Analysis of Neural Cryptography

Alexander Klimov, Anton Mityagin, and Adi Shamir

Computer Science Department, The Weizmann Institute, Rehovot 76100, Israel
{ask,mityagin,shamir}@wisdom.weizmann.ac.il

Abstract. In this paper we analyse the security of a new key exchange protocol proposed in [3], which is based on mutually learning neural networks. This is a new potential source for public key cryptographic schemes which are not based on number theoretic functions, and have small time and memory complexities. In the first part of the paper we analyse the scheme, explain why the two parties converge to a common key, and why an attacker using a similar neural network is unlikely to converge to the same key. However, in the second part of the paper we show that this key exchange protocol can be broken in three different ways, and thus it is completely insecure.

1 Introduction

Neural networks have attracted a lot of attention in the last 60 years as a plausible computational model of how the human brain operates. The model was first formalized in 1943 by Warren McCulloch and Walter Pitts, and in 1949 Donald Hebb published his highly influential book *Organization of Behavior* in which he studied a variety of neural learning mechanisms. Today the area continues to be extremely active, and attracts interdisciplinary researchers from a wide variety of backgrounds (Biology, Medicine, Psychology, Physics, Mathematics, Computer Science, etc).

Not surprisingly, researchers have also tried to use neural networks in Cryptography. In January 2002, the Physicists Kanter, Kinzel and Kanter [3] proposed a new key exchange protocol between two parties $\mathcal{A}$ and $\mathcal{B}$. It uses the new notion of chaotic synchronization, which makes it possible for two weakly interacting chaotic systems to converge even though each one of them (viewed individually) continues to move in a chaotic way. Many papers and several conferences were devoted in the last few years to this subject, and an excellent starting point for this literature can be found in [5].

In this paper we analyse the Kanter, Kinzel and Kanter (KKK) proposal, which can be viewed as a gradual type of Diffie Hellman key exchange. In both schemes the two parties start from random uncorrelated t-bit states. The DH scheme uses a single round in which each party reveals t bits of information about its state. Based on the received information, each party modifies its state once, and the new states become identical. The KKK scheme uses multiple (typically $\geq t$) rounds in which each party reveals a single bit of information about its current state, and then modifies its state according to the information revealed by the other party. If we denote the sequence of states of the two parties by $\mathcal{A}_i$ and $\mathcal{B}_i$, then $distance(\mathcal{A}_{i+1}, \mathcal{B}_{i+1}) < distance(\mathcal{A}_i, \mathcal{B}_i)$ and eventually $\mathcal{A}_i = \mathcal{B}_i$ for all $i \geq i_0$. However, the parties do not converge by moving towards a common fixedpoint which is halfway between them, and both $distance(\mathcal{A}_{i+1}, \mathcal{A}_i)$ and $distance(\mathcal{B}_{i+1}, \mathcal{B}_i)$ remain large even for $i \geq i_0$. From the point of view of the cryptanalyst the states of the parties become rapidly moving targets, and his main problem is how to combine bits of information about two converging sequences of unknown states. Such multiround synchronization is a new and unexplored idea, which makes it possible to use new types of cryptographic functions which are not based on number theory. A KKK-like scheme can thus provide a new basis for security, and can give rise to potentially faster key exchange schemes.

The concrete proposal in [3] uses two neural networks in which each network tries to learn from the other network's outputs on common random inputs. In the standard learning problem a neural network tries to learn a fixed function, and thus it converges towards it as a fixedpoint. In the mutual learning problem each network serves both as a trainer and as a trainee, and there is no fixed target to converge to. Instead, they chase each other in a chaotic trajectory which is driven primarily by the common sequence of random inputs. We first consider the nontrivial issue of why the scheme works at all, i.e., why the two chaotic behaviours become synchronized. We then explain the empirical observation in [3] that an attacker who uses an identical neural network with the same learning procedure is extremely unlikely to synchronize his network with the other parties' network even though he eavesdrops to all their communication. However, in the last part of the paper we show that the KKK scheme can be broken by at least three different attacks (using genetic algorithms, geometric considerations, and probabilistic analysis). The bottom line of our analysis is that even though this concrete cryptographic scheme is insecure, the notion of chaotic synchronization is an exciting new concept and a potential source of new insights into how parties can agree on common secret values as a result of public discussion.

