Comparing Random Forests and Feedforward Neural Networks in sentiment analysis

Kenneth Tor Blomqvist

kenneth.blomqvist@aalto.fi

Abstract

This paper compares random forests and feedforward neural networks for classification and regression. An experimental evaluation is done on a dataset of yelp reviews. A theoretical overview of the two methods are given. The implementation of a feedforward neural network is explained. The best performing method for both classification and regression was the feedforward neural network with a small margin.

I. Introduction

Neural networks are all the rage with the kids these days. News articles are constantly published about how neural nets are being applied with staggering results to image recognition, speech recognition, image synthesis, pattern recognition and other problems.

This paper does not mean to provide an in-depth analysis of the best possible approach for a given method. Instead this papers seeks to compare the performance characteristics and ease of use of a simple feed-forward neural network implementation to random forests. We conduct experiments against a dataset containing yelp reviews for a classification and a regression task.

Our dataset consists of yelp reviews of which the count of 50 different words in the review are given without any contextual information. The regression task consists of predicting how many people found a review useful based on the labeled training data. The classification task consists of trying to predict wether the review received over 3 ″stars″ from other users of the service.

Experiments are conducted against these tasks to compare the performance characteristics and other properties of these two very different methods.

This paper is laid out in several sections. In section II a general overview of the compared methods is given. Section III describes the experiments performed and section IV presents the results of the experiments. Section V gives a short discussion of the results.

II. Methods

I. Feedforward Neural Networks

Feedforward neural networks are a non-parametric estimator that can be used for classification and regression [1]. They hey are formed by composing functions connected together by weight matrices. The activation functions are considered to be layers. Each layers output is passed to the next one thus forming a network. The activation functions can be thought of as ″neurons″ and the matrices as connections between them. This is why they are called neural networks. [2]

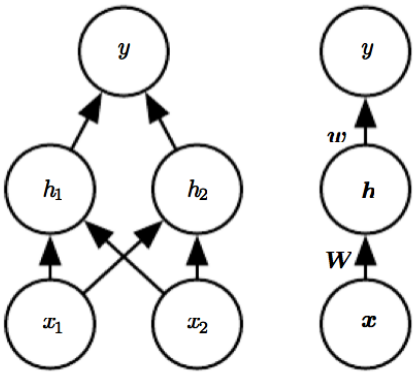

Figure 1: *A depiction of a feedforward neural network drawn in two different styles. The circles are neurons and the lines between them connections between them. x and y are the input and output layers respectively. [2]*

Neural networks can be used to approximate functions such as probability distributions. However, as opposed to many linear models, neural networks can not be trained by using a closed form optimization method. The way they are trained is that a cost function is defined as the optimization goal. The cost is then minimized using gradient based learning. The backpropagation algorithm is used to update the weights and biases of the model to minimize the cost. [2]

For neural networks, an online learning approach is typically used. Training examples are given in small batches which are evaluated against the cost function. The weights and biases are adjusted to minimize the cost function. Training in this fashion means that the model does not have to store all of the training data in external memory. This makes neural networks well suited for problems with a lot of data. [2]

In this paper we examine perhaps the most simple form of a feedforward neural network: a network with only fully connected layers. Fully connected layers are layers where the input is multiplied by a weight matrix, a bias term is added and the result is passed to some activation function. Typical activation functions include the sigmoid, tanh and restricted linear unit functions.

For classification we create a network with n neurons at the output layer where n is the amount of output classes. These neurons represent each of the output classes. At the output layer a softmax activation function is typically used. The softmax function outputs a n-dimensional vector that sums to one yielding what can be interpreted as a normalized probability distribution over the output classes. Categorical cross-entropy is used as the cost function to optimize the network. [5]

For regression tasks linear activation functions are typically used and the output values are summed to yield the predicted value [3]. Mean squared error is often used to minimize the error of the network.

Neural networks are very versatile and can be used for a wide variety of problems. They also work very well in high dimensional settings. They are however computationally expensive to train. They also have quite a lot of parameters that can be tuned. This can make finding a good set of parameters time consuming. Grid search and other search techniques can however be used to help in the search [2].

II. Random Forests

Decision trees are a parametric method for supervised learning [1]. The hypothesis is represented using a binary tree where at each node a binary decision is made. The training examples at the leaf nodes are used to make a prediction. For classification the most common training label at a leaf node represents the output class and for regression the labels for the training data are averaged to get the prediction.

