Programming Assignment #1 : Building an ANN using Backpropagation and experimenting with multiple tools

In this assignment you will build software for an Artificial Neural Network using the Vectorization facilities of Python and many of the techniques that you have learnt so far. The core backpropagation equations in operational form are given in slides 20, and 25-27 of the provided powerpoint file. You will do so by the following steps:

a. Listing out the step-by-step backpropagation algorithm, with the equations and control actions of the program (for your benefit)

b. You will first train and validate your ANN on the toy problem of learning the mathematical function y = sin(x), where $-2\pi \le x \le 2\pi$. Unless you can complete the toy problem successfully, do not get into the more complex regression functions mentioned below. The steps of this toy problem are the following:

  b1. Extract 1000 (x, y) pairs equally distributed within the domain $-2\pi \le x \le 2\pi$. The total domain is 4π, split it equally into 4 parts and extract 250 points from each, again with equal intervals. Use this for training your ANN – note that you have only 1 input and 1 output

  b2. Extract 300 points randomly within the same range $-2\pi \le x \le 2\pi$., using some uniform-distribution random-number generator in Python (Google it). Note you are creating only the x-values; your ANN will generate the y-values as outputs. This will be the validation data

  b3. Plot the x-y curves extracted from the data in b1 on the same plot extracted from the outputs of your ANN in b2. *The two plots should be almost sitting on each other, for proving correctness of your ANN*

  b4. Use each of the techniques mentioned below under items 2-10. For #5, use only tanh(.). For #8, do not use any Regularization.

c. You will train, validate and test your ANN on regression data of a *Combined Cycle Power Plant Dataset* given in http://archive.ics.uci.edu/ml/datasets/Combined+Cycle+Power+Plant which belongs to the UCI Machine Learning Data Repository. The given data consists of more than 9 thousand rows of 4 input variables and 1 output variable.

In the dataset the variables are well defined and you don't really need any knowledge of the underlying domain physics; just be aware that the relationships between the inputs and outputs are quite complicated; hence successful (i.e. highly accurate) acquisition of the same by your ANN will demonstrate the capability of these machine learning mechanisms to capture complicated functional relationships that characterize a process and are encapsulated within data generated by the running process. Building up the software from scratch (except for available Python libraries like numpy, matplotlib, etc., do not use Deep Learning libraries for this assignment) to productively-operating levels will also give you the confidence of having mastered the nuts and bolts of this complex machine learning mechanism; in future you will be using high-level libraries but not as a semi-ignoramus just using these like black boxes. Importantly, if your self-built software is indeed generating good (i.e. accurate) results at efficient speeds – it is like your property which you can use for multiple applications in diverse domains!

Regarding each of the crucial ANN development aspects, you are to broadly follow the approaches outlined below:

1. Data Splitting:

Ideally, split the available data into 72 : 18 : 10 for training : validation : testing. While your training and validation data should be interspersed all across your given data set, your test data should be one or two continuous chunks from one or the other end of your data set. Your program should be written such that every epoch of training is followed by a run over all the validation data samples so that you get both the training error and validation error at every epoch. You may wonder why the numbers 72 : 18 – reason is that you should first pull out the 10% (testing) chunk, and what remains you can easily split into 80 : 20, i.e. create a loop where index divisible by 5 is sent to "validation data" while all else is sent to "training data".

2. ANN architecture:

Apart from the fact that the number of nodes in the input layer equals the number of input variables, and correspondingly for output layer, and that there has to be at least one hidden layer, the rest is your creation. Just remember one rule of thumb, the number of unknowns should not be more than half the number of training data samples. So one hidden layer will have some number of neurons, and if you increase the number of hidden layers then the nodes per layer will reduce by the above rule.

3. Back-propagation equations:

As given in equations (A – F). The operative aspects of these should be clear by now. Definitely use vectorization operations as they easily reduce computation times by two orders of magnitude. You can take a call on whether you want to do away with a for loop over the samples by using eqs. (F) and (E1) with complete vectorization, or continue to use a for loop to traverse through the samples using partial vectorization as in eqs. (D)-(E). Ideally, do both in two alternate programs, and then check the timings and accuracies for the same input data and network architecture.

4. Granulation of training data:

Use mini-batches between sizes 64 and 256. However, at extreme ends, you should test with batch sizes of 1 (effectively SGD) and the full batch. Also, shuffle the order of presentation of mini-batches across epochs. *Observe the impact of mini-batch sizes on training and validation convergence histories*. Take the four sizes 1, 64, 256 and then full batch.

