1. structure_cellcensus_query is really just to get the list of cell types below a certain point in the ontology
    1. Top of the defined ontology could be the top of the full ontology CL:0000000 (which I think is just called “animal cell”) or the hematopoietic cell category (CL:0000988)
    2. Saved to turbo are cell_type_list.txt (hematopoietic) and cell_type_list_full.txt (full census)
2. mccell_preprocess_from_disk builds everything else you need from the cell type list
    1. IMPORTANT: filters out rarer cell types (<5000 cells)
    2. After running structure_cellcensus_query you need to change both cell type list file and the upper_limit variable in mccell_preprocess_from_disk for it to work properly
    3. Will save important variables to files that have the date at runtime as part of the filename
        1. This includes a new list of filtered cell types
3. model_from_disk
    1. Input in the date for the preprocessed data you want to use
        1. 10-11-24: all
        2. 10-14-24: hematopoietic
    2. Define batch size and soma chunk size at datapipe definition
        1. Increasing the shuffle_chunk_count parameter causes more randomness
    3. train_percent/val_percent: how we split up training and validation sets
        1. sub_percent: subset the data for testing things, using the .header() function of datapipes
            1. .header() function works by using number of batches as a parameter
            2. NOTE: preeetty sure the new shuffling code and subsetting play well. Haven’t tested it much though, so you’ll want to keep an eye on it
            3. Neural network is pretty straightforward. There are variables governing hidden layer sizes you can adjust. Leaf loss weight definition is in a nearby cell
        2. Main function for training is marginalization_classification_manual_batch
            1. I’ve never really needed to change the default function parameters, but you can specify the number of epochs you train on with num_epochs though!
            2. Can change learning rate (adam_lr) inside of function
        3. Two kinds of optimizers. I usually used SGD, Josh Fuchs used Adam (and that one works better).
            1. There’s also an lr scheduler, which I did not attempt to use. Could be interesting to check out
        4. Will save best performing model with today’s date in the name
            1. Same with a graph
    4. NOTE: for cellxgene==1.15.0, I had to replace cell_type_encoder.transform(cell_type_encoder.classes_) with the simpler np.arange(len(cell_type_encoder.classes_)) to assemble the value for the encoding_mapper parameter. Perhaps there’s a more elegant way to do this that I’m not aware of. Worth looking into!
    5. Seems the call to init_weights() was commented out. Worth uncommenting and seeing what happens
4. Submitting slurm jobs: there are three files that are designed to search over desired hyperparameters
    1. mccell_tune.py is a python program that mostly copies the model_from_disk notebook code

      1. Saves network model at the end of each epoch
      2. Uses Weights and Biases to save F1 scores at each epoch
      3. Takes in command-line input for a hyperparameter value (you need to go into the code yourself and edit which hyperparameter is affected).
      4. Also important to rename the Weights and Balances project for each hyperparameter experiment you try
         1. The job id I usually define as the input value so I know which graph was caused by which value
   2. tune_job.sh is a single slurm job that runs mccell_tune.py and feeds it the desired hyperparameter value. It also cd’s into the correct directory and activates the desired conda environment
      1. This ALSO takes in command line input fed in from send_jobs.sh
   3. send_jobs.sh contains an array with the desired hyperparameter values, loops over them, and submits a slurm job with each as an argument.
5. Preprocessed code from structure_cellcensus_query and mccell_preprocess_from_disk can be found in /nfs/turbo/umms-welchjd/mccell