Project 2:
Neural Networks

Girish K

March 24, 2016

1 Goal

The goal of this project is to implement and test a multi-layer neural network, tune its parameters, and to compare its performance with of the decision treat created in the previous project

2 Datasets

The datasets selected from the UC Irvine Machine Learning Repository for this project are, *Car, Breast Cancer, Pima Indians Diabetes, Phising, Iris, and Mushroom.*

2.1 Preprocessing

This section is the same as that for the decision tree. The preprocessing module accepts a control file that has information about the dataset location, class and attribute column numbers, names and types(discrete or continuous). The control file also has information such as separator character used in the raw data, and supports other necessary metadata we might require in future. It then converts the raw input into the format $< dataset, labels >$, and creates a metadata object that is visible throughout the program.

2.1.1 Missing Values

Missing values are replaced by the mode of the distribution in case of discrete attributes, and by the mean of the distribution in the case of continuous attributes.

2.1.2 Control Files

The control files for the datasets mentioned in this report, have already been generated in the pre-processing phase. Refer to `control.json` within the respective data folders for the template of control files. We can also add more information such as distribution statistics, thresholds, and additional data parsing parameters. The mandatory parameters in the control file include `class name, class position, data location, attribute types and attribute names`

2.2 Data Normalization

A neural network does not differentiate between continuous and discrete data, and only sees an input layer of numbers. Therefore, we need to apply certain transformations to both kinds of data and bring them to the same scale, as described in Yann Lecun's paper on efficient back propagation.

2.2.1 Numeric Attributes

We use standard z-score normalization for numeric attributes, i.e $z = \frac{(x-\mu)}{\sigma}$. Although normalization isn't required as we can expect the weights to scale themselves, the following explanation justifies the need for it.

> If the input variables are combined linearly, as in an MLP, then it is rarely strictly necessary to standardize the inputs, at least in theory. The reason is that any rescaling of an input vector can be effectively undone by changing the corresponding weights and biases, leaving you with the exact same outputs as you had before. However, there are a variety of practical reasons why standardizing the inputs can make training faster and reduce the chances of getting stuck in local optima. Also, weight decay and Bayesian estimation can be done more conveniently with standardized inputs.
>
> Source : http://www.faqs.org/faqs/ai-faq/neural-nets/part2/

2.3 Discrete Attributes

For discrete attributes, assigning numbers to each attribute may work in case of binary attributes. However, this approach fails when we have more than 2 possible values, and the neural network cannot learn the concept.

To prevent this, discrete attributes are transformed using the procedure *One-Hot Encoding*. In this method, every feature is transformed into a sparse vector representation, of size equal to that of the attribute's domain. Each feature in this new vector is a bit, and it is turned on only for the bit position corresponding to the value of the original feature. The ordering of these bits must be kept constant for all inputs to the neural network.

For example, if $f_1 = cold$, and domain for f_1 are $hot, moderate, cold$, the one-hot encoded feature will transform it into 3 new features : $< 0, 0, 1 >$

This normalization has to be done for the class value also, and therefore gives us an output vector of the same size as class domain.

3 Theory

The back-propagation algorithm implemented in this project primarily follows the derivation and back-propagation approach given in the following source.

`https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf`

I have used the following notations and formulae in my implementation.

Unit Error for output layer j:

$$\delta_j = (t_i - y_i)y_i(1 - y_i)$$

Unit Error for hidden layer j:

$$\delta_j = (\sum_{iinI_j} \delta_i w_{ji}) y_i (1 - y_i)$$

Weight Update on n^{th} iteration:

$$\Delta w_{kj}(n) = \alpha \delta_j y_k + \eta \Delta w_{kj}(n-1)$$

The notations are explained in the following section on the componenets of the neural network.

4 Neural Network Construction

4.1 Class Definition

The members of the neural network class are :

Layers : Stores the number of nodes in each layer, including bias node for each layer. eg. [3,4,5] represents a neural network with a single hidden layer containing 4 nodes.

Activations y_j : One NumPy vector for each layer, which holds the current activations for the neural network. The input layer's activations are equal to the feature vector that is supplied to the neural network, and its output layer activations correspond to predictions. The activation of the bias nodes is always 1.

Weights w_{kj} : Stores the weights for the neural network as a list of NumPy matrices, one matrix between every two adjacent layers. The dimensions of a weight matrix from a layer k to a layer j, w_{kj} is given by $n_k \times (n_j - 1)$, where n_k and n_j represent the number of nodes in the respective layers. The -1 is because there are weights *from* a bias unit, but no weights *into* it.

Unit Errors δ_j : A NumPy vector for each layer, which stores the back propagated unit errors for each node. These errors are propagated from the output layer, backwards upto the second layer. The bias units do not have unit error, as this error is never propagated (since the bias nodes are meant only to regularize or penalize the outputs due to weights going out from a specific layer). They are computed using first two equations shown in Section 3.

