The n-Tuple Method

Alex Sverdlov
alex@theparticle.com

1 Introduction

The n-Tuple method was first proposed by Bledsoe & Browning [BB59] as a mechanism to recognize printed characters. It was later shown to also be capable of regression [KA96].

A modern day view of n-tuple method is a weightless neural network. In traditional neural networks, the computational units are artificial neurons that perform an inner product of inputs and weights, followed by a threshold function—essentially a linear discriminator.

In a weightless neural network the computational units are RAM (Random Access Memory) nodes, where the input is coded into an address, and output is decoded from the RAM contents at that address. Because RAM nodes are essentially maps, they have the power to learn and compute arbitrary functions. Training involves modifying the RAM contents, unlike the traditional neural networks, where training updates the weights.

The n-tuple method is quick to train and operate—training can be accomplished via one pass through the training data, and applying the classifier is just a table lookup followed by aggregating.

The base n-tuple method is more powerful than linear models, as it is learning A to B mapping (or an arbitrary function). It can also be setup to output relative probabilities, as well as the category.

Before we get into the details of how n-tuple method works, there are a few preliminaries that need to be mentioned: indexed relations & projections notation, and quantization.

1.1 Indexed Relations & Projections

An indexed relation is a tuple (I, R), where I is a sorted set of N integers, and $R = \langle X_0, X_1, X_2, \ldots \rangle$ is a list of N-tuples, such as $X_i = (x_1, \ldots, x_N)$. The index I acts as column names for relation R

A projection $\pi_J(I, R) = (J, S)$ defines a new indexed relation, where $J \subseteq I$ and S is a list of tuples of size $|J|$, having the corresponding values indexed by J. For example, an indexed relation

$$(\{1, 2, 3\}, \langle (a, b, c), (d, e, f), (g, h, i) \rangle)$$

has an index of $\{1, 2, 3\}$, and may be projected to a subspace $\{1, 3\}$

$$\pi_{\{1,3\}}(\{1, 2, 3\}, \langle (a, b, c), (d, e, f), (g, h, i) \rangle)$$

produces

$$(\{1,3\}, \langle (a,c), (d,f), (g,i) \rangle)$$

1.2 Quantization

At its core, quantization is a way of discretizing features. If a feature is a large integer, a real number, a timestamp, or a date, quantization allows us to turn it into an integer value in a certain range.

A quantizer is a function $q(x)$ that takes a feature value (e.g. a real number) and produces an integer output between 0 and $L-1$, where L is the number of quantization levels for a given feature.

This function $q(x)$ can be trained from data samples: often setup such that each quantization level has an equal probability of occurring. This is often accomplished by taking a sample of the data, ordering it by the target feature, cutting the resulting list into L equally sized slices, and using the boundary values of such cuts as quantization ranges that function $q(x)$ uses.

Let SL be a sorted list of all values for feature x. The quantization boundaries that split x into L levels would be:

$$arr_x = (SL(\frac{N}{L}), SL(2\frac{N}{L}), SL(3\frac{N}{L}), \cdots, SL((L-1)\frac{N}{L}))$$

The function $q_x(v)$ can then be defined as:

$$q_x(v) = lower\_bound(v, arr_x)$$

where *lower_bound* does a binary search to return an index into arr_x where value v is stored, or in case v is not present, an index to the element immediately greater than v. This has the effect that all $v \le SL(\frac{N}{L})$ will get quantized to 0, all values that are $SL(\frac{N}{L}) < v \le SL(2\frac{N}{L})$ will get quantized to 1, and so on.

Alternatively, to avoid run-time calculations, the value v may be bin-quantized, followed by a lookup table.

