CS 457/557: Assignment 3

Due: December 2, 2024, by 11:00 PM (Central)

Overview

In this assignment, you will write a program that can be used to train and evaluate a neural network for multi-class classification. Your program will allow the user to specify the structure of a multi-layer feed-forward neural network and then will train that network using backpropagation in conjunction with mini-batch gradient descent.

Note: Depending on how you structured your code for Assignment 01, you may be able to reuse some parts of it here, and it is acceptable to do so. However, you might also find it easier to rewrite things from scratch with improvements based on lessons learned from Assignment 01.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "neural network in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**
3. **(NEW)** An `experimentation.txt` text file that describes the results of your experimentation (see page 10 for details) and includes the best performance that you were able to achieve on several of the sample data sets along with the command-line arguments that you used to achieve this.

Program Requirements

The general outline of the program is given below:

Algorithm 1 Program Flow

```
Process command-line arguments
Load data from the specified file
Split data into training and validation sets (use 80% of data for training)
Learn min-max normalization (feature scaling) parameters from the training set
Use the "learned" scaling parameters to rescale attribute values for all data
Set up neural network architecture based on command-line arguments
Train the neural network on the training data
Evaluate the accuracy of the neural network on the validation set
```

The program should be runnable from the command line, and it should be able to process command-line arguments to update various program parameters as needed. The command-line arguments are listed below:

- `-f <FILENAME>`: Reads data from the file named `<FILENAME>` (specified as a `String`); see the **File Format** section for more details.
- `-h <NH> <S1> <S2> ..`
  Specifies the number of hidden layers `<NH>` followed by `<NH>` additional integers corresponding to the sizes of these hidden layers. You can assume that `<NH>` is a non-negative integer no greater than 10. The following integer(s) `<S1>`, `<S2>`, etc. represent the sizes of hidden layer 1, hidden layer 2, and so on. There will be `<NH>` size values, and you can assume that each of these values will be a positive integer no greater than 500. For example, `-h 2 10 5` specifies a neural network with two hidden layers, the first containing 10 neurons and the second containing 5. Note that `<NH>` may be `0`, in which case there are no hidden layers and no additional integers following. (Using zero hidden layers is the default.)
- `-a <DOUBLE>`: Specifies the learning rate α in mini-batch gradient descent; default is `0.01`.
- `-e <INTEGER>`: Specifies the epoch limit in mini-batch gradient descent; default is `1000`.
- `-m <INTEGER>`: Specifies the batch size in mini-batch gradient descent; default is `1` for stochastic gradient descent (using `-m 0` should be interpreted as full batch gradient descent).
- `-l <DOUBLE>`: Specifies the regularization hyperparameter λ; default is `0.0` (no regularization).
- `-r`: If specified, this flag (which has no arguments) enables randomization of data for the train/validation split **and** for batch construction at the start of each epoch; if this flag is not specified, then data should **not be randomized** (which is the default behavior; this is to make testing and debugging easier).
- `-w <DOUBLE>`: Specifies the value ϵ for weight initialization; default is `0.1`.
- `-v <INTEGER>`: Specifies a verbosity level, indicating how much output the program should produce; default is `1` (See the **Output** section for details)

The `-f <FILENAME>` option is required; all others are optional. Filenames may include path information, so do not assume that the files are located in the same directory as your source code. You can assume that your program will only be run with valid arguments (so you do not need to include error checking, though it may be helpful for your own testing). Your program must be able to handle command-line arguments in **any** order (e.g., do **not** assume that the first argument will be `-f`). Several example runs of the program are shown at the end of this document.

File Format

The data for your program will be specified in files with lines corresponding to individual examples or data points. Each line will contain the example's attribute (feature) value(s) (all of which are quantitative, or continuous-valued) and an output (target) vector with length corresponding to the number of classes. The output vector will have exactly one 1, and all remaining entries will be 0. Lines that are empty or lines that start with a `#` character should be skipped. Example data with two features and three classes is shown below:

```
# This is a comment line.
(0.665 0.790) (0 1 0)
(0.272 0.412) (1 0 0)
(0.936 0.872) (0 0 1)
(0.896 0.081) (1 0 0)
(0.418 0.741) (0 1 0)
```

Several example input files are provided in the `a03-data.zip` archive on Canvas. In particular:

- The files prefixed with `iris-` contain a slightly modified version of the Iris dataset.
- The `mnist.dat` file is a slightly modified version of the MNIST database which contains handwritten images of the digits 0 through 9.
- The `image-05.dat`, `image-10.dat`, `image-15.dat`, and `image-20.dat` files contain image data of various letters and digits (specifically, the letters 'C', 'A', 'M', and 'F', and the digits '0' through '9') at several different resolutions (with `05` being the lowest and `20` being the highest).

