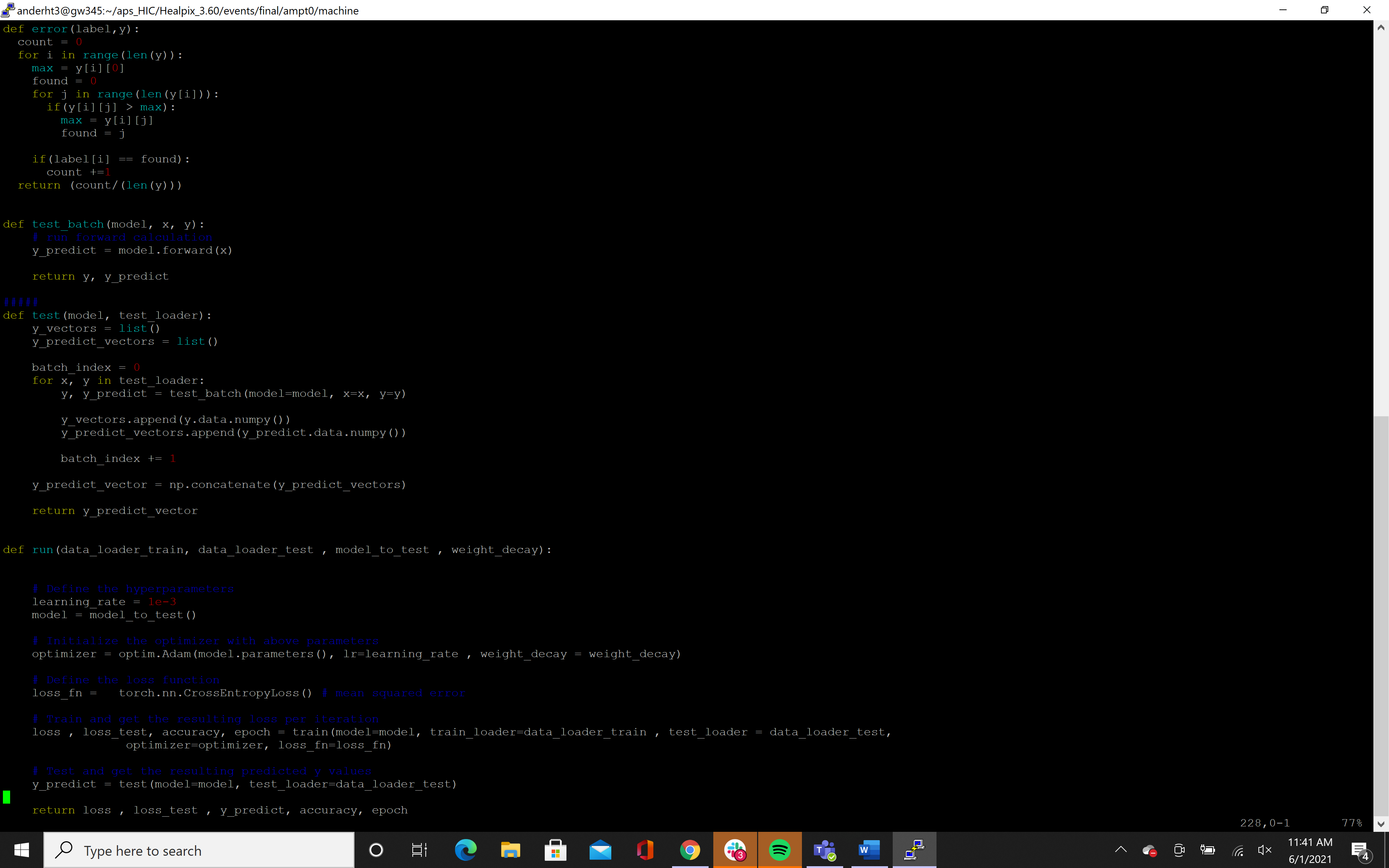
This train method does exactly what it is named; it simply trains the model on the training data sets that were created earlier in the code, but it also implements testing at the end of the code as a check to see how accurate the model has become. There are three for loops in this function. The first uses the variable epochs. The variable epochs is used as a variable to show how many times the model should loop over training and testing the models. Each time it loops, the optimizer updates and improves the model, so ultimately more epochs will be more useful. However, there may come a point where the maximum optimization is reached, so more epochs will just slow the model down rather than improve its accuracy. The second loop is where the training is done. The second loop splits the sets into x and y which are the images and labels respectively. They are also broken down into their batches so the loop processes one batch at a time. After the batches are processed and the loss added to the list of losses, the next loop looks through the training set. Although there is only one batch in the training set, the loop is in place to ensure that any data is not missed. This is able to show if the model is improving or not.