Decision trees are learned by recursively expanding the tree using one of many learning algorithms available.

Random forests are an averaging based ensemble method specifically designed for decision trees [4]. Several models are trained by sampling with replacement from the training set. The predictions of each model is averaged to construct the final predictions. This way we can use several individual models with high variance and use averaging to reduce the overall variance. Decision trees typically can have

high variance suffering from overfitting [1].

III. Experiments

In the experiments we studied two different tasks. Classifiying yelp reviews into two classes: those that received over 3 stars and those that didn't. In the regression task we try to predict how many people find the review useful.

The datasets both consist of 50 different features for each datapoint. Each feature indicates how many occurences of a word where found in the review text. The data consists of 5,000 training examples and a disjoint test set of 1,000 test examples.

I. The Neural Network

The neural network was trained by using stochastic gradient descent and the backpropagation algorithm to tune the weights and biases. The validation dataset is split off from the training data and it's size is 1/10th of all the available training data.

For optimization the RMSProp algorithm was implemented as presented in [2]. Theano [5] was used to implement the computational graph and to derive the gradients. The initial weights of each fully connected layer were initialized randomly sampled from a Gaussian distribution with a mean of 0.0 and and a variance 0.25 as suggested in Deep Learning [2].

For the regression task we used mean squared error as a cost function to minimize the error in our network. For the classification task categorical cross entropy was used.

The network was trained until the classification accuracy no longer improved on the validation set for more than 5 passes over the training dataset. The network was then evaluated on the test dataset and the test accuracy logged.

Since there is a vast amount of different hyperparameters available for tuning, the model was tuned by hand. The network parameters were adjusted after each run. Different networks depths and widths where tried along with tanh, sigmoid and rectified linear unit activation functions. Several l2 norm regularization weights were tried. Many different learning rates ranging from 0.1 to 0.0001 were tried.

The full implementation of the neural network can be found on github at `https://github.com/kekeblom/mlbp-project`. The implementation of the RMSProp algorithm is included in appendice A VI.

II. The Random Forests

For both tasks the models were tuned by running a grid search to find the best possible model parameters. The parameters that where tuned were the minimum samples left at a leaf of a tree and the maximum depth of each individual tree. The values for the minimum samples at a leaf node ranged from 1 to 10 with a step of 1. The values for the maximum depth ranged from 1 to 48 with a step of 3.

In our experiments we used the scikit-learn random forest implementation [4]. The k-fold cross-validation implementation was also from the scikit-learn library.

At each step of the search, the model was evaluated using k-fold cross validation with 5 folds. The random forest that achieved the best k-fold validation accuracy was used to evaluate the model on the test dataset.

IV. Results

The results for the classification experiments can be seen in table 1 and the results for the regression task in 2.

Method	**Accuracy [%]**
Neural Network	72.0
Random Forest	71.4

Table 1: *Results for the classification task*

Method	Mean squared error
Neural Network	0.043
Random Forest	0.129

Table 2: *Results for the regression task*

In the classification task the neural network achieved 72.0% classification accuracy on the test set with 19 hidden neurons with sigmoid activation functions and 2 softmax output units. The learning rate was set to 0.001 and the regularization weight to 1.0. The random forest achieved 71.4% test accuracy with 100 estimators, the maximum depth set to 40 and the minimum samples at a leaf node set to 100 estimators.

In the regression task the neural network achieved a mean squared error of 0.043 on the test set using 19 hidden neurons, 10 output neurons, rectified linear units at both layers and a learning rate of 0.0005. The l2 regularization weight was set to 0.5. The random forest achieved a mean squared error of 0.129 with a maximum tree depth of 49 and with 1 minimum of samples at a leaf node.

V. Discussion

We found that random forests and neural networks achieve similar performance on a small, relatively high dimensional dataset such as the one used in this paper. The neural network performed slightly better on both tasks. The neural network turned out to perform slightly better on both tasks.

Despite the fact that the neural network achieved good results on both tasks, random forests might still be a better option in many cases. Random forests turned out to be much easier to use and a well perfoming model was found very quickly. Random forests are computationally faster to train which means that more variations could be tried faster.