Training and Validation Sets

After reading the data, your program will need to construct a training set and a validation set from the data. You should split the data into these two sets, using the **first** 80% for training and the rest for validation. If the `-r` command-line flag is specified, then the ordering of records in the full data set should be randomized prior to the split.

Feature Scaling

After constructing the training and validation sets, your program will need to scale the feature values so that they are in the range $[-1, 1]$ (or close to it) using min-max normalization. Let L_j and U_j denote the minimum and maximum values observed for feature j in the **training** data. Then min-max normalization scales the feature values $\mathbf{x}_i$ for example i to $\mathbf{x}'_i$ by using

$$x'_{ij} = -1 + 2\left(\frac{x_{ij} - L_j}{U_j - L_j}\right)$$

for each feature j. If example i is in the training set, then it is guaranteed that $x'_{ij} \in [-1, 1]$ for each feature j. We cannot make this guarantee if example i is in the validation set, but it is usually close to this interval. Note that if $U_j = L_j$, then you can simply scale each feature to -1 to avoid dividing by zero.

(The reason we "learn" the feature scaling parameters from the training data is to avoid data leakage from the validation set in the form of minimum and maximum attribute values.)

Network Architecture

Next, your program should build a fully-connected feed-forward neural network. The number of input neurons in the network should match the number of features in the training data (plus the bias neuron), and the number of output neurons in the network should match the number of classes (which is the length of the output vectors).

The number of hidden layers and the size of each hidden layer should match the values provided through the command-line arguments (if the `-h` flag was not specified or was specified with `-h 0`, then no hidden layers should be included).

How you handle the bias neuron (i.e., neuron 0 from the lectures, with a constant output of $a_0 = 1$) is up to you. You can implement it as a single neuron in the input layer with connections to every other neuron in the subsequent layers, or you may wish to have a separate bias neuron in each layer that connects to just the neurons in the subsequent layer (which may make it easier to manage the weights, depending on how you store them). You could also use another configuration, as well.

By default, each processing neuron (i.e., those that are neither input neurons nor bias neurons) should use the **standard logistic activation function** given by $g(t) = \dfrac{1}{1+e^{-t}}$ for all $t \in \mathbb{R}$. Recall that the derivative of the standard logistic function is given by $g'(t) = g(t)(1 - g(t))$ for all $t \in \mathbb{R}$. You **may** add support for alternative **activation functions** for a small amount of **extra credit**; see page 11 for details.

Optimization Components

All of the weights in the network should be initialized to a random value in the range $(-\epsilon, \epsilon)$, where ϵ is a small positive constant. The default value of ϵ should be 0.01, but this can be overridden by using the `-w` command-line argument.

The cost function is the average loss over the training set plus a regularization term:

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell\left(\mathbf{y}_i, h_{\mathbf{w}}(\mathbf{x}_i)\right) + \lambda \sum_{(i,j)\in A} w_{ij}^2.$$

To keep things straightforward, you should use squared error as the loss function. In a multi-class classification context with K distinct classes, this loss function is

$$\ell\left(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x})\right) = \sum_{k=1}^{K} \left(y_k - a_k\right)^2,$$

where a_k is the output value of the output neuron k that corresponds to class k given training example $\mathbf{x}$. With the above cost function and loss function, the partial derivatives of $J(\mathbf{w})$ with respect to any weight w_{ij} can be estimated using a mini-batch of training data B as

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) \approx \frac{1}{|B|}\left(\sum_{e\in B} \Delta_j^{(e)} a_i^{(e)}\right) + 2\lambda w_{ij}$$

where $\Delta_j^{(e)}$ and $a_i^{(e)}$ are the values for Δ_j and a_i computed by passing the values $(\mathbf{x}_e, \mathbf{y}_e)$ corresponding to training example index $e \in B$ into the backpropagation algorithm (see Lectures 07-4 and 07-5 for details).

