IMPLEMENTATION OF A PARALLEL BATCH TRAINING ALGORITHM FOR DEEP NEURAL NETWORK

YUPING LIN

IFLYTEK LABORATORY FOR NEURAL COMPUTING FOR MACHINE LEARNING

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

YORK UNIVERSITY, TORONTO

NOVEMBER 10, 2015

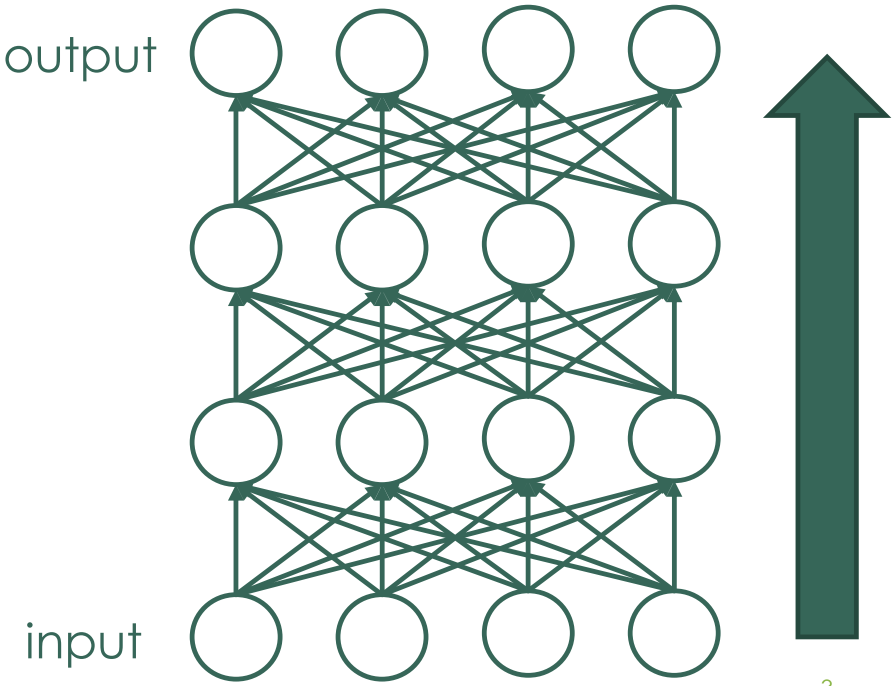
OUTLINE

- Review
- Neural Network Representation
- Sequential Trainer
- Concurrent Trainer
  - Dividing tasks
  - Collecting statistics
  - Using monitor
  - Using thread pool
- Testing
- Future Work

REVIEW -- NEURAL NETWORK TRAINING

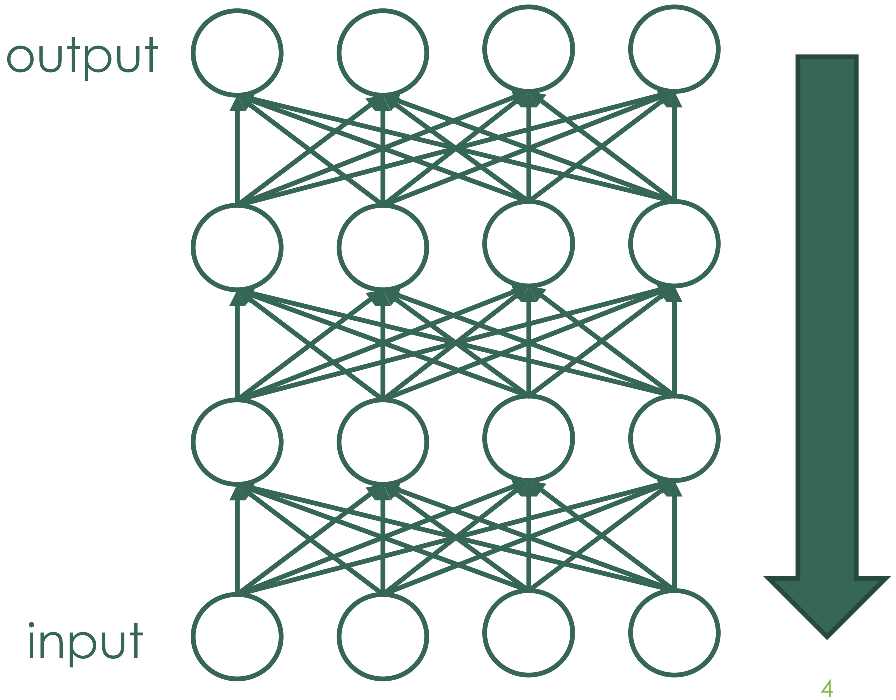
- Forward phase
  - $Z_j^{(l+1)} = F\left(\sum_i w_{ij} \cdot Z_i^{(l)} + b_j\right)$