2 The KKK Key Exchange Scheme

Each party in the proposed KKK construction uses a two level neural network: The first level contains K independent perceptrons, while the second level computes the parity of their K hidden outputs. Each one of the K perceptrons has N weights $w_{k,n}$ (where $1 \leq k \leq K$ and $1 \leq n \leq N$). These weights are integers in the range $\{-L, \ldots, L\}$ that can change over time. Given the N bit input $(x_{k,1}, \ldots, x_{k,N})$ (where $x_{k,n} \in \{-1, +1\}$), the perceptron outputs the sign (which is also in $\{-1, +1\}$) of $\vec{w_k} \cdot \vec{x_k} = \sum_{n=1}^{N} w_{k,n} x_{k,n}$. The output o_k of the perceptron has a simple geometric interpretation: the hyperplane which is perpendicular to the weight vector $\vec{w}$ divides the space into two halves, and the output of the perceptron for input $\vec{x}$ indicates whether $\vec{x}$ and $\vec{w}$ are on the same side of this hyperplane or not (i.e., whether the angle between $\vec{w}$ and $\vec{x}$ is less than or greater than 90 degrees). The output of the neural network is defined as the parity $O = \prod_{k=1}^{K} o_k$ of the outputs of the K perceptrons.

In the KKK scheme the two parties $\mathcal{A}$ and $\mathcal{B}$ start from random uncorrelated weight matrices $\{w_{k,n}\}$. At each round a new random matrix of inputs $\{x_{k,n}\}$ is publicly chosen (e.g., by using a pseudo random bit generator), and each party announces the output of its own neural network on this common input. If the two output bits are the same, the parties do nothing and proceed to the next round; otherwise each party trains its own neural network according to the output of the other party. The training uses the classical Hebbian learning rule to update the perceptron weights. However, each party knows only the parity (rather than the individual values) of the outputs of the other party's perceptrons, and thus the learning rule has to be modified: In the KKK proposal (which is also discussed and justified in [1] and [4]) each party only modifies those perceptrons in his own network whose hidden outputs differ from the announced output. With this correction, KKK observed that for some choices of K, N and L, the weight matrices of the two parties become anti parallel (i.e., $w^{\mathcal{A}}_{k,n} = -w^{\mathcal{B}}_{k,n}$ for all k and n) after a reasonably small number of rounds, and from then on they always generate negated outputs and always update their weights into new anti parallel states. The two parties could become aware of the achieved synchronization by noticing that their announced outputs were always negated for 20-30 consecutive steps. Once their networks become synchronized, the two parties could stop and compute a common cryptographic key by hashing their current weight matrix (or its negation).

3 The Synchronization Process

In this section we explain the synchronization process by using some elementary properties from the theory of random walks in bounded domains. In particular, we analyse the effect of various choices of parameters on the rate of convergence.

The standard Hebbian learning rule forces the mutually learning neural networks into anti parallel states. This is unintuitive, complicates our notation, and makes it difficult to prove convergence by using distance arguments. We thus modify the original scheme in [3], and update the two weight matrices whenever the networks agree (rather than disagree) on some input. We modify several other minor elements, and get a dual scheme in which one of the parties goes through the same sequence of states and the other party goes through a negated sequence of intermediate steps, compared to the original KKK proposal. In this dual scheme the two parties eventually becomes identical (rather than anti parallel). For $K = 3$, the modified learning procedure is defined in the following way: Given random public vectors $\vec{x_1}, \vec{x_2}, \vec{x_3} \in \{-1, 1\}^N$, each party calculates its perceptrons' hidden outputs $o_1 = \mathrm{sgn}(\vec{w_1}\vec{x_1})$, $o_2 = \mathrm{sgn}(\vec{w_2}\vec{x_2})$, $o_3 = \mathrm{sgn}(\vec{w_3}\vec{x_3})$, where $\mathrm{sgn}(x)$ is 1 if $x \geq 0$ and -1 otherwise. It then announces its final output $O = o_1 o_2 o_3$. If $O^{\mathcal{A}} \neq O^{\mathcal{B}}$ the parties end the current round without changing any weights. Otherwise, each party updates only perceptrons for which $o_k = O$ (since the common O is the product of three hidden values, each party updates the weights of either one or three perceptrons). The updated weights of perceptron k are defined by the transformation $\vec{w_k} \leftarrow \vec{\text{bound}}_{-L,L}(\vec{w_k} - o_k\vec{x_k})$, where $\text{bound}_{-L,L}$ changes any coordinate which exceeds the allowed weight bounds back to the bound (i.e., $-L-1$ is changed to $-L$ and $L+1$ is changed to L).

This learning procedure is quite delicate, and changing the identity of the updated perceptrons or the way they are updated either destroys the synchronization process or makes it trivially insecure. The goal of this section is to explain why the two parties converge, and why a third neural network cannot converge to the same weight matrix by following the same learning procedure.