Fitting the Training Data

Pseudocode for the mini-batch gradient descent algorithm, tailored for training a neural network via backpropagation, is shown in Algorithm 2. The stopping conditions that you should use are:

- The number of epochs reaches the specified epoch limit (which defaults to 1000).
- The absolute error at each output neuron k is at most 0.01 across an entire epoch. For a training example $(\mathbf{x}, \mathbf{y})$, the error at output neuron k is given by $y_k - a_k$ and the absolute error is $|y_k - a_k|$. Put another way, the training process should keep going if **any** training example produces an absolute error above 0.01 at **any** output neuron; i.e., we're making a big enough mistake on some training example.

You can adjust the epoch limit with a command-line argument. You may also wish to adjust hyperparameters like the mini-batch size m, the learning rate α, the regularization penalty λ, and/or the weight initialization value ϵ to alter how quickly or slowly the network trains.

During training, you will periodically need to compute the current cost as well as average loss and accuracy on the training set in order to display the progress (see next page for details on accuracy). This should be done only when needed, however, as otherwise it will slow down the training process. In general, your implementation should be reasonably efficient and avoid unnecessary work, though it may be helpful to deal with efficiency considerations only after you have a working implementation. Note also that weight updates in neural networks generally require more work than in multiple linear regression, so your code will probably run slower than it did for Assignment 01.

Algorithm 2 Mini-Batch Gradient Descent for Training a Neural Network

```
procedure NeuralNetworkTrain( $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$; $\mathcal{N} = (V, A)$; $m$, $\alpha$, $\lambda$, $\epsilon$ )
    for each edge $(i, j) \in A$ do
        $w_{ij}^{(0)} \leftarrow$ Rand$(-\epsilon, \epsilon)$                ▷ Random initialization of weights
    $t \leftarrow 0$                ▷ $t$ is the iteration counter for number of weight updates
    $e \leftarrow 0$        ▷ $e$ counts number of epochs; an epoch is one full pass through the training data
    while stopping conditions not met do
        Divide $\{1, 2, \ldots, n\}$ into mini-batches of $m$ data points (see note below for details)
        for each mini-batch $B$ do
            for each example index $e \in B$ do
                Run Backpropagate$((\mathbf{x}_e, \mathbf{y}_e)$, $\mathcal{N}$, $\mathbf{w}^{(t)})$ to obtain $\Delta_j^{(e)}$, $a_i^{(e)}$ values
            for each edge $(i, j) \in A$ do
                $w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \left( \frac{1}{|B|} \sum_{e \in B} \Delta_j^{(e)} a_i^{(e)} \right) - 2\alpha\lambda w_{ij}^{(t)}$
            $t \leftarrow t + 1$
        $e \leftarrow e + 1$                ▷ an epoch ends once we process all mini-batches
    return $\mathbf{w}^{(t)}$
```

For mini-batch construction, if m does not evenly divide n, then the last mini-batch should have size $n \bmod m$. Additionally, if the `-r` command-line flag is given, these mini-batches should be randomized; otherwise, the first m data points should be assigned to batch 1, the next m data points to batch 2, and so on (this is to make debugging easier).

Evaluation using the Validation Data

After the network is trained, your program should compute its accuracy on the validation set. To do this, you will need to:

1. Run each example in the validation set through the network using forward propagation to compute the output values at all output neurons.
2. Predict the class for the example as the class corresponding to the output neuron with the largest output value.
3. Check correctness by comparing the predicted class to the actual class for this example (where the actual class is the class with the sole 1 in the example's output vector).
4. Compute accuracy as the proportion of correct predictions on the validation set.

Output

The level of output produced by the program is controlled by the `-v` flag (for *verbosity*). The levels range from 1 to 4, with 4 containing the most output; the output produced is additive, meaning that the output for level i should also be produced for any level j with $j \geq i$. The desired output for each level is discussed in more detail below.

The output produced by your program does not have to match what is shown **exactly**, but it should be well-formatted and reasonably easy to read. Additional requirements for displaying numeric quantities include:

- Final training and validation accuracy should be displayed with 6 decimal places
- Time per iteration should be displayed with 4 decimal places
- In gradient descent, costs and losses should be displayed with 6 decimal places and accuracy should be displayed with 4 places

Additionally, **don't use tabs** (`\t`) in your output! (See here for a discussion of why.)