  Where $F(x)$ is the nonlinear activation function

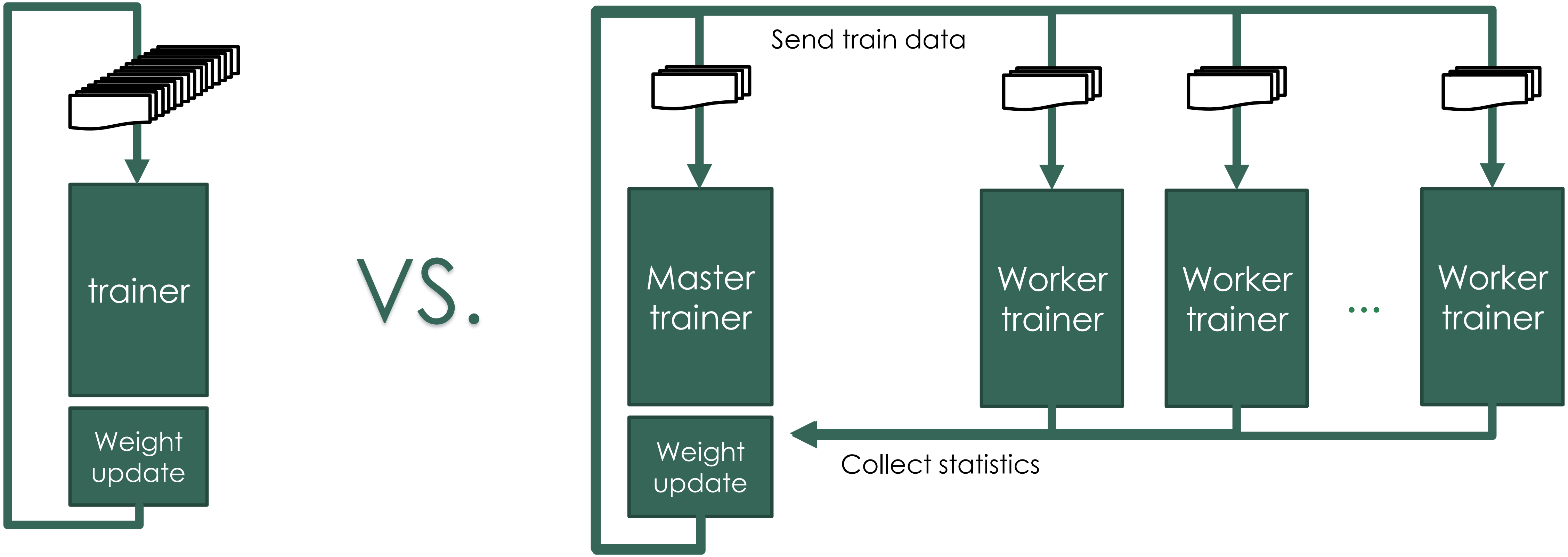


REVIEW -- NEURAL NETWORK TRAINING

- Error back propagation
  - $\delta_k^{(out)} = Z_k^{(out)} - T_k^{(out)}$
  - $\delta_i^{(l)} = F'\left(Z_i^{(l)}\right) \cdot \sum_j w_{ij} \cdot \delta_j^{(l+1)}$
  - Where $F'(x)$ is the derivative of the activation function
  - T is the desired output vector
- Weight updating
  - $\Delta w_{ij} = Z_i^{(l)} \cdot \delta_j^{(l+1)}$
  - $w_{ij} = w_{ij} - \gamma \cdot \Delta w_{ij}$
  - Where γ is the learning rate



