Experiments in Developing Discrete Particle Swarm Optimisers

Martin Gate

October 30, 2019

1 Introduction

1.1 Continuous Case model

Particle Swarm Optimisers (PSO) introduced in [KE95] have been developed to optimise problems with continuous tuning parameters. The PSO consists of a finite collection (called a swarm) of particles with current state (often called *Parameters*) x_t^i personal best state p_t^i and velocity V_t^i; where i is the particle index and t is the iteration index. These three properties of each particle are updated at every iteration using

$$V_{t+1}^i = \mu\left(x_t^i, V_t^i, N_t^i\right) \tag{1.1}$$

$$x_{t+1}^i = \xi\left(x_t^i, V_t^i\right) \tag{1.2}$$

$$p_{t+1}^i = \begin{cases} p_t^i & \text{if } x_{t+1}^i \notin \Omega \text{ or } F(p_t^i) < F(x_{t+1}^i) \\ x_t^i & \text{otherwise} \end{cases} \tag{1.3}$$

where $F(x)$ is the cost of the state x; N_t^i is the collection of personal best states used by the ith particle to update its velocity from iteration t to $t+1$: Ω is a feasible subset of states usually characterised as satisfying a collection of constraints .

The PSO has several variants and the above description is a general summery where for continuous parameters the state is a d dimensional vector space of real numbers $\mathbb{R}^d$. Typically the velocity is made up from random multiples of the vector diference between x_t^i personal bests and members ofN_t^i giving the swarms directions to go to look for improved $F(x)$ values. Also μ and ξ are not true functions since they depend on hidden random variables that are crucial for ensuring the ability to find a low cost near optimal personal best after sufficient iteration, but are otherwise processes that depend on the variables given as their arguments.

1.2 The direction of Interest

The main interest is developing an optimiser for Machine Learning that bridges the gap between neural nets and more structure determining optimisers such

as genetic algorithms. To do this the current work looks at combinatorial optimisation problems (COP) using discrete tuning parameters in the form of binary strings (BS) for particle state with a range of cost-function value types. Such a PSO will be called a Set PSO (abbreviated to SPSO) to emphasise that it interprets BS as sub sets to represent the parameters of SPSO.

1.3 Machine Learning Extensions

For Machine Learning there is a need to extend the basic SPSO to cope with:

1. noisy data mainly due to a restricted amount of available data when evaluating a cost-function.
2. represent continuous state such as design or neural net parameters by a set of discrete values some what similar to the way genetic algorithms deal with this by digitizing the continuous case.
3. treat random strings of bits as a way of generating useful programs to find optimising solutions which in turn can be used as a way of tuning meta parameters.
4. Acknowledge that only approximate solutions are needed to avoid over fitting.

1.4 Implementation and Development

The SPSO is developed mainly through biased intuitive analogy with other systems. As such it needs a programming language to explore empirically its properties and implement it. The language chosen is Golang. The treatment of SPSO here attempts to make the description language agnostic, however sometimes golang constructs such as interface are directly used to simplify the description of algorithms and help the user understand what the code is trying to do.

2 Set PSO

2.1 Cost-function Value Representation

COP problems are discrete in nature and often have an integer cost-function value, because of this big integer cost values was initially used for SPSO. The restriction to big integer is strictly not required since the cost function is used only to compare the cost between solutions so the cost function value in the software is represented by the interface type `CostValue`. This has the additional advantage that it can contain state to help deal with noisy data that needs to be evaluated several times to reduce the noise and store its variance.

Formaly `CostValue` object `cost` has the following interface methods were `x` is to be kept general to avoid a circular definition, but is expected to have a type that can be converted to the cost value object being used:

`cost.Set(x interface{})` this sets the oject to a copy of the value `x`.

`cost.Cmp(x interface{}) int` this compares the cost value object with x and returns the following values:

-2 `cost` needs to be further evaluated to establish a comparison.

-1 `cost` has a lower cost than `x`.

0 `cost` has the same cost value as `x`.

1 `cost` has a larger cost vale than `x`.

2 `x` needs to be further evaluated to establish a comparison.

`cost.Fbits() float64` this returns a logarithmic like floating point value for plotting; it attempts to give the number of bits used in the big integer case. In most cases the cost is always positive in which case a zero cost returns 0.

As can be seen the underlying implementation of the cost value is well hidden from SPSO by using this interface it allows the cost-function to have a range of implementations of its cost value which could include big integers, floating point, or even strings representing programs!

