Please submit your solutions for this assignment on Canvas by **Monday, Nov. 21st**. Post both **your code** and **a screenshot of its output** along with the answer to the bolded question below (about the number of weights).

We now wish to build and train a dense — that is, fully-connected — feed-forward neural net to solve the general digit classification problem. Let us agree to go ahead and use a non-linear model. This means that we need to **split out and evaluate on testing data** to gauge true performance on data unseen during training.

Please refer to Exercise 7 (on this DL@DU project page) where we define a so-called `LogSoftmax` model.

As discussed in class, piping the output of our hidden layers through `softmax` allows us to naturally interpret of the output as a *discrete probability distribution*. Since in our current project involves 10 categories, `softmax` looks like this:

$$\text{softmax}(x_1, x_2, \ldots, x_{10}) = \left(\frac{e^{x_1}}{\sum_{i=1}^{10} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{10} e^{x_i}}, \ldots, \frac{e^{x_{10}}}{\sum_{i=1}^{10} e^{x_i}} \right).$$

However, for optimal training in practice we output through `log_softmax` and use `NLLLoss` negative log-likelihood loss as outlined here.

You may use the model on the DL@DU page (which has a single hidden layer of width 200) or any non-linear dense model you wish. As part of your solution, **compute (or approximate) and post the number of weights (i.e., trainable parameters) in your model**.

Notes:

- Since we are using a non-linear model, we need to split out test data and use it to validate our trained model.

  Suppose, as usual, that the features of our data are housed in a tensor `xss` in such a way that the examples are indexed by the first *axis*, and that the corresponding targets live in `yss`.

  In our current digit classification problem, `xss` is a `torch.Tensor` of size `5000x400`, while `yss` is a `torch.LongTensor` of size `5000`.

  A sensible and readable way to perform a 80/20 train/test split is:

  ```
  indices = torch.randperm(len(xss))
  xss = xss[indices]; yss = yss[indices]  # coherently randomize the data
  xss_train = xss[:4000]; yss_train = yss[:4000]
  xss_test = xss[4000:]; yss_test = yss[4000:]
  ```

- In a classification problem, you do **not** need to center or normalize the outputs `yss`.
- You should likely center the features `xss` but there is no real need to normalize them.
- You can use DUlib's `train` function if you wish:

  ```
  import du.lib as dulib
  ...

  model = dulib.train(
      model = model,
  ```

```
    crit = nn.NLLLoss(),
    train_data = (xss_train, yss_train),
    valid_data = (xss_test, yss_test),
    #
    #  your other training parameters
    #
)
...
```

To graph, you can add the parameter `graph=1`.

- Feel free to use Dulib's `class_accuracy` function:

```
# gauge performance on training data
pct_training = dulib.class_accuracy(model, (xss_train, yss_train), show_cm=False)
print(f"Percentage correct on training data: {100*pct_training:.2f}")
```

When validating on test data, you may wish to print the confusion matrix by setting `show_cm=True`.