International Journal of Electrical and Computer Engineering (IJECE)
Vol. 8, No. 6, December 2018, pp. 4693 – 4704
ISSN: 2088-8708, DOI: 10.11591/ijece.v8i6.pp4693-4704

Deep Belief Networks for Recognizing Handwriting Captured by Leap Motion Controller

Abas Setiawan[1,2] **and Reza Pulungan**[1]
[1]Dept. of Computer Science and Electronics, Faculty of Mathematics and Natural Sciences, Universitas Gadjah Mada, Indonesia
[2]Dept. of Informatics Engineering, Faculty of Computer Science, Universitas Dian Nuswantoro, Semarang, Indonesia

Article Info

Article history:
Received December 29, 2017
Revised May 30, 2018
Accepted July 13, 2018

Keyword:
Handwriting
Leap Motion
Deep belief network
Accuracy
Resilient backpropagation

ABSTRACT

Leap Motion controller is an input device that can track hands and fingers position quickly and precisely. In some gaming environment, a need may arise to capture letters written in the air by Leap Motion, which cannot be directly done right now. In this paper, we propose an approach to capture and recognize which letter has been drawn by the user with Leap Motion. This approach is based on Deep Belief Networks (DBN) with Resilient Backpropagation (Rprop) fine-tuning. To assess the performance of our proposed approach, we conduct experiments involving 30,000 samples of handwritten capital letters, 8,000 of which are to be recognized. Our experiments indicate that DBN with Rprop achieves an accuracy of 99.71%, which is better than DBN with Backpropagation or Multi-Layer Perceptron (MLP), either with Backpropagation or with Rprop. Our experiments also show that Rprop makes the process of fine-tuning significantly faster and results in a much more accurate recognition compared to ordinary Backpropagation. The time needed to recognize a letter is in the order of 5,000 microseconds, which is excellent even for online gaming experience.

Corresponding Author:
Reza Pulungan
Department of Computer Science and Electronics, Universitas Gadjah Mada, Indonesia
Gedung KPTU FMIPA UGM, Sekip Utara, Bulaksumur, Sleman, Yogyakarta, 55281, Indonesia
+6282136261216
Email: pulungan@ugm.ac.id

1. INTRODUCTION

Computer aided handwriting recognition can be performed either on-line or off-line. Typically, on-line handwriting recognition is based on specific input devices, such as real-time capturing digital pen, finger-touch screen, or mouse movement. In the recent decade, such input devices have already been equipped with computer vision technology that enables recognition of natural hand or body gestures as seen in Kinect-based handwriting recognition [1]. Huang *et al.* [2] introduce digit recognition on Kinect by using Multiple Segment and Scaled Coding with 94.6% accuracy. Other methods are proposed by using Depth-Skin Background Mixture Model for hand segmentation [3] and Dynamic Time Warping combined with Support Vector Machines (SVM) for digit recognition [1].

Kinect has the ability to recognize body shape or hand movements, but it faces difficulties in obtaining precise finger position compared to Leap Motion controller. Leap Motion controller (or simply Leap Motion) is a new computer vision device that can track the position of fingers more precisely. The 3D finger data can be obtained at over 100 frames per second [4], making it possible to efficiently track the position of each finger in detail. However, Leap Motion still faces some limitations: it cannot directly recognize a specific sequence of fingers' strokes (movements) performed when drawing a letter or a digit in air.

Several algorithms have been proposed to recognize these sequences of strokes to obtain a better accuracy. Vikram *et al.* develop handwriting recognition based on Leap Motion by using Dynamic Time Warping algorithm [4]. Agarwal *et al.* [5] present segmentation and recognition of 3D handwritten text captured by Leap Motion. Hidden Markov Model (HMM) is used to train segmented words and achieves an accuracy of 77.6%. Another algorithm for rapid recognition of hand graphical gestures is SVM. SVM is used to recognize 100 testing samples with an average recognition rate of 82.4% [6].

