CIS 678 Machine Learning
Wolffe
Project #4 v2.0
Neural Network Classifier

Rob Sanchez
April 11, 2017

Introduction

This report describes a neural network classifier, *ANN.jl*, implemented in Julia[1]. Artificial neural networks are computational models that mimic biological neuron networks in the brain, and can be used to solve classification, regression, and clustering problems, among others[2]. This project focused on classification, and two datasets were considered: the classic fishing dataset with four features (wind, water, air, and forecast) and a digits dataset, a collection of bitmaps representing handwritten digits.

After a brief description of the data used and a design overview, we discuss a series of experiments comparing training and classification performance for the classifier, subject to various neural network tuning properties. We find that online training suited the fishing dataset rather well, with results converging around 20 or 21 No and 15 or 16 Yes classifications. Both mini-batch and online training worked well on the digits dataset (5-6% classification error).

Data

The fishing dataset consisted of 14 training examples with four features each: wind(strong or weak), water(cold, moderate or warm), air(cool or warm), and forecast(rainy, cloudy or sunny). In order to be processed by the NeuralNet, they were encoded according to the scheme defined in function *encfish(),* found in *ANNlib.jl*. For example, "strong wind" was encoded as [0 1] and "rainy forecast" became [1 0 0]. With this encoding scheme, the four features became ten, so the NeuralNet for fishing was defined with ten input nodes. Two output nodes, representing the answer to the question, "Is it a good day to fish?" (NO or YES) were also configured. For classification, all 36 possible fishing feature combinations were generated with the *allfishing()* function and encoded with *encfish()*.

The digits dataset consisted of 3823 training examples and 1797 test instances. Each observation in these two sub-datasets was made up of integer data, 64 values representing an 8x8 bitmap of a digit character representing intensity (0-16), and one final value representing the true class or target integer of that observation (0-9). NeuralNets for the digits dataset were defined with 64 input nodes (one for each bitmap value) and ten output nodes (one for each target digit). The digits target values were stripped from each observation and one-hot encoded with the *onehot()* function.

Design Overview

In this implementation, a new type "NeuralNet" was created and its constructor defined in file *ANNlib.jl*, the support module for the project. NeuralNet was defined as having one input layer, one hidden layer, and one output layer, each consisting of a set of nodes resembling bio-neurons for

computation. The input and hidden layers also each had one so-called "bias node" attached, a standard technique used to ensure that meaningful output was passed to the next layer when node inputs were zero. Due to time constraints, and in an effort to maintain code simplicity, no attempt was made to employ multiple hidden layers for NeuralNet. Experimental results for digits suggest that one hidden layer was suitable for classification.

Backpropagation[3] was implemented as a workflow of three functions: *feedforward()*, *backpropagate()*, and *learn()*. Using the *sigmoid()* function, new observations were fed through the NeuralNet with *feedforward()*, error delta vectors were calculated in *backpropagate()* with the sigmoid derivative *dsigmoid()*, and these deltas were then used in *learn()* in combination with the learning rate to update the network edge weights, in preparation for the next observation. These three functions were chained together in a loop implemented in *train()*, and executed on an incoming dataset after shuffling the training examples.

The edge weights in each new NeuralNet instance were initialized with small randomly-generated values, ranging from -0.125 to 0.125 as specified and assigned in *initweights()*. Function *tinyrand()* (found in *ANNlib.jl*) was used to randomly generate the weight values.

Two training modes were implemented: online, which backpropagates and updates the edge weights in the NeuralNet after each training example, and mini-batch, which delays backpropagation and weight updates until a set number of observations have been fed through the network. A *classify()* function was written to accept either fishing or digits datasets and classify them with the configured NeuralNet. In the case of fishing, *classify()* would return a count of No or Yes classifications, and for digits, the number of classification misses and an error percentage. A *softmax()* implementation from the Julia StatsFuns[4] library was used by *classify()* for counting and indexing classifications.