We first consider a highly simplified neural network which consists of a single perceptron with a single weight. The convergence of such networks is completely explained by the existence of the absorbing boundaries $-L$ and L for the range of allowed weight values. Let a_i and b_i be the current weights of the two perceptrons. At each round a new random input $x_i \in \{-1, 1\}$ is chosen, and the parties decide to either ignore it, or to simultaneously move their two weights in the same direction determined by x_i. However, if any weight tries to step beyond the allowed boundaries, it remains stuck at the boundary while the other weight moves towards it (unless it is also stuck at the same boundary). Each weight starts from a random value, and performs a one dimensional random walk which is driven by the common sequence of random inputs. When neither one of the weights is stuck at the boundary, their distance remains unchanged ($|a_{i+1} - b_{i+1}| = |a_i - b_i|$), whereas if one of them is stuck and the other one moves, their distance is reduced by one. Since each random walk is likely to hit the boundary infinitely often, their distance will eventually reduce to zero, and from then on the two random walks will always coincide.

The case of a single perceptron with multiple weights is a simple generalization of this case. The two weight vectors move in the same direction determined by $\vec{x_i}$ in a bounded multidimensional box, and along each coordinate the distance is either preserved or reduced by one. When all these distances are reduced to zero, the two random walks become identical forever.

Unfortunately, single perceptron neural networks can be trivially attacked by any neural network which starts from a random initial state and mimics the operation of the two parties. In fact, except during a short initial period, the state of all these perceptrons is uniquely determined by the (publicly known) sequence of inputs $\vec{x_i}$, and is independent of their initial state. Consequently, the synchronization process of single perceptron neural networks is trivial, and cannot be used to derive a cryptographically secure common key.

The case of neural networks with multiple perceptrons is more complicated, since the two parties may update different subsets of their perceptrons in each round. We thus have to consider a noisy version of the previous convergence argument, in which occasionally the parties perform *uncoordinated moves* which add $\vec{x_i}$ to one of $\mathcal{A}$'s perceptrons but adds zero to the corresponding perceptron of $\mathcal{B}$, which can either increase or decrease the distance between them. Initially there is some weak correlation between $o_k^{\mathcal{A}}$ and $o_k^{\mathcal{B}}$ due to the asymmetry in o caused by cases in which $\vec{w_k}\vec{x_k} = 0$. If the parties make a coordinated move (i.e.

$o_k^{\mathcal{A}} = o_k^{\mathcal{B}}$) then $\vec{w_k^{\mathcal{A}}}$ and $\vec{w_k^{\mathcal{B}}}$ become closer to each other and thus $\vec{w_k^{\mathcal{A}}}\vec{x}_k$ and $\vec{w_k^{\mathcal{B}}}\vec{x}_k$ will have an increased tendency to have the same sign (and thus make a coordinated move) in the next round with a new random input. In particular, if $\vec{w_k^{\mathcal{A}}} = \vec{w_k^{\mathcal{B}}}$ for all k then all their future moves will be coordinated, and thus their weight matrices will remain identical forever. The convergence argument becomes a delicate balance between the reduced distance caused by coordinated moves, the increased distance which may be caused by uncoordinated moves, and the probability of making an uncoordinated move as a function of the current correlation between the weights: If we completely rerandomize the weights whenever the two perceptrons make an uncoordinated move the parties will never converge, but if we do not penalize such failures then any third party will also converge to the same state in the same amount of time.

The claimed basis for the security of the scheme in [3] is the proven fact that given fewer than some number $\alpha(L)N$ of outputs of a parity machine with fixed weights for random inputs it is information theoretically impossible to calculate these weights, and the case of changing weights seems to be even harder. However, the problem of computing the initial weights and the problem of finding the final weights are completely different, and the attacker is only interested in the latter problem. To illustrate this point, consider the simple example of a one dimensional random walk with boundaries. Although it is information theoretically impossible to recover the initial position a_0 from an arbitrarily longer sequence of state signs, it is easy to predict with overwhelming probability all the states from some point onwards.

The other evidence of security given in [3] was the fact that an attacker using the same neural network and a variety of learning rules failed to converge to the same states in the same number of steps as the two parties (in some cases the attacker never converged, and in other cases its convergence was so slow that when the two parties stopped revealing their output bits its state was still completely different). This is a necessary condition for the security of the scheme, but far from being sufficient. However, the cause of this failure is not obvious, and its analysis is very instructive.

