Probabilistic Networks In ANNs

Deepak Garg, Amardeep Singh
Computer Science & Engineering Department,
Thapar Institute of Engineering & technology
{dgarg,asingh} @mail.tiet.ac.in

Abstract

In the context of classification problems, a useful interpretation of network outputs was as estimates of probability of class membership, in which case the network was actually learning to estimate a probability density function (p.d.f.). A similar useful interpretation can be made in regression problems if the output of the network is regarded as the expected value of the model at a given point in input-space. This expected value is related to the joint probability density function of the output and inputs.

1. Introduction

In the context of a classification problem, if we can construct estimates of the p.d.f.s of the possible classes, we can compare the probabilities of the various classes, and select the most-probable. This is effectively what we ask a neural network to do when it learns a classification problem - the network attempts to learn (an approximation to) the p.d.f.

A more traditional approach is to construct an estimate of the p.d.f. from the data. The most traditional technique is to assume a certain form for the p.d.f. (typically, that it is a normal distribution), and then to estimate the model parameters. The normal distribution is commonly used as the model parameters (mean and standard deviation) can be estimated using analytical techniques. The problem is that the assumption of normality is often not justified.

An alternative approach to p.d.f. estimation is *kernel-based approximation*. We can reason loosely that the presence of particular case indicates some probability density at that point: a cluster of cases close together indicate an area of high probability density. Close to a case, we can have high confidence in some probability density, with a lesser and diminishing level as we move away. In kernel-based estimation, simple functions are located at each available case, and added together to estimate the overall p.d.f. Typically, the kernel functions are each Gaussians (bell-shapes). If sufficient training points are available, this will indeed yield an arbitrarily good approximation to the true p.d.f.

This kernel-based approach to p.d.f. approximation is very similar to radial basis function networks, and motivates the probabilistic neural network (PNN) and generalized regression neural network (GRNN), both devised by Speckt (1990 and 1991). PNNs are designed for classification tasks, and GRNNs for regression. These two types of network are really kernel-based approximation methods cast in the form of neural networks.

In the PNN, there are at least three layers: input, radial, and output layers. The radial units are copied directly from the training data, one per case. Each models a Gaussian function centered at the training case. There is one output unit per class. Each is connected to all the radial units belonging to its class, with zero connections from all other radial units. Hence, the output units simply add up the responses of the units belonging to their own class. The outputs are each proportional to the kernel-based estimates of the p.d.f.s of the various classes, and by normalizing these to sum to 1.0 estimates of class probability are produced.

The basic PNN can be modified in two ways.

First, the basic approach assumes that the proportional representation of classes in the training data matches the actual representation in the population being modeled (the so-called prior probabilities). For example, in a disease-diagnosis network, if 2% of the population has the disease, then 2% of the training cases should be positives. If the prior probability is different from the level of representation in the training cases, then the network's estimate will be invalid. To compensate for this, prior probabilities can be given (if known), and the class weightings are adjusted to compensate.

Second, any network making estimates based on a noisy function will inevitably produce some misclassifications (there may be disease victims whose tests come out normal, for example). However, some forms of misclassification may be regarded as more expensive mistakes than others (for example, diagnosing somebody healthy as having a disease, which simply leads to exploratory surgery may be inconvenient but not life-threatening; whereas failing to spot somebody who is suffering from disease may lead to premature death). In such cases, the raw probabilities generated by the network can be weighted by loss factors, which reflect the costs of misclassification. A fourth layer can be specified in PNNs which includes a loss matrix. This is multiplied by the probability estimates in the third layer, and the class with lowest estimated cost is selected. (Loss matrices may also be attached to other types of classification network).

The only control factor that needs to be selected for probabilistic neural network training is the smoothing factor (i.e., the radial deviation of the Gaussian functions). As with RBF networks, this factor needs to be selected to cause a reasonable amount of overlap - too small deviations cause a very spiky approximation which cannot generalize, too large deviations smooth out detail. An appropriate figure is easily chosen by experiment, by selecting a number which produces a low selection error, and fortunately PNNs are not too sensitive to the precise choice of smoothing factor.

The greatest advantages of PNNs are the fact that the output is probabilistic (which makes interpretation of output easy), and the training speed. Training a PNN actually consists mostly of copying training cases into the network, and so is as close to instantaneous as can be expected.

The greatest disadvantage is network size: a PNN network actually contains the entire set of training cases, and is therefore space-consuming and slow to execute.

PNNs are particularly useful for prototyping experiments (for example, when deciding which input parameters to use), as the short training time allows a great number of tests to be conducted in a short period of time.

