**Elements of Statistical Learning**

**Chapter 11 – Neural Networks**

**11.1 Introduction**

The central idea of neural networks is to first derive features from the original input space, specifically as linear combination of the original inputs. Once these derived features are in place, the neural net models the response variables as a non-linear function of these features.

**11.2 Projection Pursuit Regression**

A similar, but not identical, model to neural nets is projection pursuit regression (PPR). PPR estimation is somewhat more straightforward so that is why we start here.

PPR is an additive model, but using the derived features as opposed to the raw inputs themselves. The derived features are found through applying a series of functions, , to the raw inputs. These can be thought of as smoothing functions that have been discussed in topics like ridge regression (in fact, the functions themselves are referred to as *ridge* functions). Also, these functions are unknown prior to model fit, and estimated during the model fit process; the number of functions is determined through a cross-validation or forward stagewise strategy until the model fit cannot be significantly improved.

Similarly, there is another set of unknown parameters, , which, in essence, is a weight applied to each raw input variable to create our derived features that end up going into our PPR model. The is found by applying the aforementioned series of functions to the raw inputs. Once we have found the vector of to be applied to each predictor variable, we can then find our derived features by combining and the raw input. Summing up the derived features of all predictors in our model, we can then estimate our target through the additive model eluded to above:

It is called a *projection pursuit regression* because what we are trying to find is the projection of the raw input matrix onto the vector so that the model fits well.

Although interpretation of such models are difficult, they do well in situations where all that is needed is prediction as opposed to a model used to understand the data. Because of this, we can think of PPR as a *universal approximator* since it can approximate any continuous function relatively well.

*11.2.1 Estimating the PPR models*

The goal for a given set of data is to minimize the error function:

Because both and are found during the model fitting process, we need to find the model that minimizes the above function across all possible and . Additionally, to avoid overfitting, we need to impose complexity constraints on the functions .

The order of estimating the PPR model is to first find the smoothing functions , then once these are determined, use the Gauss-Newton method. Although I won’t go into much detail, the Gauss-Newton method can be used to solve non-linear least squares problems (e.g., PPR). It can only be used to minimize a sum of squared function values, but it has the advantage that second derivatives – which can be challenging to compute – are not required.

**11.3 Neural Networks**

*Additional supplement material for these notes come from:* [*http://natureofcode.com/book/chapter-10-neural-networks/*](http://natureofcode.com/book/chapter-10-neural-networks/)

Neural nets are one type of the broader class nonlinear statistical models. They originated as a way to describe the human brain, and in order to understand neural networks it is important to understand how they were applied as models to the human brain.

The human brain can be described as a biological neural network, an interconnected web of neurons that transmit signals to one another, which ultimately lead to that human making some sort of decision (and subsequently acting upon that decision).

Unlike other types of statistical models, neural networks don’t process information in a linear fashion. In other words, they don’t examine the values of each inputs individually; neural nets process information/data collectively, in parallel with one another.

Also unlike most other types of statistical models discussed previously, the neural network has the ability to adapt and change its internal structure based on the information that flows through it. Think of a child learning not to touch a stove because he adapted after touching a hot stove. This is also called *reinforcement learning*, which contrasts to supervised and unsupervised learning with which we’re already familiar.

The simplest form of a neural network is called the single hidden layer back-propagation network, or single layer perceptron. It follows the so-called “feed-forward” model, in that inputs are sent through the model, processed, and then finally resulting in an output. How neural nets are processed are similar to the PPR; weights are applied to values of the raw inputs, resulting in derived features, and then these derived features are summed.

A difference is that this resulting sum goes through an *activation function* to determine what the interpretable prediction will be for a given observation.

For regression, there is simply one output unit that has a value in the same units as the response variable (after going through the activation function).

For -class classification, there are typically output nodes, with each output node being the probability that a given observation is a member each class. The probabilities for a single observation across all nodes sum up to 1.

