Right Whale Recognition and Classification

Tyler Allen

***Abstract*—With less than 500 North Atlantic Right Whales remaining on earth, scientists track their health through aerial photographs. This is time consuming, and can possibly be improved by machine learning. In this paper we will discuss using the results of photo recognition as input to neural networks, and the effectiveness of this method.**

I. Introduction

North Atlantic Right Whales are an endangered species. Less than 500 Right Whales remain[8]. Scientists keep track of the health conditions of the remaining Right Whales in order to help preserve the species. This process is done by examining aerial photographs of these whales. Each whale has an identification number[8]. Identifying these whales is extremely time consuming. It may be possible to improve this process by introducing machine learning. Since photographs are the only source of information, there are two problems that must be solved to complete this project. First, the whale must be recognized in the photograph. Second, the whale must be classified with its idenficiation number. This creates a very complex problem, as it requires the results of one machine learning process to be used as input to other machine learning algorithm. In this paper, we will discuss the approach fo using the *MATLAB* suite of image recognition tools to do photo recognition, the approach of using a neural network to classify recognized whales, and the approach of using a deep belief network to classify recognized whales. We will see that the approach of photo recognition presents its own set of issues as input to classification issues, and discuss the results of using this method. We will also discuss the lessons learned from this approach, and present potential future work that could provide better results. This project is based on a contest by *kaggle.com*[8].

II. Methodology

A. Data

The data provided for this competition is a set of 11469 images of Right Whales collected over the course of ten years[14]. These images vary by angle and resolution, and they may contain a number of attributes considered to be noise. This can be seen by contrasting figures 1 and 2. This noise includes lighting variation, surface reflectivity, surface disturbance, other whales, dolphins, seagulls, and particles in the water. Common information such as creation dates and geo tags were removed from the photographs prior to distribution.

Fig. 1: Sample whale photograph.

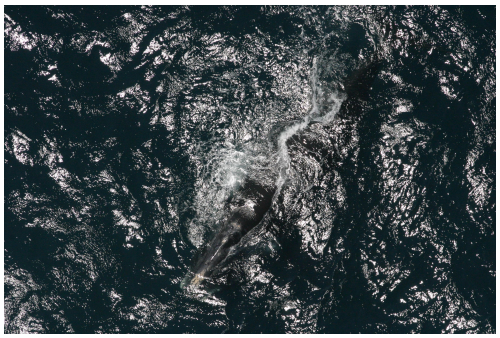

Fig. 2: Whale photos can have a large degree of variation.

The training data consists of 4544 of the original 11469 images. A list is provided containing the file name of each training image along with the correct identification number. The contest instructions claim that the remaining 6925 images may be "resized, cropped, or flipped" to discourage hand labeling of test data[14].

B. Detection

Detection was performed by using the *MATLAB* Cascade Object Detector software provided as part of the competition[2][16]. This software is a machine learning tool that can be trained to detect objects, and can be used for tasks such as facial recognition. We chose to use it as a "whale facial recognition" tool, as was suggested by the contest providers[2]. Using the full body was considered, but it was noted that the photographs frequently did not contain the full body. In addition, most of the whales seemed to have identifying features (markings) on their faces that would be ideal features for classification. Due to the previously mentioned noise in the photographs, no environmental features could be used to identify the whales.

The cascade object detector requires training input that has been labeled with a bounding box around the object to be identified. The 4544 training examples were hand labeled using the *MATLAB* Training Image Labeler tool. This tool allows the user to draw a bounding box an image, and then export the data for use in later matlab analysis as seen in figure 3.

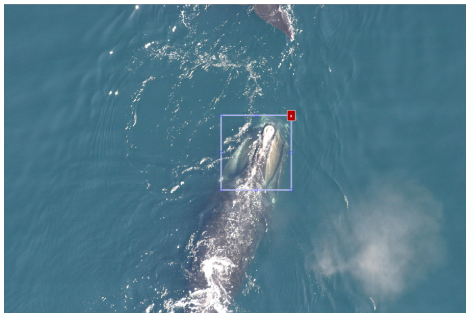

Fig. 3: A labeled training sample.