Delta Δw_{ij} : A list of NumPy matrices with same dimensions as weights. It holds updates to each weight as computed using the last equation in Section 3

Previous Delta $\Delta w_{ij}^{(n-1)}$: The previous Δw_{ij}, used for calculating momentum in next weight update

Learning Rate α : The rate at which weight updates are made with respect to backpropagated errors at each iteration.

Momentum Rate η : The factor by which the previous update influences the current update, to ensure the descent is moving in the same direction

4.2 Initializing the neural network

The constructor for the neural network object takes in 3 arguments - layer specification, learning rate, and momentum rate. It also initializes all the weights randomly using `randn` function from NumPy.

5 Implementation Details

5.1 Modules

1. Preprocess : Contains preprocessing and normalizing routines
2. Statistics : Contains functions to compute error statistics
3. Neural_Network : Contains class definition for neural network, which includes feed forward and backpropagataion functions for the neural network.
4. Driver : Core module that invokes preprocessing and normalization, splits data into 10 folds, and runs the neural network and decision tree for each fold, and reports the final results.

5.2 Training and Validation

The neural network is trained with the algorithm shown in the following psuedocode:

```
def train_neural(neural_net, trainX, trainY, validX, validY):
    max_iterations = 20000
    mse_threshold = 0.025
    i = 0
    epoch_size = len(validX)
    while(i < max_iterations):
        instance_ip, instance_op =  pick_random(trainX, trainY)
        neural.net.feedforward(instance_ip)

        #The following 3 functions are encapsulated within the backpropagation routine
        neural_net.compute_unit_errors(instance_ip, instance_op)
        neural_net.compute_deltas()
        neural_net.update weights()

        if i % epoch_size == 0:
            mse = getMSE(neural_net, validX, validY)

        if mse < threshold:
            break
```

5.2.1 Mean Squarerd Error

The `getMSE` function in the above code makes use of the follow function:

$$\theta_{mse} = \frac{1}{n}\sum_{1}^{n}\sum_{j\in op}\frac{1}{2}(t_j - y_j)^2$$

5.3 Testing

To find the final accuracy on test set, we feed forward each example in the neural network, and choose the class corresponding to the output neuron with the highest activation value, i.e *argmax* of the output vector.

```
def test_neural(neural_net, testX, testY):
    correct_predictions = 0
    for test_exampleX, test_exampleY in zip(testX, testY):
        neural_net.feedforward(test_exampleX)
        target_class = argmax(test_exampleY)
        pred_class = argmax(neural_net.outputlayer())
        if target_class == pred_class:
            correct_predictions ++


    return correct_predictions/len(testX)
```

5.4 Design Aspects

5.4.1 Stochastic Gradient Descent

This implementation uses Stochastic Gradient Descent, i.e picks a random example from the training set at every iteration, as this is computationally faster, and in turns allows us to train for a large number of iterations. The stochastic approach also has the advantage of introducing some form of randomness, which prevents getting stuck in local optima.

5.4.2 Cross Validation as Stopping Criteria

I tried the following 3 approaches of cross validation as a stopping criteria.

1. Train until MSE starts to increase on the validation set. This approach proved to be too greedy, and the training terminated within 10 iterations in some cases, and yielded poor results.

2. Monitor mean-squared error(MSE) on validation set after every iteration, i.e for every chosen random training example, until the MSE falls below a pre-determined threshold. For datasets, where 30% of the data amounted to almost 4000 examples, this resulted in that many feed forward operations. This came upto almost 10000 matrix multiplications on each iteration, which proved to be unnecessarily complex, and computationally restrictive just for validation purpose. The training was almost 10,000 slower than the naive case where we don't use a validation set.

3. Introduce the concept of epochs, where each epoch consists of many iterations. Here, we monitor the MSE only at the end of every epoch. The size of the validation set is a good approximation of the number of iterations in each epoch. This way, we ensure that over-fitting doesn't occur, and at the same time, keeping the computational complexity low. The epoch size can be further tuned to suit different datasets. Another good approach is to use a smaller validation set size.

This implementation uses approach 3, as it was fast and accurate.

5.4.3 Stopping Criteria

1. Number of Iterations > Iteration Limit (default : 20000 iterations)
2. Error on validation set < Threshold Value (default : 0.025)

5.4.4 Integration of Decision Tree and Neural Network

The decision tree from the previous project is implemented here, so as to compare the performance of the two algorithms on the same folds.

5.4.5 Learning Rate

The choice of learning rate is extremely important. With a learning rate of 0.0001, the decrease in MSE (i.e the descent) was too slow to converage in reasonable time. With a learning rate of 1.0, the algorithm tends to make a very step, and gets stuck in local minima. On the Iris dataset, it was observed that the MSE reached about 0.37 very quickly, and then remained constant. 0.1 proved to be a good learning rate for most of the datasets.

5.4.6 Momentum Rate

A higher momentum rate like 0.9 was found to make the convergence slower. Lowering the momentum to 0.01 made the training faster, and did not compromise on final test accuracy. Therefore, we cannot assertively say anything about this rate without testing on larger datasets and using statistically rigorous comparisons.