Quantization can also be applied on tuples: given a tuple we want an integer value in 0 to $Z-1$ range. For a tuple:

$$X = (x_1, \ldots, x_N)$$

We define a quantizer q_i for each feature x_i, with L_i levels. A quantized tuple $Q(X)$ is then

$$t = (q_1(x_1), \ldots, q_N(x_N))$$

Such quantized tuple is then fed into the address function $addr(t)$ which produces a single integer output given a tuple. It may be defined as:

$$\sum_{i=0}^{N-1} m_i * x_i$$

where

$$m_i = \prod_{j=0}^{i} L_j$$

The values L_i can be chosen to be proportional to the amount of entropy feature i has as compared to other features. For example, we estimate entropy for each feature:

$$e_i = -\sum_{v \in Values(x_i)} p_{iv} * log(p_{iv})$$

Where p_{iv} is relative frequency (estimated probability) of value v for feature i. The estimate of total entropy is then

$$E = \sum_i e_i$$

Each L_i should get an appropriate fraction of levels out of Z levels:

$$L_i = \max(\lceil Z^{e_i/E} \rceil, 2)$$

We need to clamp the quantization output to at least 2 levels; the minimum output of a quantization is 1 bit. This leads to a potential overflow of Z, since it may turn out that:

$$\prod_i L_i > Z$$

This means Z needs to be chosen such that it is greater than 2^N (at least 2 levels per feature).

1.3 n-Tuple Training

Let (I, R) be an indexed relation of quantized measurement tuples of Z components where

$$R = \langle X_0, X_1, X_2, \ldots \rangle$$

such as $X_i = (x_1, \ldots, x_Z)$. Let $t(X_i)$ be the known correct target class for instance X_i and $t(X_i) \in C$ (the set of class labels). This is our training data.

We define (possibly randomly) M index sets, $\langle J_1, \ldots, J_M \rangle$ where each $J_m \subset I$ and $|J_m| = N$. These M index sets are our projections for each class: these M "modules" will be trained and operate independently from each other, and their results will be combined for the final classification.

Initialize M tables, where each entry $T_{m,c}$ has an integer counter for each target class.

$$T_{m,c}(addr(x)) = \#\{x | c = t(x)\}, x \in \pi_{J_m}(I, R)$$

In other words, we build M look-up-tables from M random samples of features.

1.4 n-Tuple Classification

Let $X = (x_1, \ldots, x_Z)$ be the quantized measurement tuple we wish to classify. The classification is just an aggregate of each of the M tables for each target class, followed by picking the label that corresponds to the maximum aggregate:

$$c = \underset{c}{\operatorname{argmax}} \sum_m T_{m,c}(addr(\pi_{J_m}(I, \langle X \rangle)))$$

We could have additional logic to reserve a decision if c is not a clear winner (e.g. 2nd largest aggregate is very close). This may be beneficial, as it allows the process to not output a known ambiguous result.

1.5 Variations

There are several variations on the same theme. RAM networks often use a bit, instead of a counter (the memory location is either activated or not—the final aggregate then counts the proportion of modules that index an activated location) [AGF$^+$09].

PLN, or *probabilistic logic nodes* are RAM nodes where the content of the memory indicates a quantized probability with which the node should fire with a 1 [GVL96]. This allows for semi-training by increasing/decreasing the probability conditional on output matching desired target.

PLNs may be setup to have 3 content values: 0, 1, or d. If value is d then PLN returns 0 or 1 with equal probability. Otherwise the RAM contends of 0 or 1 are returned. Such PLNs may be arranged in a *pyramid*.

The training mechanism for this multi-layered network is to produce an output, and if it matches the target, then fix participating nodes that have a d with the actual value that they generated. This has the effect of freezing that input/output mapping for that training instance.

Bloom filters [Blo70] are a kind of n-tuple classifier. Instead of having content address the RAM directly, Bloom filters pass the input through a sequence of hash functions, using the output of the hash functions as an address into an array that is shared among all the hash functions. The classifier then checks the location of each of the results of each hash function, and if *all* the locations are populated, only then does a Bloom filter return a positive result. This has the effect of eliminating *all* false negatives. Bloom filters are used to join large datasets in distributed environments—only datasets that pass the filter are moved across the network for joining: false positives are not an issue, false negatives would be.

References

[AGF$^+$09] Igor Aleksander, Massimo De Gregorio, Felipe Maia Galvo Frana, Priscila Machado Vieira Lima, and Helen Morton. A brief introduction to weightless neural systems. In *ESANN*, 2009.

[BB59] W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 225–232, New York, NY, USA, 1959. ACM.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[KA96] Aleksander Kolcz and Nigel M. Allinson. N-tuple regression network. *Neural Networks*, 9(5):855 – 869, 1996.