Consider an attacker $\mathcal{C}$ who starts from the same parity machine with randomly chosen weights. At each step she computes her hidden outputs $o_1^{\mathcal{C}}, \ldots, o_K^{\mathcal{C}}$ with respect to the publicly available input. If the parties announce different public outputs $O^{\mathcal{A}} \neq O^{\mathcal{B}}$, $\mathcal{C}$ knows that $\mathcal{A}$ and $\mathcal{B}$ do not update their weights, and thus she also skips the current round without updating her weights. If $O^{\mathcal{A}} = O^{\mathcal{B}}$ then $\mathcal{C}$ tries to mimic the behavior of $\mathcal{A}$ and $\mathcal{B}$ by guessing which perceptrons should be updated, and she uses her hidden outputs to do so using the same rule as the two parties. In [3] it was empirically observed that this strategy does not allow $\mathcal{C}$ to converge even if she starts from a state which is strongly correlated to that of $\mathcal{B}$. In order to understand why $\mathcal{C}$ fails while $\mathcal{A}$ and $\mathcal{B}$ succeed, we have to compare the probability that $\mathcal{B}$ and $\mathcal{C}$ make the same update as $\mathcal{A}$. Consider for example a neural network with $K = 2$, i.e. each party has two perceptrons. Let's define $p_k = \Pr[o_k^{\mathcal{A}} = o_k^{\mathcal{B}}]$ for random inputs $\vec{x_k}$, and for the sake of simplicity assume that it is the same for both perceptrons ($p_1 = p_2 = p$), and for both pairs $(\mathcal{A},\mathcal{B})$ and $(\mathcal{A},\mathcal{C})$ (note that the outputs of different units are independent since they are functions of independent random inputs). There are four possible scenarios: $(o_0^{\mathcal{A}} = o_0^{\mathcal{B}}, o_1^{\mathcal{A}} = o_1^{\mathcal{B}})$, $(o_0^{\mathcal{A}} \neq o_0^{\mathcal{B}}, o_1^{\mathcal{A}} = o_1^{\mathcal{B}})$, $(o_0^{\mathcal{A}} = o_0^{\mathcal{B}}, o_1^{\mathcal{A}} \neq o_1^{\mathcal{B}})$ and $(o_0^{\mathcal{A}} \neq o_0^{\mathcal{B}}, o_1^{\mathcal{A}} \neq o_1^{\mathcal{B}})$, with probabilities p^2, $p(1-p)$, $(1-p)p$ and $(1-p)^2$ respectively. Note that in the second and the third scenarios $O^{\mathcal{A}} \neq O^{\mathcal{B}}$, and thus they will never happen in a round in which $\mathcal{A}$ and $\mathcal{B}$ decide to update their weights. However, such scenarios are possible for $(\mathcal{A},\mathcal{C})$, since their outputs can be different when $\mathcal{C}$ is forced to move, and from her perspective it is a bad idea either to keep the weights unchanged or to update the wrong collection of perceptrons. In other words, the crucial difference between the two parties and the attacker is that the parties can choose the most beneficial points in time at which to update their weights, whereas the passive eavesdropper cannot influence this choice. Consequently, the probability that $(\mathcal{A},\mathcal{B})$ make a coordinated move is $\frac{p^2}{p^2+(1-p)^2}$, while the probability that $(\mathcal{A},\mathcal{C})$ make a coordinated move is p. Since $0 < p < 1$, it is easy to see that $\frac{p^2}{p^2+(1-p)^2} > p$. Figure 1 shows the probability of making a coordinated move as a function of p for various numbers of perceptrons K. It is clear from this figure that the choice of $K = 2$ is optimal for $(\mathcal{A},\mathcal{B})$. We already demonstrated a difference of behaviour between the $(\mathcal{A},\mathcal{B})$ and $(\mathcal{A},\mathcal{C})$ cases, but in order to show why such a difference allows the pair $(\mathcal{A},\mathcal{B})$ to converge with very high probability but the pair $(\mathcal{A},\mathcal{C})$ to converge only with negligible probability, we have to consider the *speed* of convergence.