Experiments

During code development, ad-hoc trial-and-error narrowed down the possibilities of parameters that seemed most useful for classifying digits and fishing:

Parameter	digits	fishing
mini-batch size:	10, 25, or 1 (online)	1 (online-only)
hidden nodes:	40 or 50	4, 6, or 8
learning rate:	0.025 or 0.05	0.025 or 0.05
epochs:	50, 100, or 1 (online)	10, 25, 50, 100, 250 or 1000
sample size factor:	0.10, 0.15 or 1	1 (online-only)

Functions *digits_exp()* and *fishing_exp()* were used to loop through these parameter possibilities, and for each parameter value combination, to run a training and classification experiment. The results were collected into a Julia DataFrame[5]. To prevent the list of experiments from growing too large, and to allow straight-forward presentation here, the set of possible parameters (listed above) was intentionally kept small. The *NeuralNetCTL()* wrapper was used to trigger the experiment runs with the relevant options.

Due to the fishing dataset's very small size (14 training, 36 testing) all 36 fishing experiments were executed quickly, in about two minutes. For this reason, only online training (sample size factor

= 1) was used for that dataset's training and classification runs. Table 1 shows the results of the 36 fishing experiments, where the counts of No and Yes classifications appear to converge at 20 or 21 (NO) and 15 or 16 (YES) respectively, no matter the number of epochs, the learning rate, or hidden node count. For all experiments with 50 epochs or more, the accumulated training error ranged from 0.23% to 8.78%.

Row	dataset	train	batchSz	hNodes	lRate	epochs	accTrnErr	NO	YES	runtime
1	"fishing"	"ol"	1	4	0.025	10	21.09	16	20	0.2
2	"fishing"	"ol"	1	4	0.05	10	22.19	9	27	0.1
3	"fishing"	"ol"	1	6	0.025	10	10.7	16	20	0.1
4	"fishing"	"ol"	1	6	0.05	10	11.98	21	15	0.1
5	"fishing"	"ol"	1	8	0.025	10	13.2	25	11	0.2
6	"fishing"	"ol"	1	8	0.05	10	15.18	19	17	0.2
7	"fishing"	"ol"	1	4	0.025	25	10.78	21	15	0.2
8	"fishing"	"ol"	1	4	0.05	25	14.57	21	15	0.2
9	"fishing"	"ol"	1	6	0.025	25	10.69	21	15	0.4
10	"fishing"	"ol"	1	6	0.05	25	7.52	21	15	0.4
11	"fishing"	"ol"	1	8	0.025	25	8.3	21	15	0.5
12	"fishing"	"ol"	1	8	0.05	25	14.49	21	15	0.5
13	"fishing"	"ol"	1	4	0.025	50	7.27	21	15	0.5
14	"fishing"	"ol"	1	4	0.05	50	5.98	21	15	0.5
15	"fishing"	"ol"	1	6	0.025	50	5.39	21	15	0.7
16	"fishing"	"ol"	1	6	0.05	50	8.78	21	15	0.7
17	"fishing"	"ol"	1	8	0.025	50	5.76	21	15	1.0
18	"fishing"	"ol"	1	8	0.05	50	3.92	21	15	0.9
19	"fishing"	"ol"	1	4	0.025	100	4.24	21	15	1.0
20	"fishing"	"ol"	1	4	0.05	100	3.27	21	15	1.0
21	"fishing"	"ol"	1	6	0.025	100	2.55	21	15	1.4
22	"fishing"	"ol"	1	6	0.05	100	3.91	21	15	1.4
23	"fishing"	"ol"	1	8	0.025	100	3.53	20	16	1.9
24	"fishing"	"ol"	1	8	0.05	100	2.71	21	15	1.9
25	"fishing"	"ol"	1	4	0.025	250	1.3	21	15	2.5
26	"fishing"	"ol"	1	4	0.05	250	1.28	21	15	2.4
27	"fishing"	"ol"	1	6	0.025	250	1.25	20	16	3.6
28	"fishing"	"ol"	1	6	0.05	250	1.1	21	15	3.6
29	"fishing"	"ol"	1	8	0.025	250	1.29	21	15	4.7
30	"fishing"	"ol"	1	8	0.05	250	1.13	20	16	4.7
31	"fishing"	"ol"	1	4	0.025	1000	0.32	20	16	9.8
32	"fishing"	"ol"	1	4	0.05	1000	0.35	20	16	9.7
33	"fishing"	"ol"	1	6	0.025	1000	0.26	21	15	14.4
34	"fishing"	"ol"	1	6	0.05	1000	0.23	20	16	14.4
35	"fishing"	"ol"	1	8	0.025	1000	0.29	20	16	19.0
36	"fishing"	"ol"	1	8	0.05	1000	0.38	20	16	18.9