Here we talk about a cost value as if it has a numerical value; this may not be the case it could for instance be a program to implement changes to reach a goal; in this case `Fbits()` just gives an indication of the complexity of the changes while `Comp()` does the comparison between solutions.

`Comp()` has two additional values ± 2 which are used in cases where the cost value is difficult to compare without further evaluation due to randomness in the costs being evaluated. behind the scenes the cost value could for instance have a measured mean and variance associated to the cost value. If there is too much variance associated with a solution pair there cannot be a meaningful comparison between them and further cases are needed to reduce the variance through more cost-function evaluation.

Big integer cost values are used to include some of the difficult cases in combinatorial optimisation, although the chance of finding global best solutions to these problems is minimal; however, The algorithms are expected to produce good useful approximations to these hard problems; I would like to suggest that main area of application is *machine learning where one is interested in good approximate minimisation solutions of difficult problems*.

2.2 The Importance of Satisfying Constraint on Solutions

COP problems such as the Travelling Sales Person(TSP) have constraints defining Ω that must be satisfied if to be of any use so one must continuously ensure the candidate solution is in Ω before evaluating its cost. Often feasible points Ω do not form an open subset of the space of states so a raw update of x_t^i will not give a feasible state most of the time. To cope with this ξ computes a raw update $\acute{x}_{t+1}^i$ which it then attempts to convert into a feasible one before being returned as a update. Sometimes this may not be possible and the returned x_t^i is not in Ω. As indicated in the update equation this results in no change to the personal best thus ensuring that personal bests are updated to only feasible states.

2.3 Generalised Link Between COP and SPSO

The link between COP and SPSO is forged by using binary strings to represent candidate algorithms which are obtained by decoding the string. I use the term algorithm to represent any method for finding approximate solutions to a COP problem including just providing an approximate solution with no use of an algorithm. I think decoding to an algorithm rather than just a solution gives the PSO significantly more expressive power to find approximate solutions. The structure of candidate solutions may not be the same as binary strings so several binary strings may map to the same solution. The resulting solution is then evaluated by the cost-function to give its cost value indicating how well the algorithm fits; this could include some cost component representing the complexity of used algorithm; lower values being preferred to higher values as determined by `Comp()`. This procedure gives a mapping from a binary string to cost value of the binary string. The SPSO is used to learn a low cost global solution by using a swarm of proposed solutions to iteratively search for a low cost one.

Even in the case of a deterministic function it may be expedient to cut down on the evaluation time by only partially evaluating it thus introducing uncertainty. For instance the solution string may represent a *program* that has to be tested on a ridiculously large number of cases to give a perfect cost value in this case the cases are randomly chosen to test the solution and thus introducing uncertainty which is traded against speed of evaluation. Once the cost value uncertainty is measured the cost-function can determine whether two solutions can be compared and feed this back to the SPSO via `Comp()`.

For the moment we restrict the SPSO state to be at most N bits. Each binary string x can thus be regarded as a subset of integers in the range $1 \cdots N$ where $j \in x$ when $x[j] = 1$. To this extent each non zero bit represents the inclusion of some element with a feature and as such encodings that favour this interpretation is expected to do better using the following algorithms.

2.4 Reading Across from the Continuous Case

In the following let *rand* be a random number generator that produces a number in the range $0 \leq rand < 1$ for each use of it.Also let $\neg$ be the logical negation and $\leftarrow$ the assignment of left side to the right side of the arrow.

2.4.1 Semantic Reinterpretation of Notation

The notation for the continuous case can be now used where x_t^i is just a subset rather than a vector. The velocity is interpreted as a vector of probabilities where the jth component $V_t^i[j]$ is just the probability that $x_t^i[j]$ will flip during the update going from 0 to 1 or *vice versa*.

2.4.2 Adding up velocity contributions

The velocity is built up from several contributions, which will be described in detail below, so there is a need for a way of adding up the contributions. the velocity is an array of probabilities each component being regarded as independent so one is interested in how to add probabilities. direct adding

of probabilities does not work because this can give values greater than 1; multiplying probabilities just leads to a value smaller than the probabilities being combined. A possible compromise is to combine the probabilities p_1 and p_2 as follows:

$$p_1 \dot{+} p_2 = p_1 + p_2 - p_1 p_2 \tag{2.1}$$

and for two velocities we define

$$(V_1 \dot{+} V_2)[j] = V_1[j] \dot{+} V_2[j] \qquad \forall j \tag{2.2}$$

this is the method adopted although not the only one that could have been used. For instance taking the maximum value of the probabilities is popular and could be an alternative. the chosen operation (2.1) is called *pseudo-addition* since it has most properties of addition. This can be seen by putting