Researches show that algorithms with shallow architectures (*i.e.*, Decision Tree [7], Gaussian Mixture Models (GMM), HMM, Conditional Random Fields (CRF), Multi-Layer Perceptron (MLP) [8], and SVM [9]) are effective in solving many simple or well-constrained problems. However, these algorithms may face difficulties in complicated real-world applications involving natural signals, such as human speech, natural sound and language, natural images, and visual scenes [10] because of their modelling limitation. Instead, a deep architecture algorithm able to recognize handwritten digits captured by Leap Motion is available, namely Convolutional Neural Networks (CNN). However, it is hard to train deep MLP (and hence also CNN) without GPU parallelization [11].

CNN has been used to simulate 3D digit recognition model that can be projected as 2D in a 3D environment on Leap Motion tracking data [12]. This method achieves an accuracy of 92.4%. In this paper, we propose to use Deep Belief Networks (DBN) to solve this recognition problem. DBN is one of deep architecture algorithms and consists of two phases: greedy layer-wise unsupervised pre-training and discriminative fine-tuning [13, 14, 15]. Pre-training, initialized by a stacked Restricted Boltzmann Machines, can generate a better result compared to purely random configuration of the MLP. After the pre-training phase in the DBN, the network can be treated simply as an MLP and then trained with Backpropagation to enhance its weights [14, 15]. Alternatively, Resilient Backpropagation can be used to train the networks instead of ordinary Backpropagation.

2. PRELIMINARIES

2.1. Leap Motion controller

Leap Motion is an input device that allows users to recognize and track hands, fingers, and finger-like tools. Leap Motion is developed by Michael Buckwald and David Holz at Leap Motion, Inc. Leap Motion is able to track 3D finger data precisely and obtains them at over 100 frames per second [4]. Besides, Leap Motion device is also cheaper compared to other similar computer-vision devices. The dimension of the device is $1.2\times3\times7.6$ cm^3 and it uses optical sensors and infrared light, as shown in Figure 1 [16]. Leap Motion's sensors can detect hands or fingers across the upward y-axis and possess a 150°-vision with effective range of 25 to 600 millimeters [17].

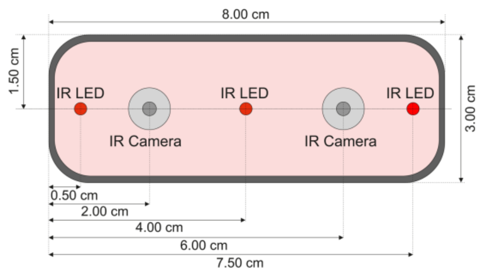


Figure 1. Detail of the Leap Motion's sensors [16]

To develop applications that use Leap Motion as input device, we can make use of an available API that can be accessed via various programming languages. Leap Motion API version 2.0 [17] introduces New Skeletal Tracking Model that provides additional information for arms, fingers, and overall tracking enhancement. In order to use Leap Motion, first, it must be connected with an existing USB port and positioned beside the monitor at a distance of the cable's range. Leap Motion device will detect gestures and the position as well as the movement of hands or fingers as the user positions her hands on its range. To retrieve the desired hand tracking data, Leap Motion API provides an update set or a frame of data. Every single frame object from the API represents a frame that contains a list of tracking entities called *motion tracking data*. Motion tracking data, such as hands, fingers, and tools, can also be accessed through the API.

2.2. Restricted Boltzmann machines

A Boltzmann machine is a stochastic recurrent neural network with binary stochastic units. There are two types of layers in Boltzmann machines: visible layer and hidden layer. Visible units or neurons on the visible layer are connected to other units and receive information from the environment. Visible units also bring the input signal from the environment and send it to other units during training. Hidden units, on the other hand, operate freely and extract features from the input signal. Interconnections of units in Boltzmann machines can be done even in a single layer itself. Unfortunately, learning in Boltzmann machines is often impractical and furthermore they also face scalability issues. In order to make learning in Boltzmann machines easier, advanced restrictions of connectivity must be enforced. This restriction is called Restricted Boltzmann Machines (RBM) [18].

In RBM, there is no connection between units of the same layer. RBM is composed of a visible layer and a hidden layer of stochastic neurons that are connected by symmetrical weights (*i.e.*, there is no weight toward itself). Hidden units in RBM are considered as feature detector.