5. Activation functions:

Use *tanh*, *logistic* and *ReLU*. *Start with tanh, and then experiment with the others*. Note that the *output layer* activations should not be *ReLU*, it has to be either of the other two, and preferably only *logistic*. Ensure that all nodes in a layer are using the same activation function.

6. I/O Normalization:

Definitely normalize inputs and outputs between -1 and +1. If using *logistic* function at output nodes, then the outputs have to scaled between 0 and 1. Follow the matter on slide #39 as closely as possible. Errors or misjudgement in normalization and denormalization (at output nodes for test cases when you convert from range 0 to 1 to actual dimensions of the variable) can severely impact your program. The easiest way to normalize is to arrange each variable's data in ascending order, evaluate the min. and max values, and then map these values to about 10% above -1 and 10% below +1. An equation you can use for linear normalization from actual range to -1 and +1:

$$X = \frac{2x - (x_{max} + x_{min})}{(x_{max} - x_{min})}$$

Note that *x* and *X* represent actual and normalized values. You will need to work out how to map actual minimum to -0.9 instead of to -1, and actual maximum to +0.9 instead of +1. That is a small exercise you have to do. A practical way is to create a virtual x_{min} 5% of the original range below the actual x_{min}, and a virtual x_{max} 5% of the original range above the actual x_{max}, and then use these virtual x_{min} and x_{max} in the above equation in place of the actual ones.

7. Weight Initialization:

Closely follow the matter in slides 33 and 34. Capping the abs values of weights at a level (say +1) is a basic but crude form of regularization which must be implemented even if you are unable to implement L2 Regularization (below).

8. Learning rate parameter and L2 Regularization:

Take η as 0.001. Refer slide 36 and eq. (17R), and the discussion that follows immediately. This is easy to implement, the complexity comes from the value of λ and its coupling with the value of η. Very difficult to provide a ball-park figure. Easier thing to do is to freeze all other parameters, and then generate a series of solutions with the value of λ increasing from 0, then 0.1 to 0.95 in three steps, and then plot the convergence histories (error values vs. epochs (could be semi-log plots of error)) for each case. Such a combined plot will give you insight into the impact of increasing bias (high regularization) on the solution.

Independently, you can freeze λ at 0, *and then see the impact of using η = 0.01, and 0.0001, on convergence histories*.

9. Momentum term:

Use eq. (K) in slide 46, with β as 0.9. When using L2 regularization, note that the second term of (K) is actually the eqn. (17R) on slide 36, rather than the eq. (17) which is shown for simplicity.

10. Stopping of training and final error calculation of test data:

The importance of concurrently observing both the training and validation convergence histories has been stressed before and is necessary for critically analysing your ANN performance. *This is also important for stopping of training at the lowest level of overfitting*. The method is explained using realistic data in slide 59. Since the data set considered here is realistic noisy data, the approach should work.

After observing the stoppage point as described in the slide, you should run once again with the maximum number of epochs set at the stopping value. You should write the computed weights at the end of training into a file in your hard disk properly maintaining the multiple layer and then matrix structure. When running test data, you should read from this file and perform only the forward calculations, and then extract the rms error of this test data, between actual and predicted outputs. So your code should be able to run in both modes – forward-plus-backward-training mode, and forward-only mode.

For calculating errors in prediction on validation or test data, use the Mean Absolute Percentage Error (MAPE) with the formula $MAPE = \frac{1}{N}\sum_{j=1}^{N}\frac{|y_j - d_j|}{|d_j| + \varepsilon}\times 100$, where ε is a small +(ve) number like (say) 0.001, d_j and y_j represent sample output and ANN output for sample *j*, taken over all validation and test data samples, separately. Hence you shall have a $MAPE_{val}$ and a $MAPE_{test}$.

Your submission should be a folder containing your code, and a word doc (PDF) containing your step-by-step algorithm, detailed comparisons of the impact of different parametric variations stated in italics in items # 2, 4, 5, 8 and 10. You should also explicitly state the best combination of parameters that work on each data set, and the corresponding *MAPE's*.

<u>Note</u>:

1. Do not try out Batch Normalization. Do not try out more than 3 *hidden* layers.
2. For a good software which will stand the test of time, all these experimentations should be performable using hyper-parameters that the program reads at start of execution. You should not have to change code internally for each of the above experimentations.
3. Run the code on the DGX-1 machine, rather than your laptops. Try to use the GPUs for parallelism. Check out speed-ups you may have obtained when compared to serial runs.

<u>Bonus</u>:

1. Use Adam optimization over and above SGD-with-momentum. Compare the convergence histories.
   (10% above baseline marks).