Table 1. 36 classification experiments on the fishing dataset, sorted by 'epochs'.

Overall, the NeuralNet was very good at recognizing the handwritten characters encoded in the bitmap integer data, provided that the network was tuned appropriately. Both online and mini-batch training performed well on digits. Table 2 presents 40 experiments, with a mixture of online and mini-batch training type. In this early set of experiments, when online training was selected, the mini-batch size and number of epochs was fixed to 1. Again, as it was for fishing, the sample size factor was fixed to 1 in online mode, so that all training examples would be processed with backpropagation and update in function *train()*. Early ad-hoc experiments during development showed that only 1 epoch was necessary for digits classification in online mode. Table 2 column "accTrnErr" shows the accumulated total network error, the total error summed after each backpropagation and update, divided by the number of epochs. For all online experiments (epoch = 1) this figure is quite low, suggesting that training converged quickly in online mode.

The best classification result however, from this initial set of experiments, was a result of mini-batch training. Row 1 in Table 2 shows a 5.18 % classification error, with 40 hidden nodes, batch

size 10, learning rate 0.05, and a sample size factor of 15% (0.15 * 3823 = 573). The best online training result is listed in row 5, at 6.07% classification error.

Row	dataset	train	batchSize	hNodes	lRate	epochs	sampleSz	accTrnErr	misses	classErr	runtime
1	"digits"	"mb"	10	40	0.05	100	573	11.61	93	5.18	225.4
2	"digits"	"mb"	10	50	0.05	100	573	8.65	93	5.18	282.6
3	"digits"	"mb"	10	40	0.05	100	382	11.63	103	5.73	150.8
4	"digits"	"mb"	10	50	0.025	100	382	14.39	108	6.01	195.5
5	"digits"	"ol"	1	40	0.025	1	3823	0.43	109	6.07	134.6
6	"digits"	"ol"	1	40	0.05	1	3823	0.03	112	6.23	130.8
7	"digits"	"ol"	1	40	0.05	1	3823	0.05	118	6.57	131.3
8	"digits"	"mb"	10	40	0.025	100	573	10.77	119	6.62	226.4
9	"digits"	"mb"	10	40	0.025	100	382	9.94	122	6.79	151.0
10	"digits"	"mb"	10	50	0.05	100	382	10.79	122	6.79	203.1
11	"digits"	"mb"	10	50	0.05	50	573	11.46	125	6.96	148.0
12	"digits"	"ol"	1	50	0.025	1	3823	0.01	125	6.96	169.9
13	"digits"	"ol"	1	50	0.05	1	3823	0.41	125	6.96	170.1
14	"digits"	"mb"	10	40	0.025	50	573	17.68	127	7.07	112.7
15	"digits"	"mb"	10	40	0.05	50	573	12.25	131	7.29	112.9
16	"digits"	"mb"	25	50	0.025	100	573	18.19	131	7.29	130.7
17	"digits"	"mb"	10	50	0.025	100	573	13.01	131	7.29	294.3
18	"digits"	"mb"	10	40	0.05	50	382	16.14	138	7.68	77.3
19	"digits"	"mb"	25	40	0.05	100	573	15.67	138	7.68	105.8
20	"digits"	"ol"	1	50	0.025	1	3823	0.02	138	7.68	169.1
21	"digits"	"ol"	1	40	0.025	1	3823	0.03	139	7.74	135.2
22	"digits"	"mb"	25	40	0.025	100	573	18.72	140	7.79	103.9
23	"digits"	"mb"	25	50	0.05	100	573	13.95	148	8.24	123.6
24	"digits"	"mb"	10	50	0.05	50	382	18.2	154	8.57	100.1
25	"digits"	"ol"	1	50	0.05	1	3823	0.01	161	8.96	170.2
26	"digits"	"mb"	10	50	0.025	50	573	10.44	163	9.07	143.2
27	"digits"	"mb"	25	50	0.05	100	382	22.18	171	9.52	85.2
28	"digits"	"mb"	25	50	0.025	100	382	22.5	180	10.02	89.3
29	"digits"	"mb"	25	50	0.05	50	573	27.89	191	10.63	65.8
30	"digits"	"mb"	25	40	0.05	50	573	27.19	192	10.68	53.1
31	"digits"	"mb"	10	50	0.025	50	382	21.99	199	11.07	95.0
32	"digits"	"mb"	10	40	0.025	50	382	18.17	202	11.24	75.3
33	"digits"	"mb"	25	40	0.025	100	382	21.11	211	11.74	71.0
34	"digits"	"mb"	25	40	0.05	100	382	22.68	218	12.13	70.8
35	"digits"	"mb"	25	50	0.025	50	573	29.76	241	13.41	66.5
36	"digits"	"mb"	25	40	0.025	50	573	27.65	285	15.86	52.5
37	"digits"	"mb"	25	40	0.05	50	382	32.49	413	22.98	35.9
38	"digits"	"mb"	25	40	0.025	50	382	38.83	465	25.88	35.6
39	"digits"	"mb"	25	50	0.05	50	382	33.2	515	28.66	45.3
40	"digits"	"mb"	25	50	0.025	50	382	36.93	624	34.72	45.6