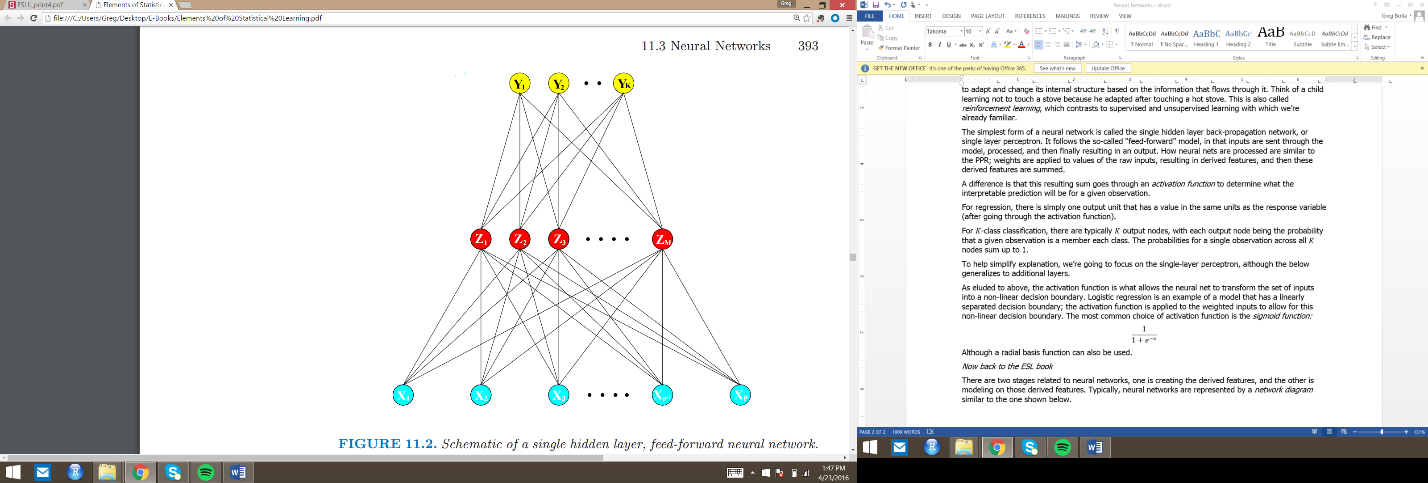
To help simplify explanation, we’re going to focus on the single-layer perceptron, although the below generalizes to additional layers.

As eluded to above, the activation function is what allows the neural net to transform the set of inputs into a non-linear decision boundary. Logistic regression is an example of a model that has a linearly separated decision boundary; the activation function is applied to the weighted inputs to allow for this non-linear decision boundary. The most common choice of activation function is the *sigmoid function:*

Although a radial basis function can also be used.

*Now back to the ESL book*

There are two stages related to neural networks, one is creating the derived features, and the other is modeling on those derived features. Typically, neural networks are represented by a *network diagram* similar to the one shown below.

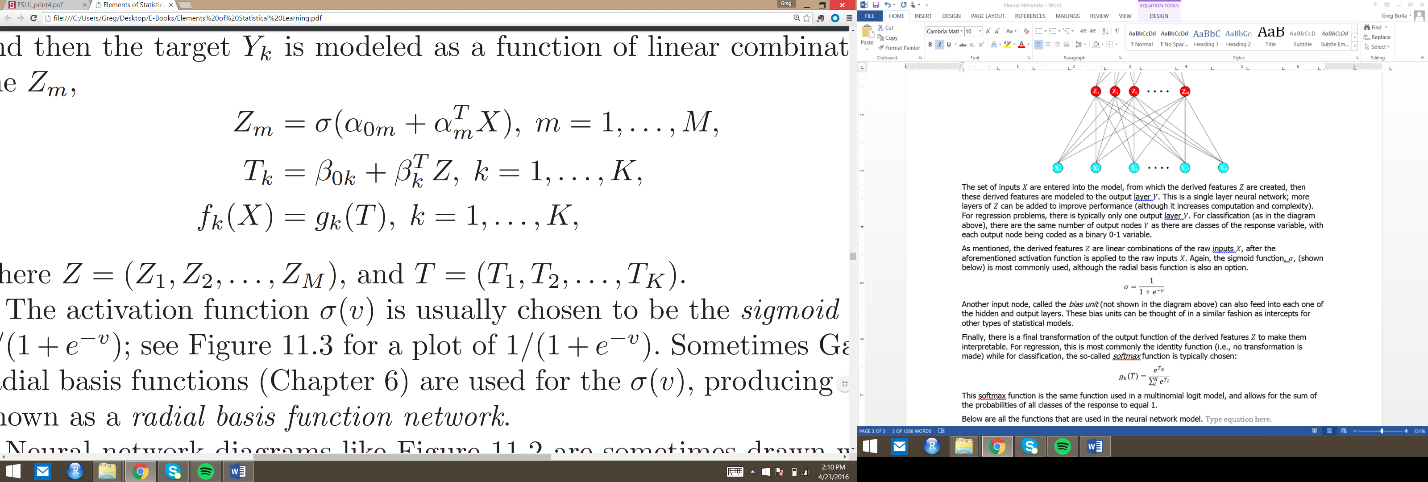
The set of inputs are entered into the model, from which the derived features are created, then these derived features are modeled to the output layer . This is a single layer neural network; more layers of can be added to improve performance (although it increases computation and complexity). For regression problems, there is typically only one output layer . For classification (as in the diagram above), there are the same number of output nodes as there are classes of the response variable, with each output node being coded as a binary 0-1 variable.