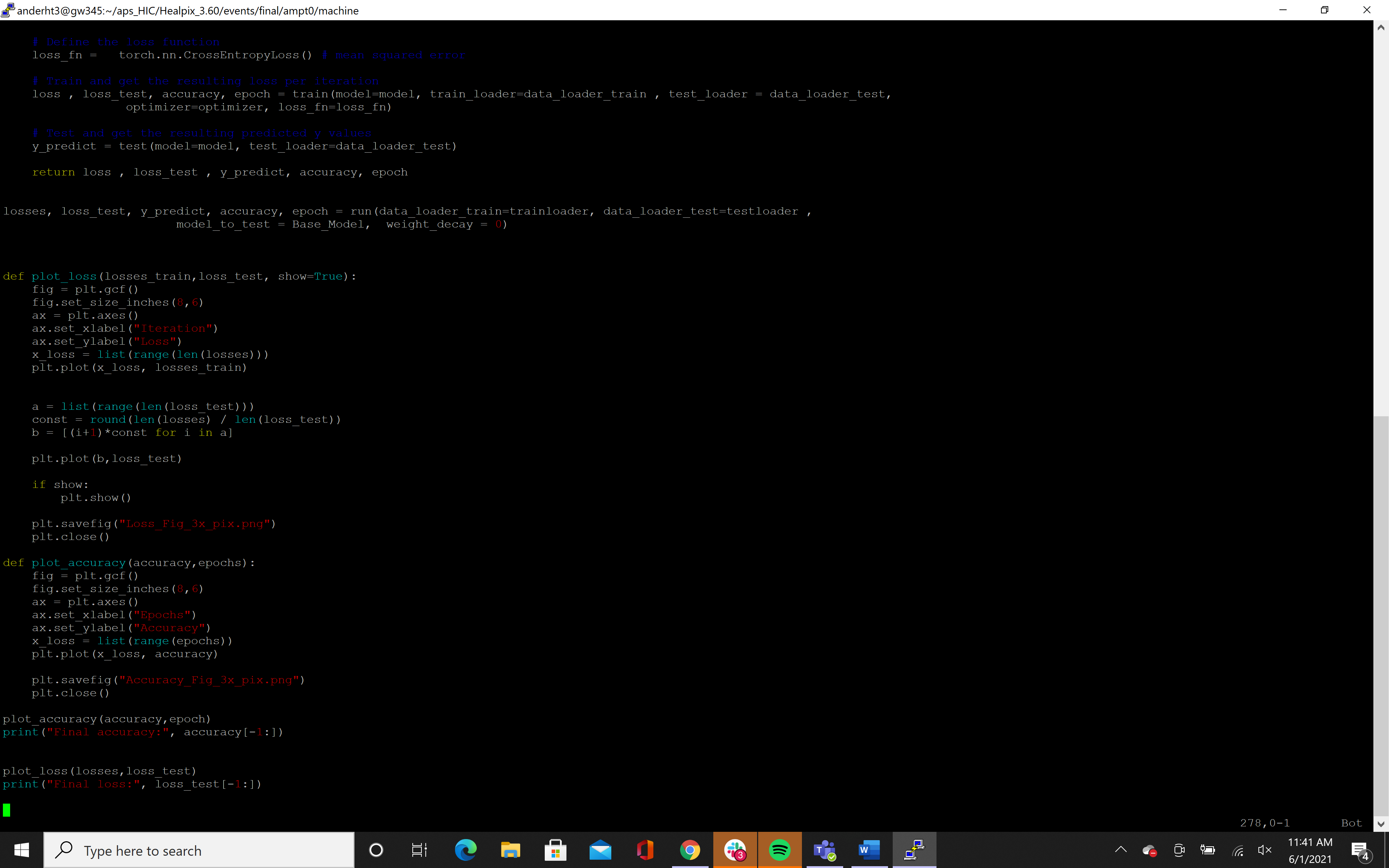
This function simply creates an error function that can be used in place of loss to see how close the two are. This is not necessary and is not used in the current code.