REVIEW -- SEQUENTIAL TRAINING VS. CONCURRENT TRAINING



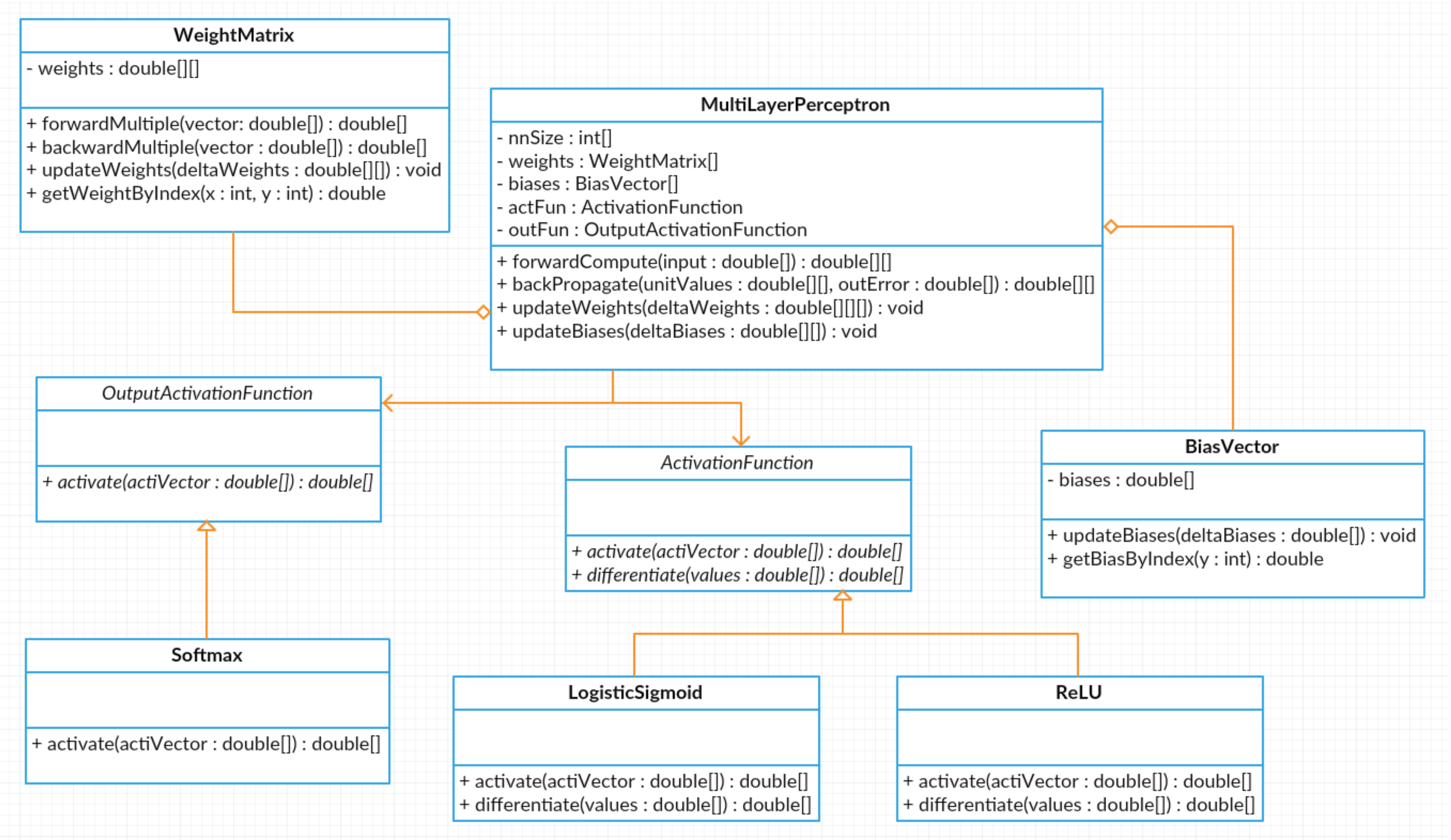
Sequential training

Concurrent training

BROAD VIEW OF THE IMPLEMENTATION

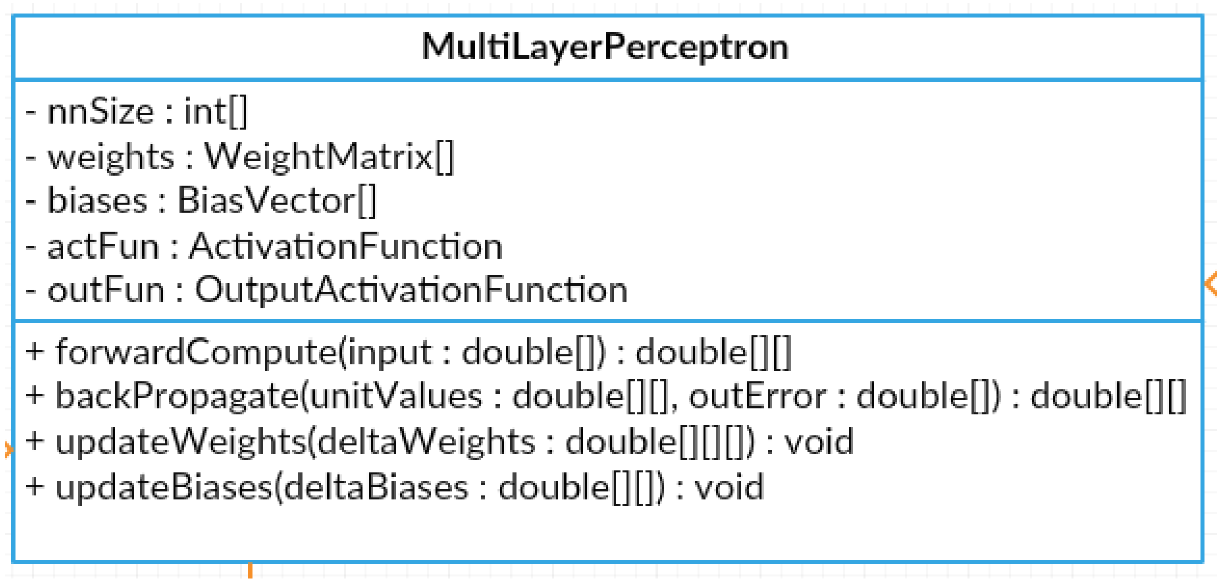
- There are 3 major components in our implementation:
  - **The neural network representation**: package of classes that form a neural network.
  - **Sequential trainers**: classes that implement the sequential training algorithm.
  - **Concurrent trainers**: classes that implement the concurrent training algorithm.