Output Level 1

Level 1 prints a brief summary of the program flow followed by the results.

```
shell$ java Driver -f data/iris-shuf.dat
* Reading data/iris-shuf.dat
* Doing train/validation split
* Scaling features
* Building network
* Training network (using 120 examples)
* Evaluating accuracy
  TrainAcc: 0.941667
  ValidAcc: 0.933333
```

Output Level 2

Level 2 includes a few more details during each part of the program flow, such as the minimum and maximum values for each feature within the training set, the network layer sizes, and the gradient descent parameters, training time, and stopping condition:

```
shell$ java Driver -f data/iris-shuf.dat -v 2
* Reading data/iris-shuf.dat
* Doing train/validation split
* Scaling features
  * min/max values on training set:
    Feature 1: 4.400, 7.900
    Feature 2: 2.000, 4.200
    Feature 3: 1.000, 6.900
    Feature 4: 0.100, 2.500
* Building network
  * Layer sizes (excluding bias neuron(s)):
    Layer  1 (input) :   4
    Layer  2 (output):   3
* Training network (using 120 examples)
  * Beginning mini-batch gradient descent
    (batchSize=1, epochLimit=1000, learningRate=0.0100, lambda=0.0000)
  * Done with fitting!
    Training took 180ms, 1000 epochs, 120000 iterations (0.0015ms / iteration)
    GD Stop condition: Epoch Limit
* Evaluating accuracy
  TrainAcc: 0.941667
  ValidAcc: 0.933333
```

Output Level 3

Level 3 prints additional details about the model's performance on the training data (cost, loss, and accuracy) before, during, and after the training process:

```
shell$ java Driver -f data/iris-shuf.dat -v 3 -h 1 2
* Reading data/iris-shuf.dat
* Doing train/validation split
* Scaling features
  * min/max values on training set:
    Feature 1: 4.400, 7.900
    Feature 2: 2.000, 4.200
    Feature 3: 1.000, 6.900
    Feature 4: 0.100, 2.500
* Building network
  * Layer sizes (excluding bias neuron(s)):
    Layer  1 (input) :   4
    Layer  2 (hidden):   2
    Layer  3 (output):   3
* Training network (using 120 examples)
  * Beginning mini-batch gradient descent
    (batchSize=1, epochLimit=1000, learningRate=0.0100, lambda=0.0000)
    Initial model with random weights : Cost = 0.757058; Loss = 0.757058; Acc = 0.3083
    After    100 epochs ( 12000 iter.): Cost = 0.365432; Loss = 0.365432; Acc = 0.7000
    After    200 epochs ( 24000 iter.): Cost = 0.289164; Loss = 0.289164; Acc = 0.8750
    After    300 epochs ( 36000 iter.): Cost = 0.198624; Loss = 0.198624; Acc = 0.9583
    After    400 epochs ( 48000 iter.): Cost = 0.125109; Loss = 0.125109; Acc = 0.9667
    After    500 epochs ( 60000 iter.): Cost = 0.087541; Loss = 0.087541; Acc = 0.9667
    After    600 epochs ( 72000 iter.): Cost = 0.067797; Loss = 0.067797; Acc = 0.9750
    After    700 epochs ( 84000 iter.): Cost = 0.056384; Loss = 0.056384; Acc = 0.9750
    After    800 epochs ( 96000 iter.): Cost = 0.049125; Loss = 0.049125; Acc = 0.9750
    After    900 epochs (108000 iter.): Cost = 0.044119; Loss = 0.044119; Acc = 0.9750
    After   1000 epochs (120000 iter.): Cost = 0.040448; Loss = 0.040448; Acc = 0.9833
  * Done with fitting!
    Training took 262ms, 1000 epochs, 120000 iterations (0.0022ms / iteration)
    GD Stop condition: Epoch Limit
* Evaluating accuracy
```

```
  TrainAcc: 0.983333
  ValidAcc: 0.966667
```

Note: **Unlike Assignment 01**, where model performance was printed after every 1000 epochs, here the printing should occur after every `epochLimit`/10 epochs. So with a default `epochLimit` of 1000, this means printing should occur after every 100 epochs; however, with an `epochLimit` of 10000, printing should occur after every 1000 epochs, while an `epochLimit` of 100 should result in printing after every 10 epochs. For `epochLimit` values less than 10, printing should be done after every epoch.

