Electronic Preprint for Journal of Information Processing Vol.26

Regular Paper

A Privacy Preserving Protocol for Cloud-Based Implementation of Aware Agents

Masato Hashimoto[1,a)] Qiangfu Zhao[1,b)]

Received: September 28, 2017, Accepted: March 6, 2018

Abstract: Aware agents (A-agents) are programs that can be aware of various situations and support human decisions. The aim of this study is to implement A-agents on portable/wearable computing devices (P/WCDs) like smartphones. Since P/WCDs usually have limited resources, it is difficult to implement many A-agents together in one P/WCD. Cloud-based system is a solution, but this kind of system has privacy problem. To use a cloud server while preserving privacy, we propose a new protocol that has four features, namely, 1) divide each A-agent into two parts, and implement the computationally expensive part using the cloud server; 2) encrypt the data before sending them to the server; 3) share the same black-box computing model on the server side by various A-agents; and 4) make the final decisions on the P/WCD side with selected results obtained from the server. Experimental results show that the performance of the A-agents does not change significantly even if they share the same black-box model. In addition, the P/WCD can be more energy efficient. Therefore, the proposed protocol can be very useful for improving the usability of P/WCDs.

Keywords: neural network, extreme learning machine, privacy preserving, mobile computing device, wearable computing device, cloud computing

1. Introduction

In this study, we define aware agents, or A-agents in short, as computer programs for supporting human users to make decisions in their daily lives. The A-agents can be aware of user situations, intentions, preferences, etc., and provide useful information to the user. In practice, the A-agents are realized by machine learning models, and many learning algorithms proposed in the literature can be used to customize or personalize the A-agents. In this study, we aim to implement the A-agents on portable/wearable computing devices (P/WCDs) like smart phones or smart watches. Since resources (e.g. CPU, memory, and battery) of P/WCDs are usually limited, it is difficult to implement many A-agents together in one P/WCD. It will be more difficult if we want to implement high performance A-agents. Thus, how to implement the A-agents more efficiently or economically is a challenging problem.

Figure 1 is the concept of the system based on A-agents. For developing A-agents as mobile applications, we suppose the following flow (see **Fig. 2**):

(1) Train the A-agents on developer's or user's personal computer (PC).
(2) Distribute the A-agents to the mobile device.
(3) Use the A-agents as daemons. If an A-agent observes a user datum, it classifies the datum and provides information.

To solve the problem, we may try to design compact (i.e., with low calculation cost) and high performance A-agents and implement them in the P/WCD (**Fig. 3**), or use a cloud server to implement all A-agents based on a client-server model (**Fig. 4**). In our earlier study, we tried the first approach and proposed the

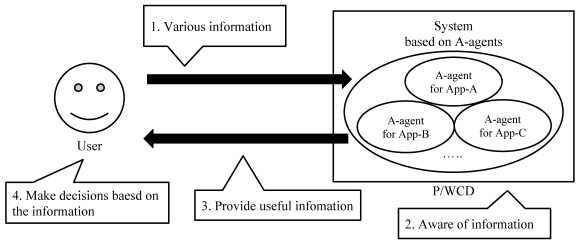


Fig. 1 An example of systems based on A-agents for user assistance.

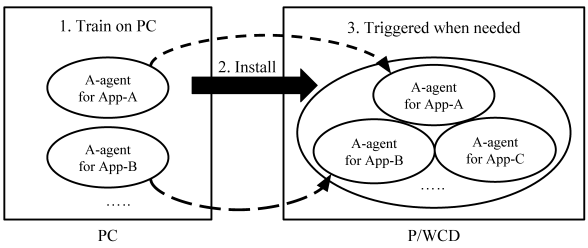


Fig. 2 A-agents set up flow.

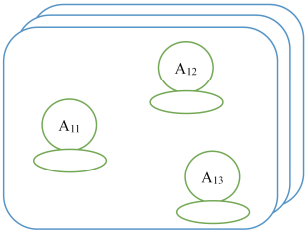


Fig. 3 The all-in-P/WCD approach.

[1] University of Aizu, Aizu-wakamatsu, Fukushima 965–0005, Japan
[a)] m5201129@u-aizu.ac.jp
[b)] qf-zhao@u-aizu.ac.jp

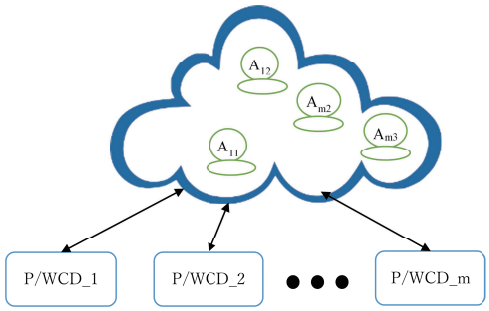


Fig. 4 The all-in-server approach.

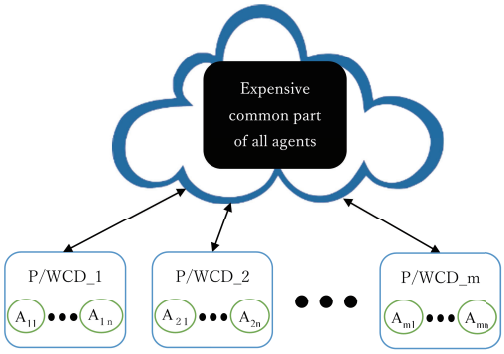


Fig. 5 The proposed approach.

decision boundary making (DBM) algorithm [1], [2]. The DBM algorithm uses a relatively expensive model as a bridge to obtain a high performance compact model. The flow for training is as follows. First, we train the expensive model M_0 to achieve a high performance. Then, we generate new data around the decision boundary based on M_0, and add them to the training set. Using the new training set, we can train a compact model M_1 to approximate M_0. In our research, a multilayer perceptron (MLP) is used for M_1, and a support vector machine (SVM) is used for M_0. Experiments on several public databases have verified the performance of the DBM algorithm [3]. However, if we want to implement many A-agents (one for each application) in a single P/WCD, this all-in-P/WCD approach is still not enough.

The all-in-server approach is a basic method for implementing many mobile or web applications. In this approach, a client (e.g., a P/WCD or web browser) is used as a user interface (UI) for inputting and outputting data, and all computations are conducted in the cloud server. The clients and server are connected by a network (e.g., internet). By this approach, we can implement any number of A-agents without significantly increasing the load of the P/WCD. However, this approach also poses information leakage and privacy invasion problems [4]. If the cloud server holds the user data and the A-agent model, it is easy for the server to analyze the user intention. Even if the server itself might be trustable, a malicious person can see sensitive personal data easily, if he/she can visit the system using some illegal method. In this sense, the all-in-server approach may not be trustable.

A well-known technology for user data protection is (fully) homomorphic encryption (HE). For example, HE can be used to implement a neural network (NN) using a cloud server [5], [6]. The HE-based NN can provide secure classification because all computations are performed on the encrypted data. For example, in Cryptonets proposed in Ref. [6], the client first encrypts a datum using HE with a public key and sends it to a cloud server that hosts an NN. The cloud server then conducts all computations of the NN while keeping the datum and all intermediate results in encrypted form, and sends back the final result, which is also encrypted, to the client. Finally, the client decrypts the result with a secret key. This method is secure in the sense that the server can only see the encrypted data. However, the computational cost is usually high to implement the HE-based NN. In addition, the size of the encrypted datum is also larger than that of the original one. There is another serious problem in using the HE-based approach. That is, the computing model (e.g., the NN) must be hosted by the server, and therefore, some malicious person may analyse the data (e.g., the statistics, the data type, etc.) indirectly based on the computing model and the input-output pairs. Therefore, the HE-based approach is not P/WCD oriented and may not be secure.

Recently, stealing machine learning model attack, also known as model extraction attack, was shown by Florian Tramèr et al. [7]. Based on their research, some malicious third person can copy and analyse the machine learning model (i.e., the A-agents) using some request queries for prediction or classification, in existing machine learning cloud services like Amazon ML and BigML. Therefore, the all-in-server approach itself may not be trustable.