NEURAL NETWORK REPRESENTATION



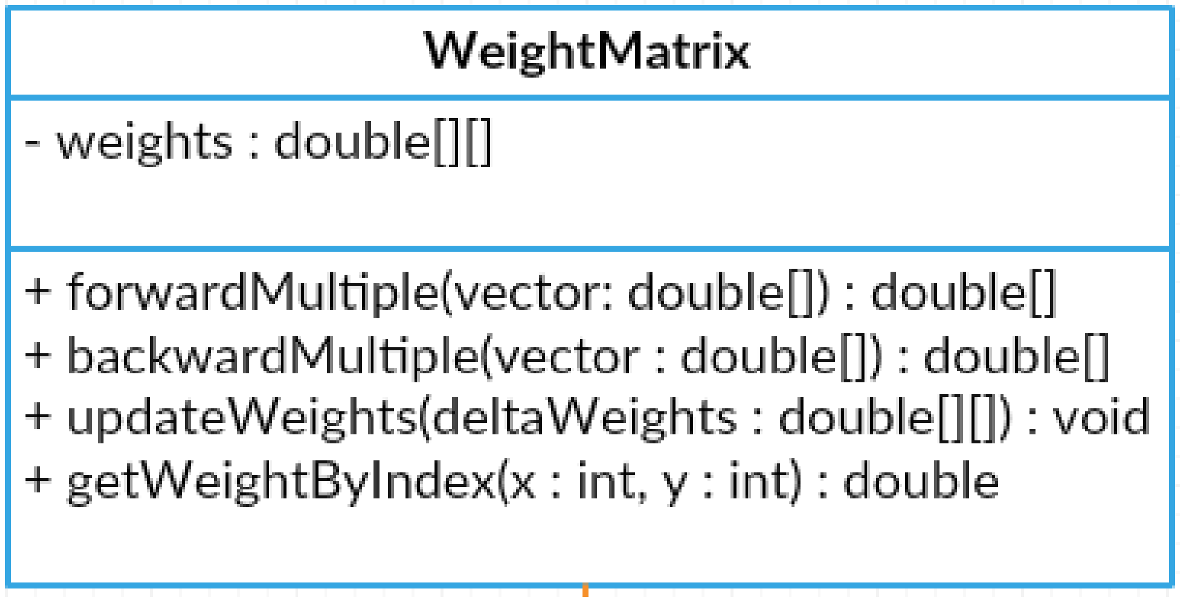
NEURAL NETWORK REPRESENTATION

- Main class that represents a multi-layer perceptron.
- Has attributes representing the components of a neural network.
  - Weights
  - Biases
  - Activation Functions