$$p_i = 1 - e^{-\xi_i} \tag{2.3}$$

and then noting that

$$p_1 \dot{+} p_2 = (1 - e^{-\xi_1}) + (1 - e^{-\xi_2}) - (1 - e^{-\xi_1})(1 - e^{-\xi_2}) \tag{2.4}$$

$$= 1 - e^{-\xi_1} e^{-\xi_2} \tag{2.5}$$

$$= 1 - e^{-(\xi_1 + \xi_2)} \tag{2.6}$$

so the mapping $p_1 \mapsto \xi_i$ is a homomorphism between $\dot{+}$ and $+$. In particular the pseudo-sum is commutative and associative.

2.4.3 Velocity Components for Directing Towards N_t^i

As for continuous case the velocity, as an indicator of the direction to go to find better solutions, is roughly increased along the diference between x_t^i and p_t^k for each $k \in N_t^i$. For the SPSO the difference is represented by the exclusive or of the corresponding subsets giving the difference:

$$del_t^{(i,k)} = x_t^i \oplus p_t^k$$

From this a velocity increment can be produced as $Vdel_t^{(i,k)}$ where

$$r(i,k) \leftarrow \phi \, rand \tag{2.7}$$

$$r_t^{(i,k)} \leftarrow \begin{cases} r(i,k) & \text{if } r(i,k) < 1 \\ 2 - r(i,k) & \text{otherwise} \end{cases} \tag{2.8}$$

$$Vdel_t^{(i,k)}[j] = \begin{cases} r_t^{(i,k)} & \text{if } j \in del_t^{(i,k)} \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

the ϕ is a heuristic for tuning how aggressively the velocity points towards the target particle k and here takes on a predetermined heuristic value between 0 and 2 (following tradition of continuous case). However, the velocity is limited to be in the range 0 to 1 since it is a probability of flipping, so values above 1 are reflected back. This doubles the density of $Vdel_t^{(i,k)}$ near 1 when $\phi > 1$. The treatment of how to cope with $r(i,k) > 1$ case is not well derived and alternatives such as limiting all such cases to 1 may be better. The velocity increment encourages movement to the target p_t^k by only contributing to the difference between x_t^i and p_t^k.

2.4.4 Shotgun Blurring of Targets

Typicaly there are a large number of elements (corresponding to dimensions in the continuous case) while the number of particles is kept to just a few say at most 30. This means the use of just the velocity contributions directed to targets will explore only a small subset of changes. In the continuous case this has been solved in [BM14] by adding a noise component to the velocity proportional to distance of particle from corresponding target, which effectively gives a shotgun effect. Further more this simple device makes the PSO converge to a local optimal solution.

For SPSO the distance is the Hamming distance between the particle and target given by

$$d_H(i,k) = |del_t^{(i,k)}| \tag{2.10}$$

where $|z|$ is just the number of elements in z.this gives a blurring velocity

$$b = rand(L_{factor} d_H(i,k) + L_{offset})/N \tag{2.11}$$

$$Vblur_t^{(i,k)}[j] = b \ \forall j \tag{2.12}$$

where L_{factor} and L_{offset} are heuristics used to tune the shotgun effect.

2.4.5 Inertia Term

A heuristc ω borrowed from the continuous case is an inertia term that multiplies the velocity probability by a reducing factor to slow down the progress of a particle and stop the velocity terms from over saturation.

Using the multiplying operation is a crude and simple and may not be the best; the mapping from pseudo-addition to addition given in Equation 2.3 suggests a more complex operation involving raing the probability of not flipping to some power. However, for the moment, simpler operation of multiplying which has the desired reduction of velocity is used here.

2.4.6 Combining the Velocity Contributions

these various contributions are combined to give

$$V_{t+1}^i = \omega \left(V_t^i \dot{+} \sum_{k \in N_t^i}^{\bullet} Vblur_t^{(i,k)} \right) \dot{+} \sum_{k \in N_t^i}^{\bullet} Vdel_t^{(i,k)} \tag{2.13}$$

where multiplying by ω multiply each component of the velocity by ω. In the continuous case ω just multiplies V_t^i; in the discrete case blurring tends to saturate and mask the update that moves the solution in direction of targets, so ω is applied to this as well.