Table 2. 40 classification experiments on the digits dataset, sorted by 'classErr'.

Backpropagation and update can be quite expensive in terms of runtime. Comparing rows 1 and 5 from Table 2, in 100 epochs of 573 samples, the mini-batch run processes roughly 262 observations per second ((573 * 100 + 1797) / 225.4), whereas the online run processes just under 42 per second ((3823 + 1797) / 134.6). At 7.07% classification error, the mini-batch experiment in row 14 offers comparable classification to the online experiment in row 5 in about 16% less runtime.

The reason mini-batch training was explored for this project in the first place was the rather slow (2-3 minutes) training time required for 1 online training epoch on digits. Since online training for digits was quite good in some configurations (6% error) after only 1 epoch, it became necessary to investigate how much better the NeuralNet might perform after multiple online epochs. Could online training results be improved while also reducing runtime? To examine this, the volume of training instances used in online and mini-batch training was compared.

For mini-batch with 100 epochs (Table 2, row 1), 573 digits observations were sampled in each epoch. The NeuralNet saw 57,300 training instances, and with a mini-batch size of 10 it executed

5,730 backpropagation updates. In contrast, during online training with 1 epoch, the NeuralNet saw just 3,823 training instances and 3,823 backpropagation updates. To run more online epochs (thereby feeding a higher training instance volume to the NeuralNet), while at the same time reducing the training runtime, a "breakout" condition was added to the *train()* function. During training, if 150 *totalerror()* calculations in the network smaller than 0.001 were counted, then the program would break out of the current epoch and move on to the next epoch. This gave *train()* the opportunity to re-shuffle and re-sample, based on the notion that networks learn fastest from the most *unexpected* training example[6]. Table 3 presents 36 online experiments using the new breakout condition, with 5, 10, or 25 epochs. Clearly more epochs yield generally higher classification accuracy, and "online with breakout" was able to edge out the best mini-batch run from Table 2 (4.23% vs. 5.18% error). Table 3, row 1 runtime with breakout was comparable to Table 2 rows 1 & 2 mini-batch (264.8 vs. 225.4 and 282.6), but clearly beat running through 25 complete epochs for that same online configuration (from Table 2 row 6, approximately 130 seconds per epoch): 3,250 seconds or 54.2 minutes.