2.3. Deep belief networks

A deep belief network is a generative model, consisting of multiple stochastic layers. A stochastic layer has several latent variables. Latent variables typically have binary values and are called hidden units or feature detectors [13]. Each DBN layer is an RBM and in order to create many layers, several RBMs are stacked on top of each other to construct a DBN construction. Hinton *et al.* in [13] propose an idea to learn one layer at a time and restrict the connectivity of binary stochastic units in order to make learning more efficient and simpler [18].

DBN training begins with RBM training: starting from input samples and then sequentially progressing toward the next RBM training. Generated patterns formed by the top RBM can deliver a reversed propagation to the input layer by using conditional probability as in belief network or sigmoid belief network. This training procedure is called DBN's greedy layer-wise training [13]. DBN can work with or without supervision. Although DBN can be used in supervised or discriminative mode, basically, greedy layer-wise training is an unsupervised training for each layer of stacked RBMs. It is called a pre-training procedure [19, 20, 21]. The purpose of pre-training procedure on each layer's training is to put the parameters of all layers in a region of certain parameter ranges that contain a good generalization of local optimum that can be obtained by local gradient descent [20, 21].

In an RBM, the joint distribution $p(v, h; w)$ of visible units v and hidden units h with parameters w is defined by the energy function $E(v, h; w)$. For Bernoulli-visible units that lead to Bernoulli-hidden units, the energy function is defined by:

$$E(v, h; w) = -\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij} v_i h_j - \sum_{i=1}^{I} b_i v_i - \sum_{j=1}^{J} a_j h_j,$$

where w_{ij} represents the symmetrical interaction between visible unit v_i and hidden unit h_j, b_i and a_j are the bias terms of, and I and J are the numbers of visible and hidden units, respectively. Therefore, the conditional probabilities can be calculated by:

$$p(h_j = 1 \mid v; w) = \varphi\left(\sum_{i=1}^{I} v_i w_{ij} + a_j\right), \text{ and} \tag{1}$$

$$p(v_i = 1 \mid h; w) = \varphi\left(\sum_{j=1}^{J} h_j w_{ij} + b_i\right), \tag{2}$$

where $\varphi(x) = 1/(1 + \exp(-x))$.

Greedy procedure on stacked RBMs is used to raise the maximum likelihood learning. Equation (3) is a learning rule for updated weights at each RBM to collect the gradient log-likelihood:

$$\Delta w_{ij} = E_{data}(v_i, h_j) - E_{model}(v_i, h_j), \tag{3}$$

where $E_{data}(v_i, h_j)$ is the expectation of observation on the training data and $E_{model}(v_i, h_j)$ is the expectation under the distribution defined by the model. Calculating $E_{model}(v_i, h_j)$ is intractable, so Contrastive Divergence (CD) algorithm is used to solve this problem [13].

CD algorithm performs Gibbs sampling to replace $E_{model}(v_i, h_j)$ in one or more steps. CD-1, namely CD algorithm that only performs one step of Gibbs sampling on the data, is used to train the model. The following are the steps of CD-1. First, input sample x and updated hidden units are used to compute $p(h_j = 1, v; w)$. Second, units are activated by comparing the sample result from the previous step with random values to obtain the binary state for each unit. Third, those binary units are reconstructed by calculating $p(v_i = 1, h; w)$. Fourth, the output probability of the reconstruction is calculated, and the reconstruction result becomes an input. Fifth, after all collections of statistics from the previous step are obtained, the weights on the RBM are updated.

After completing pre-training procedure in DBN, the next phase is to perform discriminative fine-tuning with Backpropagation [13, 14]. Fine-tuning with Backpropagation is carried out by adding a final output layer, which is a labeled class excluded from the pre-training procedure.

3. PROPOSED APPROACH

Figure 2 depicts the architecture of our proposed approach for in-air handwriting recognition using Leap Motion. The proposed approach is developed based on the idea laid out in [22]. Data acquisition is a process of collecting every letter sample by using Leap Motion. This captured letter sample can be used for recognition, or else, it can be stored in a dataset. The letter dataset is split into two parts, namely: training and testing datasets. The training dataset will be incorporated into a learning system by using several proposed algorithms; and this basically constitutes the training process. After training process has been completed, several model algorithms will be constructed. These model algorithms are then used for testing (by using the testing dataset) in order to obtain and compare their accuracy. The best model algorithm is then selected for further recognition of letter input data captured by Leap Motion.