2. Generalized Regression Neural Networks

Generalized regression neural networks (GRNNs) work in a similar fashion to PNNs, but perform regression rather than classification tasks. As with the PNN, Gaussian kernel functions are located at each training case. Each case can be regarded, in this case, as evidence that the response surface is a given height at that point in input space, with progressively decaying evidence in the immediate vicinity. The GRNN copies the training cases into the network to be used to estimate the response on new points. The output is estimated using a weighted average of the outputs of the training cases, where the weighting is related to the distance of the point from the point being estimated (so that points nearby contribute most heavily to the estimate).

The first hidden layer in the GRNN contains the radial units. A second hidden layer contains units that help to estimate the weighted average. This is a specialized procedure. Each output has a special unit assigned in this layer that forms the weighted sum for the corresponding output. To get the weighted average from the weighted sum, the weighted sum must be divided through by the sum of the weighting factors. A single special unit in the second layer calculates the latter value. The output layer then performs the actual divisions (using special division units). Hence, the second hidden layer always has exactly one more unit than the output layer. In regression problems, typically only a single output is estimated, and so the second hidden layer usually has two units.

The GRNN can be modified by assigning radial units that represent clusters rather than each individual training case: this reduces the size of the network and increases execution speed. Centers can be assigned using any appropriate algorithm (i.e., sub-sampling, *K*-means or Kohonen).

GRNNs have advantages and disadvantages broadly similar to PNNs - the difference being that GRNNs can only be used for regression problems, whereas PNNs are used for classification problems. A GRNN trains almost instantly, but tends to be large and slow (although, unlike PNNs, it is not necessary to have one radial unit for each training case, the number still needs to be large). Like an RBF network, a GRNN does not extrapolate.

3. Linear Networks

A general scientific principal is that a simple model should always be chosen in preference to a complex model if the latter does not fit the data better. In terms of function approximation, the simplest model is the linear model, where the fitted function is a hyperplane. In classification, the hyperplane is positioned to divide the two classes (a linear discriminant function); in regression, it is positioned to pass through the data. A linear model is typically represented using an *NxN* matrix and an *Nx1* bias vector.

A neural network with no hidden layers, and an output with dot product synaptic function and identity activation function, actually implements a linear model. The weights correspond to the matrix, and the thresholds to the bias vector. When the network is executed, it effectively multiplies the input by the weights matrix then adds the bias vector.

The linear network provides a good benchmark against which to compare the performance of your neural networks. It is quite possible that a problem that is thought to be highly complex can actually be solved as well by linear techniques as by neural networks. If you have only a small number of training cases, you are probably anyway not justified in using a more complex model.

4. SOFM Networks

Self Organizing Feature Map (SOFM, or Kohonen) networks are used quite differently to the other networks. Whereas all the other networks are designed for supervised learning tasks, SOFM networks are designed primarily for *unsupervised learning* .

Whereas in supervised learning the training data set contains cases featuring input variables together with the associated outputs (and the network must infer a mapping from the inputs to the outputs), in unsupervised learning the training data set contains only input variables.

At first glance this may seem strange. Without outputs, what can the network learn? The answer is that the SOFM network attempts to learn the structure of the data.

One possible use is therefore in exploratory data analysis. The SOFM network can learn to recognize clusters of data, and can also relate similar classes to each other. The user can build up an understanding of the data, which is used to refine the network. As classes of data are recognized, they can be labeled, so that the network becomes capable of classification tasks. SOFM networks can also be used for classification when output classes are immediately available - the advantage in this case is their ability to highlight similarities between classes.

A second possible use is in novelty detection. SOFM networks can learn to recognize clusters in the training data, and respond to it. If new data, unlike previous cases, is encountered, the network fails to recognize it and this indicates novelty.

A SOFM network has only two layers: the input layer, and an output layer of radial units (also known as the *topological map* layer). The units in the topological map layer are laid out in space - typically in two dimensions (although *ST Neural Networks* also supports one-dimensional Kohonen networks).

SOFM networks are trained using an iterative algorithm. Starting with an initially-random set of radial centers, the algorithm gradually adjusts them to reflect the clustering of the training data. At one level, this compares with the sub-sampling and *K*-Means algorithms used to assign centers in RBF and GRNN networks, and indeed the SOFM algorithm can be used to assign centers for these types of networks. However, the algorithm also acts on a different level.

The iterative training procedure also arranges the network so that units representing centers close together in the input space are also situated close together on the topological map. You can think of the network's topological layer as a crude two-dimensional grid, which must be folded and distorted into the N-dimensional input space, so as to preserve as far as possible the original structure. Clearly any attempt to represent an N-dimensional space in two dimensions will result in loss of detail; however, the technique can be worthwhile in allowing the user to visualize data which might otherwise be impossible to understand.