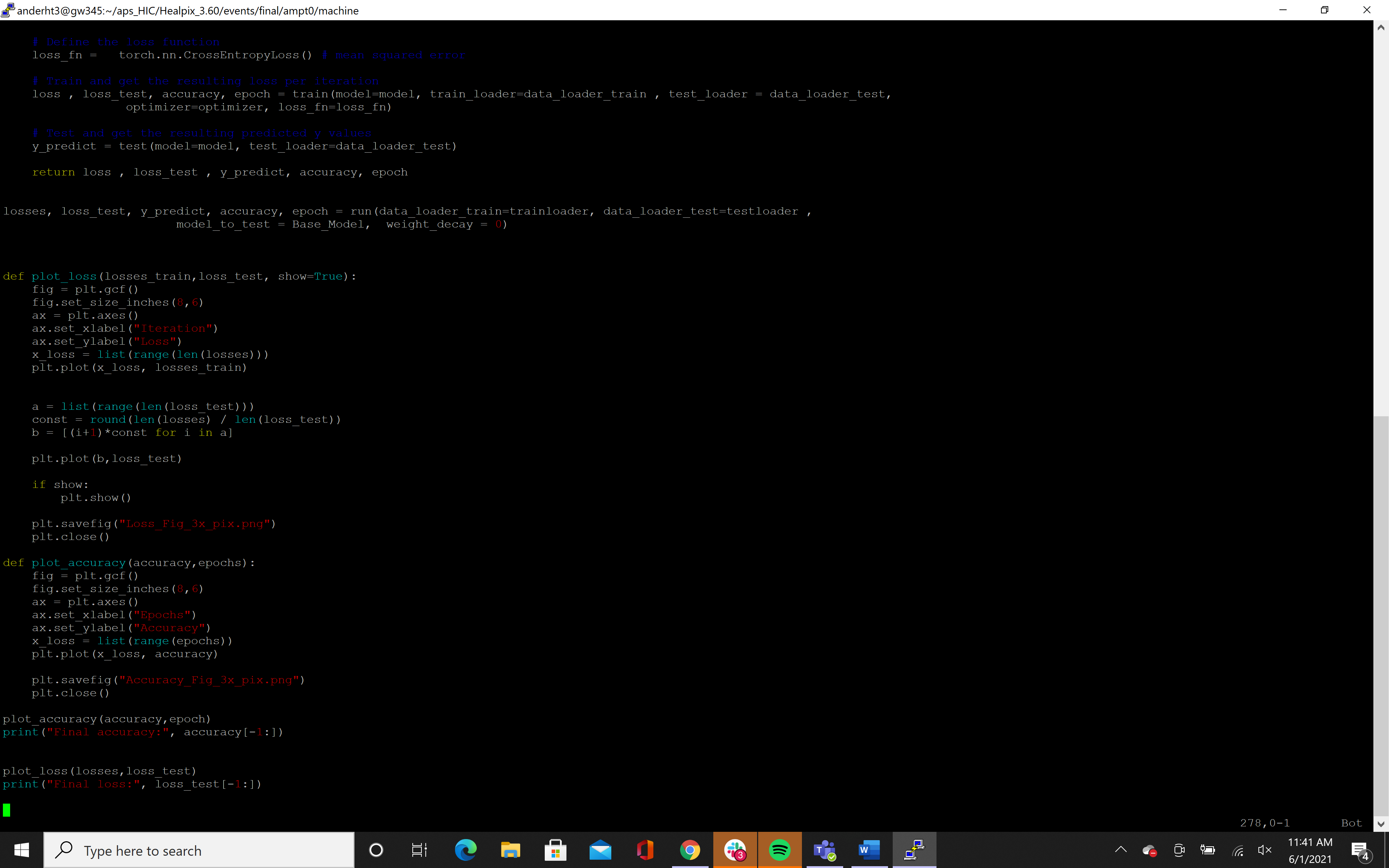
The test and test batch methods are similar to the train methods, but in this method, the model is simply tested with the test images. The value that is returned is just the predicted labels that the model is able to come up with after the training.



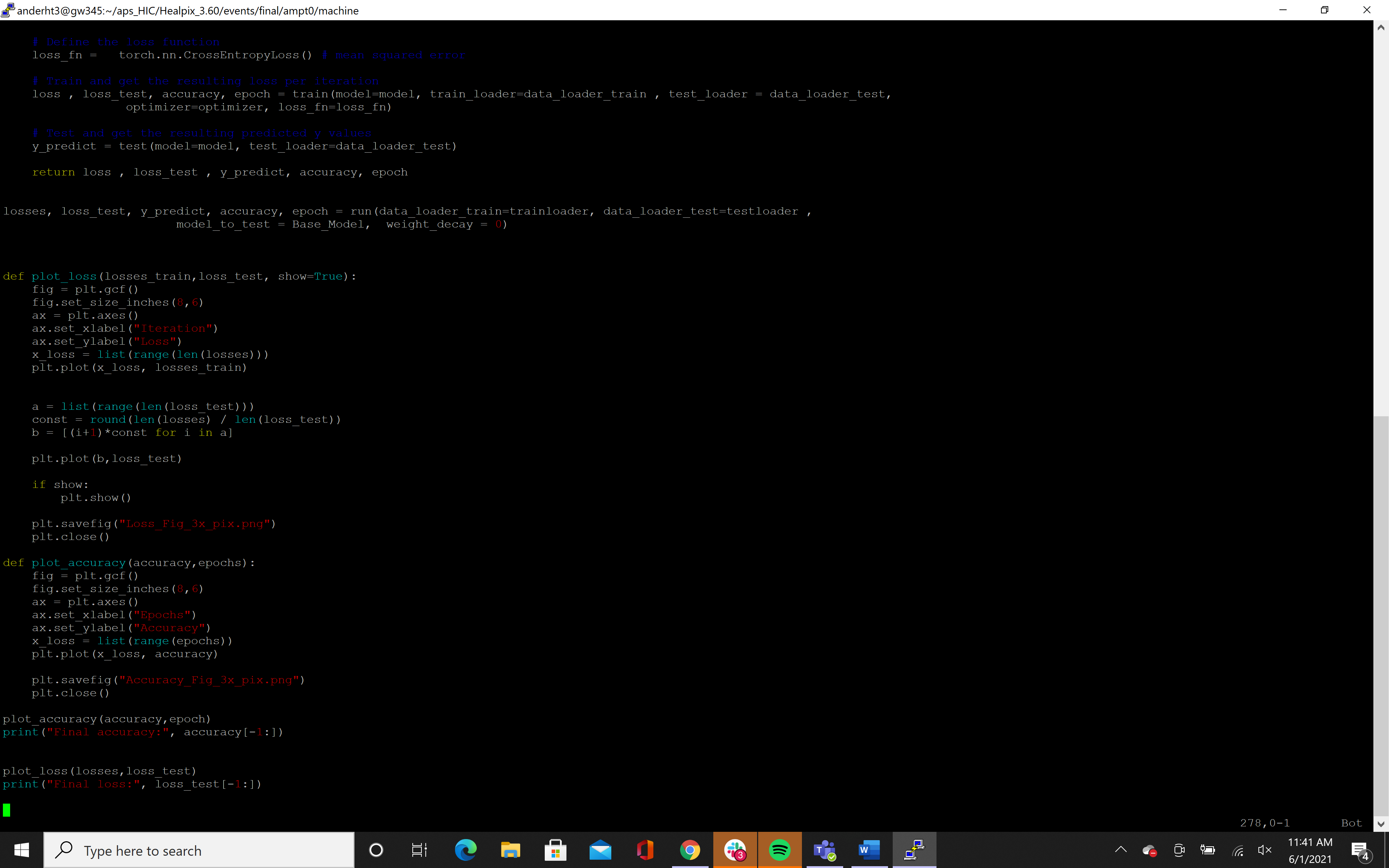
Finally, the run function incorporates all of the functions that have been previously defined. This function also defines the loss function that is used as well as the optimizer that is used. In this specific run, the optimizer that is used is the Adam optimizer, although others can be implemented and may be tested in the future. The loss function is defined by the cross entropy rather than the mean square error since cross entropy works better with variables that may not be the same size. Then this is all compiled and sent through the train and test functions. From these functions we get loss from both our training set and our test set as well as accuracy, the number of epochs used, and the predicted labels.



