Multi-Layer Perceptron

Christopher Parker
Biologically Inspired Computing
University of Oslo
Fall 2014

***Abstract*—I developed a single hidden layer perceptron classifier, hereafter referred to as a 'network', applied to control of a prosthesis. The input was provided as a time series of 10 samples of 4 sensors, yielding a 40-dimensional input vector. The output was 8 dimensional, and its argmax corresponded to the classification result. Test error as low as 2.7% was achieved in five minutes of training. This was comparable to results using reference implementations of the support vector classifiers in the Scikit-Learn package. Hill climbing was used for parameter tuning, though I discuss that this could have been significantly improved through genetic search.**

The relevant code & example output images may be found at: github.com/csp256/INF3490_Homework_2

I. Introduction

Only 447 labelled examples were available. Using a constant seed for reproducibility these examples were shuffled and divided into training, validation, and test sets with ratio 2:1:1. The seed was chosen so that each class had a roughly constant representation in each of the data sets. It was also found that simply not sorting the data was suitable to these purposes as well.

II. Linear Support Vector Classifier

Unfortunately this data set was extremely small and it was difficult to perform normal statistical analysis on the quality of my solution because virtually all errors were "one off" events. To get a baseline of performance I used Scikit-Learn, a popular Python machine learning package, to perform classification on the training data with support vectors with linear kernel. Training it on the training set only (not the validation set), and testing using the test set the following results were found:

		Result							
		0	1	2	3	4	5	6	7
Target	0	12							1
	1		15						
	2			13					
	3				12			1	
	4					15			
	5						13	1	
	6							14	
	7				1				13

$$P(correct) = 0.964 \tag{1}$$

$$min(P(correct|result)) = 0.875 \tag{2}$$

$$min(P(correct|target)) = 0.923 \tag{3}$$

III. Manifold Hypothesis

A perceptron of sufficiently many units can decide any linearly seperable function in any number of dimensions. It does this by performing a linear transformation of the input vector to the output vector. If the hidden layer of a multilayer perceptron had no nonlinearity it would not improve the generality of a perceptron: the result would still simply be a linear map of the inputs.

Adding a nonlinear activation function allows the network to learn any function, given sufficiently many units. However, the number of hidden units required grows as an exponential of the dimensionality of the space required to linearly seperate the input. For now we will ignore this problem (and significant, subsequent problems), by only considering networks of a single hidden layer.

The linearly seperable space is necessarily higher dimensional than the input space for nonlinear functions. The full multilevel perceptron is then a linear transformation of the input followed by a nonlinear map, and finally a linear transformation to the output. In the case of smooth nonlinear activation functions, such as a sigmoid, the result is a continuous, smooth, nonlinear map into a space where each class is linearly seperable from all other classes. This linearly seperable space is then a manifold. A step activation, or rectified linear activation, produces sharp folds in this map, and while this produces theoretical difficulties, they are often desirable in practice.

Technically, a final nonlinear mapping is applied to the output layer, but this is not necessary in generality.

IV. Batching & Low-Pass Filtering

The optimization of this map (the training stage) is an iterative, linear task: the local gradient of weights as a function of the error is computed and the units' weights (the linear portion of the total mapping) are updated so as to approach a local minima of the error space. Rather, this is what we wish would happen. Because sampling the error gradient must be done as a function of a training example, the gradient that is recovered is actually a projection of the error gradient along some axis dependent upon the map itself, the input vector, and its class. To offset this, the gradient is sampled with several training examples and their average is used to approximate the true error gradient. This is known as "batching".

The updates applied to each unit pass through a linear recursive low pass filter (an exponential filter), which is given by a free parameter. This parameter is on the interval of 0 (no filtering applied) to 1 (local gradient is ignored; update is constant). These updates are finally scaled by a free parameter η and applied.

The author stresses that this low pass filter is not analogous to ”momentum” or ”inertia”. As a pedagogical example, a constant nonzero error gradient would not cause update values to diverge, as a constant force on a free particle would cause its velocity to diverge.

V. Stopping Criterion

Unfortunately, simply optimizing the network for the training set is certain to result in a suboptimal network. As a trivial but provocative example, binary classification of points on the plane may result in a delta function projection into a third dimension on all training instances. This space fulfils the requirements of being linearly separable, and would have a 100% accuracy on the training data. This general phenomenon (including its less severe forms) is known as overfitting and results in a network that is unlikely to generalize well to new, previously unseen data (which is the objective of a classifier).