The basic iterative Kohonen algorithm simply runs through a number of epochs, on each epoch executing each training case and applying the following algorithm:

- Select the winning neuron (the one who's center is nearest to the input case);
- Adjust the winning neuron to be more like the input case (a weighted sum of the old neuron center and the training case).

The algorithm uses a time-decaying learning rate, which is used to perform the weighted sum and ensures that the alterations become more subtle as the epochs pass. This ensures that the centers settle down to a compromise representation of the cases which cause that neuron to win.

The topological ordering property is achieved by adding the concept of a neighborhood to the algorithm. The neighborhood is a set of neurons surrounding the winning neuron. The neighborhood, like the learning rate, decays over time, so that initially quite a large number of neurons belong to the neighborhood (perhaps almost the entire topological map); in the latter stages the neighborhood will be zero (i.e., consists solely of the winning neuron itself). In the Kohonen algorithm, the adjustment of neurons is actually applied not just to the winning neuron, but to all the members of the current neighborhood.

The effect of this neighborhood update is that initially quite large areas of the network are "dragged towards" training cases - and dragged quite substantially. The network develops a crude topological ordering, with similar cases activating clumps of neurons in the topological map. As epochs pass the learning rate and neighborhood both decrease, so that finer distinctions within areas of the map can be drawn, ultimately resulting in fine-tuning of individual neurons. Often, training is deliberately conducted in two distinct phases: a relatively short phase with high learning rates and neighborhood, and a long phase with low learning rate and zero or near-zero neighborhood.

Once the network has been trained to recognize structure in the data, it can be used as a visualization tool to examine the data. The *Win Frequencies Datasheet* (counts of the number of times each neuron wins when training cases are executed) can be examined to see if distinct clusters have formed on the map. Individual cases are executed and the topological map observed, to see if some meaning can be assigned to the clusters (this usually involves referring back to the original application area, so that the relationship between clustered cases can be established). Once clusters are identified, neurons in the topological map are labeled to indicate their meaning (sometimes individual cases may be labeled, too). Once the topological map has been built up in this way, new cases can be submitted to the network. If the winning neuron has been labeled with a class name, the network can perform classification. If not, the network is regarded as undecided.

SOFM networks also make use of the accept threshold, when performing classification. Since the activation level of a neuron in a SOFM network is the distance of the neuron from the input case, the accept threshold acts as a maximum recognized distance. If the activation of the winning neuron is greater than this distance, the SOFM network is regarded as undecided. Thus, by labeling all neurons and setting the accept threshold appropriately, a SOFM network can act as a novelty detector (it reports undecided only if the input case is sufficiently dissimilar to all radial units).

SOFM networks are inspired by some known properties of the brain. The cerebral cortex is actually a large flat sheet (about 0.5m squared; it is folded up into the familiar convoluted shape only for convenience in fitting into the skull!) with known topological properties (for example, the area corresponding to the hand is next to the arm, and a distorted human frame can be topologically mapped out in two dimensions on its surface).

5. Classification in ST Neural Networks

In classification problems, the purpose of the network is to assign each case to one of a number of classes (or, more generally, to estimate the probability of membership of the case in each class). Nominal output variables are used to indicate a classification problem. The nominal values correspond to the various classes.

Nominal variables are normally represented in networks using one of two techniques, the first of which is only available for two-state variables; these techniques are: *two-state*, *one-of-N*. In two-state representation, a single node corresponds to the variable, and a value of 0.0 is interpreted as one state, and a value of 1.0 as the other. In *one-of-N* encoding, one unit is allocated for each state, with a particular state represented by 1.0 on that particular unit, and 0.0 on the others.

Input nominal variables are easily converted using the above methods, both during training and during execution. Target outputs for units corresponding to nominal variables are also easily determined during training. However, more effort is required to determine the output class assigned by a network during execution.

The output units each have continuous activation values between 0.0 and 1.0. In order to definitely assign a class from the outputs, the network must decide if the outputs are reasonably close to 0.0 and 1.0. If they are not, the class is regarded as undecided.

Confidence levels (the accept and reject thresholds) decide how to interpret the network outputs. These thresholds can be adjusted to make the network more or less fussy about when to assign a classification. The interpretation differs slightly for *two-state* and *one-of-N* representation:

Two-state. If the unit output is above the accept threshold, the 1.0 class is deemed to be chosen. If the output is below the reject threshold, the 0.0 class is chosen. If the output is between the two thresholds, the class is undecided.

One-of-N. A class is selected if the corresponding output unit is above the accept threshold and all the other output units are below the reject threshold. If this condition is not met, the class is undecided.