The below example uses an epoch limit of 5000, two hidden layers with sizes 3 and 2, and a regularization parameter of $\lambda = 0.0001$ (which is why the cost and the loss are not always the same):

```
shell$ java Driver -f data/iris-shuf.dat -v 3 -e 5000 -h 2 3 2 -l 0.0001
* Reading data/iris-shuf.dat
* Doing train/validation split
* Scaling features
  * min/max values on training set:
    Feature 1: 4.400, 7.900
    Feature 2: 2.000, 4.200
    Feature 3: 1.000, 6.900
    Feature 4: 0.100, 2.500
* Building network
  * Layer sizes (excluding bias neuron(s)):
    Layer  1 (input) :   4
    Layer  2 (hidden):   3
    Layer  3 (hidden):   2
    Layer  4 (output):   3
* Training network (using 120 examples)
  * Beginning mini-batch gradient descent
    (batchSize=1, epochLimit=5000, learningRate=0.0100, lambda=0.0001)
    Initial model with random weights : Cost = 0.735975; Loss = 0.735964; Acc = 0.3250
    After    500 epochs ( 60000 iter.): Cost = 0.664997; Loss = 0.664892; Acc = 0.3667
    After   1000 epochs (120000 iter.): Cost = 0.664995; Loss = 0.664888; Acc = 0.3667
    After   1500 epochs (180000 iter.): Cost = 0.664991; Loss = 0.664882; Acc = 0.3667
    After   2000 epochs (240000 iter.): Cost = 0.664846; Loss = 0.664722; Acc = 0.3667
    After   2500 epochs (300000 iter.): Cost = 0.304122; Loss = 0.293052; Acc = 0.8750
    After   3000 epochs (360000 iter.): Cost = 0.138447; Loss = 0.109819; Acc = 0.9667
    After   3500 epochs (420000 iter.): Cost = 0.086505; Loss = 0.044383; Acc = 0.9833
    After   4000 epochs (480000 iter.): Cost = 0.081379; Loss = 0.035461; Acc = 0.9917
    After   4500 epochs (540000 iter.): Cost = 0.079597; Loss = 0.032547; Acc = 0.9917
    After   5000 epochs (600000 iter.): Cost = 0.078498; Loss = 0.031244; Acc = 0.9917
  * Done with fitting!
    Training took 1472ms, 5000 epochs, 600000 iterations (0.0025ms / iteration)
    GD Stop condition: Epoch Limit
* Evaluating accuracy
  TrainAcc: 0.991667
  ValidAcc: 0.966667
```

Output Level 4

Level 4 provides additional details about the forward and backward propagation passes used during training. Specifically, whenever a training example is run through the BACKPROPUPDATE procedure, the in_j and a_j values computed during forward propagation and the Δ_j values computed during backward propagation should be printed for the neurons in each layer.

This level of output is **extremely verbose**, and so is intended primarily for debugging purposes when training on small data sets with a very small epoch limit. The example below illustrates this on a small data set with a learning rate of $\alpha = 0.5$ and a weight initialization value of $\epsilon = 0$ (which means that all the weights start at 0; this is good for debugging but bad for training!):

```
shell$ java Driver -f data/iris-small.dat -v 4 -h 1 2 -e 1 -w 0 -a 0.5
* Reading data/iris-small.dat
* Doing train/validation split
* Scaling features
  * min/max values on training set:
    Feature 1: 4.800, 7.600
    Feature 2: 2.300, 3.400
    Feature 3: 1.600, 6.600
    Feature 4: 0.200, 2.500
* Building network
  * Layer sizes (excluding bias neuron(s)):
    Layer  1 (input) :   4
    Layer  2 (hidden):   2
    Layer  3 (output):   3
* Training network (using 8 examples)
  * Beginning mini-batch gradient descent
    (batchSize=1, epochLimit=1, learningRate=0.5000, lambda=0.0000)
    Initial model with random weights : Cost = 0.750000; Loss = 0.750000; Acc = 0.2500
    * Forward Propagation on example 1
      Layer 1 (input) :     a_j:  1.000  0.214  0.273  0.440  0.565
      Layer 2 (hidden):    in_j:  0.000  0.000
                            a_j:  0.500  0.500
      Layer 3 (output):    in_j:  0.000  0.000  0.000
                            a_j:  0.500  0.500  0.500
             example's actual y:  0.000  0.000  1.000
    * Backward Propagation on example 1
      Layer 3 (output): Delta_j:  0.250  0.250 -0.250
      Layer 2 (hidden): Delta_j:  0.000  0.000

    * Forward Propagation on example 2
      (OUTPUT OMITTED FOR BREVITY)

    * Forward Propagation on example 8
      Layer 1 (input) :     a_j:  1.000  0.071 -1.000  0.120 -0.043
      Layer 2 (hidden):    in_j: -0.005 -0.005
                            a_j:  0.499  0.499
      Layer 3 (output):    in_j: -0.507 -0.338 -0.171
                            a_j:  0.376  0.416  0.457
             example's actual y:  0.000  1.000  0.000
    * Backward Propagation on example 8
      Layer 3 (output): Delta_j:  0.176 -0.284  0.227
      Layer 2 (hidden): Delta_j: -0.003 -0.003
    After      1 epochs (     8 iter.): Cost = 0.675760; Loss = 0.675760; Acc = 0.3750
  * Done with fitting!
    Training took 36ms, 1 epochs, 8 iterations (4.5000ms / iteration)
    GD Stop condition: Epoch Limit
* Evaluating accuracy
  TrainAcc: 0.375000
  ValidAcc: 0.500000
```