To minimize overfitting we use a separate set of validation data. It provides a ”sanity check” on how well the network can provide broad, general classification of unseen data. Training continues until the derivative of the error on the validation data becomes positive. A sixth-order accurate one-sided discrete first derivative of the validation data was used in my implementation instead of an explicit number of iterations.

A future implemention might store that state as a local optima, and exit once a superior local optima has not been found after a constant number of iterations.

A. Nonstandard Validation Error Metric

I noticed that even with long training times, low learning rates, and large batch sizes that some classes had a huge (often 100%) error rate. I addressed this by updating the validation error metric to be a weighted sum of the squares of not just $(1 - P(correct))$ but also $min(P(correct|result))$ & $min(P(correct|target))$. The initial, arbitrarily chosen, weights were 5, 1, & 1. This caused a huge increase in test accuracy, so they were kept.

Another potential approach would be too scale the target error before backpropogation to more strongly emphasize the classes with weak conditional accuracies. Either of these methods could be made adaptive, or parameterized in the meta-optimization heuristic.

VI. Parameter Optimization Through Hill Climbing

I originally used time-limited hill climbing on the curvature of the maps (the β_1 & β_2 parameters; where 1 pertains to the input to hidden weights, and 2 pertains to the hidden to output weights), the learning rates of each weight matrix (η_1 & η_2), the low pass filters, the batch size, and the random number seeds for the initial weights and the training process. New potential locations in the parameter space were chosen with a Boltzmann-like, unimodal, scale invariant distribution with the mean centered at the current location. When necessary, a transformation (magnitude, scaling to unit interval, converting to integer, etc) of this distribution was performed to extract that the actually used parameters. This distribution had the unintended side effect of driving the ’raw’ (untransformed) parameters to large magnitudes (a sample at $+1\sigma$ was further from the mean than a sample at -1σ), but the resulting networks performed well regardless so this was not corrected.

Sampling random seeds like this did not result in any meaningful correlation between points in the parameter space (due to the high entropy property of pseudo random number generators), and thus did not conform to the assumptions which empower hill climbing over random search. Accordingly, seeds were randomly initialized instead.

The other parameters were strongly linked. β was a factor of the interval of initial weights. Recognizing that each training point was inside a unit hypercube (with one vertex at the origin), a value of $\beta_i = 0.1$ was selected as a constant. This fairly small value was chosen so the nonlinear section of the manifold would be initially smooth over the region of likely initial values, so as to have minimal initial bias.

It was necessary to have unique values of every parameter for each weight matrix, due to the vanishing gradient problem. Accordingly, the hill climbing algorithm repeatedly elected to select $\beta_1 > \beta_2$, where the scale of the inequality (roughly a factor of 50 to 200) was roughly constant but strongly dependent upon the size of the hidden layer.

Low pass coefficients of near zero were often chosen by the algorithm. Optimization performed well with low pass filtering disabled altogether, however the hillclimbing algorithm consistently chose values of about 0.005.

Improvements on this technique would involve selection methodologies which do not bias towards large magnitudes, and allow for smooth sampling of initial parameters. A genetic search algorithm would be very interesting to investigate for optimizing the initial weights.

VII. Plotting

Plotting performance on the validation and test sets as a function of time was implemented, but disabled because there was no impetus to finish: early results simply show that classifier error follows a power law, and nothing more interesting.

VIII. Results

In the following, hill climbing was only performed on $beta_i$ and the low pass filter parameters. Results correspond to the network with lowest validation error (note, this is not equal to $1 - P(correct)$) found after 60 seconds of training. A probability of -1 corresponds to division by zero from no data being available: that class was never recognized correctly. This is considered a 100% error for purposes of my custom validation error metric calculation. Iterations refer to the number of hill climbing iterations, and $P(MoveAccept)$ indicates the portion of those which were successful.