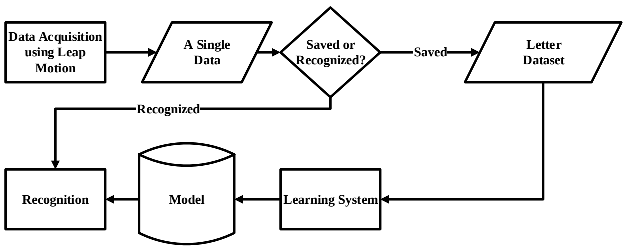


Figure 2. Our proposed approach

3.1. Data acquisition

Data acquisition is a mechanism to obtain handwritten sample data by using Leap Motion. A sample data is a sequence of information about the movement of hand (inputs only use one hand and one finger) in the air as captured by Leap Motion. The motion of finger-hand draws a letter stroke, which is then displayed on the screen of the simulation application integrated with Leap Motion API. Figure 3 depicts the flow for creating one sample data using the existing API.

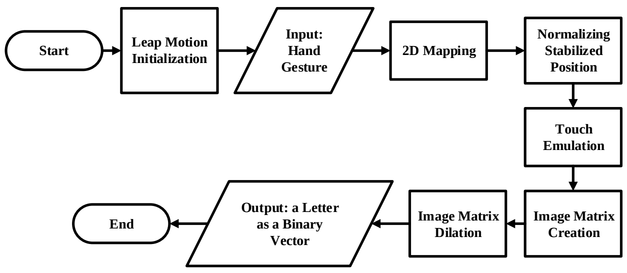


Figure 3. Data acquisition workflow

Leap Motion initialization Initialization is performed to connect the simulation application to the API. In the simulation application, the user is given information about what letter should be drawn with the Leap Motion. As the user starts drawing the letter in the air, we can access the captured Frame object of the Leap Motion through the API.

2D mapping Through the Frame we can access the Finger object, and hence the tip positions of all fingers can be obtained through the API. However, we do not need the positions of all fingers, but only the dominant finger as when pointing with the forefinger. The dominant finger can be obtained via a property of the Finger object, namely Frontmost. In addition, in order to obtain the position of the dominant finger's tip with minimal jitter, a property named StabilizedTipPosition can be used. Because of 2D mapping, the z-position of the forefinger produced by StabilizedTipPosition is ignored, and the x and y-position are then denoted by x_{pos} and y_{pos}, respectively.

Normalizing stabilized position InteractionBox property of Frame is an object on the API to ensure the user's finger-hand always stays on the device's visibility range. While finger-hand is inside InteractionBox, movements of the fingers will be detected on 3D coordinates, as shown in Figure 4(left). When the application is mapped to 2D coordinates, the InteractionBox is as shown in Figure 4(right).

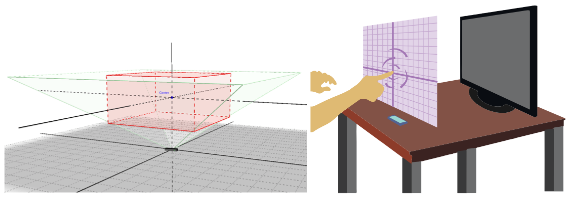


Figure 4. Field-of-view of the (left) 3D and the (right) 2D Interactionbox [17]

InteractionBox simplifies the mapping of every point of finger's tip position by normalizing the range of captured hands from left to right or top to bottom to values between 0 to 1, relatively to the application screen. If w_{app} and h_{app} are the width and height, respectively, of the 2D field in the application screen, InteractionBox provides the normalized position of the final stroke relative to the application's coordinate (x_{app}, y_{app}), which is given by:

$$x_{app} = NP(x_{pos})w_{app} \qquad \text{and} \qquad y_{app} = (1 - NP(y_{pos}))h_{app},$$

where NP is a normalization function in InteractionBox called normalizePoint.

Touch emulation From the positions of all strokes representing a single letter, a sequence of strokes $\vec{S} = (S_1, S_2, S_3, S_4, \ldots, S_m)$ is formed, where each $S_i = (x_{app}, y_{app})$ for $1 \leq i \leq m$, and m is the number of strokes required to draw the letter. Strokes can be in two states: hovering and touching. Not all strokes are used; those that are in hovering state can simply be ignored, as illustrated in Figure 5(left).