The model and the data loaders are put into the parameter list and the function runs to give us the outputs that were discussed above.

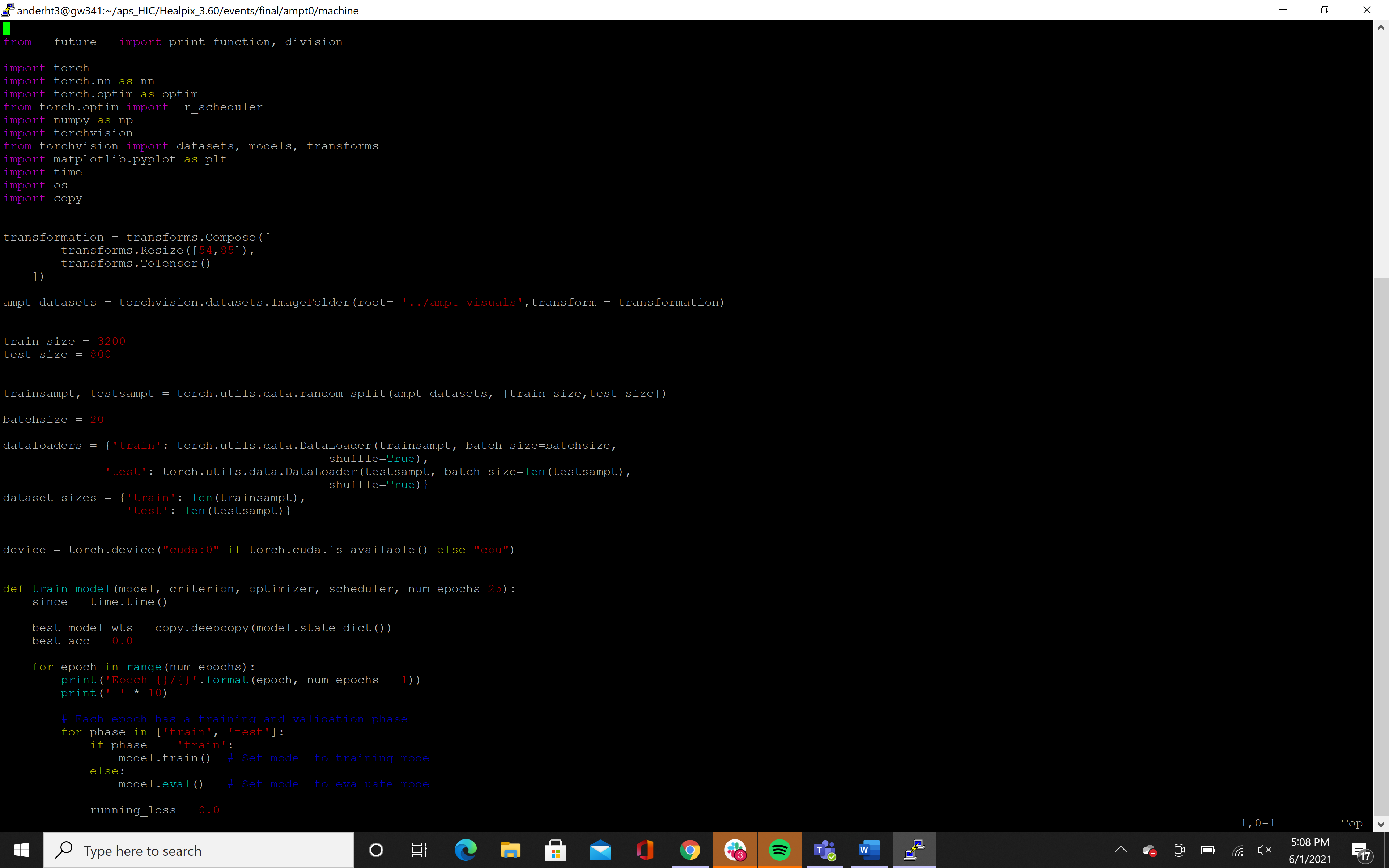


The loss can be plot through this function which does not require much explanation.