To solve the problem, we proposed a privacy preserving protocol in Ref. [8]. The main point of the proposed protocol is to realize each A-agent using an extreme learning machine (ELM) which was proposed in Refs. [9] and [10]. Briefly speaking, ELM is a single hidden layer MLP. The basic idea of the protocol is to divide ELM into two parts, and implement these two parts separately by the client (P/WCD) and the server (see **Fig. 5**). The server holds the weight matrix ($\boldsymbol{W}$) of the hidden layer, and the P/WCD holds the weight matrix ($\boldsymbol{\beta}$) (which is a vector for two-class classification problems) of the output layer. Since the hidden layer weight matrix $\boldsymbol{W}$ of an ELM is generated at random and fixed, the server or some malicious third party cannot interpret the decision making model based on information available on the server side, therefore we can call it black-box computing. Thus, the proposed protocol can protect the user decision model effectively.

To protect the personal data, we may use HE to encrypt the data and decrypt the results on the client (P/WCD) side. To improve the computational efficiency, however, we do not use HE, but use the transposition cypher instead. That is, each datum, which is usually represented as a real vector, is encrypted via permutation by the client. In addition, instead of decrypting the results, we generate a different $\boldsymbol{\beta}$ for each transposition cypher key, and save it in the client. This way, we can make decisions efficiently while protecting the user data and the user computing model.

To further improve the usefulness of the proposed protocol, we proposed two methods in Ref. [11]. The first method is to share the same hidden layer weight matrix for various A-agents (i.e., agents for different classification tasks or for different users). That is, we can generate a large $\boldsymbol{W_0}$ at random, fix it, and use a sub-matrix of $\boldsymbol{W_0}$ for each specific A-agent. This method not

only makes the user decision model more secure, but also reduces the response time of the server because it is not necessary to load or re-load $\boldsymbol{W}$ for requests from different clients. The second method given in Ref. [11] is to use part of the results obtained from the server to make the final decision on the client side. This method makes it more difficult for the malicious person to analyse the user intention because the input-output relation cannot be observed on the server side.

This paper is the journal version of two conferences papers [8], [11]. In this paper, we have summarized the background and some related concepts into one place, and extended the explanations about the protocol, the training and testing phases. This journal version is more self-contained and can be more useful for the readers to understand the whole story. In addition, we have added more results (e.g., results for more datasets, and results for battery consumption). As for the system itself, we have upgraded the method for fast data transmission (i.e., changed the method from JSON to Protcol buffer), and the method for fast memory access inside the P/WCD.

This paper is organized as follows. In Section 2, we introduce very briefly ELM and its learning algorithm. Section 3 explains the proposed protocol in detail. Section 4 investigates the performance, the computational cost, and the battery consumption of the P/WCD via experiments on several public datasets. Section 5 draws some conclusions and suggests some topics for future work.

2. Preliminaries

To make this paper relatively self-contained, here we introduce ELM very briefly. For more detail, the readers may refer to Ref. [9] or its improved versions. The most fundamental form of an ELM is a single hidden layer MLP. The only difference between an ELM and a normal MLP is that the hidden neuron weights are given at random and fixed. For training, only the output neurons are tuned based on the training data. Therefore, training of an ELM is usually more cost efficient. In this study, we just adopt the training algorithm given in Ref. [9].

For convenience of discussion, here we introduce some notations. We use N_f, N_h, $\boldsymbol{x}$, $\boldsymbol{W}$, and $\boldsymbol{\beta}$ to denote, respectively, the dimension of the feature space, the number of hidden neurons, the input vector, the weight matrix of the hidden layer, and the weight matrix of the output layer. For two-class problems, $\boldsymbol{\beta}$ is a vector. Here, we consider two-class problems only because all discussions can be generated to multi-class problems straightforwardly. In addition, we use the augmented input defined by $\boldsymbol{x} = (x_1 \cdots x_{N_f}\ 1)^t$, and therefore the weight vector of the ith hidden neuron is $\boldsymbol{w}_i = (w_{i1} \cdots w_{i,N_f}\ b_i)^t$, where b_i is the bias. Note that the hidden layer weight matrix can also be represented by $\boldsymbol{W} = (\boldsymbol{w}_1 \cdots \boldsymbol{w}_{N_h})$. Based on these notations, the output function of an ELM is give below.

$$f(\boldsymbol{x}) = \mathrm{sign}(\mathbf{h}(\boldsymbol{x}) \cdot \boldsymbol{\beta}) \tag{1}$$

where $\mathbf{h}(\boldsymbol{x})$ is the output vector of the hidden layer given by

$$\mathbf{h}(\boldsymbol{x}) = \boldsymbol{G}(\boldsymbol{W}^{\boldsymbol{t}} \cdot \boldsymbol{x}) = [g(z_1) \ \cdots \ g(z_{N_h})]^t \tag{2}$$

and where $z_i = \boldsymbol{w}_i^t \cdot \boldsymbol{x}$ and $g(z_i)$ are, respectively, the effective input and the output of the i-th hidden neuron, and g is the activation function defined by

$$g(z) = \frac{1}{1 + \exp(-\lambda \cdot z)} \tag{3}$$

where λ is a positive real number.

To train an ELM, we first define the training set as follows:

$$\Omega = \left\{(\boldsymbol{x}_j, t_j) \mid \ \boldsymbol{x}_j \in \boldsymbol{R}^{N'_f}, t_j \in \{-1, 1\}, j = 1, ..., N_d\right\} \tag{4}$$

where N_d is the number of training data, $N'_f = N_f + 1$ is the dimension of the augmented feature vector, $\boldsymbol{x}_j$ is the j-th datum, and t_j is the desired output or the teacher signal for the j-th datum. The training data are often represented in a matrix form as follows:

$$\boldsymbol{X} = [\boldsymbol{x}_1 \ \cdots \ \boldsymbol{x}_{N_d}]. \tag{5}$$

The training process is given below.

(1) Initialize the hidden layer weight matrix $\boldsymbol{W}$ at random. In our study, the range of each weight is $[-1, 1]$.

(2) Calculate the output vector of the hidden layer.

$$\begin{aligned} \boldsymbol{H} &= [\mathbf{h}(\boldsymbol{x}_1) \ \cdots \ \mathbf{h}(\boldsymbol{x}_{N_d})]^t \\ &= [\boldsymbol{G}(\boldsymbol{W}^{\boldsymbol{t}} \cdot \boldsymbol{x}_1) \ \cdots \ \boldsymbol{G}(\boldsymbol{W}^{\boldsymbol{t}} \cdot \boldsymbol{x}_{N_d})]^t \end{aligned} \tag{6}$$

(3) Find the output weight vector.

$$\boldsymbol{T} = \boldsymbol{H} \cdot \boldsymbol{\beta} \tag{7}$$

$$\boldsymbol{\beta} = \boldsymbol{H}^+ \cdot \boldsymbol{T} \tag{8}$$

where $\boldsymbol{H}^+$ is the Moor-Penrose generalized inverse matrix of $\boldsymbol{H}$, and

$$\boldsymbol{T} = (t_1 \cdots \ t_{N_d})^t. \tag{9}$$

Theoretically speaking, the neural network model used in this paper is able to realize any aware agents, because an MLP is known to be a universal approximator. However, in this paper, we consider only A-agents that can be implemented in a P/WCD, and that can be used for decision support in the user's daily life. For the time being, we do not consider agents for "optimization" (i.e., for finding the best solution for a given problem). An A-agent is trained based on permutated user data, and the weights of the output layer are saved in the P/WCD. The agent works as a "daemon." That is, it is triggered only when needed (e.g., the P/WCD captures a new "health related datum" from the user's smart watch, or a new "situation related datum" from the user's smart home). A triggered agent first encrypts the datum, sends it to the server, and makes a decision (e.g., health/home condition is normal or abnormal) based on the results received from the server.

3. ELM-Based Privacy Preserving Protocol

The main point of the protocol proposed in our study is to use ELM for protecting the agent model [8], [11]. In the proposed protocol, the hidden neuron weights, which are given at random, are put in the cloud server, and the weights for the output neuron are put in the P/WCD. It is impossible to estimate the agent model using information available on the server side only. To

make the protocol more practically useful, we also introduced three methods for protecting the user data, for sharing the same hidden layer matrix between various agents/applications, and for protecting the user intention. These three methods are discussed in detail below.