The required training parameters to the cascade object detector are `outputXMLFilename`, `positiveInstances`, and `negativeImages`[1]. The parameter `outputXMLFilename` is simply the location to store data. `positiveInstances` is the exported data from the Training Image Labeler tool. `negativeImages` is a directory of negative images. The Cascade Object Detector requires a number of images not containing whales to help distinguish between the whales present in the photo and the background noise[1]. These photos were procedurally generated by cropping parts of photos where whales had already been identified. Two optional parameters are the `FalseAlarmRate` and `NumCascadeStages`. The false alarm rate is "the fraction of negative training samples incorrectly classified as positive samples", and the number of cascade stages are the number of times detector should iterate over the training data. The suggested values were 15 stages, with a false alarm rate of 0.01. Experimentation led to the use of 7 stages, and 0.005 for the false alarm rate. Stages 9 and above generated a program warning indicating that it would not be able to sufficiently identify images. Reducing the stages and decreasing the false alarm rate seemed to provide more accurate results.

C. Classification

Classifying large images that have not been normalized is a very challenging problem. Out of the 4544 training examples provided, there are 447 unique whale identification numbers. Each whale identification number has a varying number of correlated training examples. One of the 447 only has a single associated training image. The appropriate classification methods for this type of data must have the ability to understand image input. The correct methods must also be able to handle noise, because noise is present in the iamges. Additional noise is also generated by the Cascade Object Detector output, since the correct whale image may sometimes not be chosen. The two methods chosen were a neural network, and a deep belief network.

The input to the machine learning programs is the output of the image recognition stage. To provide a good input format for the neural network, the data was normalized. First, the bounding boxes detected by the image recognition stage were cropped out of their original image. These images were resized to $32x30$ resolution. This value was pulled from a "face pose" study of humans contained in the Machine Learning textbook by Mitchell[18]. This resolution is low, but still maintains the features to some extent. Larger images require more processing time, so this size was chosen

to preserve features while reducing processing time. Figure 4 is an example of one of these resized images. In addition, the photos were converted to greyscale for the neural network, to reduce the amount of image data by a third, while preserving the contrast between whales and their facial markings.

Fig. 4: Resized sample image at true resolution.

The neural network was implemented in the Java programming language. It is a three layer feedforward neural network using backpropogation for training, as described in Mitchell[18]. The input layer consists of 960 input nodes, which is the product of the $32x30$ resolution. Each input is a single, in-order, greyscale pixel. The output layer is 447, one node for each possible whale identification number classification. The hidden layer is 704, the average of the two values. This is due to it being the average of the input output layers. This may not be optimal, but training times did not allow for validation of this value. The neural network also offered functionality to store the state of the network, so that the training results could be reused. The learning rate was 0.1, and the weights began as values between -0.5 and $.5$, as referenced from Mitchell[18]. The node activation function was the standard sigmoid function[18]. Each greyscale input was resized from the range of $0-255$ to a floating point value between -1 and 1.

The neural network was written in three iterations. The initial version used online learning. Due to the extreme runtime required by this version, it was converted to batch learning as an intermediate step, and then rewritten to make use of MPI. This would allow training to be done in parallel on the palmetto cluster in hopes of reducing training time. The MPI version divide the training images in each epoch to other nodes. Since batch learning was used, each image simply had to return its results to be summed by the master. The training section of the MPI code is available in appendix A.

The deep belief network was implemented using the *Deep Learning 4 Java* library[4]. The configuration of this network is very similar to the previous neural network. It again consists three layers with 960 input nodes, 704 hidden layer nodes, and 447 output nodes. A learning rate of $.0000001$ was used, as suggested by the Deep Belief Network reference page[3]. As is consistent with the definition of a Deep Belief Network, the layers are constructed of Restricted Boltzmann machines[3]. The activation function is a rectified linear unit function, again recommended by the Deep Belief Network construction page[3]. Appendix B provides the code used in the network setup.

III. Results

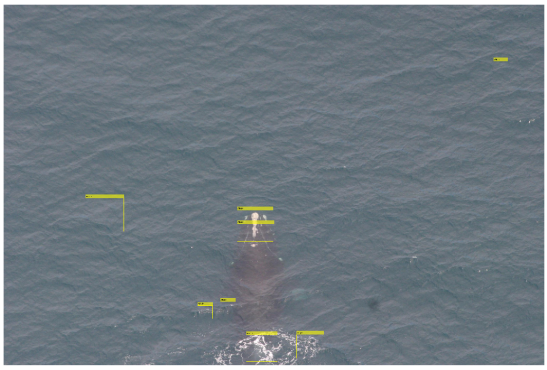

Fig. 5: Objects detected by the CascadeObjectDetector.