6 Discussion of Results

The results of the experiments are included in as separate file, named *Results.pdf.*

6.1 Neural Network V.S Decision Tree

Iris Both algorithms perform almost the same. The class boundary is well defined for this dataset, and test accuracy is high even if MSE doesn't fall below threshold value.

Mushroom The decision tree learns the concept better than the neural network, which seems to make a small chunk of mistakes all the time. This could be because the neural network tries to tune the parameters to accommodate noisy examples, while the decision tree simply assigns such examples a majority class label.

Breast Cancer The neural network performs marginally better

Pima Indians Both algorithms perform equally poorly, which shows that the dataset itself is noisy, and does not have a well defined concept for all examples

Car The decision tree performs marginally better

Phishing The pruned decision tree times out with this dataset (due to having over 100 nodes in the tree). The neural network however, is able to converge to a reasonably low MSE value, and yields good accuracy.

6.2 Choice of Hidden Layer Specification

The second part of the experiment illustrates the use of different hidden layer combinations to find out which combination yields the best results. While the results are not entirely conclusive on what the best combination would be, there are some observations we can make by observing the gradient descent values (displayed while running the program).

- Without a hidden layer, the algorithm fails drastically, and is unable to converge
- Increasing the number of nodes in a layer, up to a certain level, helps the algorithm converge much faster(in less iterations). However, increasing it too much results in the gradient descent move down the wrong surface, and get stuck at local minima. It was observed during experimentation that, in certain folds, MSE got stuck at about 0.3 for 200000 iterations, while for some folds, the MSE fell from 0.5 to 0.01 in less than 1000 iterations. These kind of results, as shown below, are extremely dangerous.

```
Fold 1
MSE on Epoch 0 : 0.464144
MSE on Epoch 1 : 0.011701
Fold 2
MSE on Epoch 0 : 0.482943
MSE on Epoch 1 : 0.280521
MSE on Epoch 2 : 0.268447
MSE on Epoch 3 : 0.264751
MSE on Epoch 4 : 0.264245
MSE on Epoch 5 : 0.264647
MSE on Epoch 6 : 0.263899
MSE on Epoch 7 : 0.263795
MSE on Epoch 8 : 0.263582
MSE on Epoch 9 : 0.261503
```

- Simpler networks, while more time consuming, do not exhibit this characteristic of getting stuck, and tend to gradually converge.
- A small number of layers, such as 2 or 3 is often sufficient to handle the fairly simple datasets using in this project. A largern number of layers makes the network overly complex, and it fails to converge.

7 Thoughts and Analysis

- Setting epoch size equal to that of validation set is an adaptive choice. For larger datasets, the probability of the training set covering the target concept is high, and therefore validation can be done periodically to prevent overfitting. If the data is small (eg. Iris), chance of overfitting is high, and validation is done frequently to make the algorithm stops beyond a threshold.

- The paired T-test is more intuitive and statistically rigorous, as it compares performance of the classifiers on the same folds. The t-scores show that the decision tree and neural network perform almost the same, although the difference is statistically significant in some cases.

- We cannot assume any performance guarantees about the network configuration based on a single execution. Even though we are using 10-fold validation, the performances were found to vary drastically across multiple executions with the same configuration. This occurred primarily due to the algorithm getting stuck in one of the folds, in the case of more complex neural networks. The best result from the Mushroom dataset, using 2 hidden layers of 100 neurons each, seems to be best choice, but on successive runs, it failed to replicate the same performance, and was stuck on a few folds.

- The MSE threshold (θ_{mse}) and maximum limit on iterations are closely tied together, since both are used as stopping criteria (although the iteration limit exists more as practical concern than theoretical one for this project). If the MSE threshold is too high, the iteration limit becomes pointless, as the algorithm will converge very quickly. If the MSE is too low, the iteration limit dominates, and the training stops before convergence. In this case, the entire validation set is wasted. In most of our experiments, even with a limit of 20000 iterations, the training could not converge on some datasets. In all these cases, the use of the validation set proved to be pointless.

- With θ_{mse} set to 0.01, the algorithm barely converged within 20000 iterations, which meant the cross validation wasn't actually serving any purpose. Increasing it marginally, to 0.025, ensured the algorithm converged on most datasets, within 20000 iterations!

- Using adaptive formulae to compute MSE threshold and iteration limit for each dataset, is a good approach to solve this tradeoff, and requires further mathematical analysis of the results.

- Learning rate must not be fixed for all datasets, and should ideally be adaptive, to ensure that the gradient descent makes rapid steps intially down the surface, and then slows down over time, so as to not jump past the global minima. Some of the commonly used technqiues are annealing and local rate adaptions.

- From the observations, I think it might be a better idea to use the cross-validation set to learn model paramters ($\eta, \alpha, \theta_{mse}, iteration\_limit$). Our practical limit on number of iterations often seems to contradict the use of the mean squared error.