3.1 Method for Protecting the Data

In any cloud-based computing, protection of user data is a basic request. The most commonly known technology for this purpose is (fully) homormophic encryption (HE). However, HE in its current status is not suitable for P/WCD because it is computationally expensive. In our study, we propose to use a transposition cipher. That is, for any input feature vector, we just encrypt it by using permutation. In the following discussion, the key for a transposition cipher is a vector whose elements are indices of the encrypted vector. We call the key for a transposition cipher the "trans-key." When the feature vector to encrypt is $\boldsymbol{x} \in \boldsymbol{R}^{N'_f}$, the trans-key is $K_t = (k_1, k_2, ..., k_{N'_f})$, then the d-th elements of $\boldsymbol{x}$ will be the c-th element of the encrypted vector if $k_c = d$ $(1 \leq c \leq N'_f)$. For example, if $\boldsymbol{x} = (x_1, x_2, x_3)$ and $K_t = (2, 3, 1)$, the encrypted vector will be $Enc(\boldsymbol{x}, K_t) = (x_2, x_3, x_1)$.

In practice, for any given classification problem, we can generate many trans-keys at random, and design the same number of models for this problem. These models share the same hidden layer weight matrix, but have different output layer weight vectors. The former is stored in the cloud server, and the latter are stored in the P/WCD. For any input, if we encrypt it using the i-th trans-key, we can get the correct answer by using the i-th output layer weight vector. Thus, even if we do not decrypt the results obtained from the server, we can make correct decisions.

We may consider three types of attacks. One is on the client side, the second is on the server side, and the third is on the communication channel side. In this research, we assume that the trans-key (along with other keys) are stored in a secure region in the P/WCD, and we do not consider attack to the client or P/WCD. The main concern here is to protect the user data from some malicious third party on the server side. Denoting the number of inputs (features) by N_f, there are $N_f!$ possible patterns for the permutation. Theoretically, it is difficult for the third party to understand the original input vector if the trans-key is unknown and if N_f is large. In our research, we consider only "homogeneous" data. That is, all elements of the input vector have the same or very similar "statistical properties" (e.g., dynamic range, mean, variance, etc.). In this sense, permutation alone is quite safe. If each element of the input vector has a unique property, permutation is not good because the third party may find the original features using their statistical properties. One way to solve the problem is to normalize the original input vector, so that all elements share the same (or similar) statistical properties. In fact, this way is useful not only for security, but also recommended for improving the performance of machine learners (e.g., SVM, MLP, etc.). To protect the user data from third party attacks on the communication channel side, we should use a secure method like HTTPS [12] for data communication between the server and the P/WCD. In fact, HTTPS is now recommended for all web-based applications.

3.2 Method for Sharing the Hidden Layer Weight Matrix

The basic idea here is to generate a very large random weight matrix $\boldsymbol{W_0}$, and store it in the cloud server. Different tasks from different users can share $\boldsymbol{W_0}$ by using a sub-matrix of $\boldsymbol{W_0}$. A sub-matrix of $\boldsymbol{W_0}$ is defined by its position (i.e., row number and column number) and size (i.e., the number of hidden neurons and the number of features). The size of the sub-matrix for a certain task is $N'_f \times N'_h$ where N'_h is the number of hidden neurons (to be defined in the next sub-section). We call the position of the sub-matrix the "pos-key." A pos-key contains 2 elements, namely the row number and the column number. For example, if the pos-key is $K_p = (e, f)$, where e and f are integers, then the (e, f)-th element of $\boldsymbol{W_0}$ will be the "upper-left" or the (1,1)-th element of the sub-matrix. When the size of $\boldsymbol{W}_0$ is $W_x \times W_y$, the range of e is $1 \leq e \leq W_x - N'_f$, and that of f is $1 \leq f \leq W_y - N'_h$.

There are mainly two advantages by sharing the random hidden weights. The first one is cost reduction for loading/reloading the hidden weight. The second advantage is to hide the user intention. That is, the server or some third party cannot see what kind of model a user/application uses.

Similar to the trans-keys, we can generate many pos-keys for a given problem, and extract a sub-matrix $\boldsymbol{W}$ from $\boldsymbol{W_0}$ for each pos-key. Corresponding to each sub-matrix $\boldsymbol{W}$, we can design a model, and save the output layer weight vector on the P/WCD side.

3.3 Method for Protecting the User Intention

If we use all results obtained from the cloud server for making the final decisions, some malicious person may still try to guess the user intention by observing the relation between the input and the output of the hidden layer. To protect the user intention further, we can drop some of the results obtained from the server, and use only some selected results for making the final decisions. That is, we can drop some elements of $\boldsymbol{h} \in \boldsymbol{R}^{N'_h}$ and generate $\boldsymbol{h'} \in \boldsymbol{R}^{N_h}$. That is, the length of $\boldsymbol{h'}$ is smaller than that of $\boldsymbol{h}$ $(N_h \leq N'_h)$. Here, we define a redundancy rate r $(1 \leq r \leq t$, $t \in \boldsymbol{R})$, and $N'_h = r \cdot N_h$, where N_h is the number of hidden neurons actually used by the agent, and N'_h is the number of hidden neurons for computation. We call the key for selecting against some un-used elements of $\boldsymbol{h}$ the "drop-key." The drop-key should be kept in the P/WCD. A drop-key K_d is a vector that defines the indices of the dropped elements. The range of the elements of K_d is $1 \leq K_d \leq N'_h$, and the number of elements of K_d is $N'_h - N_h = (r - 1) \cdot N_h$. For example, if $N_h = 2$, $r = 2.0$, then $N'_h = r \cdot N_h = 4$. Therefore, the server calculates $\boldsymbol{h}$ using a "redundant" number of hidden neurons $N'_h = 4$. After that, if the output of the hidden layer is $\boldsymbol{h} = (h_1, h_2, h_3, h_4)^t$ (i.e., the dimensions of $\boldsymbol{h}$ is N'_h) and $K_d = (1, 3)$, then $drop(\boldsymbol{h}, K_d) = (h_2, h_4)^t$. In this example, hidden neurons 1 and 3 are "dummy neurons", and their outputs are not used by the agent to make the final decision.

In practice, we can generate many drop-keys for any given problem, and design a model for each drop-key. Using different drop-keys, it will be extremely difficult for a malicious person to guess the user intention.

The same as trans-keys, drop-keys should also be stored in a secure place of the P/WCD. In this study, we assume that the

final decisions made by the P/WCD are not observable by any third person.

3.4 The Training Process

The flow for training using the proposed protocol is as follows.

(1) Define a large enough real matrix $\boldsymbol{W}_0$ at random. Various classification tasks can share this $\boldsymbol{W}_0$.

(2) Define a redundancy rate r, the number of hidden neurons N_h, and a redundant number of hidden neurons $N'_h = r \cdot N_h$.

(3) Generate a set of trans-keys $\boldsymbol{K_t} = \{K_t^1, K_t^2, ..., K_t^n\}$.

(4) Generate a set of pos-keys $\boldsymbol{K_p} = \{K_p^1, K_p^2, ..., K_p^m\}$.

(5) Generate a set of drop-keys $\boldsymbol{K_d} = \{K_d^1, K_d^2, ..., K_d^l\}$.
Note that n trans-keys, m pos-keys, and l drop-keys are all generated at random.

(6) Generate a new training set $\boldsymbol{X}'$ by encrypting all data in the original training set $\boldsymbol{X}$ using each trans-key. For example, if the number of training data is N_d, and the dimension N'_f of the augmented feature vectors is 4, the original training set is given by Eq. (10). If the trans-key is given by Eq. (11), the encrypted training set is Eq. (12).

$$\boldsymbol{X} = \begin{pmatrix} x_{11} & x_{12} & x_{13} & 1 \\ x_{21} & x_{22} & x_{23} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{N_d 1} & x_{N_d 2} & x_{N_d 3} & 1 \end{pmatrix}^t \tag{10}$$

$$K_t = \begin{pmatrix} 4 & 3 & 1 & 2 \end{pmatrix} \tag{11}$$

$$\boldsymbol{X}' = transposition(\boldsymbol{X}, K_t) = \begin{pmatrix} 1 & x_{13} & x_{11} & x_{12} \\ 1 & x_{23} & x_{21} & x_{22} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{N_d 3} & x_{N_d 1} & x_{N_d 2} \end{pmatrix}^t \tag{12}$$

For n trans-keys, we can generate n training sets, and an agent can be designed from each of them.