Row	dataset	train	batchSz	hNodes	lRate	epochs	sampleSz	accTrnErr	misses	classErr	runtime
1	"digits"	"ol"	1	40	0.05	25	765	0.14	76	4.23	264.8
2	"digits"	"ol"	1	40	0.05	25	573	0.33	80	4.45	265.1
3	"digits"	"ol"	1	50	0.05	25	382	1.05	87	4.84	301.8
4	"digits"	"ol"	1	50	0.05	25	765	0.06	88	4.9	325.0
5	"digits"	"ol"	1	40	0.05	10	765	2.11	93	5.18	146.6
6	"digits"	"ol"	1	50	0.05	25	573	2.28	93	5.18	315.8
7	"digits"	"ol"	1	40	0.05	10	382	3.74	95	5.29	125.5
8	"digits"	"ol"	1	40	0.05	10	573	0.07	95	5.29	143.7
9	"digits"	"ol"	1	50	0.025	10	382	2.81	99	5.51	153.6
10	"digits"	"ol"	1	40	0.05	25	382	5.99	100	5.56	246.9
11	"digits"	"ol"	1	40	0.05	5	765	0.22	101	5.62	102.1
12	"digits"	"ol"	1	40	0.025	25	765	0.51	102	5.68	273.3
13	"digits"	"ol"	1	40	0.025	25	382	2.31	106	5.9	253.2
14	"digits"	"ol"	1	40	0.025	10	765	1.7	107	5.95	147.0
15	"digits"	"ol"	1	50	0.025	10	573	4.36	111	6.18	178.9
16	"digits"	"ol"	1	50	0.025	25	382	0.81	111	6.18	303.3
17	"digits"	"ol"	1	50	0.025	25	765	0.1	112	6.23	330.8
18	"digits"	"ol"	1	40	0.05	5	573	4.8	114	6.34	87.5
19	"digits"	"ol"	1	40	0.025	10	382	0.96	114	6.34	120.6
20	"digits"	"ol"	1	40	0.025	25	573	2.69	114	6.34	258.6
21	"digits"	"ol"	1	50	0.025	5	573	2.32	116	6.46	114.2
22	"digits"	"ol"	1	50	0.025	25	573	1.92	121	6.73	320.8
23	"digits"	"ol"	1	50	0.025	5	765	1.3	122	6.79	115.8
24	"digits"	"ol"	1	50	0.05	5	765	6.29	123	6.84	119.9
25	"digits"	"ol"	1	40	0.025	10	573	0.57	123	6.84	145.4
26	"digits"	"ol"	1	50	0.05	10	765	0.31	123	6.84	174.2
27	"digits"	"ol"	1	50	0.05	10	573	5.78	127	7.07	171.8
28	"digits"	"ol"	1	50	0.05	5	382	5.99	129	7.18	82.9
29	"digits"	"ol"	1	50	0.05	10	382	8.9	135	7.51	146.8
30	"digits"	"ol"	1	50	0.05	5	573	13.63	137	7.62	109.0
31	"digits"	"ol"	1	40	0.025	5	573	22.49	146	8.12	87.9
32	"digits"	"ol"	1	40	0.025	5	765	9.12	150	8.35	100.2
33	"digits"	"ol"	1	40	0.05	5	382	5.03	151	8.4	64.5
34	"digits"	"ol"	1	50	0.025	10	765	4.47	154	8.57	175.7
35	"digits"	"ol"	1	40	0.025	5	382	10.13	157	8.74	64.5
36	"digits"	"ol"	1	50	0.025	5	382	13.05	158	8.79	83.0

Table 3. 36 "online with breakout" classification experiments on the digits dataset.