The final results of the recognition seem to vary quite a bit from image to image. Ideally, only one bounding box would appear per image. Unfortunately, there are typically more than one objects detected. Figure 5 is one example. The face was accurately detected, but there are several incorrect detections as well. Detecting the correct whale out of the selections made is another challenging task, as adjusting the parameters only seemed to provide minor improvements. No adjustments prevented most pictures from having false positives. Unfortunately, we must chose a single one of the detected bounding boxes to use for classification. The current implementation chooses a random box of the ones found. Other methods were investigated, such as searching for the "most gray" bounding box, but the lighting conditions and reflectivity found in various images cause this method to skew the data towards certain image qualities.

Results for classification are evaluated using a multi-class logarithmic loss method[12]. The perfect result set would have a score of 0, and higher values indicate less correctness. The highest score is currently 3.02633, as of December 5^{th}.

The basic neural network using batch learning scored a result of 34.49601 on the leaderboard. The neural network was only trained through 100 iterations. This is because training had to restart several times throughout the course of this project, and it requires roughly 26 minutes to do one epoch over the 4544 training data samples. This means that 100 iterations required nearly 2 consecutive days. It was not possible to run the simulation continuously for several days, so it had to be trained in parts over time. It would have required roughly 18 consecutive days to complete 1000 iterations. Mitchell succeeded in classifying human face position in only 100 iterations, but the network was much smaller, had only 260 training samples, and had only 4 possible classifications[18]. Training time is probably not the only issue, as the input from the facial recognition has already been shown to be inaccurate. When input from facial recognition is a false positive, it's possible that the neural network partially learns features outside of expectation, such as water color.

The MPI version of the neural network was unable to produce results. When launched on palmetto, every node died when the Java Virtual Machine (JVM) failed to initialize due to lack of memory. This occurred with both the preinstalled versions of Java and MPI, as well as the versions installed locally to the user account. This persisted despite passing arguments to the Java Virtual Machine to allocate more memory on startup. Requesting up to 12GB of memory per node was also insufficient, despite the non-MPI version requiring far less than 12GB. Since the JVM crashed on initialization, the neural network code was never able to initialize, so it is most likely a configuration issue with Palmetto. There was insufficient time to contact CCIT to resolve this issue in time to provide results in this paper.

The deep belief network had similar issues to the neural network. With 500 epochs, using 80% of the data as training data, and the remaining 20% as test data, it received an accuracy score of $.001$, precision of 0.0217, recall of $.0909$, and F1 score of 0.0351. These scores are far below expectation for a sophisticate neural network. The number of epochs is not recommended directly for problems like this by Deep Learning 4 Java, but in at least one case 500 iterations are used[17].

IV. Discussion

A. Lessons Learned

Machine Learning using images is a challenging problem. Using the output of a machine learning algorithm as input to other machine learning algorithms should not have been my first goal with machine learning. It has been a very interesting problem to work on, but some prior experience would have prevented a lot of difficulty and issues, and likely provided much better results over the course of this project.

The *Matlab* Cascade Object Detector may not have been the optimal tool for facial recognition. An alternative tool, `haarcascade` from the *OpenCV* library was also considered as a replacement for the Cascade Object Detector[5]. This tool was suggested by some users on the Kaggle website because in some cases it performed better than the appropriate matlab tool[6]. This was ruled out, because the classification process would be too time consuming to run on two different data sets for comparison.

The neural network was written in Java with the intention of allowing different machine learning methods to be designed in an object-oriented fashion so that they could be modularly exchanged within the same program. This would have provided a good design for analysis and comparison. In hindsight, the language of implementation probably should have been C or C++ to facilitate faster runtimes. However, it is abundantly clear that libraries such as DL4J provide far greater performance than custom software for challenging these problems in the real world[15].

The Deep Learning 4 Java library, while powerful, is very poorly documented. The website contains several examples, but none that fit my specific needs. The associated API documentation is largely absent. It seemed to be the only common deep learning library available for Java. Java was chosen for compatability with some of the existing code from the neural network. This is another area where perhaps C, C++, or other languages may be better due to better support.

B. Future Work

The first step in continuing this project would have to be sorting out the issues in the initial photo recognition stage. One possibility would be using the alternative photo recognizer from *OpenCV*[5]. Using the current deep belief network on larger scale images, or images that have not been cropped, may be a worthwhile experiment. In some cases, it is recommended by the Deep Learning 4 Java library not to normalize the data[7]. Removing all layers of normalization, and then adding them back one-by-one, may give an indication of what the deep belief network expects. Improving the photo recognition stage or finding a way to collapse the two steps into one step is definitely the first goal to continue this project.