(7) Define a sub-matrix $\boldsymbol{W}$ of $\boldsymbol{W_0}$ using each pos-key. Here, we suppose N'_h and N'_f are given. For m pos-keys, we can get m sub-matrices. For example, if $\boldsymbol{W}_0$ is given by Eq. (13), where N is a sufficiently large integer, pos-key is given by Eq. (14), $r = 1.6$, $N_h = 5$, $N'_h = r \cdot N_h = 1.6 \cdot 5 = 8$, and $N'_f = 4$, then, the sub-matrix is defined by Eq. (15). The size of $\boldsymbol{W}$ is $N'_f \times N'_h = 4 \times 8$.

$$\boldsymbol{W}_0 = \begin{pmatrix} w_{1,1} & \cdots & w_{1,N} \\ \vdots & \ddots & \vdots \\ w_{N,1} & \cdots & w_{N,N} \end{pmatrix} \tag{13}$$

$$K_p = \begin{pmatrix} 2 & 9 \end{pmatrix} \tag{14}$$

$$\boldsymbol{W} = position(\boldsymbol{W}_0, K_p, N'_h, N'_f) = \begin{pmatrix} w_{2,9} & \cdots & w_{2,16} \\ \vdots & \ddots & \vdots \\ w_{5,9} & \cdots & w_{5,16} \end{pmatrix} \tag{15}$$

(8) For each $\boldsymbol{X}'$ generated in Eq. (12) and each $\boldsymbol{W}$ defined in Eq. (15), find the output of the hidden layer, and select the elements for making the final decision using each of the drop-keys. For example, when $r = 1.6$, $N_h = 5$, $N'_h = r \cdot N_h = 1.6 \cdot 5 = 8$, $N'_f = 4$, the drop-key is given by Eq. (17), and the output of the hidden layer is given by Eq. (16). The result after dropping is given by Eq. (18) because the 5-th, the 4-th, and the 7-th rows are dropped based on Eq. (17).

$$\boldsymbol{H} = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,N_d} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{2,N_d} \\ h_{3,1} & h_{3,2} & h_{3,3} & \cdots & h_{3,N_d} \\ h_{4,1} & h_{4,2} & h_{4,3} & \cdots & h_{4,N_d} \\ h_{5,1} & h_{5,2} & h_{5,3} & \cdots & h_{5,N_d} \\ h_{6,1} & h_{6,2} & h_{6,3} & \cdots & h_{6,N_d} \\ h_{7,1} & h_{7,2} & h_{7,3} & \cdots & h_{7,N_d} \\ h_{8,1} & h_{8,2} & h_{8,3} & \cdots & h_{8,N_d} \end{pmatrix}^t \tag{16}$$

$$K_d = \begin{pmatrix} 5 & 4 & 7 \end{pmatrix} \tag{17}$$

$$\boldsymbol{H}' = drop(\boldsymbol{H}, K_d) = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \cdots & h_{1,N_d} \\ h_{2,1} & h_{2,2} & h_{2,3} & \cdots & h_{2,N_d} \\ h_{3,1} & h_{3,2} & h_{3,3} & \cdots & h_{3,N_d} \\ h_{6,1} & h_{6,2} & h_{6,3} & \cdots & h_{6,N_d} \\ h_{8,1} & h_{8,2} & h_{8,3} & \cdots & h_{8,N_d} \end{pmatrix}^t \tag{18}$$

(9) Find an output layer weight vector $\boldsymbol{\beta}$ for each data set $\boldsymbol{X}'$ generated in Eq. (12), each sub-matrix $\boldsymbol{W}$ defined in Eq. (15), and each $\boldsymbol{H}'$ obtained in Eq. (18). Altogether there are $n \cdot m \cdot l$ output layer weight vectors.

(10) Store $\boldsymbol{W}_0$ in the cloud server, and $\boldsymbol{\beta}_s = \{\boldsymbol{\beta}^{111}, \boldsymbol{\beta}^{112}, ..., \boldsymbol{\beta}^{nml}\}$ in the P/WCD. The keys, that is, $\boldsymbol{K_t}$, $\boldsymbol{K_p}$, and $\boldsymbol{K_d}$, and the parameter N'_h are also stored in the P/WCD.

3.5 The Classification Process

The classification flow of the protocol is as follows.

(1) On the P/WCD side:

- Obtain a feature vector $\boldsymbol{x} \in \boldsymbol{R}^{N'_f}$ from the user through some sensors (which can be physical sensors or software sensors).
- Generate i at random from $1 \le i \le n$, and load the i-th trans-key K_t^i.
- Encrypt $\boldsymbol{x}$ using K_t^i and Eq. (19), and get $\boldsymbol{x}'$.

$$\boldsymbol{x}' = transposition(\boldsymbol{x}, K_t^i) \tag{19}$$

- Generate j at random from $1 \le j \le m$, and load the pos-key K_p^j.
- Send $\boldsymbol{x}'$, K_p^j and $N'_h = r \cdot N_h$ from P/WCD to the cloud server.

(2) On the server side:

- Extract $\boldsymbol{W}$ from $\boldsymbol{W}_0$ using K_p^j, N'_h and N'_f using Eq. (20).

$$\boldsymbol{W} = position(\boldsymbol{W}_0, K_p^j, N'_h, N'_f) \tag{20}$$

- Calculate the hidden layer output using Eq. (21).

$$\boldsymbol{h}(\boldsymbol{x}') = \boldsymbol{G}(\boldsymbol{W^t} \cdot \boldsymbol{x}') \tag{21}$$

- Return $\boldsymbol{h}(\boldsymbol{x}')$ to the P/WCD.

(3) On the P/WCD side:

- Upon receiving $\boldsymbol{h}(\boldsymbol{x}')$, generate k at random from $1 \le k \le l$, and load the drop-key K_d^k.

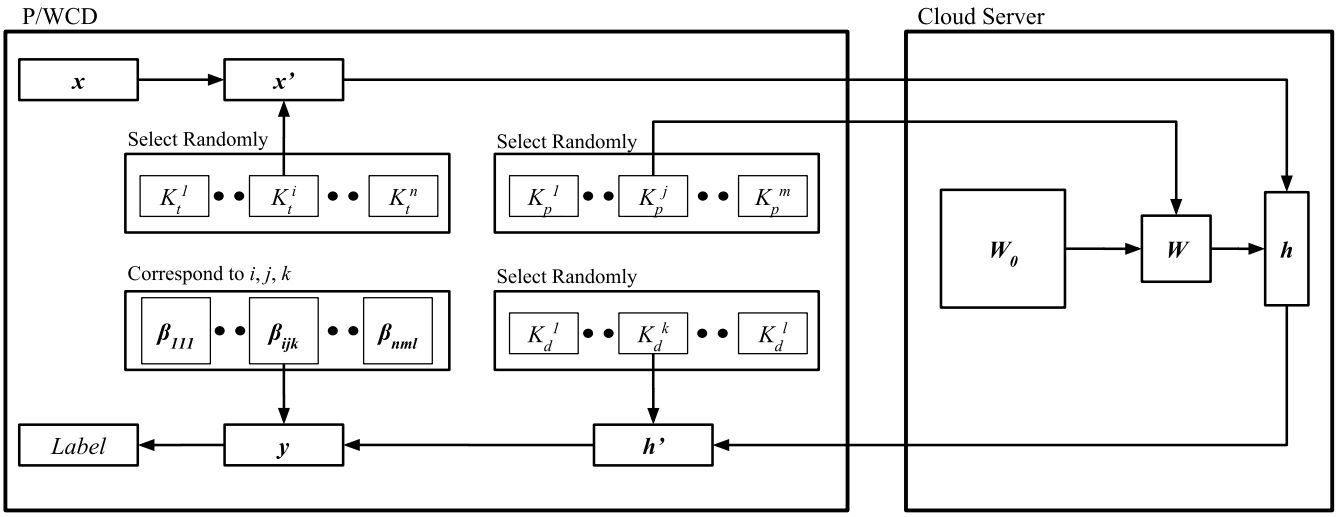


Fig. 6 Block diagram for classification phase.