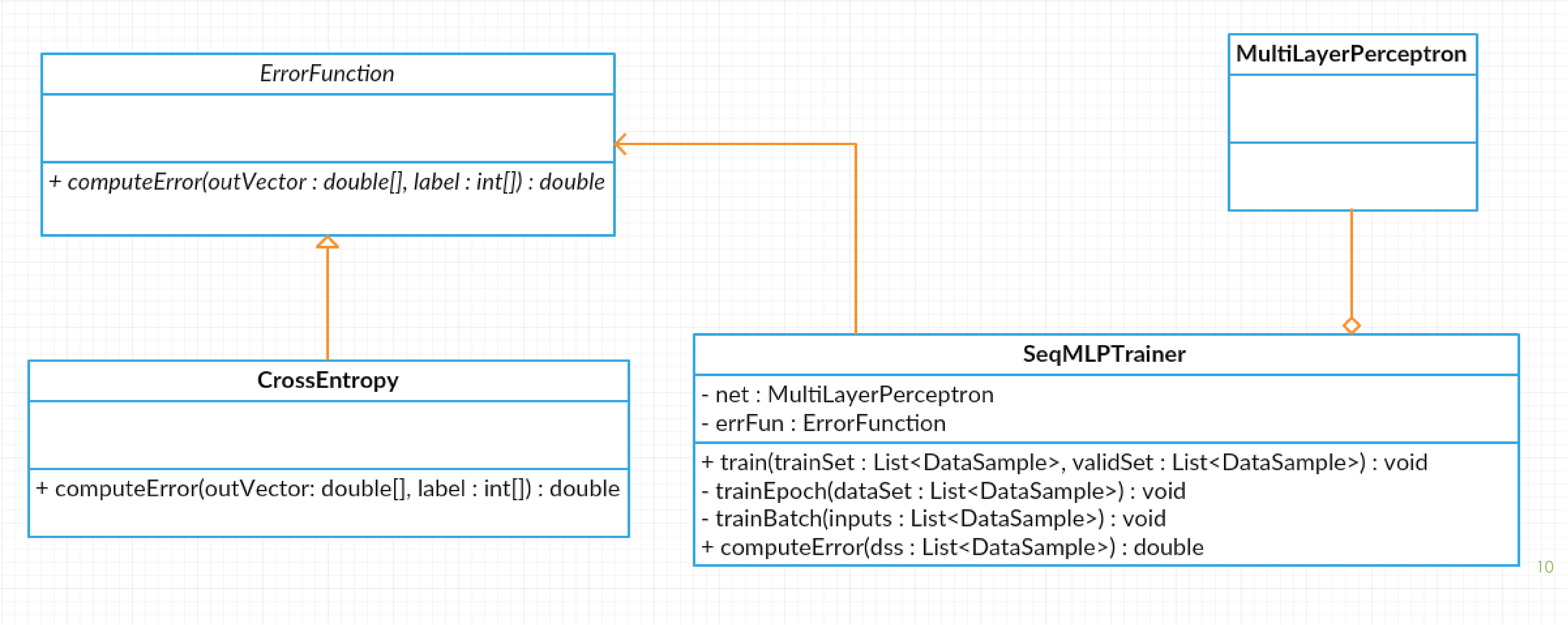


NEURAL NETWORK REPRESENTATION

- Represents a weight matrix
- Support forward/backward multiplications



THE SEQUENTIAL TRAINER

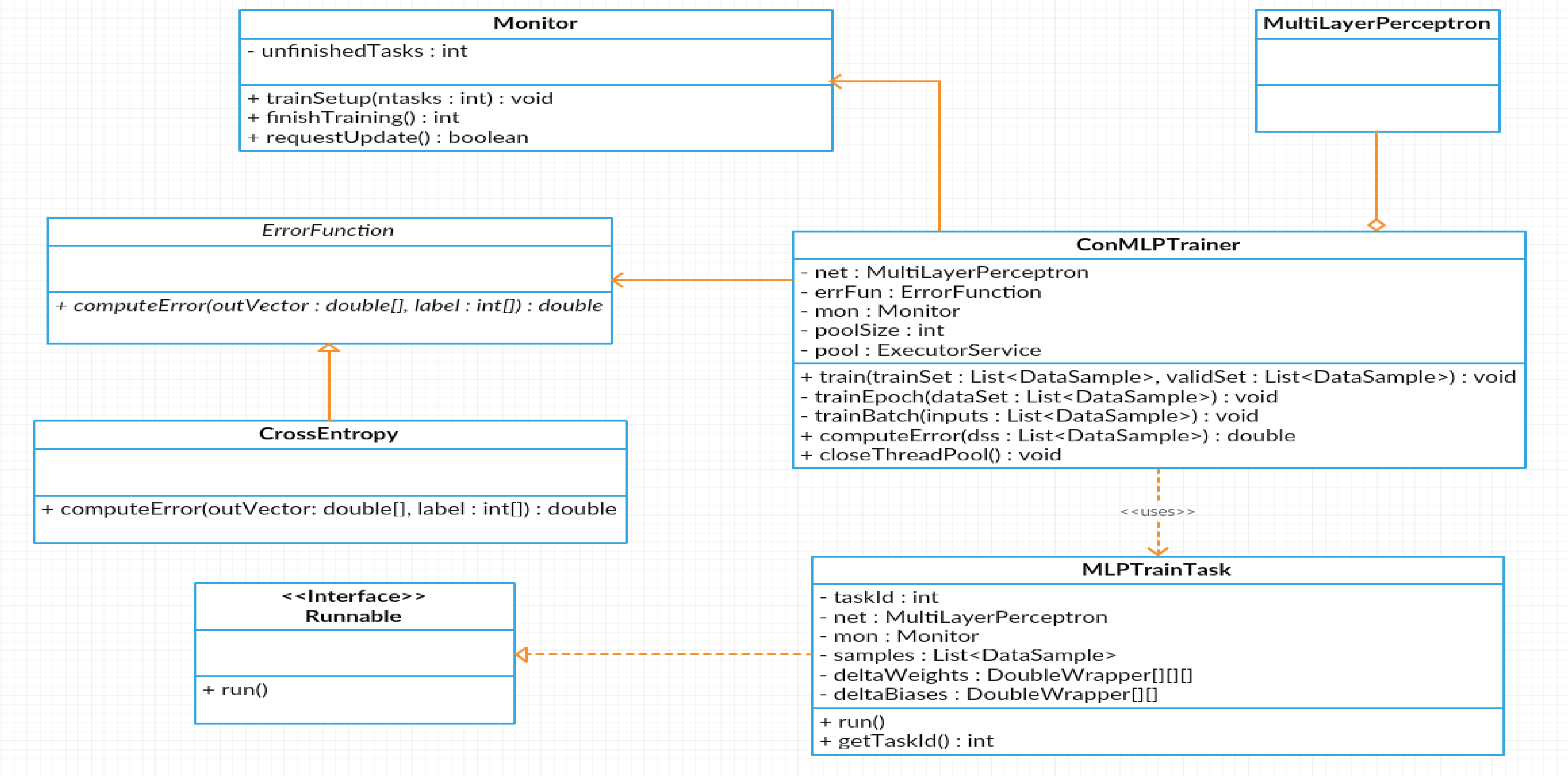


THE SEQUENTIAL TRAINER

- Divide training into 3 layers:
  - **train()** for the whole training process
  - **trainEpoch()** for the training of each epoch
  - **trainBatch()** for the training of each mini-batch