A. j = 10, batchSize = 10

		Result							
		0	1	2	3	4	5	6	7
Target	0			9					4
	1		15						
	2			13					
	3			1		5			7
	4					15			
	5		12				2		
	6		11				1	2	
	7			2					12

$$P(correct) = 0.532 \tag{4}$$

$$min(P(correct|result)) = -1 \tag{5}$$

$$min(P(correct|target)) = 0.0 \tag{6}$$

$$Iterations = 546 \tag{7}$$

$$P(MoveAccept) = 0.0110 \tag{8}$$

$$BestError = 6.99 \tag{9}$$

$$eta1 = 12.8 \tag{10}$$

$$eta2 = 4.65 \tag{11}$$

$$lowPass1 = 0.00603 \tag{12}$$

$$lowPass2 = 0.0200 \tag{13}$$

B. j = 10, batchSize = 30

		Result							
		0	1	2	3	4	5	6	7
Target	0	12				1			
	1		14				1		
	2	1		11		1			
	3					11	2		
	4					15			
	5						13	1	
	6						1	4	
	7	13				1			

$$P(correct) = 0.622 \tag{14}$$

$$min(P(correct|result)) = -1 \tag{15}$$

$$min(P(correct|target)) = 0.0 \tag{16}$$

$$Iterations = 420 \tag{17}$$

$$P(MoveAccept) = 0.0190 \tag{18}$$

$$BestError = 6.84 \tag{19}$$

$$eta1 = 41.3 \tag{20}$$

$$eta2 = 4.64 \tag{21}$$

$$lowPass1 = 0.0212 \tag{22}$$

$$lowPass2 = 0.00243 \tag{23}$$

C. j = 10, batchSize = 100

		Result							
		0	1	2	3	4	5	6	7
Target	0	13							
	1		15						
	2	1		12					
	3					4			9
	4					15			
	5	4				2	8		
	6	1				2	10		1
	7	1							13

$$P(correct) = 0.685 \tag{24}$$

$$min(P(correct|result)) = -1 \tag{25}$$

$$min(P(correct|target)) = 0.0 \tag{26}$$

$$Iterations = 272 \tag{27}$$

$$P(MoveAccept) = 0.0221 \tag{28}$$

$$BestError = 5.87 \tag{29}$$

$$eta1 = 21.14 \tag{30}$$

$$eta2 = 4.388 \tag{31}$$

$$lowPass1 = 0.00379 \tag{32}$$

$$lowPass2 = 0.03 \tag{33}$$

D. j = 30, batchSize = 10

		Result							
		0	1	2	3	4	5	6	7
Target	0	9							4
	1		15						
	2			13					
	3			1	2	3		2	5
	4					15			
	5			2			12		
	6			2			4	8	
	7								14

$$P(correct) = 0.793 \tag{34}$$

$$min(P(correct|result)) = 0.609 \tag{35}$$

$$min(P(correct|target)) = 0.154 \tag{36}$$

$$Iterations = 545 \tag{37}$$

$$P(MoveAccept) = 0.00734 \tag{38}$$

$$BestError = 1.27 \tag{39}$$

$$eta1 = 52.3 \tag{40}$$

$$eta2 = 5.97 \tag{41}$$

$$lowPass1 = 0.00423 \tag{42}$$

$$lowPass2 = 0.03 \tag{43}$$

E. j = 30, batchSize = 30

		Result							
		0	1	2	3	4	5	6	7
Target	0	13							
	1		15						
	2	1		11		1			
	3				3	7		2	1
	4					15			
	5					1	9	4	
	6							14	
	7					1			13
	]								

$$P(correct) = 0.838 \tag{44}$$

$$min(P(correct|result)) = 0.6 \tag{45}$$

$$min(P(correct|target)) = 0.231 \tag{46}$$

$$Iterations = 566 \tag{47}$$

$$P(MoveAccept) = 0.00883 \tag{48}$$

$$BestError = 0.675 \tag{49}$$

$$eta1 = 36.8 \tag{50}$$

$$eta2 = 2.40 \tag{51}$$

$$lowPass1 = 0.0105 \tag{52}$$

$$lowPass2 = 0.00860 \tag{53}$$

F. j = 30, batchSize = 100

		Result							
		0	1	2	3	4	5	6	7
Target	0	12							1
	1		15						
	2			13					
	3			1	7			3	2
	4				2	12		1	
	5						12	2	
	6			1			2	11	
	7								14