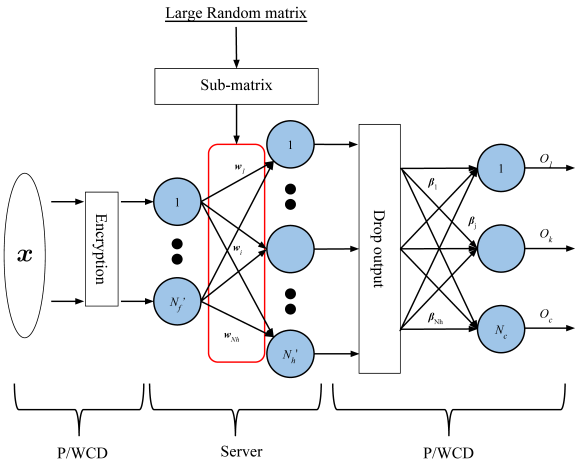


Fig. 7 Neural Network diagram for ELM based privacy preserving protocol.

- Select the hidden layer outputs useful for making the final decision using Eq. (22).

$$\boldsymbol{h}' = drop(\boldsymbol{h}(\boldsymbol{x}'), K_d^k) \tag{22}$$

- Load $\boldsymbol{\beta}_{ijk}$ and calculate the final result y as follows:

$$y = \text{sign}(\boldsymbol{h}' \cdot \boldsymbol{\beta}_{ijk}) \tag{23}$$

Figure 6 shows a block diagram of the classification process, and **Fig. 7** is a diagram focusing on the learning model. Here, we show only for classifying one datum.

3.6 Computational Costs

In the following, we discuss the theoretical time cost and battery cost (energy consumption) of each method. For convenience of discussions, cost functions are listed as follows.

- $C_{hid}(N_f, N_h)$: Time cost for calculating the hidden layer outputs.
- $C_{out}(N_h, N_c)$: Time cost for calculating the final outputs.
- $C_{send}(N_f)$: Time cost for sending an N_f-dimensional datum to the server.
- $C_{receive}(N_h)$: Time cost for receiving an N_h-dimensional result form the server.
- $C_{trans}(N_f)$: Time cost for encrypting the datum using transposition cipher.
- $C_{drop}(N_h')$: Time cost for selecting the results received from the server.
- C_{pos}: Time cost for extracting a sub-matrix from $\boldsymbol{W}_0$ stored in the server.

Each cost is proportional to its parameters. If the values of parameters are increased, the costs are also increased. Note that for the proposed protocol, N_h used in $C_{out}(N_h, N_c)$ is slightly different from the one used in $C_{hid}(N_f, N_h)$ because some of the results received from the server are dropped.

In practice, the last three types of costs ($C_{trans}(N_f)$, $C_{drop}(N_h')$, and C_{pos}) can be ignored because they are relatively small compared with other costs. For $C_{hid}(N_f, N_h)$ and $C_{out}(N_h, N_c)$, they are much smaller and can be ignored if the calculations are conducted on the server side, because a cloud server is usually much more powerful compared with a P/WCD. Thus, the time costs of the proposed approach can be given approximately as follows

$$C_{proposed} \simeq C_{out}(N_h, N_c) + C_{send}(N_f) + C_{receive}(N_h) \tag{24}$$

On the other hand, the time cost of the all-in-P/WCD approach is given by

$$C_{all-in-p/wcd} = C_{hid}(N_f, N_h) + C_{out}(N_h, N_c) \tag{25}$$

Therefore, the proposed approach can make a decision faster if the following condition is satisfied

$$\begin{aligned} &C_{proposed} < C_{all-in-p/wcd} \\ &\Leftrightarrow C_{send}(N_f) + C_{receive}(N_h) < C_{hid}(N_f, N_h) \end{aligned} \tag{26}$$

That is, the proposed approach is faster if the data transmission time is shorter than the time used by the P/WCD to find the hidden layer outputs.

As for battery cost, the energy consumption of a P/WCD consists of the following two parts: 1) The energy consumed for computations conducted inside the device, and 2) The energy consumed for transmitting the data. From Eq. (26) we can see that the proposed approach is more energy efficient if energy consumed for $C_{send}(N_f) + C_{receive}(N_h)$ is less than that consumed for $C_{hid}(N_f, N_h)$.

4. Experiments and Discussions

In this section, we look at the usefulness of the proposed protocol from three different view points. The first one is to see if we can obtain good A-agents using only one $\boldsymbol{W}_0$; the second one is to see if the response time is fast enough for data with various sizes; and the third one is to see if the proposed protocol is really energy efficient.

4.1 Experiment 1: The Classification Performance

The intention of this experiment is to confirm that the proposed protocol can work correctly. That is, the accuracy of the system will not be degraded even if a large random weight matrix is shared by various applications. We compared the classification performance of two methods. The first one is the original ELM in which each agent is implemented using a different model; and the second method is the proposed protocol in which all agents share the same $\boldsymbol{W}_0$. To make the results more reliable, we conducted 10 times 5-fold cross-validation.

Before the experiments, the best number of hidden neurons N_h for each problem was found via grid search from 10, 20, 30, ..., 100, 200, ..., 1,000, 1,500, and 2,000. The standard sigmoid function was used as the activation function of all hidden neurons. The programs were written by us using Python3, Numpy 1.10.4 [13], Scipy 0.17.0 [14] and Scikit-learn 0.18.1 [15]. The datasets we used are shown in **Table 1**. All datasets were taken from the UCI Machine Learning Repository [16] and normalized (the norm equals to one) before the experiment.

Table 2 shows the accuracy of the original ELM and that of our protocol with the best parameter of N_h. We can see that the accuracies of both methods are almost the same. That is, even if all datasets share the same random matrix $\boldsymbol{W}_0$, there is no significant change in the performance.

4.2 Experiment 2: Time Cost for Classification

In the second experiment, we actually implemented the protocol, and tested the response time for each datum. We compared the classification times of the following two methods:

- Local: The all-in-P/WCD method.
- Protocol: The proposed protocol.

The main purpose of this experiment is to see if the response time is acceptable for data with different sizes. The classification time is the total time used in the whole classification process, i.e., from loading a datum to finding the final result. Again, the number of hidden neurons N_h was set to the best value found in Experiment 1. The number n of trans-keys, the number m of pos-keys, and the number l of drop-keys were all set to 5 in the experiment. The redundancy rate r was set to 1.4, and the size of W_0 was $10{,}000 \times 10{,}000$. The system contains a smartphone, a router, and a server. The router and the server were connected by a LAN cable. The router and the smartphone were connected via Wi-Fi (i.e., wireless connection). NEC Aterm WR8370N was used as the router. Two different smartphones were used in the experiment to see if the results depends on the platforms or not. The environments of the smartphones and the server are shown in **Table 3** and **Table 4**.

Table 1 Parameters of data sets.

	Number of classes	Number of features	Number of data
Australian	2	14	690
QSAR	2	41	1,055
Satimage	6	36	6,435
MNIST	10	784	70,000
Gisette	2	5,000	7,000

Table 2 Classification performance: Original ELM V.S. Proposed protocol.

Dataset	N_h	Original (%)	Protocol (%)
Australian	100	79.4	79.0
QSAR	60	85.5	85.3
Satimage	2,000	94.3	93.5
MNIST	2,000	95.8	95.8
Gisette	2,000	96.6	96.3

Table 3 Smartphone environment.

Machine	Google Nexus6	Motorola Moto Z play
OS	Android 7.0	Android 7.0
Chipset	Snapdragon 805	Snapdragon 625
CPU	Quad-core 2.7 GHz Krait 450	Octa-core 2.0 GHz Cortex-A53
Memory	3 GB	3 GB
Wi-Fi	IEEE 802.11 b/g/n	IEEE 802.11 ac/a/b/g/n

Table 4 Server environment.