2.4.7 Updating to the Next Frame Using the Probabilistic Velocity

In the continuous case the raw update is given by adding the velocity component to the current state. For the SPSO case by analogy we use flipping to give a raw update from the velocity, thus put for each component:

$$(V_{t+1}^i[j]), \breve{x}_{t+1}^i[j]) \leftarrow \begin{cases} (0, \neg x_t^i[j]) & \text{if } V_{t+1}^i[j] > rand \\ (V_{t+1}^i[j], x_t^i[j]) & \text{otherwise} \end{cases} \tag{2.14}$$

At this stage the raw state $\breve{x}^i_{t+1}$ may not satisfy the constraint so to keep things general a function `ToConstraint()`[1] supplied by the cost-function interface is applied to attempt to make the raw state satisfy the constraint based on the raw state as a starting hint and the old state,x^i_t to facilitate this attempt.If it fails then revert back to the old state.Thus:

$$\breve{x}^i_{t+1} \leftarrow \quad \text{ToConstraint}(x^i_t, \breve{x}^i_{t+1}) \tag{2.15}$$

$$x^i_{t+1} \leftarrow \quad \begin{cases} \breve{x}^i_{t+1} & \text{if } \breve{x}^i_{t+1} \in \Omega \\ x^i_t & \text{otherwise} \end{cases} \tag{2.16}$$

this gives an update that satisfies the constraints, Ω where the initalisation of the state x^i_0 is chosen to ensure it also satisfies the constraint. this is done by random selection until the application of `ToConstraint()` gives something that satisfies the constraint. Note during this operation the first argument will be the empty set and should be treated as a special case by the cost function.

2.5 Updating the personal best

To cope with noisy cost-functions the comparator function Comp() has the option of not being able to compare two cases but will indicate which case needs to be further evaluated to improve the chance of a successful comparison. let x and y be two constraint satisfying subset cases from this we get the cost function values $F(x)$ and $F(y)$ we now attempt a comparison say by using Comp() method on $F(x)$ and we use the following symbolic notation depending on the result as follows:

$F(x)$.`Comp`$(F(y))$	Operator	Comment
0	$F(x) = F(y)$	
-1	$F(x) < F(y)$	
1	$F(x) > F(y)$	
-2	$F(x) \nsucc F(y)$	$F(x)$ needs further evaluation
2	$F(x) \nprec F(y)$	$F(y)$ needs further evaluation

2.6 grouping Swarm Particles

It is anticipated that the main things that improve the SPSO performance is change in heuristics and the method used to choose the target sets N^i_t. Further more it is expected that particles in the swarm will adopt specialist behaviour by belonging to a group of particles. To support this the swarm is partitioned into groups where each group of swarm particles has the same set of targets and heuristics. Groups are each given a name to aid describing a group.To begin with there is one group called "root" which contains all particles. Heuristics are often the same between groups so when forming a new group the heuristics are linked to the "root" group, although this can be modified and in general groups share heuristics to simplify changing a heuristic. Groups are populated by moving swarm particles from other groups thus maintaining a partition of swarm. By default the heuristics are chosen to have useful values and need not be explicitly given.

[1] in the code this function has a diferent interface to minimise the API but is used to acheive the same thing.

3 How to deal with a noisy cost function?

In the targeted application of Machine Learning the cost-function $F(x)$ is not a direct mathematical function of x and has a random component so it has to be evaluated several times before comparing the fitness of x.

The Particle Swarm Optimiser uses a personal best and global best set of parameters and costs to guide it in its optimisation. in the standard implementation this assumes that the cost function is deterministic to the extent that the cost is assumed to be a direct function of the parameters submitted to the cost function. However, this may not be the case when dealing with real data where there are hidden parameters that are not accounted for or the cost is a random function parametrised by x. This means that the personal best may be due to parameters with a fluke low cost which is rearly improved on during the optimisation and thus cause the process to stick at a non optimal value for typical behaviour. Furthermore the underlying statistics may be slowly changing as the cost is sampled.