Experimentation

Once you have finished your implementation (and debugged it!), you should experiment with different network architectures and hyperparameter values to see what yields good performance on the `image-NN.dat` and `mnist.dat` sample data sets. (*Note:* These are **large** files with a high number of features, so expect the training process to be slow!) Try to find options that lead to high accuracy on the validation data. Make notes of any observations you might have regarding the influence of various hyperparameters on performance. Provide, in a text file, a list of the best training and validation set accuracy values that you were able to achieve on these data sets, along with the command-line flags that you used to generate those results.

Extra Credit Options

The following pages detail two extra credit options that are available for this assignment. You may choose to implement any subset of these options.

Extra Credit Option 1: Additional Activation Functions

For a small amount of extra credit (about 2–3% of the overall assignment), you can implement support for additional activation functions via another command-line argument:

- `-g <NAME>`: Specifies the name of the activation function to be used at neurons in the **hidden layers**, where `<NAME>` is a `String` that matches one of the following (case-insensitive):
  - `logistic`: The **standard logistic** function, with

    $$g(t) = \frac{1}{1+e^{-t}} \qquad g'(t) = g(t)(1-g(t))$$

  - `relu`: The **rectified linear unit** function, with

    $$g(t) = \max\{0, t\} \qquad g'(t) = \begin{cases} 1 & \text{if } t > 0 \\ 0 & \text{if } t < 0 \end{cases}$$

    (Technically, g' is undefined at 0 for `ReLU`, but we can use $g'(0) = 0$ in code.)
  - `softplus`: The **softplus** function, with

    $$g(t) = \ln\left(1+e^{t}\right) \qquad g'(t) = \sigma(t) = \frac{1}{1+e^{-t}}$$

  - `tanh`: The **hyperbolic tangent** function, with

    $$g(t) = \frac{e^{t}-e^{-t}}{e^{t}+e^{-t}} \qquad g'(t) = 1 - g(t)^2$$

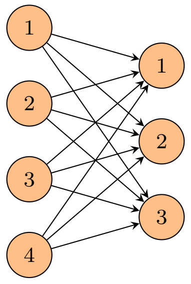
Note that **neurons in the output layer should always use** the logistic activation function in order to ensure that outputs are in the range $[0, 1]$.

Extra Credit Option 2: Deterministic Weight Initialization

For a small amount of extra credit (about 3–5% of the overall assignment), you can implement support for **deterministic** weight initialization, which should be activated by passing a negative value to the `-w` command-line argument. In deterministic weight initialization, the edge weights should be initialized as follows:

- The weights on the edges leaving the bias neuron(s) should be initialized to 0.1.
- For the edges between non-bias neurons in layers l and $l+1$:
  - Temporarily relabel the neurons in layer l from 1 to m
  - Temporarily relabel the neurons in layer $l+1$ from 1 to n
  - Initialize the weight on the edge from neuron i in layer l to neuron j in layer $l+1$ as $1/\left(i \cdot 2^{j-1}\right)$.
    (I.e., the first edge out of neuron i has weight $1/i$, the second edge out of neuron i has weight $(1/i)/2$, the third edge out of neuron i has weight $(1/i)/2/2$, etc.)
  - Restore neuron labels as needed

  (Note: This initialization should be used even if layer l is the input layer; i.e., $l = 1$.)



With a deterministic weight initialization, it should be possible for your program's behavior to exactly match that of a reference implementation (e.g., mine). This is useful for double-checking the overall training process as well as forward and backward propagation calculations with verbosity level 4 on small examples.