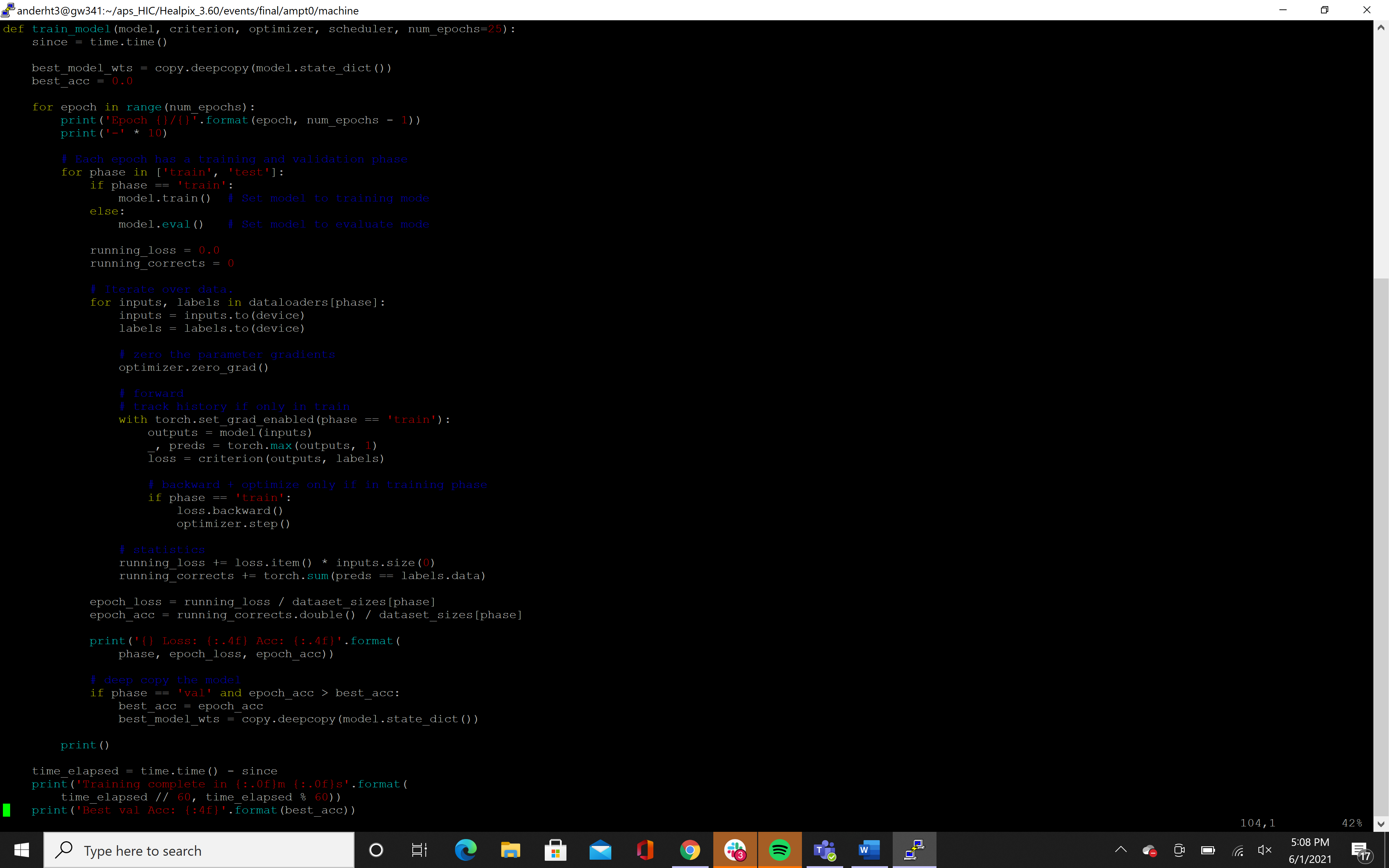


The accuracy can be plotted with this function as well and is very similar to the previous function. These figures are then shown and the final accuracy and loss is printed.