Different options for the count of the random forest estimators in the models were not tried. This could be an interesting experiment to run to see wether better results could be achieved by adding more estimators. Other parameters could also have been included in the search, such as the maximum number of samples at a leaf node or different node splitting criteria. Some feature selection scheme could have been tried as decision trees are known to not perform very well in high dimensional settings as they use a heuristic search for growing the tree to fit the dataset.

Some metrics could have been measured such as how long it takes to evaluate the trained model. This could be an interesting comparison as it could be critical for some applications.

Better results and less biased results could have been achieved using a grid search for finding the neural network parameters, but since there were so many parameters to be tuned and a lack of time, the decision was made to hand tune the parameters.

Implementing the neural network model and conducting both experiments was a good way to learn more about the ergonomics and performance characteristics of these machine learning methods.

References

[1] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.

[2] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," 2016, book in preparation for MIT Press. [Online]. Available: http://www.deeplearningbook.org

[3] "Using neural networks for regression," 2016. [Online]. Available: https://deeplearning4j.org/linear-regression

[4] "Scikit-learn: Machine learning in python," 2016. [Online]. Available: http://scikit-learn.org/stable/

[5] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

VI. Appendix A: Implementation of the RMSProp algorithm

This appendice contains the code for the RMSProp algorithm. The full code can be found at `https://github.com/kekeblom/mlbp-project`.

```
    def updates(self, cost_function, parameters, minibatch_size):
        shapes = [param.get_value().shape for param in parameters]
        accumulators = [theano.shared(np.zeros(shape)) for shape in shapes]
        gradients = self.get_gradients(cost_function, parameters, minibatch_size)
        updates = []
        for parameter, gradient, accumulator in zip(parameters, gradients, accumulators):
            new_accumulator = self.rho * accumulator + (1. - self.rho) * (gradient**2)
            updates.append((accumulator, new_accumulator))

            new_parameter = parameter - self.learning_rate * (
                    gradient / (np.sqrt(new_accumulator) + self.epsilon)
                )

            updates.append((parameter, new_parameter))

        return updates
```

VII. Appendix B: Implementation of stochastic gradient descent

This appendice contains part of the code for running stochastic gradient descent for the neural network.

```
    def minibatch_SGD(self, training_data):
        training_data = training_data.shuffle()
        batch_index = T.lscalar()

        minibatch_size = self.minibatch_size

        updates = self.optimizer.updates(self.cost, self.get_params(), minibatch_size)

        training_x = theano.shared(training_data.x)
        training_y = theano.shared(training_data.y)

        self.train = theano.function([batch_index], self.cost, updates=updates,
            givens={
                self.x: training_x[batch_index * minibatch_size:(batch_index+1) * minibatch_size],
                self.y: training_y[batch_index * minibatch_size:(batch_index+1) * minibatch_size]
            })

        training_batch_count = int(len(training_data.x) / self.minibatch_size)
        costs = []
        for batch in range(0, training_batch_count):
            cost = self.train(batch)
            costs.append(np.mean(cost))

        return costs
```

VIII. Appendix C: Implementation of the fully connected layer

This appendice contains the code for the fully connected neural network layer implementation.

```
import numpy as np
import theano
from theano import tensor

class FullyConnected(object):
    def __init__(self, input_dimensions=None, output_dimensions=None, activation=tensor.nnet.relu):
        self.input_dimensions = input_dimensions
        self.output_dimensions = output_dimensions

        initial_weights = np.random.normal(loc=0.0, scale=0.25, size=(input_dimensions, output_dimensions))

        biases = np.zeros((output_dimensions), dtype=theano.config.floatX)

        self.weights = theano.shared(
                value=initial_weights.astype(theano.config.floatX),
                name='W', borrow=True)
        self.biases = theano.shared(value=biases, name='b', borrow=True)
        self.activation = activation

    def build(self, input, batch_size):
        reshaped = input.reshape((batch_size, self.input_dimensions))
        output = self.activation(tensor.dot(reshaped, self.weights) + self.biases)
        return output

    def get_params(self):
```

```
    return [self.weights, self.biases]

def get_weights(self):
    return [self.weights]

def accuracy(self, labels, output):
    return tensor.mean(tensor.eq(labels, tensor.argmax(output, axis=1)))
```