As mentioned, the derived features are linear combinations of the raw inputs , after the aforementioned activation function is applied to the raw inputs . Again, the sigmoid function, , (shown again below) is most commonly used, although the radial basis function is also an option.

Another input node, called the *bias unit* (not shown in the diagram above) can also feed into each one of the hidden and output layers. These bias units can be thought of in a similar fashion as intercepts for other types of statistical models.

Finally, there is a final transformation of the output function of the derived features to make them interpretable. For regression, this is most commonly the identity function (i.e., no transformation is made) while for classification, the so-called *softmax* function is typically chosen:

This softmax function is the same function used in a multinomial logit model, and allows for the sum of the probabilities of all classes of the response to equal 1.

Below are all the functions that are used in the neural network model. refers to how the derived features are, well, derived. is the function employed to get from the derived features to the output layer. Finally, is the function applied to the output layer to get it into an interpretable format.

The derived features are known as hidden units because they’re not directly observable. Additionally, you can think of the function that takes the raw inputs and outputs them into the derived features as basis functions which we’ve seen in ensemble methods like random forests.

When these basis functions are the identity function, neural networks are simply linear models. However, there is a nuance to the basis functions that are unique to neural networks. In other statistical models that make use of basis functions, the parameters of the basis functions are known before fitting the model. This is not the case with neural networks. The basis functions that take the raw inputs into are learned from the data. Because of this, neural networks can be thought of as a nonlinear expansion of a linear model.

**11.4 Fitting Neural Networks**

We seek values of the weights of each input and hidden node that make the neural network model fit well.

weights, applied to raw inputs

weights, applied to derived inputs from hidden layer

To determine which model fits is best, we use sum-of-squared error for regression and cross-entropy (aka deviance) for classification. The equation shown below is for cross entropy, with representing the total error of the given model with the weights applied to both input and hidden layers.

Note that with the softmax activation function and cross-entropy error function, the neural network is the exact same as a linear logistic regression model in the hidden units, with parameters estimated by maximum likelihood.

However, we shy away from the global minimizer because this is likely to yield an overfit solution. Therefore, we implement some kind of regularization via a penalty term or early stopping (more on this in a later section).

Generally, the approach to minimize is through gradient descent, called *back-propagation* in the neural network setting. I’m going to keep this at a high level because this gets into some heavy mathematics with derivatives involved. However, the gradient can be derived using something called the chain rule for differentiation, computed by a forward, and subsequent backward, sweep of the network, keeping track only of the quantities local to each unit.

At a high level, here are the steps that the computer undergoes:

1. Find the derivatives of the error function (the book uses the SSE function on page 396).
2. Given those derivatives, at each iterative step of the model fitting process, a gradient descent update is implemented, using something called the *learning rate*, represented by
3. From there, we have enough information to calculate the errors of the *forward pass* over the network, represented by in the book.
   * The errors from this forward pass are also called the *back-propagation equations* because they are used to inform how the model fits itself in the backwards pass through the network
4. The errors of the backwards pass (represented by ) are then computed and subsequently back-propagated to give the errors from the forward pass

The advantage of this procedure includes its local nature, in that each hidden unit passes and receives only information to and from other nodes with which a connection is shared; therefore it can more efficiently implements in a parallel architecture.

The previous few paragraphs assume that the parameters updates occur across all training cases. Model fitting can also take place one observation at a time, updating the gradient at each case, and cycling through the training set many times.

There is also the concept of learning rate which determines when the neural net will complete the fitting process. Because the ANN applies weights to the various layers, the learning rate applies a greater or lesser adjustment to the weights of the previous iteration. A larger value for the learning rate will allow the algorithm to arrive at a solution more quickly; whether or not it fits well with a larger learning rate is another issue altogether. Usually, it is better to set the learning rate to a small value and edge upward if the neural net fits slowly.