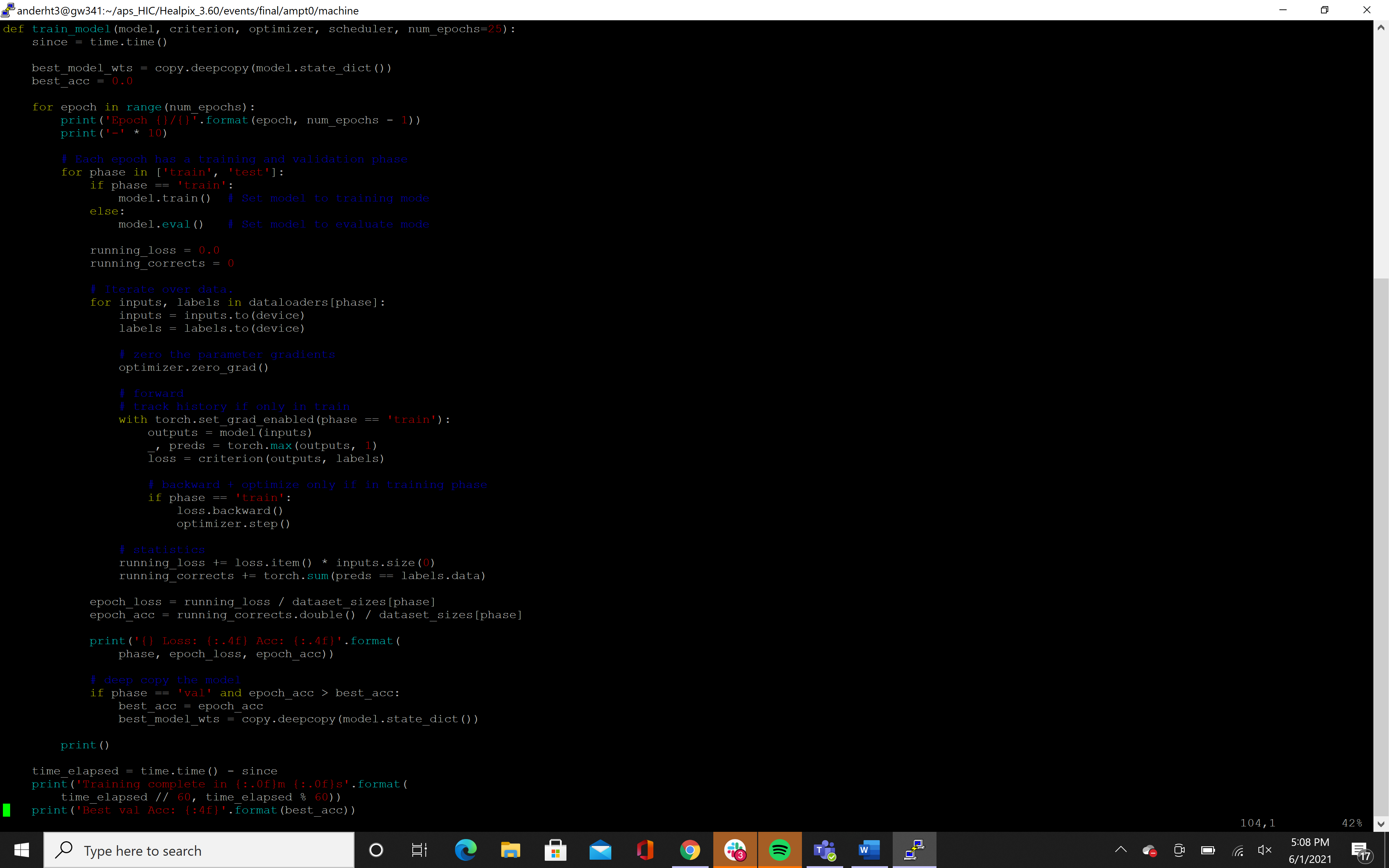
**Documentation for the file CNN.py**



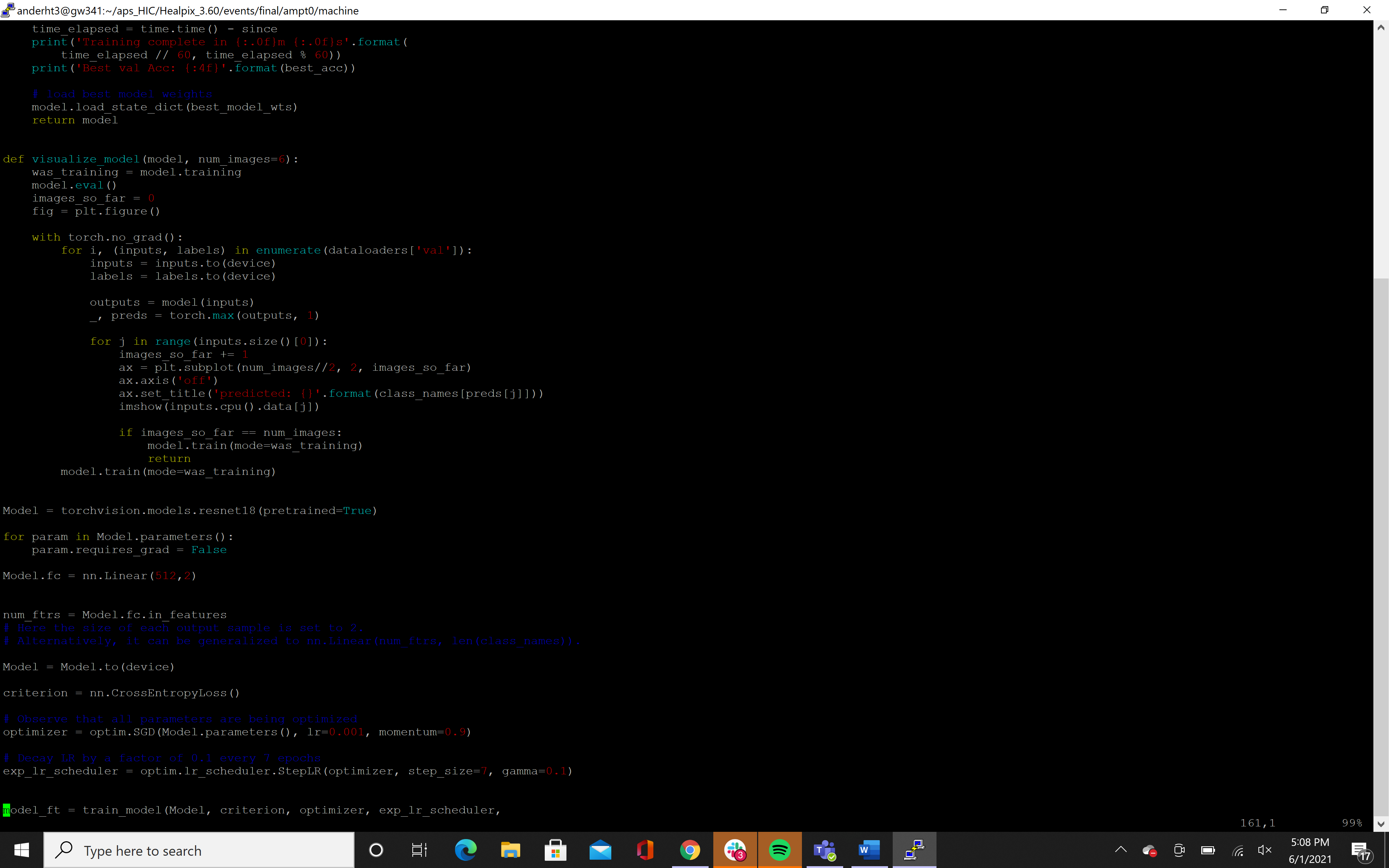
This first piece of code is similar to how the data was loaded into the previous code. The one major difference is in how the dataloaders are processed. Instead of creating separate variables for test and train sets, they are split within the data loaders. This is done to make some of the loops more simple in the next method.