Some caveats:

- I make no claims or guarantees regarding this particular weight initialization scheme; in my limited testing, it seems to prevent any two edges from always having the same weight, thus allowing the training process to take full advantage of all the weight parameters available for tuning. However, I don't (yet) have a formal proof that this will always work as intended. For production purposes, using random initial weights is recommended.
- I am fairly confident that the output below from my reference implementation is correct, but if you spot any differences between my output and yours, let me know and I can take a closer look (I've been known to make the occasional mistake!).

```
shell$ java Driver -f data/iris-shuf.dat -h 2 3 2 -w -1 -v 3 -a 0.5 -e 10
* Reading data/iris-shuf.dat
* Doing train/validation split
* Scaling features
  * min/max values on training set:
    Feature 1: 4.400, 7.900
    Feature 2: 2.000, 4.200
    Feature 3: 1.000, 6.900
    Feature 4: 0.100, 2.500
* Building network
  * Layer sizes (excluding bias neuron(s)):
    Layer  1 (input) :   4
    Layer  2 (hidden):   3
    Layer  3 (hidden):   2
    Layer  4 (output):   3
```

```
* Training network (using 120 examples)
  * Beginning mini-batch gradient descent
    (batchSize=1, epochLimit=10, learningRate=0.5000, lambda=0.0000)
    Initial model with determ. weights: Cost = 1.028471; Loss = 1.028471; Acc = 0.3083
    After      1 epochs (   120 iter.): Cost = 0.697988; Loss = 0.697988; Acc = 0.3667
    After      2 epochs (   240 iter.): Cost = 0.690464; Loss = 0.690464; Acc = 0.3667
    After      3 epochs (   360 iter.): Cost = 0.670832; Loss = 0.670832; Acc = 0.3667
    After      4 epochs (   480 iter.): Cost = 0.577479; Loss = 0.577479; Acc = 0.6750
    After      5 epochs (   600 iter.): Cost = 0.464022; Loss = 0.464022; Acc = 0.6750
    After      6 epochs (   720 iter.): Cost = 0.414322; Loss = 0.414322; Acc = 0.6750
    After      7 epochs (   840 iter.): Cost = 0.383633; Loss = 0.383633; Acc = 0.6750
    After      8 epochs (   960 iter.): Cost = 0.356763; Loss = 0.356763; Acc = 0.6750
    After      9 epochs (  1080 iter.): Cost = 0.334091; Loss = 0.334091; Acc = 0.6750
    After     10 epochs (  1200 iter.): Cost = 0.318682; Loss = 0.318682; Acc = 0.6750
  * Done with fitting!
    Training took 13ms, 10 epochs, 1200 iterations (0.0108ms / iteration)
    GD Stop condition: Epoch Limit
* Evaluating accuracy
  TrainAcc: 0.675000
  ValidAcc: 0.633333
```

```
shell$ java Driver -f data/iris-small.dat -v 4 -h 1 2 -e 1 -w -1 -a 0.5
* Reading data/iris-small.dat
* Doing train/validation split
* Scaling features
  * min/max values on training set:
    Feature 1: 4.800, 7.600
    Feature 2: 2.300, 3.400
    Feature 3: 1.600, 6.600
    Feature 4: 0.200, 2.500
* Building network
  * Layer sizes (excluding bias neuron(s)):
    Layer  1 (input) :   4
    Layer  2 (hidden):   2
    Layer  3 (output):   3
* Training network (using 8 examples)
  * Beginning mini-batch gradient descent
    (batchSize=1, epochLimit=1, learningRate=0.5000, lambda=0.0000)
    Initial model with determ. weights: Cost = 1.006318; Loss = 1.006318; Acc = 0.2500
     * Forward Propagation on example 1
       Layer 1 (input) :     a_j:  1.000  0.214  0.273  0.440  0.565
       Layer 2 (hidden):    in_j:  0.739  0.419
                             a_j:  0.677  0.603
       Layer 3 (output):    in_j:  1.078  0.589  0.345
                             a_j:  0.746  0.643  0.585
              example's actual y:  0.000  0.000  1.000
     * Backward Propagation on example 1
       Layer 3 (output): Delta_j:  0.283  0.295 -0.201
       Layer 2 (hidden): Delta_j:  0.083  0.045

     * Forward Propagation on example 2
       Layer 1 (input) :     a_j:  1.000 -1.000  1.000 -1.000 -1.000