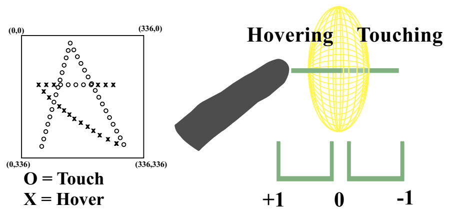


Figure 5. Touch emulation: the difference between hovering and touching [17]

The state is determined by a normalized distance in the range of $+1$ to -1. First, the finger enters the hovering state from normalized distance of $+1$ and the distance decreases toward 0 where the finger is near the touching surface. When the finger penetrates the surface, the normalized distance decreases to the maximum -1, which indicates that it is right now in the touching state. Stroke points are only recorded when the finger's tip is in the touching state.

Image matrix creation The tracing in the previous process is related to the tile-based editor of the simulation application. Tile-based editors are often developed in game technology, specifically to create the game environment [23, 24]. In this research, all tiles are of the same width and height, namely 12 pixels, and the tile-map is a 28×28 tile matrix.

If the tile-map is considered as a representation of an image, each tile will have an intensity value and an index position on the tile-map. Each tile indicates a pixel image drawn by the user in the air. Therefore, in order to select a tile that corresponds to the user's creation, a fixed stroke point position with the hovering or touching state is used. Figure 6 provides an illustration on how the selection of tile-based position for each stroke captured by Leap Motion is done.

In Figure 6, m_t is map-top distance, m_l is map-left distance, T_s is the size of a tile, and (x_t, y_t) is the selected index of a tile. The sequence of strokes $\vec{S}$ will be converted into an indexed (vector) sequence of strokes $\vec{V} = (V_1, V_2, V_3, V_4, \ldots, V_n)$ with $V_i = (x_i, y_i)$ for $1 \leq i \leq n$ and n is the vector size of the 28×28 image matrix, namely 784. Each V_i has an intensity value of 0 or 1 determined by:

$$V_i = \begin{cases} 1, & \text{if } d > 0, \\ 0, & \text{otherwise,} \end{cases}$$

where d is the touch distance, which defines whether the finger is in hovering or touching state. A coordinate (x_t, y_t) of each stroke point is then defined by:

$$x_t = \frac{x_{app} - m_l}{T_s} \qquad \text{and} \qquad y_t = \frac{y_{app} - m_t}{T_s}.$$

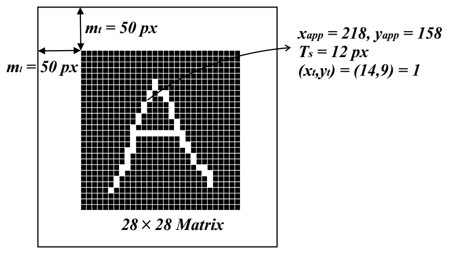


Figure 6. Selecting an indexed tile on the tile-map

In the end, this phase basically converts an image (tile) matrix obtained by the Leap Motion into a vector with binary elements representing the image.

Image matrix dilation Sometimes when the user writes a letter in the air by using Leap Motion too quickly, the handwritten letter becomes unclear. To clarify the handwritten letter, dilation operation can be applied to the image matrix. Dilation simply takes several specific neighboring points of a point and gives them an intensity value of 1, as depicted in Figure 7.

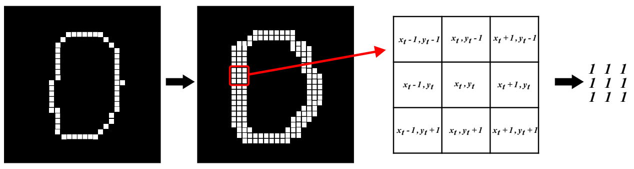


Figure 7. Image matrix dilation for letter “D”

3.2. Data, dataset, and data preparation

Each instance of data acquisition creates a single sample data. A single sample data is represented by a 784-size vector, which encodes the 28×28 matrix produced in image matrix creation. The data is used for two purposes: as an addition to dataset collection used as training or testing dataset with labeled data, and to be used in the recognition process, where the label will be searched out based on the learning algorithm trained before.