**11.5 Some Issues in Training Neural Networks**

*Side note: This website could provide some useful tips on the interworkings of neural networks:* [*http://www.cheshireeng.com/Neuralyst/nnbg.htm*](http://www.cheshireeng.com/Neuralyst/nnbg.htm)

Because of the complexity inherent in neural networks and its associated parameters, the book goes into detail about an assortment of things to keep in mind when fitting neural net models.

*11.5.1 Starting Values*

As the weights associated with the various layers of the neural network are near zero, then the sigmoid is roughly linear and therefore the network collapses into an approximately linear model. Usually, starting weights near zero are chosen before the first iteration, then the algorithm adds weight (and introduces more nonlinearity) as needed.

*11.5.2 Overfitting*

Because of the plethora of weights employed by the neural network algorithm, the global minimum of the error function often overfit the data. There are a couple different ways to counteract this that have been developed.

First was the early stopping rule eluded to above. You simply train the model for a certain period of time, and stop training before it hits the global minima. This has the effect of training a model that is more linear and regularized.

Another more explicit form of regularization is the concept of *weight decay*, which is analogous to ridge regression for linear models. It is simply a penalty term added to the error function:

In the equation above, is the adjustable penalty parameter, with calculated as a sum of the squared weights:

Larger values of will tend to shrink the weights to zero; typically cross-validation is used to arrive at .

Although the above is one way to calculate the penalty term, there have been others proposed, including below which has the effect of shrinking smaller weights more so than the above:

*11.5.3 Scaling of the Inputs*

Since the scales of the inputs determines the scale of each inputs associated weights in the bottom layer, it is best to standardize all inputs to have mean zero with standard deviation of one, thus ensuring all inputs are treated equally in the regularization process.

*11.5.4 Number of Hidden Units and Layers*

The optimal choice of the number of hidden layers is guided by background knowledge and experimentation. However, generally speaking, it is better to have too many hidden units than too few. This is especially true when regularization techniques are employed; the extra weights can simply be shrunk toward zero.

Typically, ANNs are trained with anywhere between 5 and 100 hidden units, with the number increasing as the number of inputs and training cases increase.

*11.5.5 Multiple Minima*

The error function is nonconvex and possesses many local minima. Because of this, the final model fit depends heavily on the (random) choice of starting weights.

At the very least, you must try a series of starting configurations and choose the solution that gives the lowest penalized error.

Another, probably better, solution is to implement an ensemble method that averages the predictions over a collection of neural networks as the final prediction.

Finally, another approach is through bagging, averaging the predictions from the ANNs from randomly perturbed versions of the training set.

**11.6 Example: Simulated Data**

The book goes into detail about some fake data they ran and how the neural network models performed relative to the Bayes error rate. One thing to note is that neural networks that employed the sum of sigmoid activation function performed much, much better than the associated models with the radial basis activation function.

One thing to call out from this section is that there are two free parameters that analysts must select to fit a neural net model – the amount of weight decay and the number of hidden units of the network, . To quote directly from the book:

*As a learning strategy, one could fix either parameter at the value corresponding to the least constrained model, to ensure that the model is rich enough, and use cross-validation to choose the other parameter.*

I’m not exactly sure what is meant by the “least constrained model”, however it appears the recommended strategy is to thoughtfully select one of those two parameters, then use cross-validation to choose the other (given the value of the first parameter).

**11.8 Discussion**

Both PPR and neural networks take nonlinear functions of linear combinations (referred to as “derived features” above) of the input variables. This approach has shown to work well for many different types of problems.

These approaches are tailor made for in problems with a high signal-to-noise ratio, and settings where prediction without interpretation is the goal. Conversely, if one of the goals of the model is to describe the roles of individual inputs, ANNs may not be the optimal choice.

**11.9 Bayesian Neural Nets and the NIPS 2003 Challenge**

The authors go into detail about a data science competition where two data scientists used a Bayesian approach to neural networks, using a Markov Chain Monte Carlo method to sample from the posterior distribution of a binary response. The theory behind the success of a Bayesian neural network is that the MCMC approach provides an efficient way of exploring the important parts of the parameter space, and subsequently averaging the predictions of the models that use the parameter space that is more likely to be appear in actual data.

A bit more digging is needed for me to really understand how Bayesian inference is actually applied to neural nets, but it is possible to do so.