3.1 Recursive formula for calculating expected values

Let b_i be a sequence of independent binary samples $b_i : S \to \{0,1\}$ of a probability P. By defenition we have

$$P(b_{1,} \cdots, b_n) = \prod_{i=1}^{n} P(b_i)$$

For a subset $A \subset S$ and a map $g : 2^A \to \mathbb{R}$ we can get a sequence of estimates of expected values

$$\mathbb{E}_n[g \parallel A] = \frac{1}{n} \sum_{i=1}^{n} g(b_i \cap A)$$

of

$$\mathbb{E}[g \parallel A] = \sum_{B \subset S} P(B) g(B \cap A)$$

which just calculates the expected value of g acting on subsets of A . The double bar notation $\parallel$ is used to indicate that we are looking on to the probability disribution through the sub set window A. Now for $n > 1$

$$\mathbb{E}_n[g \parallel A] = \left(1 - \frac{1}{n}\right) \mathbb{E}_{n-1}[g \mid A] + \frac{1}{n} g(b_n \cap A)$$

we can generalise this to the sequence

$$\check{\mathbb{E}}_n[g \parallel A] = (1 - \lambda_n)\, \check{\mathbb{E}}_{n-1}[g \parallel A] + \lambda_n g(b_n \cap A)$$

with

$$\check{\mathbb{E}}_1[g \parallel A] = g(b_1 \cap A)$$

taking expected values and putting $\tilde{\mathbb{E}}_n[g \parallel A] = \mathbb{E}\left[\check{\mathbb{E}}_n[g \parallel A]\right]$ we get

$$\begin{aligned} \tilde{\mathbb{E}}_n[g \parallel A] &= (1 - \lambda_n)\tilde{\mathbb{E}}_{n-1}[g \parallel A] + \lambda_n \mathbb{E}[g \parallel A] && (3.1) \\ &= (1 - \lambda_n)\mathbb{E}[g \parallel A] + \lambda_n \mathbb{E}[g \parallel A] && (3.2) \\ &= \mathbb{E}[g \parallel A] && (3.3) \end{aligned}$$

by induction on n. So even this sequence is an unbiased estimate of the expected value. The important thing is to check for variance. We have

$$\begin{aligned}
\mathbb{E}\left[\left(\check{\mathbb{E}}_n[g \parallel A] - \mathbb{E}[g \parallel A]\right)^2\right] &= \mathbb{E}\left[\left(\check{\mathbb{E}}_n[g \parallel A]\right)^2\right] - 2\mathbb{E}\left[\check{\mathbb{E}}_n[g \parallel A]\right]\mathbb{E}[g \parallel A] + \mathbb{E}[g \parallel A]^2 \\
&= \mathbb{E}\left[\left(\check{\mathbb{E}}_n[g \parallel A]\right)^2\right] - \mathbb{E}[g \parallel A]^2 \\
&= (1-\lambda_n)^2\mathbb{E}\left[\left(\check{\mathbb{E}}_{n-1}[g \parallel A]\right)^2\right] + \lambda_n^2\mathbb{E}\left[g(b_n \cap A)^2\right] + (2\lambda_n(1-\lambda_n) - 1)\mathbb{E}[g \parallel \\
&= (1-\lambda_n)^2\mathbb{E}\left[\left(\check{\mathbb{E}}_{n-1}[g \parallel A]\right)^2\right] + \lambda_n^2\mathbb{E}\left[g(b_n \cap A)^2\right] - ((1-\lambda_n)^2 + \lambda_n^2)\mathbb{E}[g \parallel \\
&= (1-\lambda_n)^2\mathbb{E}\left[\left(\check{\mathbb{E}}_{n-1}[g \parallel A] - \mathbb{E}[g \parallel A]\right)^2\right] + \lambda_n^2\mathbb{E}\left[\left(g(b_n \cap A) - \mathbb{E}[g \parallel A]\right)^2\right]
\end{aligned}$$

where we have used the independence of the b_i's to give

$$\begin{aligned}
\mathbb{E}\left[\check{\mathbb{E}}_{n-1}[g \parallel A]g(b_n \cap A)\right] &= \mathbb{E}\left[\check{\mathbb{E}}_{n-1}[g \parallel A]\right] E\left[g(b_n \cap A)\right] \\
&= \mathbb{E}[g \parallel A]^2
\end{aligned}$$

put

$$\begin{aligned}
\sigma^2 &= \mathbb{E}\left[\left(g(b_n \cap A) - \mathbb{E}[g \parallel A]\right)^2\right] \\
\alpha_n &= \mathbb{E}\left[\left(\check{\mathbb{E}}_n[g \parallel A] - \mathbb{E}[g \parallel A]\right)^2\right] / \sigma^2
\end{aligned}$$

then we have

$$\alpha_n = (1-\lambda_n)^2\alpha_{n-1} + \lambda_n^2$$

as a formular that gives the change in variance of the estimates $\check{\mathbb{E}}_n[g \parallel A]$.