In this research, 22,000 samples of training and 8,000 samples of testing datasets are collected from 10 respondents. The respondents are undergraduate students in Universitas Dian Nuswantoro, Semarang, Indonesia, aged 20-21 years old. Each of them produces around 3,000 handwritings with Leap Motion, captured within 1 month. The training and testing datasets are restricted for upper-case letters from A to Z. Because the learning task is supervised, the dataset used for the DBN is in the form of $[(\vec{V}_1, L_1), (\vec{V}_2, L_2), \ldots, (\vec{V}_k, L_k)]$. For each letter sample, input data is a vector representing the sequence of strokes $\vec{V}_i$ for $1 \leq i \leq k$ whose label is L_i, and k is the number of samples in the dataset. All elements of input vector $\vec{V}_i$ are of binary value.

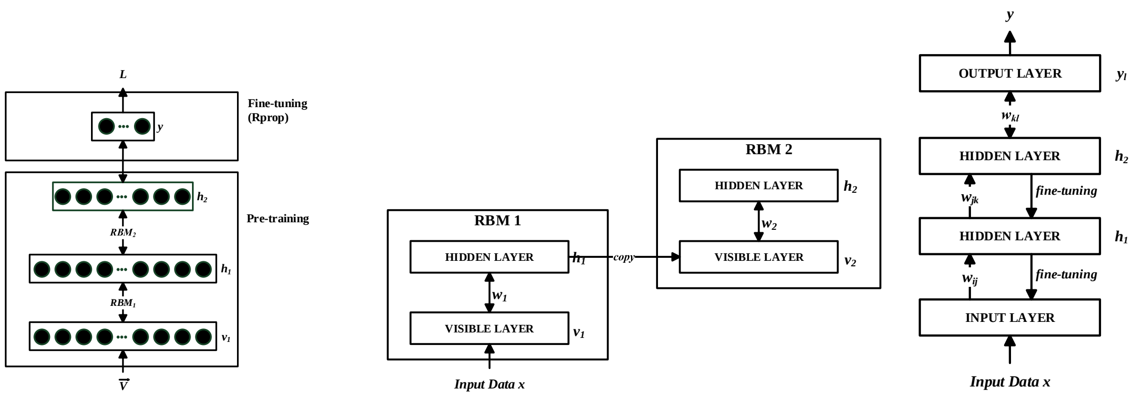
Encoding the label L_i, which has value from “A” to “Z”, such that $L_i \in \{0, 1, 2, \ldots, 25\}$ cannot be included directly into a DBN. In order to map L_i as an element of a vector, one-hot encoding mechanism is used. In one-hot encoding, label L_i containing the value “A” is encoded by $[1, 0, 0, \ldots, 0]$, “B” is encoded by $[0, 1, 0, \ldots, 0]$, etc.

3.3. Learning system

This research makes use of Deep Belief Networks (DBN) in supervised fashion. DBN basically learns the model of $F_w : \{\vec{V}\} \rightarrow \{L\}$, which maps an input vector $\vec{V}$ into a labeled vector L with parameter weights w. Figure 8(a) shows the architecture of a DBN with two hidden layers, h_1 and h_2. Two main processes exist on learning this DBN model. Firstly, the process of greedy layer-by-layer training for each stacked RBM by using Contrastive Divergence algorithm, which is called pre-training. Secondly, the process of fine-tuning the entire network using Resilient Backpropagation or *Rprop*.

3.3.1. RBM initialization and construction

As described earlier, a DBN consists of stacked RBMs [13], therefore, each RBM layer first needs to be initialized. Figure 8(b) shows a DBN constructed by two layers of RBMs. The following is the pre-training process,



(a) (b) (c)

Figure 8. (a) A DBN model with Rprop, (b) Constructing two RBMs in a DBN (c) Fine-tuning DBN with Rprop

which uses unsupervised learning algorithm on DBN to construct the stacked RBMs:
`Step 0`: In the first RBM, visible layer v_1 gets direct input from sample $\vec{V}$.
`Step 1`: Train the RBM's visible v_1 and hidden h_1 layers to adjust weights w_1.
`Step 2`: After the training is completed, use the pattern activity of the first RBM's hidden layer h_1 as input to visible layer v_2 to train the second RBM.
`Step 3`: By copying h_1 as v_2 in the second RBM, train visible layer v_2 and hidden layer h_2 to adjust weights w_2. Note that in this process, class labels of data are not involved.