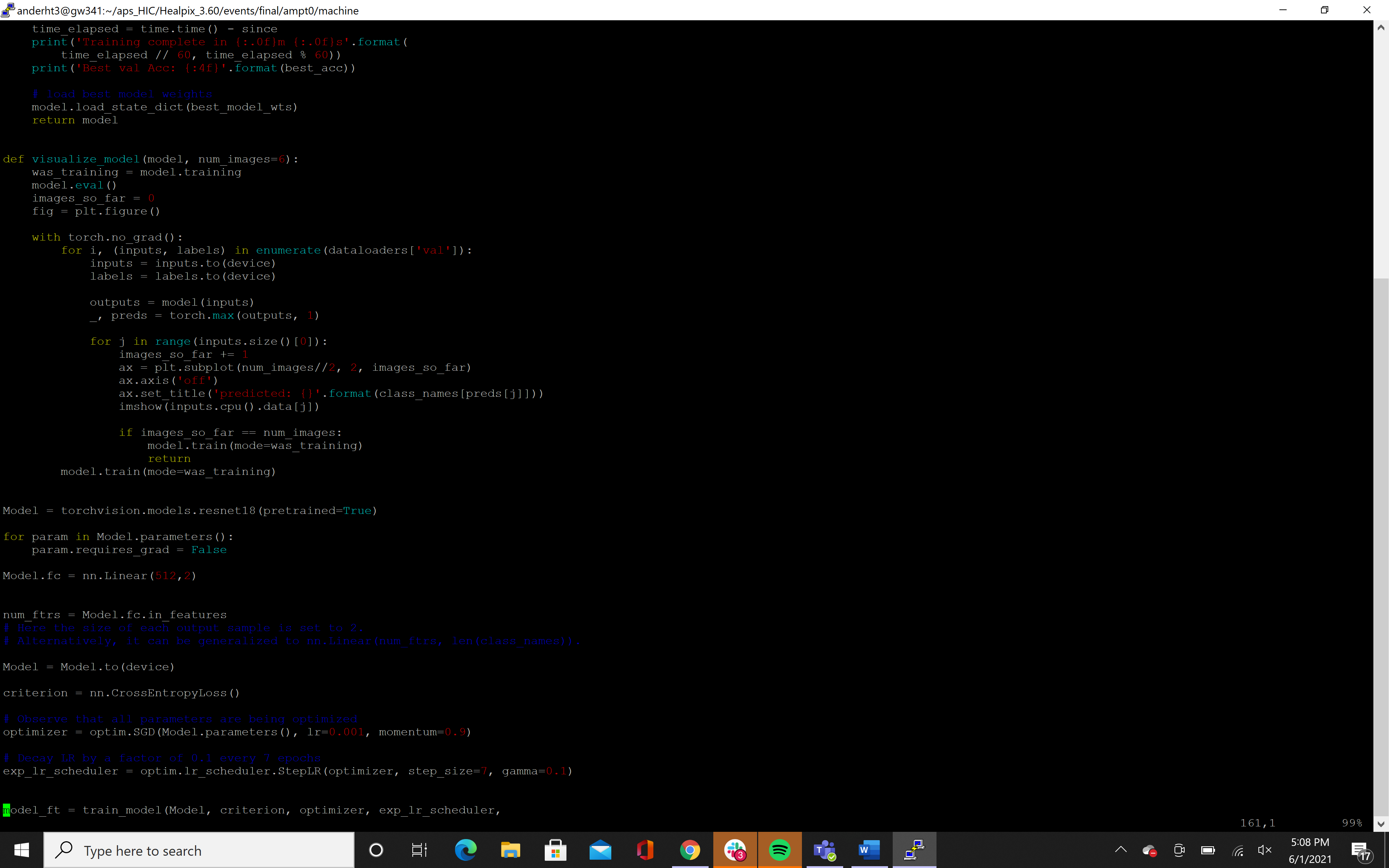
The main method that is used in this file is the train method. There will be no test method as I have yet to implement that piece of code, but there is a section within the training that is used to test. The first for loop is to go through the train and test sets and their respective batchsizes. If the model is set to “test” which is the else in the if statement, then the model evaluates itself, otherwise the model will be trained during the iteration of the for loop. For the next for loop, if the model is set for “train” the data for each batch size is put into arbitrary variables for inputs and labels in order to train the model.