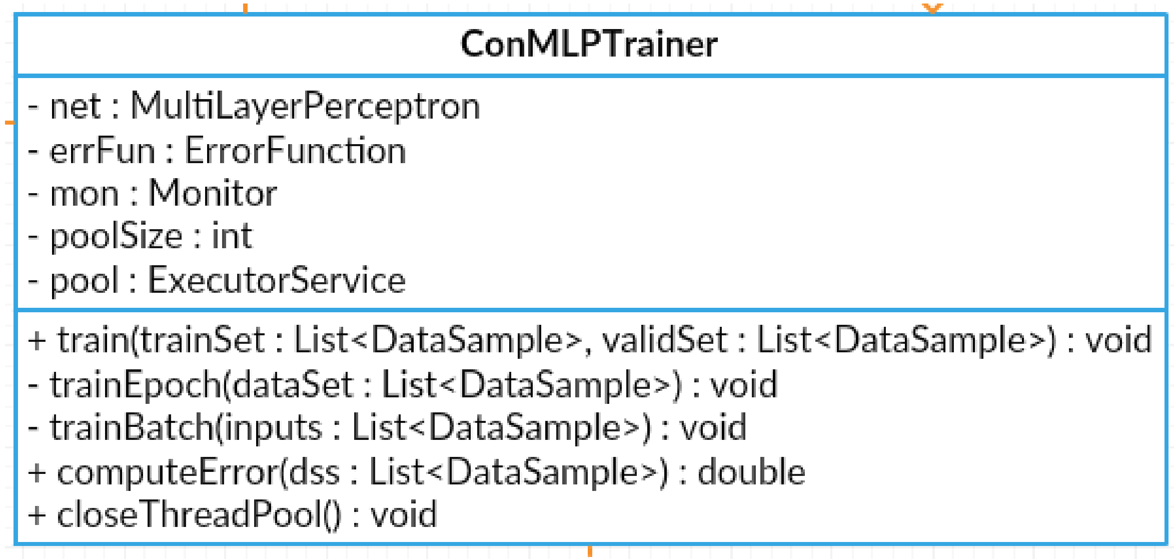
SeqMLPTrainer
- net : MultiLayerPerceptron - errFun : ErrorFunction
+ train(trainSet : List<DataSample>, validSet : List<DataSample>) : void - trainEpoch(dataSet : List<DataSample>) : void - trainBatch(inputs : List<DataSample>) : void + computeError(dss : List<DataSample>) : double

THE CONCURRENT TRAINER



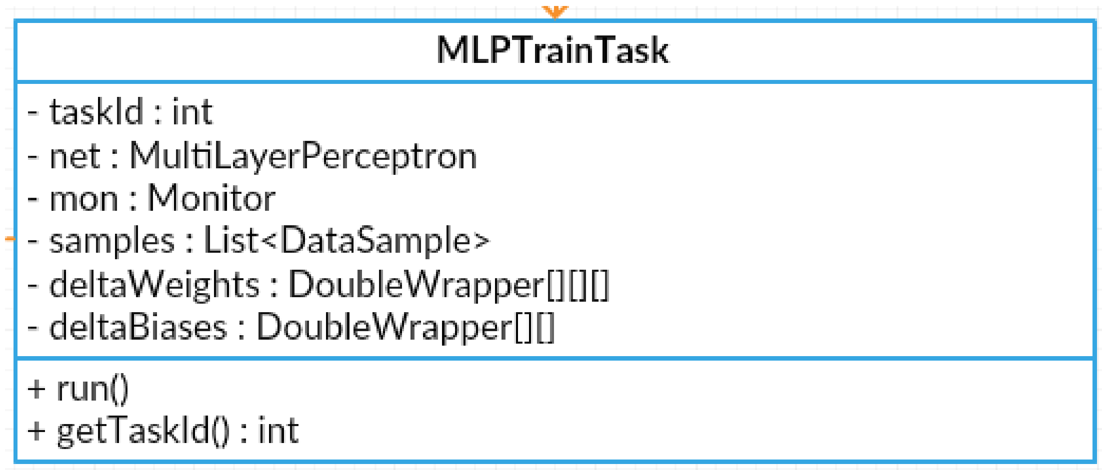
THE CONCURRENT TRAINER

- Similar to the sequential trainer.
- Keep references to the monitor object and a thread pool.
- Concurrency occur within the **trainBatch()** method.