       Layer 2 (hidden):    in_j: -0.986 -0.443
                             a_j:  0.272  0.391
       Layer 3 (output):    in_j:  0.367  0.124  0.360
                             a_j:  0.591  0.531  0.589
              example's actual y:  1.000  0.000  0.000
     * Backward Propagation on example 2
       Layer 3 (output): Delta_j: -0.198  0.264  0.285
       Layer 2 (hidden): Delta_j:  0.003  0.003

     * Forward Propagation on example 3
       Layer 1 (input) :     a_j:  1.000 -0.571  1.000 -0.960 -1.000
       Layer 2 (hidden):    in_j: -0.556 -0.232
                             a_j:  0.364  0.442
       Layer 3 (output):    in_j:  0.598  0.001  0.217
                             a_j:  0.645  0.500  0.554
              example's actual y:  1.000  0.000  0.000
     * Backward Propagation on example 3
       Layer 3 (output): Delta_j: -0.162  0.250  0.274
       Layer 2 (hidden): Delta_j:  0.004 -0.003

     * Forward Propagation on example 4
       Layer 1 (input) :     a_j:  1.000  0.357  0.818  0.640  1.000
       Layer 2 (hidden):    in_j:  1.241  0.666
                             a_j:  0.776  0.661
       Layer 3 (output):    in_j:  1.208 -0.022  0.145
                             a_j:  0.770  0.494  0.536
              example's actual y:  0.000  0.000  1.000
     * Backward Propagation on example 4
       Layer 3 (output): Delta_j:  0.273  0.247 -0.231
       Layer 2 (hidden): Delta_j:  0.050  0.029

     * Forward Propagation on example 5
       Layer 1 (input) :     a_j:  1.000 -0.357  0.273  0.040 -0.130
       Layer 2 (hidden):    in_j: -0.210 -0.057
                             a_j:  0.448  0.486
       Layer 3 (output):    in_j:  0.579 -0.342  0.250
                             a_j:  0.641  0.415  0.562
              example's actual y:  0.000  1.000  0.000
     * Backward Propagation on example 5
       Layer 3 (output): Delta_j:  0.295 -0.284  0.277
       Layer 2 (hidden): Delta_j:  0.069  0.041

     * Forward Propagation on example 6
       Layer 1 (input) :     a_j:  1.000  1.000  0.273  1.000  0.652
       Layer 2 (hidden):    in_j:  1.555  0.814
                             a_j:  0.826  0.693
       Layer 3 (output):    in_j:  0.733 -0.022  0.165
                             a_j:  0.676  0.495  0.541
              example's actual y:  0.000  0.000  1.000
     * Backward Propagation on example 6
       Layer 3 (output): Delta_j:  0.296  0.247 -0.228
       Layer 2 (hidden): Delta_j:  0.035  0.019

     * Forward Propagation on example 7
       Layer 1 (input) :     a_j:  1.000  0.357  0.455  0.120  0.043
       Layer 2 (hidden):    in_j:  0.575  0.329
```

```
                              a_j:  0.640  0.581
       Layer 3 (output):    in_j:  0.264 -0.318  0.328
                              a_j:  0.566  0.421  0.581
              example's actual y:  0.000  1.000  0.000
     * Backward Propagation on example 7
       Layer 3 (output): Delta_j:  0.278 -0.282  0.283
       Layer 2 (hidden): Delta_j:  0.054  0.029

     * Forward Propagation on example 8
       Layer 1 (input) :     a_j:  1.000  0.071 -1.000  0.120 -0.043
       Layer 2 (hidden):    in_j: -0.393 -0.152
                              a_j:  0.403  0.462
       Layer 3 (output):    in_j: -0.133 -0.141  0.010
                              a_j:  0.467  0.465  0.503
              example's actual y:  0.000  1.000  0.000
     * Backward Propagation on example 8
       Layer 3 (output): Delta_j:  0.232 -0.266  0.251
       Layer 2 (hidden): Delta_j:  0.030  0.011

    After      1 epochs (     8 iter.): Cost = 0.736744; Loss = 0.736744; Acc = 0.3750
  * Done with fitting!
    Training took 24ms, 1 epochs, 8 iterations (3.0000ms / iteration)
    GD Stop condition: Epoch Limit
* Evaluating accuracy
  TrainAcc: 0.375000
  ValidAcc: 0.500000
```