There may also be more appropriate methods for use on this project. In late November, a post appeared on the contest forums with information about Convolution autoencoders[13]. This information was written and provided by the person ranked 2^{nd} in the Right Whale competition with the intention to "level the playing field." This seems like a strong direction to take this project. Another student presented work on the MNIST data set using this method as well, and had the best results with it.

V. Conclusion

The main products of this project were a trained whale face recognizer, a neural network written in Java, and a deep belief network using the Deep Learning 4 Java library. In general, the resulting accuracy of these scores was very low. It is believed that this is largely caused by false positives generated by the recognition step. Since these results were fed into the following classification step, the classification step can be at most as accurate as the original recognition. Accuracy could be greatly improved by reducing the impact of the recognition step, or by somehow collapsing the first and second step together. Future work will involve reducing or removing the recognition step.

References

[1] *Cascade object detector*. http://www.mathworks.com/help/vision/ref/traincascadeobjectdetector.html, Accessed: 2015-10-04.

[2] *Creating a "face detector" for whales*. http://www.kaggle.com/c/noaa-right-whale-recognition/details/creating-a-face-detector-for-whales, Accessed: 2015-09-15.

[3] *Deep-belief networks*. http://deeplearning4j.org/deepbeliefnetwork.html, Accessed: 11/7/2015.

[4] *Dl4j*. http://deeplearning4j.org/, Accessed: 11/7/2015.

[5] *Face detection using haar cascades*. http://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html#gsc.tab=0, Accessed: 2015-10-04.

[6] *Face detection using haar cascades*. https://www.kaggle.com/c/noaa-right-whale-recognition/forums/t/17016/matlab-roi-and-opencv-classifier, Accessed: 2015-10-04.

[7] *Facial reconstruction*. http://deeplearning4j.org/facial-reconstruction-tutorial.html, Accessed: 12/5/2015.

[8] *Identify endangered right whales in aerial photographs*. http://www.kaggle.com/c/noaa-right-whale-recognition, Accessed: 2015-09-15.

[9] *Labeling images for classification model training*. http://www.mathworks.com/help/vision/ug/label-images-for-classification-model-training.html, Accessed: 2015-09-15.

[10] *New england aquarium*. www.neaq.org/index.php, Accessed: 2015-09-15.

[11] *Noaa fisheres*. www.nmfs.noaa.gov, Accessed: 2015-09-15.

[12] *Object recognition for fun and profit*. https://www.kaggle.com/c/noaa-right-whale-recognition/details/evaluation.

[13] *Object recognition for fun and profit*. https://github.com/anlthms/whale-2015/blob/master/object-recognition.pdf.

[14] *Right whale recognition data*. http://www.kaggle.com/c/noaa-right-whale-recognition/data, Accessed: 2015-09-15.

[15] *Scaleout*. http://deeplearning4j.org/scaleout.html, Accessed: 12/5/2015.

[16] *Train a cascade object detector*. www.mathworks.com/help/vision/ug/train-a-cascade-object-detector.html, Accessed: 2015-09-15.

[17] *The welldressed recommendation engine*. http://deeplearning4j.org/welldressed-recommendation-engine.html.

[18] Thomas M. Mitchell, *Machine learning*, 1st ed., McGraw-Hill, Inc., New York, NY, USA, 1997.