For most purposes, a mini-batch approach to NeuralNet training was good enough to generate acceptable results for the digits dataset, especially considering the runtime requirements of complete, multi-epoch online training runs. However, to squeeze out the last bit of classification performance in a reasonable amount of time, online training "with breakout" provided a good solution in terms of finding a sweet spot compromise of classification vs. runtime performance.

Challenges:

1. The first NeuralNet design used for this project was much more object-oriented. Abstractions were developed for different node types: InputNode, OutputNode, HiddenNode, and BiasNode. Separate abstractions for InputLayer, HiddenLayer, and OutputLayer were created, and an Edge type was also defined. Unfortunately, this made the *feedforward()* computations awkward and so the original design was abandoned for a simpler matrix manipulation approach.
2. At first, during development, training would not converge. No matter how many epochs were configured, total error would hover and remain around 50%. This was fixed by normalizing the input data, using *zscore()* from the StatsBase[7] package. To accommodate this, *prepdata()* had to be re-written to normalize the dataset as a whole, rather than row-by-row.
3. The fishing input data had to be encoded. A separate function *encfish()* (found in *ANNlib.jl*) was written to convert the fishing feature values into ones and zeroes.
4. The classes for both datasets had to be encoded. A "one-hot" encoding was used (function *onehot()*), whereby the class is indicated as a 1 in a specific position of a target values array; the other positions are zeroes. In fishing for example, with a ["NO" "YES"] target values array, No is encoded as [1 0], and Yes as [0 1]. For digits, with a [0 1 2 3 4 5 6 7 8 9] target values array, a three (3) is encoded as [0 0 0 1 0 0 0 0 0 0].
5. During early digits classification experiments, no matches were found. Zero. This was curious. Surely at least some training examples should match, no? While inspecting the debugging output, when comparing the predicted class to the true class, the true class column occasionally would show a numeral "10". Aha! Julia arrays are 1-indexed, not 0-indexed, but our digits target values array is effectively 0-indexed, so a quick adjustment in *classify()* eliminated this problem.
6. Runtime performance in general was acceptable, with each digits experiment completing in between 1-5 minutes. It took about 80 minutes to run all 40 experiments listed in Table 2. I spent some time thinking about how to vectorize every matrix operation to improve runtime, but given the time constraints I left this for another time. Other research indicates that a momentum factor could also speed up runtime processing. I will explore this in a future implementation.
7. I spent some time exploring an approach for multiple hidden layers, but I was constrained by time and was having difficulty making the calling interface easy to use. Should the network just "know" how many hidden layers are configured, or should the calling function have to specify it each time? This idea needs more design time.

References

[1] Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98, doi:10.1137/141000671, http://julialang.org/publications/julia-fresh-approach-BEKS.pdf

[2] Artificial neural networks. (n.d.). Retrieved 2017-04-04, from *Wikipedia*: https://en.wikipedia.org/wiki/Artificial_neural_network

[3] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. http://doi.org/10.1038/323533a0

[4] JuliaStats. (n.d.). *StatsFuns.jl, Mathematical functions related to statistics.* Retrieved 2017-04-04 from https://github.com/JuliaStats/StatsFuns.jl

[5] JuliaStats. (n.d.). *DataFrames.jl, Library for working with tabular data in Julia.* Retrieved 2017-04-04 from https://github.com/JuliaStats/DataFrames.jl

[6] LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K. R. (2012). Efficient backprop. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7700 LECTURE NO, 9–48. http://doi.org/10.1007/978-3-642-35289-8-3

[7] JuliaStats. (n.d.). *StatsBase.jl, Basic statistics for Julia.* Retrieved 2017-04-04 from https://github.com/JuliaStats/StatsBase.jl

v2.0 Changelog

Design Overview:

- trailing (!) removed from function names, as well as the footnote
- updated section on random number generation, added function *tinyrandom()*

Experiments:

- NeuralNet parameter lists for digits and fishing were switched
- "breakout" condition was added to *train()*
- "breakout" results presented in Table 3

Challenges:

- section #4 now mentions function *onehot()*
- removed section #8 on random number generation