$$P(correct) = 0.865 \tag{54}$$

$$min(P(correct|result)) = 0.647 \tag{55}$$

$$min(P(correct|target)) = 0.538 \tag{56}$$

$$Iterations = 388 \tag{57}$$

$$P(MoveAccept) = 0.0258 \tag{58}$$

$$BestError = 0.353 \tag{59}$$

$$eta1 = 47.8 \tag{60}$$

$$eta2 = 5.843 \tag{61}$$

$$lowPass1 = 0.0134 \tag{62}$$

$$lowPass2 = 0.0155 \tag{63}$$

G. j = 100, batchSize = 10

		Result							
		0	1	2	3	4	5	6	7
Target	0	13							
	1		15						
	2			13					
	3			1	12				
	4					15			
	5			1		1	12		
	6			2				12	
	7			1	1				12

$$P(correct) = 0.937 \tag{64}$$

$$min(P(correct|result)) = 0.722 \tag{65}$$

$$min(P(correct|target)) = 0.857 \tag{66}$$

$$Iterations = 622 \tag{67}$$

$$P(MoveAccept) = 0.0161 \tag{68}$$

$$BestError = 0.161 \tag{69}$$

$$eta1 = 92.2 \tag{70}$$

$$eta2 = 7.25 \tag{71}$$

$$lowPass1 = 0.00753 \tag{72}$$

$$lowPass2 = 0.03 \tag{73}$$

H. j = 100, batchSize = 30

		Result							
		0	1	2	3	4	5	6	7
Target	0	13							
	1		15						
	2			13					
	3				10	1		2	
	4					15			
	5						11	3	
	6							14	
	7		1		3				10

$$P(correct) = 0.910 \tag{74}$$

$$min(P(correct|result)) = 0.737 \tag{75}$$

$$min(P(correct|target)) = 0.714 \tag{76}$$

$$Iterations = 386 \tag{77}$$

$$P(MoveAccept) = 0.0130 \tag{78}$$

$$BestError = 0.158 \tag{79}$$

$$eta1 = 29.4 \tag{80}$$

$$eta2 = 1.37 \tag{81}$$

$$lowPass1 = 0.00731 \tag{82}$$

$$lowPass2 = 0.00786 \tag{83}$$

I. j = 100, batchSize = 100

		Result							
		0	1	2	3	4	5	6	7
Target	0	12		1					
	1		15						
	2			13					
	3			1	6	4		2	
	4					15			
	5						14		
	6						2	12	
	7			1	1				12

$$P(correct) = 0.892 \tag{84}$$

$$min(P(correct|result)) = 0.789 \tag{85}$$

$$min(P(correct|target)) = 0.462 \tag{86}$$

$$Iterations = 342 \tag{87}$$

$$P(MoveAccept) = 0.0234 \tag{88}$$

$$BestError = 0.268 \tag{89}$$

$$eta1 = 12.8 \tag{90}$$

$$eta2 = 1.88 \tag{91}$$

$$lowPass1 = 0.00876 \tag{92}$$

$$lowPass2 = 0.0252 \tag{93}$$

IX. CONCLUSION

The network, unsurprisingly, performed better with more hidden units and moderate batch sizes (30-50). This trend was present in other configurations tested (not listed here). The tendency for offdiagonal terms in the confusion matrix to appear in the 6 column was high, even when the data was shuffled before being divided into training, validation, and test sets. This might imply that the 6 class is too similar to other classes.

It is impossible to draw strong conclusions with only 447 data points. As is, 2% error is likely unacceptable for practical use. Errors of both kinds are strongly prevalent: a class may be consistently mislabeled, and a specific label may be applied too zealously. Interestingly, no strong confusion patterns appear (except that mentioned above) after repeated training suggesting that the model can be accurately fitted with enough training time. Furthermore, there are time series specific methods in neural networks (such as deep recurrent restricticted Boltzmann machines) which maybe be more capable of handling this sort of method, as they can hold state.

While training the network is computationally expensive, thankfully the computational power of a quadcore laptop is not necessary to run the network forward (that is, to use it to classify). Though in application some type of online learning (with a user selectable penalty for poor behavior) may be desirable.

The author also notes that using the classifier is embarassingly parallel, and the slower clock speeds permitted by parallel architectures result in lower power consumption, an important consideration for prosthesis.