Machine	Dell Precision-WorkStation-T3400
OS	Ubuntu 14.04
CPU	Intel Core2 Duo E8500 (3.16 GHz)
Memory	4 GB

The communication method for the protocol was implemented as a REST-like API, so that we used HTTP/1.1. The model ($\boldsymbol{W_0}$ and $\boldsymbol{\beta}$) and the keys (K_t, K_p, K_d) were generated by using Python3, and saved as npy format. The npy format is a general binary format for Numpy array. The program for implementing ELM was also written in Python3. The sever was implemented by Python3 bottle [17], Nginx [18], uWSGI [19]. The HTTP cache was set to off (Add "Cache-Control: no-store" to the HTTP header), and the keep-alive was set to disable. The program for the Android smart phone was written in Kotlin (a programming language for Java virtual machine (JVM)) and C++. Kotlin is an official programming language for Android application development. The network connection for the Android was implemented in Rxjava (ReactiveX) [20], Retrofit [21]. The format for data transmission was Google protocol buffers [22]. And we used wire [23] to compile protocol buffers scheme to Java code for Android. The methods for loading the weights ($\boldsymbol{W}$ and $\boldsymbol{\beta}$), the keys (K_t, K_p, K_d), and for matrix multiplication were implemented in Android NDK. Android NDK provides higher calculation performance compared with JVM. To connect Kotlin programs and C++ programs, we used Java Native Interface (JNI). Single thread three-for-loop method was used for matrix multiplication. In our experiments, time used for model loading and classification were all included in measuring the computing time. We measured 30 times for classification of each datum, and took the average computing time.

Figures 8–12 are results of the classification time using Nexus6, and **Figs. 18–22** are results of Moto Z play. The vertical axis indicates average time (in seconds) for classifying one datum. The error bar shows the standard deviations. The left bar

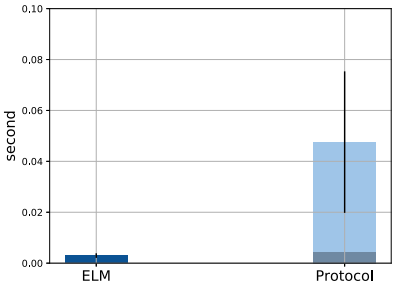


Fig. 8 Classification time of Australian on Nexus6.

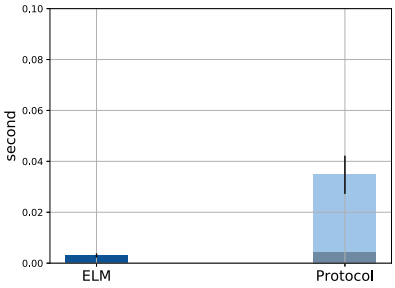


Fig. 9 Classification time of QSAR on Nexus6.

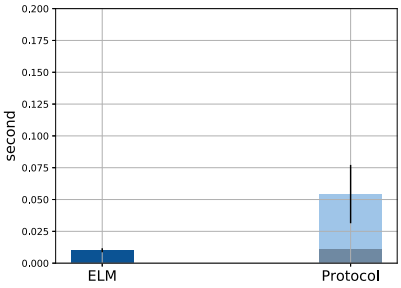


Fig. 10 Classification time of Satimage on Nexus6.

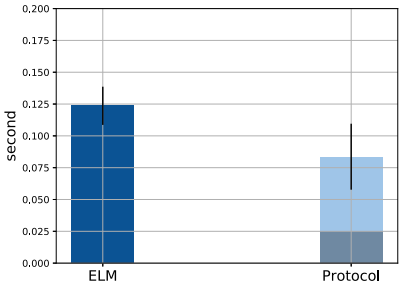


Fig. 11 Classification time of MNIST on Nexus6.

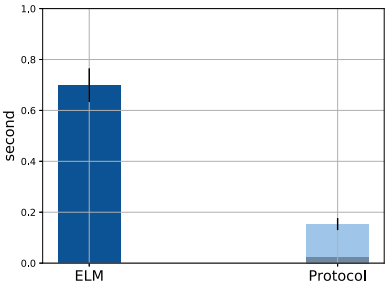


Fig. 12 Classification time of Gisette on Nexus6.

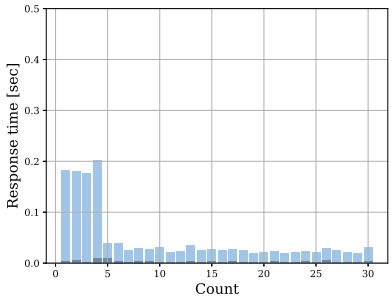


Fig. 13 Time details of Australian on Nexus 6.

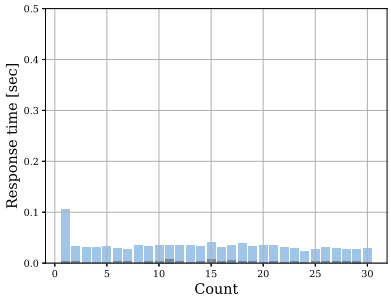


Fig. 14 Time details of QSAR on Nexus 6.

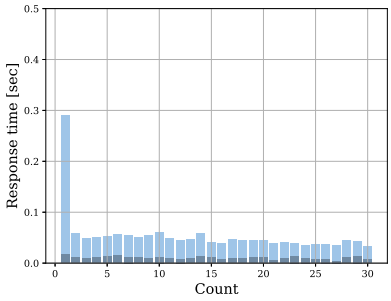


Fig. 15 Time details of Satimage on Nexus 6.

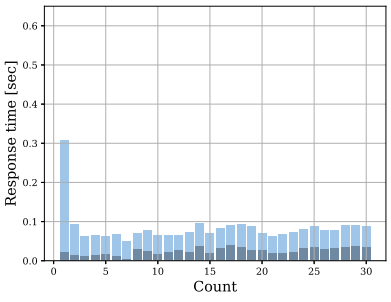


Fig. 16 Time details of MNIST on Nexus 6.

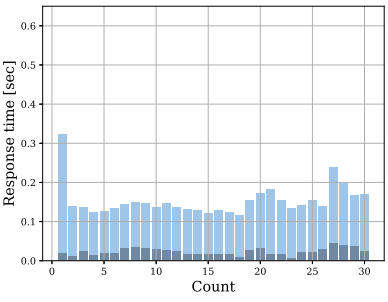


Fig. 17 Time details of Gisette on Nexus 6.

shows the time of the local method, and the right bar shows the time of our protocol. The right bar has two parts (bottom and top). The top one is the time between sending a request to the server and receiving a response (The sum of transmission time and the server calculation time), and the bottom one is the time for calculation in a P/WCD (transposition encryption and calculation of the output layer). The details of classification time of the protocol using Nexus6 are shown in **Figs. 13–17**. The details of Moto Z play are given in **Figs. 23–27**. The vertical axis shows the classification time. The horizontal axis shows the classification count which ranges from 1 to 30 because we have 30 trials. There are also two patterns on each bar. The bottom one is time for local calculation, and upper one is time for data transmission and server calculation.

For small datasets (i.e., Australian and QSAR), the classification of local method is faster than our protocol. This means that our protocol may not be needed for classification of simple problems. But in the Gisette and MNIST datasets, which have large N_f and require large N_h, our protocol can classify faster in both smartphones. This means the Gisette and MNIST databases satisfies Eq. (26). Since the calculation cost depends on the numbers N_f and N_h, our protocol is efficient if these two values are large.

In both devices and all databases, the standard deviations of classification time of protocol are relatively large. We can see the reason from Figs. 13–17 and Figs. 23–27. In 30 classification trials, the first (or start) trial takes the largest time in all databases. This is because the first (or start) trial needs to generate some objects for classification on Android side and server side, and for

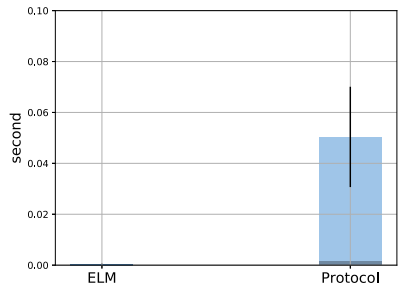


Fig. 18 Classification time of Australian on Moto Z.