First we have to define the notion of closeness between perceptrons: $\rho(\vec{w},\vec{w'}) = \Pr_x[\text{sgn}(\vec{w}\vec{x}) = \text{sgn}(\vec{w'}\vec{x})]$ for a random input $\vec{x}$ (by definition, $0 \leq \rho \leq 1$). To calculate the expected change of ρ after one round in which the parties update their weights, we use the following formula:

$$\mathrm{E}[\Delta\rho] = \Pr[\top]\Delta\rho_\top + \Pr[\perp]\Delta\rho_\perp,$$

where we use $\top$ to denote coordinated moves (in which the hidden outputs are the same) and $\perp$ to denote uncoordinated moves. For the sake of simplicity consider again the case of $K = 2$ and $\rho(\vec{w}_1,\vec{w'_1}) = \rho(\vec{w}_2,\vec{w'_2})$. Using a large number of numerical experiments we found the forms of $\rho'_\top$ and $\rho'_\perp$. The results are shown in figure 2, which describes the closeness before and after a coordinated and an uncoordinated move in the various experiments. In order to combine these results we approximated $\Delta\rho_\top$ and $\Delta\rho_\perp$ by two third degree polynomials which are described in figure 3. Using this approximation, figure 4 shows the expected increase of ρ as a function of the current value of ρ for the whole system.

Using figure 4 we can easily explain why the pair $(\mathcal{A},\mathcal{B})$ quickly converges: $\Delta\rho(\vec{w}^{\mathcal{A}},\vec{w}^{\mathcal{B}}) > 0$, so each step is expected to increase ρ until eventually $\rho = 1$. However, for $(\mathcal{A},\mathcal{C})$ the drift is positive only before approximately 0.8, but if $\mathcal{C}$ gets any closer then her drift is negative and thus her strategy is counterproductive. This explains the experimental result described in [3] — even if $\rho(\vec{w}^{\mathcal{B}},\vec{w}^{\mathcal{C}})$ is relatively high it tends to decrease, and thus such an adversary has a negligible probability to converge to the common states of $\mathcal{A}$ and $\mathcal{B}$.

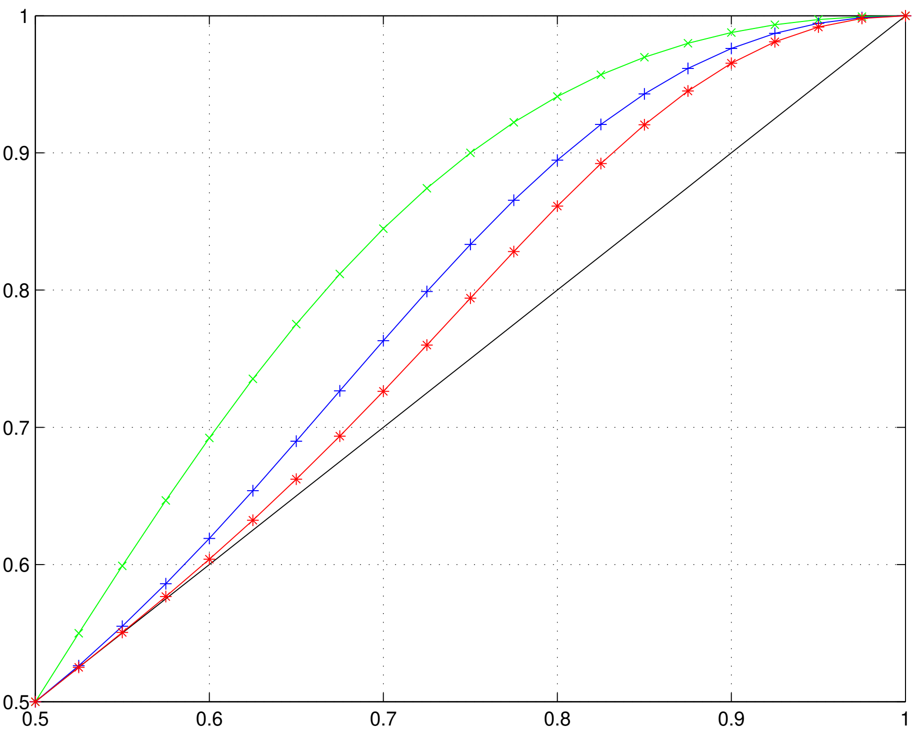

Fig. 1. The probability of making a coordinated move for one perceptron when $K = 2$ ("$\times$"), $K = 3$ ("$+$"), or $K = 4$ ("$\star$")

4 Cryptanalytic Attacks

The security of the scheme was analysed in [3] in terms of its robustness against a particular attacker who simulates the actions of the two parties. The security of the scheme against such an attack was experimentally verified in [3], and mathematically explained in the previous section. In this section we consider other types of attacks, and show that the KKK scheme can be broken by three completely different cryptanalytic techniques. Since the proposed cryptographic scheme is very different from standard schemes, the attacks are also somewhat unusual.