APPENDIX A
MPI TRAINING METHOD

```
public void runEpoch(java.util.List<WhaleImage> training) throws MPIException
{
    int rank = MPI.COMM_WORLD.getRank();
    int size = MPI.COMM_WORLD.getSize();

    double[][] odws = new double[output.length][];
    int oSize = 0;
    for (int i = 0; i < output.length; i++)
    {
        oSize += output[i].weights.keySet().size();
        odws[i] = new double[output[i].weights.keySet().size()];
    }

    double[][] hdws = new double[hidden.get(0).length][];
    int hSize = 0;
    for (int i = 0; i < hidden.get(0).length; i++)
    {
        Perceptron current = hidden.get(0)[i];
        hSize += current.weights.keySet().size();
        hdws[i] = new double[current.weights.keySet().size()];
    }

    if (rank == 0)
    {

        double[][] mOut = new double[output.length][];
        for (int j = 0; j < output.length; j++)
        {
            odws[j] = new double[output[j].weights.keySet().size()];
        }

        double[][] mHid = new double[hidden.get(0).length][];
        for (int j = 0; j < hidden.get(0).length; j++)
        {
            Perceptron current = hidden.get(0)[j];
            hdws[j] = new double[current.weights.keySet().size()];
        }

        int counter = 0;
        for (int i = 0; i < training.size(); )
        {
            Status status = MPI.COMM_WORLD.recv(null, 0, MPI.DOUBLE, MPI.ANY_SOURCE, MPI.ANY_TAG)
                ↪ ;
            if (status.getTag() == TAGS.WORK_NEED.ordinal())
            {
                MPI.COMM_WORLD.send(i, 1, MPI.INT, status.getSource(), TAGS.WORK_TODO.ordinal());
                i++;
            }
            else if (status.getTag() == TAGS.WORK_DONE.ordinal())
            {
                MPI.COMM_WORLD.send(null, 0, MPI.INT, status.getSource(), TAGS.WORK_TODO.ordinal
                    ↪ ());
                MPI.COMM_WORLD.recv(mOut, oSize, MPI.DOUBLE, status.getSource(), TAGS.WORK_DONE.
                    ↪ ordinal());
                MPI.COMM_WORLD.recv(mHid, hSize, MPI.DOUBLE, status.getSource(), TAGS.WORK_DONE.
                    ↪ ordinal());

                for (int j = 0; j < output.length; j++)
                {
                    for (int k = 0; k < output[j].weights.keySet().size(); k++)
                    {
                        odws[j][k] += mOut[j][k];
                    }
                }

                for (int j = 0; j < hidden.get(0).length; j++)
                {
                    Perceptron current = hidden.get(0)[j];
```

```
                    for (int k = 0; k < current.weights.keySet().size(); k++)
                    {
                        hdws[j][k] += mHid[j][k];
                    }
                }
            }
        }
        while (size > 0)
        {
            Status status = MPI.COMM_WORLD.recv(null, 0, MPI.DATATYPE_NULL, MPI.ANY_SOURCE, TAGS.
                ↪ WORK_DONE.ordinal());
            MPI.COMM_WORLD.send(null, 0, MPI.DATATYPE_NULL, status.getSource(), TAGS.WORK_DONE.
                ↪ ordinal());
            MPI.COMM_WORLD.recv(mOut, oSize, MPI.DOUBLE, status.getSource(), TAGS.WORK_DONE.
                ↪ ordinal());
            MPI.COMM_WORLD.recv(mHid, hSize, MPI.DOUBLE, status.getSource(), TAGS.WORK_DONE.
                ↪ ordinal());

            for (int j = 0; j < output.length; j++)
            {
                for (int k = 0; k < output[j].weights.keySet().size(); k++)
                {
                    odws[j][k] += mOut[j][k];
                }
            }

            for (int j = 0; j < hidden.get(0).length; j++)
            {
                Perceptron current = hidden.get(0)[j];
                for (int k = 0; k < current.weights.keySet().size(); k++)
                {
                    hdws[j][k] += mHid[j][k];
                }
            }
            size--;
        }


        // Apply weight change.
        for (int i = 0; i < output.length; i++)
        {
            int j = 0;
            for (Perceptron p : output[i].weights.keySet())
            {
                double cw = output[i].weights.get(p);
                output[i].weights.put(p, cw + odws[i][j]);
                j++;
            }
        }

        for (int i = 0; i < hidden.get(0).length; i++)
        {
            int j = 0;
            Perceptron current = hidden.get(0)[i];
            for (Perceptron p : current.weights.keySet())
            {
                double cw = current.weights.get(p);
                current.weights.put(p, cw + hdws[i][j]);
                j++;
            }
        }
    }
    else
    {
        boolean done = false;
        while (!done)
        {
            Integer mInt = 0;
            MPI.COMM_WORLD.send(null, 0, MPI.DATATYPE_NULL, 0, TAGS.WORK_NEED.ordinal());
            Status status;
            MPI.COMM_WORLD.recv(mInt, 1, MPI.INT, 0, TAGS.WORK_TODO.ordinal());
```

```
WhaleImage anInput = training.get(mInt);

BufferedImage image = null;
int[] colorBuffer;
try
{
    // get the BufferedImage, using the ImageIO class
    image = ImageIO.read(anInput.getFile());
}
catch (IOException e)
{
    System.err.println(e.getMessage());
    System.err.println("File: " + anInput.getFile());
    return;
}
colorBuffer = image.getRGB(0, 0, WIDTH, HEIGHT, null, 0, WIDTH);
Color color;
for (int i = 0; i < INPUTS; i++)
{
    color = new Color(colorBuffer[i]);
    // derive gray and normalize...
    int gray = (((color.getRed() + color.getGreen() + color.getBlue()) / 3) / 255) *
        ↪ 2 - 1;
    this.input[i].receiveInput(gray);
}
for (Perceptron p : input)
{
    p.activate();
}

double[] hidden_out = new double[hidden.get(0).length];
for (int i = 0; i < hidden.get(0).length; i++)
{
    Perceptron p = hidden.get(0)[i];
    hidden_out[i] = p.activate();
}
// Stores all output node results for backprop.
double[] out = new double[output.length];
double[] hiddenErr = new double[hidden.get(0).length];
double[] outErr = new double[output.length];
for (int i = 0; i < out.length; i++)
{
    double o_k = output[i].activate();
    out[i] = o_k;
    double val = outputMap.get(anInput.getWhaleId()) == output[i] ? .9 : .1;
    outErr[i] = o_k * (1 - o_k) * (val - o_k);
}

// backprop for hidden...
for (int i = 0; i < hidden.get(0).length; i++)
{
    Perceptron p = hidden.get(0)[i];
    double o_h = hidden_out[i];
    double wSum = 0;
    for (int j = 0; j < p.connections.size(); j++)
    {
        wSum += outErr[j] * p.connections.get(j).getWeight(p);
    }
    hiddenErr[i] += o_h * (1 - o_h) * wSum;
}

// Apply weight change.
for (int i = 0; i < output.length; i++)
{
    double temp = N * outErr[i];
    int j = 0;
    for (Perceptron p : output[i].weights.keySet())
    {
        double dw = temp * output[i].in.get(p);
        odws[i][j] += dw;
```

```
                    j++;
                }
            }

            for (int i = 0; i < hidden.get(0).length; i++)
            {
                double temp = N * hiddenErr[i];
                Perceptron current = hidden.get(0)[i];
                int j = 0;
                for (Perceptron p : current.weights.keySet())
                {
                    double dw = temp * current.in.get(p);
                    hdws[i][j] += dw;
                    j++;
                }
            }
            MPI.COMM_WORLD.send(null, 0, MPI.DATATYPE_NULL, 0, TAGS.WORK_DONE.ordinal());
            status = MPI.COMM_WORLD.recv(null, 0, MPI.DATATYPE_NULL, 0, MPI.ANY_TAG);
            if (status.getTag() == TAGS.WORK_DONE.ordinal())
            {
                done = true;
            }
            MPI.COMM_WORLD.send(hdws, oSize, MPI.DATATYPE_NULL, 0, TAGS.WORK_DONE.ordinal());
            MPI.COMM_WORLD.send(odws, hSize, MPI.DATATYPE_NULL, 0, TAGS.WORK_DONE.ordinal());
        }
    }
}
```