As one can see there are several ways of combining measured means to give unbiased estimates of a noisy function's mean that gives a trade off on variance.

To directly calculate the expected cost at each iteration would be too expensive an operation especially if the parameters are way off optimal. Let $C(X)$ be the random variable representing the cost function for a given parameter vector X. for a given particle the current parameter changes rapidly while when it matters the personal best parameter, X_b varies comparatively slowly so when the X_b is not updated one can take the opportunity to update the estimated expected cost function value $\hat{C}(X_b)$ of $\mathbb{E}[C(X_b)]$ in this case.This reduces the variance of the expected value compared to its raw value $C(X_b)$.

Lets look at some ways in which the estimate in principle can be updated iteratively. Given samples $C_1, \cdots, C_i$ and a forgetting gain α we could produce an unbiased estimate

$$\hat{C}(X_b)_i = \frac{\sum_{j=1}^{i} \alpha^{j-1} C_{i-j+1}}{\sum_{j=1}^{i} \alpha^{j-1}} \tag{3.4}$$

Put

$$A_i = \sum_{j=1}^{i} \alpha^{j-1} \tag{3.5}$$

$$= \frac{1-\alpha^i}{1-\alpha} \tag{3.6}$$

then using (3.4) we get

$$A_i\hat{C}(X_b)_i - \alpha A_{i-1}\hat{C}(X_b)_{i-1} = C_i \tag{3.7}$$

rearranging and using (3.6) we get

$$\hat{C}(X_b)_i = \frac{\alpha A_{i-1}\hat{C}(X_b)_{i-1} + C_i}{A_i} \tag{3.8}$$

$$= \frac{\alpha(1-\alpha^{i-1})\hat{C}(X_b)_{i-1} + (1-\alpha)C_i}{1-\alpha^i} \tag{3.9}$$

now if we put

$$\lambda_i = \frac{1-\alpha}{1-\alpha^i} \tag{3.10}$$

we get substituting into (3.9) the iterative update

$$\hat{C}(X_b)_i = (1-\lambda_i)\hat{C}(X_b)_{i-1} + \lambda_i C_i \tag{3.11}$$

In fact any update of the form given in (3.11) always gives an unbiased estimate for arbitrary λ_i . So we can use this as a generalised update. the choice of parameter is tuned to cater for how rapidly one wants to converge and the required variance of the result.

Looking at the variance update assuming independent samples. Put σ_i^2 as the variance of $\hat{C}(X_b)_i$ and σ^2 as the variance of C_i then we get

$$\sigma_i^2 = (1-\lambda_i)^2\sigma_{i-1}^2 + \lambda_i^2\sigma^2 \tag{3.12}$$

The minimum value of the update variance as a function of λ_i occurs at the point of inflexion given by

$$0 = \frac{\partial\sigma_i^2}{\partial\lambda_i} \tag{3.13}$$

$$= -2(1-\lambda_i)\sigma_{i-1}^2 + 2\lambda_i\sigma \tag{3.14}$$

$$= 2\{\lambda_i(\sigma_{i-1}^2 + \sigma^2) - \sigma_{i-1}^2\} \tag{3.15}$$

This gives the smallest variance λ_i as

$$\overset{*}{\lambda}_i = \frac{\sigma^2}{\sigma_{i-1}^2 + \sigma^2} \tag{3.16}$$

$$= \frac{1}{1 + (\frac{\sigma^2}{\sigma_{i-1}^2})} \tag{3.17}$$

$$= \frac{\sigma^{-2}}{\sigma_{i-1}^{-2} + \sigma^{-2}} \tag{3.18}$$

From (3.18) we get

$$1 - \overset{*}{\lambda}_i = \frac{\sigma_{i-1}^{-2}}{\sigma_{i-1}^{-2} + \sigma^{-2}} \tag{3.19}$$

(3.17) show that the result depends only on the ratio of variances while (3.18) gives the result in form of inverse variance. using this gives easy to use results

as will shortly be seen. Based in this remark substitute (3.18),(3.19) into (3.12) and calculate the updated inverse variance as

$$\overset{*}{\sigma_i}^{-2} = ((1-\overset{*}{\lambda_i})^2\sigma_{i-1}^2 + \overset{*}{\lambda_i}^2\,\sigma^2)^{-1} \tag{3.20}$$

$$= \frac{\sigma_{i-1}^{-2}\sigma^{-2}}{(1-\overset{*}{\lambda_i})^2\sigma^{-2} + \overset{*}{\lambda_i}^2\,\sigma_{i-1}^{-2}} \tag{3.21}$$