4.1 The Genetic Attack

Since the cryptosystem is based on the biological notion of neural networks, we decided to apply a biologically motivated attack based on genetic algorithms. The general idea behind any genetic algorithm is to simulate a population of virtual organisms and to impose evolutionary rules which prefer organisms with certain

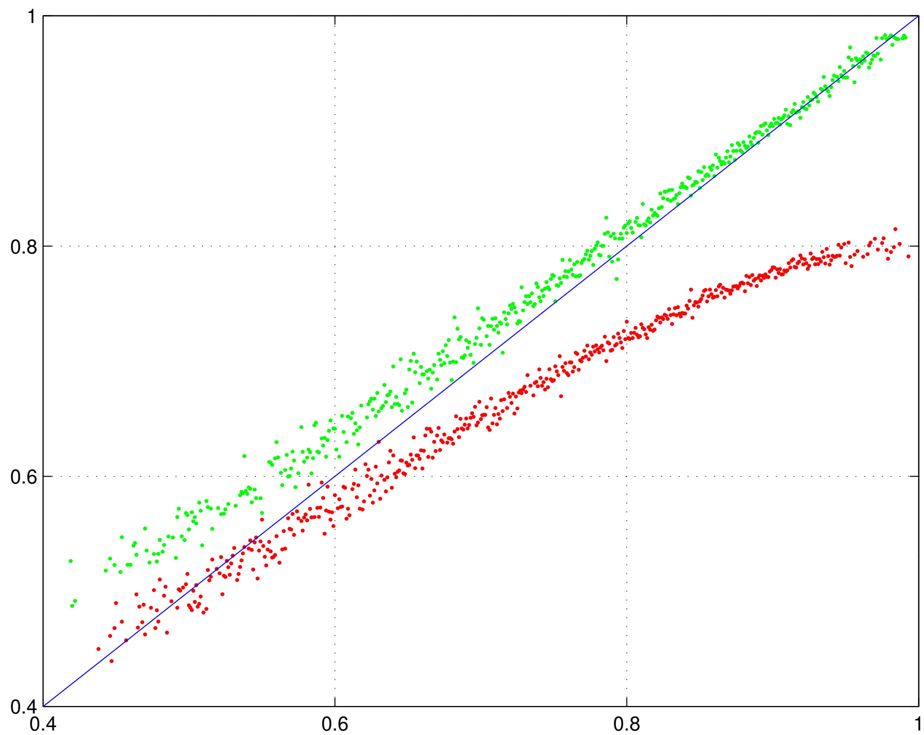

Fig. 2. Experimental form of $\rho'_\perp$ (the lower distribution) and $\rho'_\top$ (the upper distribution) for $L = 3$ and $N = 101$.

desirable properties. The literature contains very few successful cryptanalytic applications of such techniques, but a recent exception is the simulated annealing attack on the PPP scheme described in [2].

In our attack, we simulate a large population of neural networks with the same structure as the two parties, and train them with the same inputs. At each stage about half the simulated networks announce an output of $+1$, and half announce an output of -1. Networks whose outputs mimic those of the two parties breed and multiply, while unsuccessful networks die.

We start the attack with one network with randomly chosen weights. At each step a population of networks evolves according to three possible scenarios:

- $\mathcal{A}$ and $\mathcal{B}$ have different outputs $O^{\mathcal{A}} \neq O^{\mathcal{B}}$, and thus do not change their weights. Then all the attacker's networks remain unchanged as well.
- $\mathcal{A}$ and $\mathcal{B}$ have the same outputs $O^{\mathcal{A}} = O^{\mathcal{B}}$, and the total number of attacking networks is smaller than some limit M. In this case there are 4 possible combinations of the hidden outputs agreeing with the final output. So, the attacker replaces each network $\mathcal{C}$ from the population by 4 variants of itself,

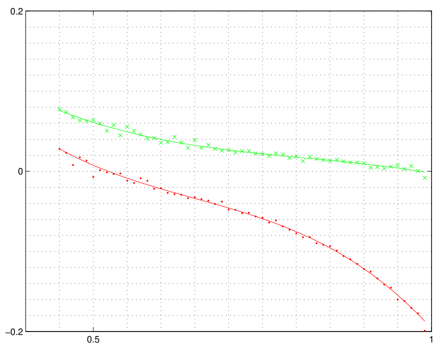

Fig. 3. Approximation of $\Delta\rho_\perp$ (the lower distribution) and $\Delta\rho_\top$ (the upper distribution).
The polynomials are $f_c(p) = -0.6364p^3 + 1.5516p^2 - 1.3409p + 0.4231$ and $f_n(p) = -1.8666p^3 + 3.6385p^2 - 2.5984p + 0.6304$.

