Part 1: Multi-Layer Perceptrons (9.5 points)

The file `iml/dl.py` defines a basic framework for building multi-layer models. The `forward` method of a `Module` is meant to implement a "forward pass," an arbitrary operation mapping some inputs to some outputs. The `backward` method is meant to carry gradients backwards and perform parameter updates.

More specifically, if the `forward` method of a `Module` produces an output y from an input x and an internal parameter θ, then the `backward` method should receive a gradient $\partial L/\partial y$, perform some parameter update based on $\partial L/\partial \theta$, and finally return the gradient $\partial L/\partial x$. See the implementation of the `Linear` module for an example.

`Module`s admit a natural kind of composition which I have implemented in the `Sequential` class. Specifically, `forward` methods compose as functions in the usual way, and `backward` methods are composed in reverse. As long as you've defined `backward` correctly for all sub-modules, `Sequential.backward` automatically carries out the "back-propagation algorithm."

The goal of this problem is to use the `Module` system to train an multi-layer perceptron (MLP) on the MNIST handwritten character dataset.

(a) Using the `mnist_iter()` method from `dl.py`, load and display some datapoints from MNIST. Using `LogisticRegression` from sklearn, train a baseline model on 2000 datapoints from the training set and report its accuracy on the test set. (1 point)

(b) Using the `Linear` module from `dl.py`, train your own logistic regression model on the MNIST dataset using gradient descent on cross-entropy loss. Run for two epochs on minibatches of size 256, and ensure your model has at least 90% accuracy on the test set. (2 points)

(c) Write a `ReLU` module that applies the ReLU activation. (Remember that ReLU replaces negative inputs with zero and passes positive inputs through.) Explain how you derived the backward pass. (1.5 points)

(d) Using `Linear` and `ReLU` modules, build some kind of multi-layer perceptron and train it on MNIST. Ensure your model has at least 95% accuracy on the test set. (3 points)

(e) **Bonus**: Still using only `numpy` and `iml/`, design a MLP that achieves at least 98% test accuracy and takes less than 5 minutes to train on a typical consumer CPU. (2 points)

Part 2: Linear Autoencoders (7.5 points)

In this exercise you'll implement a simple *autoencoder*. An autoencoder has two parts: an *encoder* f which sends an input $x \in \mathbb{R}^d$ to a "code" $f(x) \in \mathbb{R}^k$, and a *decoder* g which sends a code back to a vector in $\mathbb{R}^d$. The reconstruction loss of an autoencoder is an expectation like

$$L = \mathrm{E}_X \|g(f(X)) - X\|^2$$

where X is a vector drawn from some dataset. In this exercise, we measure reconstruction loss as mean squared Euclidean distance between autoencoder output and our *original data.*

As in Part 1, do not use any libraries besides numpy and `iml/dl.py`.

(a) Let's start by training the model `Linear(3, 3)` on `ulu.csv` with the reconstruction loss objective. Run full-batch gradient descent on the raw data. Show your loss curve over 50 iterations and determine, without doing any additional calculations, if gradient descent seems to have a good rate of convergence. (1 point)

(b) Consider the `ulu.csv` dataset. Make an autoencoder that encodes datapoints as two-dimensional vectors. Using only `numpy` and `iml/`, minimize its reconstruction loss (in squared Euclidean distance) using gradient descent. Report the final reconstruction loss and confirm that it is close to the theoretical minimum. (2 points)

(c) Now implement a `DropoutOne` module whose forward pass zeros out the first dimension of each input vector with probability 0.5. What is the correct way to implement a backwards pass for this module? (1.5 points)

(d) By incorporating `DropoutOne` into the forward pass, train an autoencoder in such a way that the first dimension of the code reliably distinguishes the foreground and background parts of `ulu.csv`. Ensure that your model converges within 500 iterations of full-batch gradient descent. What is the optimal expected reconstruction loss for this model? (3 points)

Part 3: Bouba–Kiki (7 points)

Almost everyone has an intuitive grasp for the concepts of roundness and sharpness. In fact, it's possible to talk about roundness and sharpness without reference to a pre-established language; in 1929, Wolfgang Köhler showed that subjects with different cultures and languages usually agree on how to relate the nonsense words *bouba* and *kiki* with round and sharp shapes. In this exercise, you'll investigate whether a small image model (MobileNetV3-small) also has a notion of roundness and sharpness.

The file `data/bouba_kiki.npz` contains a dataset of 200 100×100 black and white images, which can be loaded with `numpy.load`. Half have round shapes, and the other half have sharp shapes.

(a) Load MobileNetV3-small with `torchvision.models.mobilenet_v3_small()`. Briefly describe the idea of a convolutional linear layer, and count the number of weights in this model. (1 point)

(b) Using MobileNetV3-small, separate the images in `bouba_kiki.npz` into a bouba class and a kiki class. Confirm that your technique works by showing a few examples of the classification. (I recommend upscaling the images to 244×244 with `Resize` from `torchvision.transforms` before passing them to the model.) (2.5 points)

(c) Fetch some images of frogs and airplanes from the CIFAR100 dataset. Using your method from the previous problem, determine whether frogs or airplanes are more kiki. Do you agree with the model's judgment? (1.5 points)

(d) **Bonus**: Can you identify the earliest layer of MobileNetV3 that reliably distinguishes between the bouba and kiki examples in our dataset? (2 points)