$$= \frac{\sigma_{i-1}^{-2}\sigma^{-2}(\sigma_{i-1}^{-2}+\sigma^{-2})^2}{\sigma_{i-1}^{-4}\sigma^{-2}+\sigma_{i-1}^{-2}\sigma^{-4}} \tag{3.22}$$

$$= \sigma_{i-1}^{-2} + \sigma^{-2} \tag{3.23}$$

This shows the inverse variance using the optimal λ is additive and the expression is particularly simple so it can be calculated first and then $\overset{*}{\lambda_i}$ is evaluated from

$$\overset{*}{\lambda_i} = \sigma^{-2}/\,\overset{*}{\sigma_i}^{-2} \tag{3.24}$$

This gives the minimum variance update. However in the long run as it repeats the λ_i becomes so small that it ignores the current measurement and any change in the underlying statistics is ignored. We can avoid this by deliberately choosing a λ_i which is larger than the optimal given by (3.24). To this end put

$$\lambda_i = (1+\delta)\,\overset{*}{\lambda_i} \tag{3.25}$$

then evaluate the updated inverse variance by substituting into (3.12) the expression (3.25) and simplifying using (3.23),(3.18) and (3.19)

$$\sigma_i^{-2} = \frac{\sigma^{-2}\sigma_{i-1}^{-2}}{[(1-\overset{*}{\lambda_i}) - \delta\,\overset{*}{\lambda_i}]^2\,\sigma^{-2} + \overset{*}{\lambda_i}^2\,(1+\delta)^2\,\sigma_{i-1}^{-2}} \tag{3.26}$$

$$= \frac{\sigma^{-2}\sigma_{i-1}^{-2}}{(1-\overset{*}{\lambda_i})^2\,\sigma^{-2} + \overset{*}{\lambda_i}^2\,\sigma_{i-1}^{-2} - 2(1-\overset{*}{\lambda_i})\,\sigma^{-2} + 2\,\overset{*}{\lambda_i}\,\sigma_{i-1}^{-2} + \overset{*}{\lambda_i}^2\,\delta^2(\sigma^{-2}+\sigma_{i-1}^{-2})} \tag{3.27}$$

$$= \frac{\sigma^{-2}\sigma_{i-1}^{-2}\,\overset{*}{\sigma_i}^{-4}}{\sigma^{-2}\sigma_{i-1}^{-2}\,\overset{*}{\sigma_i}^{-2} + \sigma^{-}4\,\overset{*}{\sigma_i}^{-2}\,\delta^2} \tag{3.28}$$

$$= \frac{\overset{*}{\sigma_i}^{-2}}{1+\sigma^{-2}/\sigma_{i-1}^{-2}\,\delta^2} \tag{3.29}$$

Equation (3.29) is a particularly simple expression that shows how the inverse variance depends on δ.

Most models of probability distribution start with combining pairs of variables to give more elaborate models of the probability function. Often in the case of neural nets the pairs are combined linearly to give a single linear function which is then thresholded before being passed onto an output or hidden layer. Unfortunately this in the case of neural nets gives rise to difficult to interpret models with with often pathological behaviour. The aim here is to use

the multilinear approach that encourages more unique representation of the data which can also be interpreted. However multi-linear functions are only linear on each variable in isolation and are in general non linear, hopefully in a useful way. When combining pairs of variables we will combine to produce a result for a hidden variable (or output from this) and not threshold it. This section gives results for combining pairs of variables.

4 Multi-linear functions and binary data probabilities

Multi-linear expressions are a natural way of generating functions of binary data. Let S be a finite ordered set of size n containing the elements $x_1, x_2, \cdots, x_n$ then we can for each binary sequence $a : S \to \{0, 1\}$ of n 0 and 1's associate a subset $\varphi(a) \subset S$ given as the elements x_i of S such that the sequence a maps $a(i)$ to 1. Put for $A \subset S$

$$Q(A)[x_1, x_2, \cdots, x_n] = \prod_{x_{i \in A}} x_i \prod_{x_j \notin A} (1 - x_j)$$