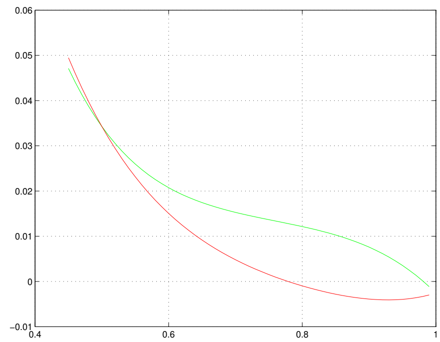

Fig. 4. The speed of convergence for $(\mathcal{A}, \mathcal{B})$ (the upper line) and $(\mathcal{A}, \mathcal{C})$ (the lower one) for $L = 3$, $N = 101$ and $K = 2$.

$\mathcal{C}_1, \ldots, \mathcal{C}_4$. which are the results of updating $\mathcal{C}$ with the standard learning rule but pretending that the hidden outputs were equal to each one of these combinations.

– $\mathcal{A}$ and $\mathcal{B}$ have the same outputs $O^{\mathcal{A}} = O^{\mathcal{B}}$ but the total number of simulated networks is larger than M. In this case the attacker computes the outputs of all the networks, deletes the unsuccessful networks whose output is different from $O^{\mathcal{A}}$, and updates the weights in the successful networks by using the standard learning rule with the actual hidden outputs of the perceptrons.

Shortly after $\mathcal{A}$ and $\mathcal{B}$ synchronize for the first time, they know this fact, and the attacker uses the same test to check whether any one of his networks has the same weights as $\mathcal{A}$. For the recommended choice of parameters ($K = 3$, $N = 101$, $L = 3$), we tried the attack with a threshold of $M = 2500$ networks, and in more than 50% of our tests at least one of the attacking networks $\mathcal{C}$ became synchronized with $\mathcal{A}$ even before $\mathcal{A}$ and $\mathcal{B}$ themselves became fully synchronized.

We successfully applied this attack to several variants of the KKK scheme using different parameters as well as different rules for updating the weights and computing the output. The attack is particularly effective for variants in which the genetic attack has a small local branching rate (e.g., when $K = 2$).

4.2 The Geometric Attack

We have already described the geometric interpretation of the action of a perceptron. Now we are going to exploit this characterization in order to gain useful information about the unknown weights of neural networks which are defined as the parity of several perceptrons.

Each input can be viewed as K random hyperplanes $X_1, \ldots, X_K$ corresponding to K perceptrons. Each X_i is a hyperplane

$$f_i(z_1, \ldots, z_N) = \sum_{j=1}^{N} x_{ij} \cdot z_j = 0$$

in the N-dimensional discrete space $\mathbf{U} = \{-L, \ldots, L\}^N$. The weights of a network could be also viewed as K points $W_1, \ldots, W_K$ in $\mathbf{U}$, $W_i = (w_{i1}, \ldots, w_{iK})$, while the i-th hidden output is just the side of the half-space (with respect to X_i) which contains W_i.

Consider an attacking network $\mathcal{C}$ that is close enough to the unknown network $\mathcal{A}$ but has a different output for a given input. In fact they have either 1 or 3 different hidden outputs. The second case is less likely to occur so we assume that only one hidden output of the network $\mathcal{C}$ is different from the corresponding hidden output of $\mathcal{A}$. Consequently, only one pair $(W_i^{\mathcal{A}}, W_i^{\mathcal{C}})$ is separated by the known input hyperplane X_i. Of course, we are interested in detecting its index i.

If the points $W_i^{\mathcal{C}}$ and $W_i^{\mathcal{A}}$ are separated by X_i then the distance between them is greater than the distance from $W_i^{\mathcal{C}}$ to the hyperplane X_i. $W_i^{\mathcal{C}}$ and $W_i^{\mathcal{A}}$ are close to each other, so the distance from $W_i^{\mathcal{C}}$ to X_i has to be small. On the other hand, if $W_i^{\mathcal{C}}$ and $W_i^{\mathcal{A}}$ are in the same half-space with respect to X_i then they are more likely to be far away from the random input X_i (even though we know that they are close to each other). We thus guess that the index of the incorrect hidden output is the i for which $W_i^{\mathcal{C}}$ is closest to the corresponding hyperplane X_i, where we compute the distance by $\rho(W_i^{\mathcal{C}}, X_i) = |f_i(W_i^{\mathcal{C}})|$.