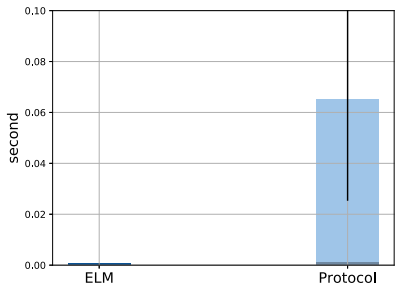


Fig. 19 Classification time of QSAR on Moto Z.

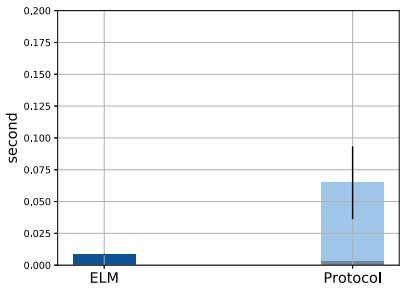


Fig. 20 Classification time of Satimage on Moto Z.

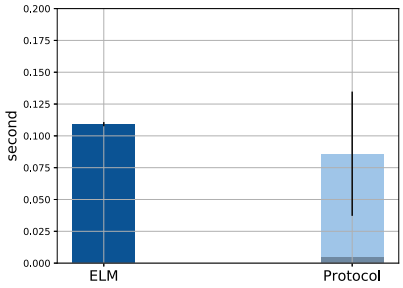


Fig. 21 Classification time of MNIST on Moto Z.

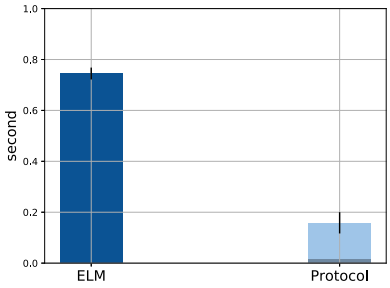


Fig. 22 Classification time of Gisette on Moto Z.

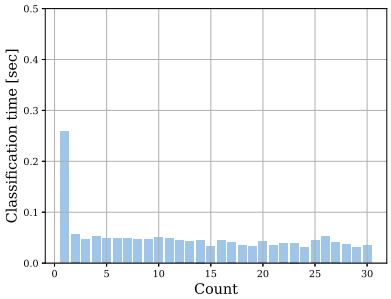


Fig. 23 Time details of Australian on Moto Z.

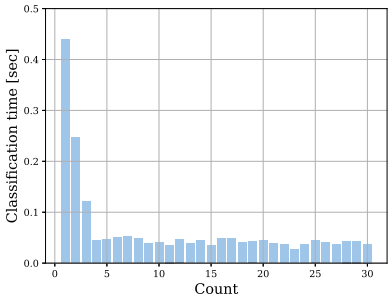


Fig. 24 Time details of QSAR on Moto Z.

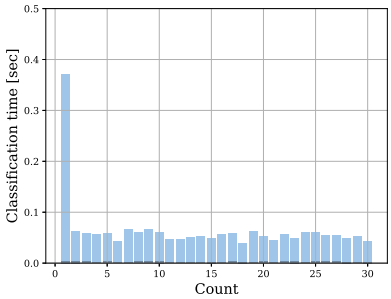


Fig. 25 Time details of Satimage on Moto Z.

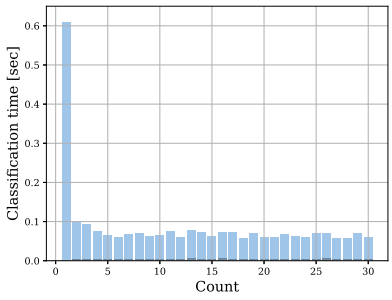


Fig. 26 Time details of MNIST on Moto Z.

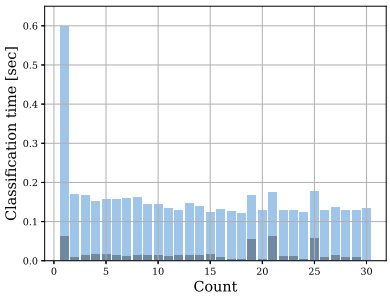


Fig. 27 Time details of Gisette on Moto Z.

establishment of network connection. Hence the overhead cost is larger in the first step.

The upper regions in the bar graphs of Figs. 13–17 and Figs. 23–27 correspond roughly to $C_{send}(N_f) + C_{receive}(N_h)$. From these figures we can see that, the proposed approach is not necessarily the best choice if the time cost used for finding the hidden layer outputs is relatively small compared with data transmission cost.

4.3 Experiment 3: Investigation of Battery Consumption

We compared the battery consumption speeds of two methods used in experiment 2. The smart phone classifies 1,000 data one by one. After classification of each datum, we measured the battery level. The maximum battery level is 100%, and the minimum is 0% which means no power. The GPS and blue-tooth were set to off and the display brightness was not changed during the experiment. The smart phone was not connected to cellular network. The implementation environment, the used devices, the datasets, and the ELM parameters were the same as those used in experiment 2.

Table 5 shows the battery level reduction after 1,000 classifications. Because of the specifications, Android battery level can be measured only by an integer number. For datasets with small data, there is no significant difference between the two methods (i.e., local and the proposed protocol). For the Gisette dataset, however, the battery consumption of the proposed protocol is smaller.

From Table 5, we can make a conclusion similar to the previous

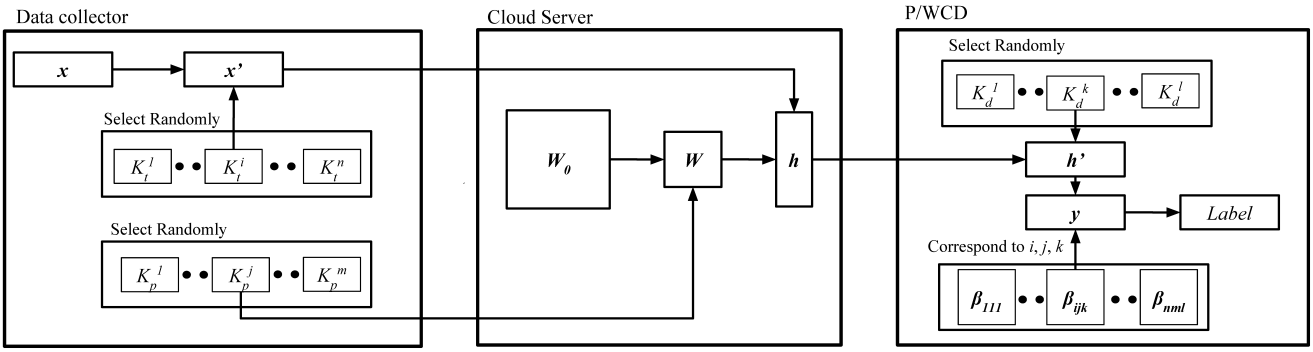


Fig. 28 Another framework.

Table 5 Battery Consumption:
Using only P/WCD V.S. Proposed protocol (%).

Dataset	N_h	Nexus 6		Moto Z play	
		Local	Protocol	Local	Protocol
Australian	100	-0	-0	-0	-0
QSAR	60	-0	-0	-0	-0
Satimage	2,000	-0	-0	-0	-0
MNIST	2,000	-1	-1	-0	-0
Gisette	2,000	-5	-2	-1	-0

experiment from the point of view of energy consumption. That is, the proposed approach can be useful for computationally heavy tasks. For example, in the future, if we implement A-agents using deep neural networks, the cost for data transmission will be smaller compared with that for computing the hidden neuron outputs. In addition, next generation communication technology (e.g., the 5-th generation wireless communication or 5G) may also provide strong support to the proposed method.

5. Conclusion

The purpose of our research is to implement multiple A-agents in a single P/WCD (e.g., smart phone). To improve the computation efficiency while preserving privacy, we have proposed ELM-based privacy preserving protocol. The proposed protocol has several features. The first one is to divide an A-agent into two parts. This division makes it possible to protect the agent model and at the same time to reduce the computational cost and energy consumption on the P/WCD side. To protect the agent model further, the proposed protocol uses an ELM to implement each A-agent. Using the ELM, we can put a random matrix on the server side, and this makes it difficult for any malicious person to estimate the agent model based on information available from the server. To protect the user data, the proposed protocol uses the transposition cipher, which is very easy to implement in a P/WCD. To improve the computational efficiency, we have also proposed to share a large hidden layer matrix, so that for any request from any user, the server can respond without loading or re-loading the matrix again and again. Finally, to protect the user intention further, we have proposed to use a drop-key, to use the results returned from the server selectively.