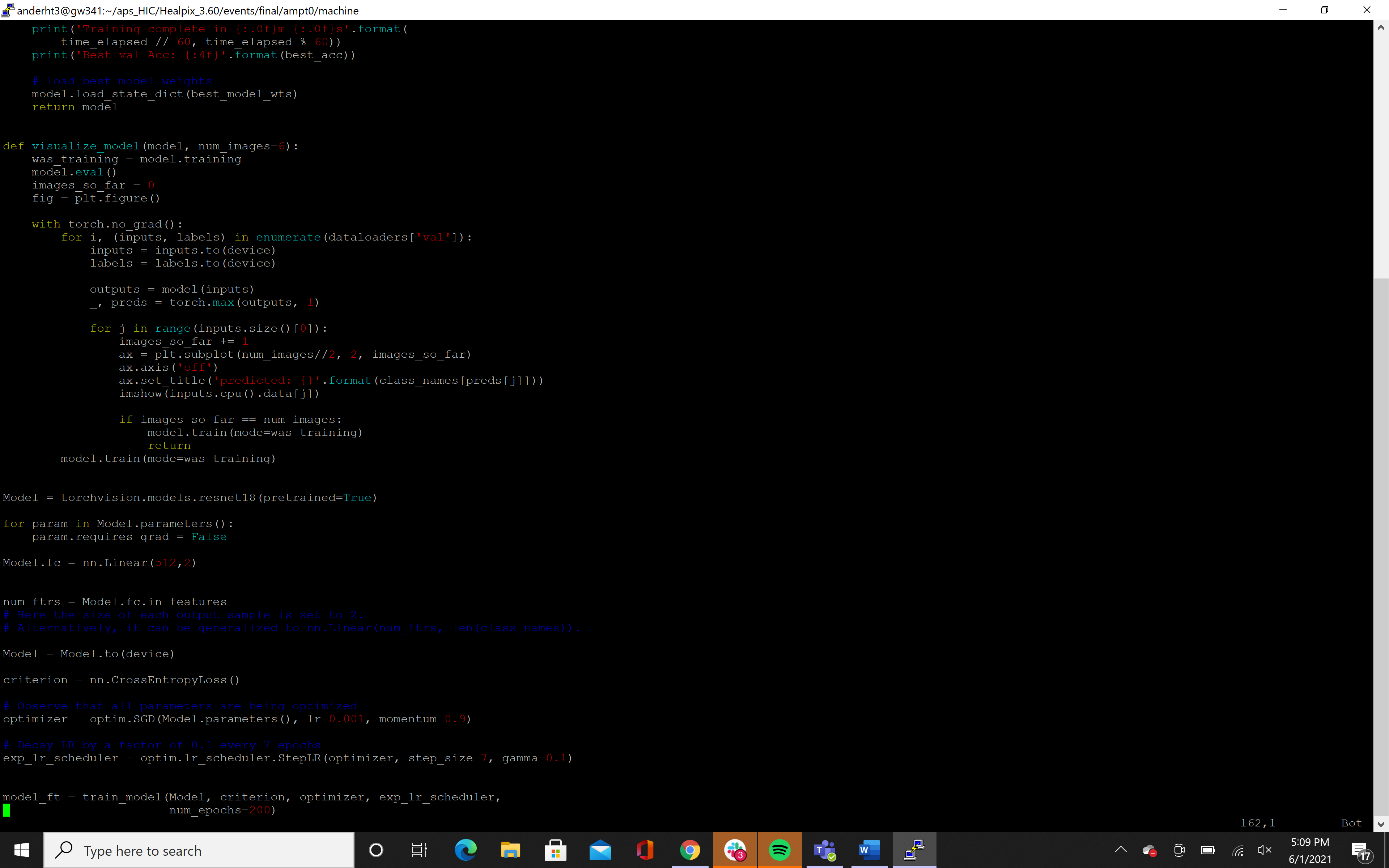
The model then runs the inputs, and the outputs are compared to the labels. The loss then moves backward and the optimizer is updated much like the previous file. The loss is stored in variables, and there are variables created to look at the loss during the different epochs as well. The final if statement should say ‘test’ not val, but it just tests the model again.



When this code is finished running, then a model is returned that is fully trained.



This simply shows what the model is doing, and is not necessary for our purposes.



Finally, the model that we are working with is a resnet18 with one linear neuron. The resnet model is able to create its own internal neurons that move from the beginning number of features to the final number of features. The optimizer is a SGD optimizer which works more smoothly for convolutional neural networks than the Adam optimizer. Then the model is run.