For one-of-N encoding, the use of thresholds is optional. If not used, the "winner-takes-all" algorithm is used (the highest activation unit gives the class, and the network is never undecided). There is one peculiarity when dealing with *one-of-N* encoding. On first reading, you might expect a network with accept and reject thresholds set to 0.5 is equivalent to a "winner takes all" network. Actually, this is not the case for *one-of-N* encoded networks (it is the case for two-state). You can actually set the accept threshold lower than the reject threshold, and only a network with accept 0.0 and reject 1.0 is equivalent to a winner-takes-all network. This is true since the algorithm for assigning a class is actually:

- Select the unit with the highest output. If this unit has output greater than or equal to the accept threshold, and all other units have output less than the reject threshold, assign the class represented by that unit.

With an accept threshold of 0.0, the winning unit is bound to be accepted, and with a reject threshold of 1.0, none of the other units can possibly be rejected, so the algorithm reduces to a simple selection of the winning unit. In contrast, if both accept and reject are set to 0.5, the network may return undecided (if the winner is below 0.5, or any of the losers are above 0.5).

Although this concept takes some getting used to, it does allow you to set some subtle conditions. For example, accept/reject 0.3/0.7 can be read as: "select the class using the winning unit, provided it has an output level at least 0.3, and none of the other units have activation above 0.7" - in other words, the winner must show some significant level of activation, and the losers mustn't, for a decision to be reached.

If the network's output unit activations are probabilities, the range of possible output patterns is of course restricted, as they must sum to 1.0. In that case, winner-takes-all is equivalent to setting accept and reject both to 1/N, where N is the number of classes. The above discussion covers the assignment of classifications in most types of network: MLPs, RBFs, linear and Cluster. However, SOFM networks work quite differently.

In a SOFM network, the winning node in the topological map (output) layer is the one with the lowest activation level (which measures the distance of the input case from the point stored by the unit). Some or all of the units in the topological map may be labeled, indicating an output class. If the distance is small enough, then the case is assigned to the class (if one is given). The accept threshold indicates the largest distance which will result in a positive classification. If an input case is further than this distance away from the winning unit, or if the winning unit is unlabelled (or its label doesn't match one of the output variable's nominal values) then the case is unclassified. The reject threshold is not used SOFM networks.

The discussion on non-SOFM networks has assumed that a positive classification is indicated by a figure close to 1.0, and a negative classification by a figure close to 0.0. This is true if the logistic output activation function is used, and is convenient as probabilities range from 0.0 to 1.0. However, in some circumstances a different range may be used. Also, sometimes ordering is reversed, with smaller outputs indicating higher confidence.

First, the range values used are actually the min/mean and max/SD values stored for each variable. With a logistic output activation function, the default values 0.0 and 1.0 are fine. Some authors actually recommend using the hyperbolic tangent activation function, which has the range (-1.0,+1.0) . Training performance may be enhanced because this function (unlike the logistic function) is symmetrical. Alternatively (and we recommend this practice) use hyperbolic tangent activation function in hidden layers, but not in the output layer.

Ordering is typically reversed in two situations. We have just discussed one of these: SOFM networks, where the output is a distance measure, with a small value indicating greater confidence. The same is true in the closely-related Cluster networks. The second circumstance is the use of a loss matrix (which may be added at creation time to PNNs, and also manually joined to other types of network). When a loss matrix is used, the network outputs indicate the expected cost if each class is selected, and the objective is to select the class with the lowest cost. In this case, we would normally expect the accept threshold to be smaller than the reject threshold.

Conclusion

Estimating probability density functions from data has a long statistical history, and in this context fits into the area of Bayesian statistics. Conventional statistics can, given a known model, inform us what the chances of certain outcomes are (e.g., we know that a unbiased die has a 1/6th chance of coming up with a six). Bayesian statistics turns this situation on its head, by estimating the validity of a model given certain data. More generally, Bayesian statistics can estimate the probability density of model parameters given the available data. To minimize error, the model is then selected whose parameters maximize this p.d.f.

References

1. **Kay(1994a)** Information Theoretic Neural Networks for Contextually Guided Unsupervised Learning: Mathematical and Statistical Considerations. Research report.
2. Neural Networks : Simon Haykin, a comprehensive foundation
3. P.D. WasserMan, Neural Computing: Theory and Practice , Van Nortrand Reinhold
4. Bose, Neural Network fundamentals with graphs, algorithms and applications
5. R.H.Neilson, Neurocomputing, Addison Wesley
6. J. Anderson et al., Neurocomputing Vol. 1 & Vol 2, MIT Press, 1986 & 1988