Experimental results show that our protocol keeps original ELM classification accuracy score even if the same random hidden weight matrix is shared by different agents. We also confirmed that the protocol classification is faster and more energy efficient when the dimension of the data is high or the number of hidden neurons needed is large. Currently, we are trying to implement the system using a real cloud server, and try to solve some practical problems using the protocol.

It is interesting to notice that the proposed protocol can also be extended easily to the framework shown in **Fig. 28**. In this framework, the data collector (i.e., someone who collects and distributes the data) and the end-user (i.e., P/WCD user) are different. The data collector (e.g., a local server in a smart home) collects a datum, transforms it into a feature vector, encrypts the vector using a trans-key, and sends the feature vector to the cloud server along with a pos-key. The Server calculates the hidden layer outputs, and sends them to the P/WCD of the end-user. The P/WCD selects the results using a drop-key, and makes the final decision using a $\boldsymbol{\beta}$ corresponding to the selected trans-key, pos-key, and drop-key. This approach can protect user privacy and the agent model from the server or some third person. The server can see only the encrypted data, a random matrix, and their multiplication results that contains some dummy elements. The P/WCD can get the final decision without reducing accuracy.

Note that in the above framework, we need a method for sharing i, j, and k (the indices of the keys). This can be easily implemented if the data collector and the P/WCD share a common random seed. That is, every time when a decision is needed, i, j, and k are generated using the same random process based on the same seed. Of course, we may also use some more advanced scheme for key exchange between the data collector and the P/WCD.

The above framework will be used for the smart home project conducted in our laboratory. The purpose of this project is to support elderly people while protecting their privacy. In this situation, the data collector is supposed to be a local server in a smart home (in which an elderly person lives alone), and the end-user is supposed to be a trusted person (e.g., the doctor or the children of the elderly person). If the data collected contain something unusual, the system can send a notification to some reliable person. So far, we have investigated the life-cycles of a resident using one infrared sensor [24], and studied methods for sensor array-based location and activity recognition [25], [26]. In the next step, we are going to implement the smart home system using the above framework, so that the system can be more computationally efficient and more trustable for the users.

Acknowledgments This research has been supported in part through a JSPS KAKENHI Grant Number 16K00334.

References

[1] Kaneda, Y. and Zhao, Q.: Inducing high performance and compact neural networks based on decision boundary making, *The Trans. Institute of Electrical Engineers of Japan, C*, Vol.134, No.9, pp.1299–1309 (2014).

[2] Kaneda, Y., Pei, Y., Zhao, Q. and Liu, Y.: Improving the Performance of the Decision Boundary Making Algorithm via Outlier Detection, *J. Inf. process.*, Vol.23, No.4, pp.497–504 (2015).

[3] Hashimoto, M., Kaneda, Y., Zhao, Q. and Liu, Y.: DBM vs ELM: A study on effective training of compact MLP, *2016 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC*), pp.001291–001296, IEEE (2016).

[4] Haris, M., Haddadi, H. and Hui, P.: Privacy leakage in mobile computing: Tools, methods, and characteristics, arXiv preprint arXiv:1410.4978 (2014).

[5] Chabanne, H., de Wargny, A., Milgram, J., Morel, C. and Prouff, E.: Privacy-Preserving Classification on Deep Neural Network, *IACR Cryptology ePrint Archive*, Vol.2017, p.35 (2017).

[6] Dowlin, N., Gilad-Bachrach, R., Laine, K., Lauter, K., Naehrig, M. and Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy, *International Conference on Machine Learning ICML*, Vol.48, pp.201–210 (2016).

[7] Tramèr, F., Zhang, F., Juels, A., Reiter, M.K. and Ristenpart, T.: Stealing machine learning models via prediction apis, *USENIX Security* (2016).

[8] Hashimoto, M., Kaneda, Y. and Zhao, Q.: An ELM-based privacy preserving protocol for cloud systems, *2016 IEEE Symposium Series on Computational Intelligence* (*SSCI*), pp.1–6, IEEE (2016).

[9] Huang, G.-B., Zhu, Q.-Y. and Siew, C.-K.: Extreme learning machine: Theory and applications, *Neurocomputing*, Vol.70, No.1, pp.489–501 (2006).

[10] Huang, G.-B., Zhou, H., Ding, X. and Zhang, R.: Extreme learning machine for regression and multiclass classification, *IEEE Trans. Systems, Man, and Cybernetics, Part B: Cybernetics*, Vol.42, No.2, pp.513–529 (2012).

[11] Hashimoto, M. and Zhao, Q.: An ELM-Based Privacy Preserving Protocol for Implementing Aware Agents, *2017 3rd IEEE International Conference on Cybernetics* (*CYBCONF*), pp.1–6, IEEE (2017).

[12] Rescorla, E.: Rfc 2818: Http over tls, *Internet Engineering Task Force*, available from ⟨http://www.ietf.org⟩ (2000).

[13] Van Der Walt, S., Colbert, S.C. and Varoquaux, G.: The NumPy array: A structure for efficient numerical computation, *Computing in Science & Engineering*, Vol.13, No.2, pp.22–30 (2011).

[14] Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python (2001) (online), available from ⟨http://www.scipy.org/⟩ (accessed 2018-01-20).

[15] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E.: Scikit-learn: Machine Learning in Python, *Journal of Machine Learning Research*, Vol.12, pp.2825–2830 (2011).

[16] Lichman, M.: UCI Machine Learning Repository (2013), available from ⟨http://archive.ics.uci.edu/ml⟩

[17] Hellkamp, M.: bottle (2014). available from ⟨http://bottlepy.org/⟩

[18] nginx: Nginx, Nginx (online), available from ⟨https://nginx.org⟩ (accessed 2018-01-20).

[19] Unbit: uWSGI, Unbit (online), available from ⟨https://uwsgi-docs.readthedocs.io/en/latest/⟩ (accessed 2018-01-20).

[20] ReactiveX: ReactiveX, ReactiveX (online), available from ⟨http://reactivex.io⟩ (accessed 2018-01-20).

[21] Square: Retrofit, Square (online), available from ⟨https://square.github.io/retrofit/⟩ (accessed 2018-01-20).

[22] Google: Protocol Buffers, Google (online), available from ⟨http://code.google.com/apis/protocolbuffers/⟩ (accessed 2018-01-20).

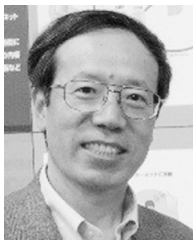
[23] Square: wire, Square (online), available from ⟨https://github.com/square/wire⟩ (accessed 2018-01-20).

[24] Kobiyama, Y., Zhao, Q., Omomo, K. and Taya, M.: Analyzing correlation of resident activities based on infrared sensors, *2015 IEEE 7th International Conference on Awareness Science and Technology* (*iCAST*), pp.1–6, IEEE (2015).

[25] Kobiyama, Y., Zhao, Q. and Omomo, K.: Privacy preserving infrared sensor array based indoor location awareness, *2016 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC*), pp.001353–001358, IEEE (2016).

[26] Kobiyama, Y., Zhao, Q., Ota, R. and Ichimura, S.: Recognition of Frequently Appeared Locations/Activities Based on Infrared Sensor Array, *2017 3rd IEEE International Conference on Cybernetics* (*CYBCON*), pp.1–5, IEEE (2017).

Masato Hashimoto was born in 1993. He received his B.E. from University of Aizu of Japan in 2016. He is pursuing his M.E. at University of Aizu. His research interests include neural network, privacy preserving machine learning and awareness computing. He is a student member of IEEE.



Qiangfu Zhao received the Ph.D degree from Tohoku University of Japan in 1988. He joined the Department of Electronic Engineering of Beijing Institute of Technology of China in 1988, first as a post-doctoral fellow and then an associate professor. He was associate professor from Oct. 1993 at the Department of Electronic Engineering of Tohoku University of Japan. He joined the University of Aizu of Japan from April 1995 as an associate professor, and became tenure full professor in April 1999. His research interests include image processing, pattern recognition, machine learning, and awareness computing.