Let $A, B \subset S$ such that $A \neq B$ then there exists an k such that either $x_k \in A$ and $x_k \notin B$ or $x_k \in B$ and $x_k \notin A$. assuming the former and let b be the associated binary sequence of B. Evaluating $Q(A)$ with b we get that $b(k) = 0$ so

$$Q(A)(b) \triangleq Q(A)(b(1), b(2), \ldots, b(n)) = \prod_{x_{i \in A}} b(i) \prod_{x_j \notin A} (1 - b(j)) = 0$$

similarly for the other case. Also $Q(B)(b) = 1$ by inspection. This also shows that the $Q(A)[x_1, x_2, \cdots, x_n]$ are linearly independent since if

$$\sum_{A \subset S} c(A) Q(A)[x_1, x_2, \cdots, x_n] = 0$$

for coefficients $c(A) \in \mathbb{R}$ by substituting $a = \varphi^{-1}(A)$ we get

$$c(A) = c(A) Q(A)(a) = 0$$

If we put for $A \subset S$

$$\widetilde{Q}(A)[x_1, x_2, \cdots, x_n] = \prod_{x_{i \in A}} x_i$$

then by induction on n we can show that $Q(A)$ is a linear combination of $\widetilde{Q}(A)$. Also by definition the $\widetilde{Q}(A)$ are independent and have the same number of dimensions 2^n so span the same subspace of polynomials. Because of this the $Q(A)$ for $A \subset S$ span the space of *multilinear polynomials*.

Further for each map $f : 2^S \to \mathbb{R}$ we can associate a a polynomial $Q(f)$ over the elements of S given by

$$Q(f)[x_1, x_2, \cdots, x_n] = \sum_{A \subset S} f(A) Q(A)$$

and we get

$$Q(f)(b) = f(\varphi(b))$$

This multi-linear or polynomial representation of the map f is in one to one correspondence so there is nothing added by using the more complex polynomial representation of the map from subsets of S other than providing a smooth continuous representation of such functions that can be easily approximated by simple combination of polynomials that can be used to provide a more compact approximation of f. In the general case $Q(f)$ is a large expression to evaluate or store in a computer since it has 2^n terms. Also the coefficients by themselves do not convey any useful structure or pattern without further analysis. The question is can one introduce more restrictive polynomial expressions that approximate $Q(f)$ and give easy interpretations of the underlying patterns of sequences that generate significant values of f without significant loss of information.

In particular if $P(a)$ is the probability of the sequence a then $Q(P \circ \varphi^{-1})$ is a polynomial that maps binary sequences to their respective probability. Call this the *probability polynomial* and Use the notation $Q_P[x_1, x_2, \cdots, x_n]$ for this. The φ map is an equivalence between subsets and binary sequences so when not ambiguous leave it out. For instance $P(A) \triangleq P(\varphi^{-1}(A))$ for a subset A.

As a general rule low order polynomial representations avoid over fitting so polynomials involving a small number of variables (that is elements of S) are useful. in particular combining variables in pairs is useful especially if we create intermediate variables called hidden that represent linear polynomials of variables and hidden variables.

4.1 bivariate binary table polynomial representation

let x and y be binary valued variables. let $a : \{0,1\} \times \{0,1\} \to \{0,1\}$ be a function. this is represented by a polynomial

$$Q(a)[x,y] = a(0,0)(1-x)(1-y) + a(0,1)(1-x)y + a(1,0)x(1-y) + a(1,1)xy \tag{4.1}$$

$$= a(00) + [a(1,0) - a(0,0)]x + [a(0,1) - a(0,0)]y + [a(0,0) + a(1,1) - a(0.1) - a(1,0)]xy \tag{4.2}$$

that agrees with the binary function a for binary values. we can get a clean uncluttered parametrisation by putting

$$X_0 = a(0,0) \tag{4.3}$$

$$X_1 = a(1,0) - a(0,0) \tag{4.4}$$

$$X_2 = a(0,1) - a(0,0) \tag{4.5}$$

$$x_3 = a(1,1) - a(0,0) \tag{4.6}$$

$$\tag{4.7}$$

to give

$$Q(a)[x,y] = X_0 + X_1 x + X_2 y + (X_3 - X_1 - X_2)xy \tag{4.8}$$

References

[BM14] Mohammad Reza Bonyadi and Zbigniew Michalewicz. A locally convergent rotationally invariant particle swarm optimization algorithm. *Swarm Intell,Springer*, pages 1–40, June 2014.

[BM16] Mohammad Reza Bonyadi and Zbigniew Michalewicz. Analysis of stability, local convergence, and transformation sensitivity of a variant of particle swarm optimization algorithm. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 20(3):370–385, July 2016.

[KE95] J. Kennedy and R. Eberhart. Particle swarm optimization. *International conference on neural net- works, IEEE*, 4(3):1942–1948, July 1995.