Formally, the attacker constructs a single neural network $\mathcal{C}$ with the same structure as $\mathcal{A}$ and $\mathcal{B}$, and randomly initializes its weights. At each step she trains $\mathcal{C}$ with the same input as the two parties, and updates its weights with the following rules:

– If $\mathcal{A}$ and $\mathcal{B}$ have different outputs $O^{\mathcal{A}} \neq O^{\mathcal{B}}$, then the attacker doesn't update $\mathcal{C}$.
– If $\mathcal{A}$ and $\mathcal{B}$ have the same outputs $O^{\mathcal{A}} = O^{\mathcal{B}}$ and $O^{\mathcal{C}} = O^{\mathcal{A}}$, then the attacker updates $\mathcal{C}$ by the usual learning rule.
– If $\mathcal{A}$ and $\mathcal{B}$ have the same outputs $O^{\mathcal{A}} = O^{\mathcal{B}}$ and $O^{\mathcal{C}} \neq O^{\mathcal{A}}$, then the attacker finds $i_0 \in \{1, \ldots, K\}$ that minimizes $|\sum_{j=0}^{N} w_{ij}^{\mathcal{C}} \cdot x_{ij}|$. The attacker negates $o_{i_0}^{\mathcal{C}}$ and updates $\mathcal{C}$ assuming the new hidden bits and output $O^{\mathcal{A}}$.

Different attackers starting from randomly chosen states behave independently and thus multiple attackers have a higher probability to be successful. We tried this attack with 100 random initial states and at least one of them synchronized with $\mathcal{A}$ faster than $\mathcal{B}$ with probability $> 90\%$.

4.3 The Probabilistic Attack

As was described in the previous section, it is much easier to predict the position of a point in a bounded multidimensional box after several moves in its random walk than to guess its original position. A simple way to do it is to consider each coordinate separately, and to associate with each possible value i in the interval $\{-L,\ldots,L\}$ the probability $p_t(i) = \Pr[x_t = i]$. Initially $\forall i, p_0(i) = \frac{1}{2L+1}$ and after each move $p_{t+1}(i) = \sum_j p_t(j)$, where j are such that if $x_t = j$ then $x_{t+1} = i$. Applying this technique to the original scheme we face the problem that the moves are not known — the attacker does not know which perceptrons are updated in each round. Fortunately, if we know the distribution of the probabilities $P_{k,n,i} = \Pr[w_{k,n} = i]$ then using dynamic programming we can calculate the distribution[1] of $\vec{w_k}\vec{x_k}$ for a given vector $\vec{x_k}$ and thus the probabilities $u_k(s) = \Pr[o_k = s]$. Using these probabilities we can calculate the conditional probabilities $U_k = \Pr[o_k = 1|O]$:

$$U_k = \frac{\sum_{(\alpha_1,\ldots,\alpha_K):\prod_i \alpha_i = O \& \alpha_k = 1} \prod_i u_i(\alpha_i)}{\sum_{(\alpha_1,\ldots,\alpha_K):\prod_i \alpha_i = O} \prod_i u_i(\alpha_i)},$$

because O is publicly known. We can now update the distribution of the weights: $P^{t+1}_{k,n,i} = \sum_j P^t_{k,n,j} \Pr[w^t_{k,n} = j \Rightarrow w^{t+1}_{k,n} = i]$, where $\Pr[w^t_{k,n} = j \Rightarrow w^{t+1}_{k,n} = i]$ is calculated using U_k. Experiments show that in most cases, when $\mathcal{A}$ and $\mathcal{B}$ converge to a common $\hat{w}_{k,n}$, the probabilities $\Pr[w_{k,n} = \hat{w}_{k,n}] \approx 1$ and thus the adversary can easily find $\hat{w}_{k,n}$ when $\mathcal{A}$ and $\mathcal{B}$ decide to stop the protocol.

References

1. M. Biehl, N. Caticha, "Statistical Mechnics of On-Line Learning and Generalization", The Handbook of Brain Theory and Neural Networks, 2001.
2. John A. Clark, Jeremy L. Jacob, "Fault Injection and a Timing Channel on an Analysis Technique", Proceedings of Eurocrypt 2002, p. 181.
3. Ido Kanter, Wolfgang Kinzel, Eran Kanter, "Secure exchange of information by synchronization of neural networks", Europhys., Lett. 57, 141, 2002.
4. M. Opper, W. Kinzel, "Statistical Mechanics of Generalization", Models of Neural Networks III, 151-20, 1995.
5. `http://rfic.ucsd.edu/chaos`

[1] Note that when we calculate the distribution of $\vec{w_k}\vec{x_k}$, we assume that the random variables $w_{k,n}$ are independent. This seems to be true for the original scheme.