APPENDIX B
DEEP BELIEF NETWORK SETUP

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed) // Locks in weight initialization for tuning
        .iterations(iterations) // # training iterations predict/classify & backprop
        .learningRate(1e-6f) // Optimization step size
        .optimizationAlgo(OptimizationAlgorithm.CONJUGATE_GRADIENT) // Backprop to calculate
            ↪ gradients
        .l1(1e-1).regularization(true).l2(2e-4)
        .useDropConnect(true)
        .list(2) // # NN layers (doesn't count input layer)
        .layer(0, new RBM.Builder(RBM.HiddenUnit.RECTIFIED, RBM.VisibleUnit.GAUSSIAN)
                .nIn(imgSize) // # input nodes
                .nOut(hiddenSize) // # fully connected hidden layer nodes. Add list if multiple
                    ↪ layers.
                .weightInit(WeightInit.XAVIER) // Weight initialization
                .k(1) // # contrastive divergence iterations
                .activation("relu") // Activation function type
                .lossFunction(LossFunctions.LossFunction.RMSE_XENT) // Loss function type
                .updater(Updater.ADAGRAD)
                .dropOut(0.5)
                .build()
        ) // NN layer type
        .layer(1, new OutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
                .nIn(hiddenSize) // # input nodes
                .nOut(outputNum) // # output nodes
                .activation("softmax")
                .build()
        ) // NN layer type
        .build();
MultiLayerNetwork network = new MultiLayerNetwork(conf);
```