THE CONCURRENT TRAINER -- DIVIDING TASKS

- Implements the ***java.lang.Runnable*** interface
- Represents a training task for worker thread.
- Keep references to the monitor object and the global shared variables for update statistics.



THE CONCURRENT TRAINER -- COLLECTING STATISTICS

- The update statistics are accumulated locally within each task.
- Then update in the shared global variables concurrently upon finish of the task.
- Need synchronization: synchronized blocks, compare and set etc.

```java
        // update delta weights and biases in shared variables
        for (int ly = 0; ly < nnSize.length - 1; ly++) {
            for (int x = 0; x < nnSize[ly]; x++) {
                for (int y = 0; y < nnSize[ly+1]; y++) {
                    synchronized (this.deltaWeights[ly][x][y]) {
                        this.deltaWeights[ly][x][y].setValue(this.deltaWeights[ly][x][y].getValue() + delW[ly][x][y]);
                    }
                }
            }

            for (int y = 0; y < nnSize[ly+1]; y++) {
                synchronized (this.deltaBiases[ly][y]) {
                    this.deltaBiases[ly][y].setValue(this.deltaBiases[ly][y].getValue() + delB[ly][y]);
                }
            }
        }
```

THE CONCURRENT TRAINER -- USING MONITOR

```java
public class Monitor {
    private volatile int unfinishedTasks;

    public Monitor()

    public synchronized void trainSetup(int ntasks)

    public synchronized int finishTraining()

    public synchronized void requestUpdate()
}
```

THE CONCURRENT TRAINER -- USING THREAD POOL

- Repeatedly creating and destroying threads can waste a lot of resource and time.
- Can pre-define a fixed size thread pool to avoid this problem.

```
        this.pool = Executors.newFixedThreadPool(poolSize); // create a fixed size thread pool
```

```
        // assign tasks to worker threads
        for (int i = 0; i < taskCount - 1; i++)
        {
            this.pool.execute(tasks[i]);
        }

        // execute one of the tasks in this thread
        tasks[taskCount - 1].run();
```

TESTING

- The concurrent training algorithm only parallelizes the computations over data samples within each mini-batch.
- The computed update statistics should be the same for both the sequential and concurrent algorithms.
- Define the concurrent implementation as correct if the model trained by the concurrent trainer is **equivalent** to the same model trained by the sequential trainer.
- Two models are considers equivalent if the differences between all their weights and biases are within some small error ε.

TESTING

```
Problems  @ Javadoc  Declaration  Console ☒
<terminated> Demo (1) [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (2015年10月30日 上午4:38:48)
readed in 60000 data samples
train set size: 54000
validation set size: 6000
[epsilon=0.000001] initial test: 318010 out of 318010 elements are considered equal
train model1 sequentially...
train model2 concurrently...
[epsilon=0.000001] test 1: 318010 out of 318010 elements are considered equal
[epsilon=0.000001] test with original model: 318003 out of 318010 elements are considered equal
train model1 sequentially...
train model2 concurrently...
[epsilon=0.000001] test 2: 318010 out of 318010 elements are considered equal
[epsilon=0.000001] test with original model: 318009 out of 318010 elements are considered equal
train model1 sequentially...
train model2 concurrently...
[epsilon=0.000001] test 3: 318010 out of 318010 elements are considered equal
[epsilon=0.000001] test with original model: 318008 out of 318010 elements are considered equal
train model1 sequentially...
train model2 concurrently...
[epsilon=0.000001] test 4: 318010 out of 318010 elements are considered equal
[epsilon=0.000001] test with original model: 318007 out of 318010 elements are considered equal
train model1 sequentially...
train model2 concurrently...
[epsilon=0.000001] test 5: 318010 out of 318010 elements are considered equal
[epsilon=0.000001] test with original model: 318006 out of 318010 elements are considered equal
train model1 sequentially...
```

- Have ran 100 comparison tests and all of them are considered equal.

FUTURE WORK

- Run the sequential and the concurrent algorithm on a multicore machine to see how much training time can be reduced by using the concurrent algorithm.