3.3.2. Training each RBM layer

Every RBM is trained by using Contrastive Divergence with one step of Gibbs sampling algorithm (or CD-1). CD-1 follows the gradient difference function called Kullback-Leibler divergence. Here are the steps of CD-1:
`Step 1`: While maximum epoch is not reached, do step 2 to 7 repeatedly.
`Step 2`: Use data sample $\vec{V}$ as visible layer v.
`Step 3`: Update hidden layer by calculating $p(h_j = 1, v; w)$ (cf. Equation (1)), then activate stochastic binary neurons h_j on the hidden layer by taking the probability $p(h_j = 1, v; w)$.
`Step 4`: Reconstruct visible neurons v_i^1 by calculating $p(v_i = 1, h; w)$ (cf. Equation (2)) using the value of stochastic binary neurons calculated before as input.
`Step 5`: Calculate the output probability of h_j^1 from previous reconstruction step, namely $p(h_j^1 = 1, v^1; w)$.
`Step 6`: Calculate weights (cf. Equation (3)) by using the learning rate α as follows:

$$\Delta w_{ij} = \alpha(\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1),$$

where $\langle v_i h_j \rangle^0 = E_{data}(v_i h_j)$ and $\langle v_i h_j \rangle^1 = E_{model}(v_i h_j)$.
`Step 7`: Update the new weights by adding momentum μ and weight decay w_d as follows:

$$w_{grad} = \langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1, \text{ and } w_{ij} = w_{ij}(t-1) + (\mu \Delta w_{ij} + \alpha(w_{grad}) - w_d w_{ij}(t-1)).$$

Learning rate is a parameter that influences the learning speed of the algorithm in reaching convergence. Momentum is a parameter used to prevent the system from being trapped in local minima. Weight decay works by adding "additional terms" to the normal gradient. The additional term is a derivative of a function that penalizes an excessive weight [25].

The process of updating weights of RBM refers to the update rules of the gradient descent. In the case of large-scale datasets, using batch gradient descent requires quite an expensive computation when it only needs one step for the entire data training. Therefore, we use a more efficient learning mode that learns the dataset for each predetermined batch. This mode is called mini-batch gradient descent [25].

3.3.3. Fine-tuning Rprop

Because DBN is to be used as supervised (after pre-training process is completed), which involves label classes, weights have to be fine-tuned. Fine-tuning is conducted by training the network with Backpropagation algorithm [13, 14]. In this research, besides using fine-tuning with Backpropagation, we also perform Resilient Backpropagation (Rprop) algorithm as shown in Figure 8(c).

Theoretically, Rprop algorithm is more efficient in terms of learning speed compared to ordinary Backpropagation. The difference between Rprop and Backpropagation is located on the weight update, where Rprop makes use of the sign that identifies the value of the last gradient update [26]. In the Rprop algorithm, we use weights and biases obtained from pre-training process, namely $w_{ij} = w_1$ (from input to the first hidden layer) and $w_{jk} = w_2$ (from the first hidden layer to the second hidden layer). The weights and bias connection between the second hidden layer to output layer uses random initialization in the interval $(-1, 1)$. We initialize the maximum epoch and set the values of $\Delta_0 = 0.0125$, $\Delta_{max} = 50.0$, $\Delta_{min} = 10^{-6}$, $\eta^+ = 1.2$, and $\eta^- = 0.5$.

4. RESULT AND ANALYSIS

A learning algorithm naturally minimizes the cost function in order to obtain optimal result. When the algorithm is being trained using some training dataset, we are basically creating a model algorithm that closely reflects the available data. Afterwards, the obtained model algorithm can then be applied to testing dataset to assess the performance of the algorithm.

4.1. Experimental setup

All methods and algorithms in our proposed approach are implemented in C#. All experiments are conducted using this implementation on a computer with a Core(TM) i7-2600 CPU @ 3.40GHz (8 CPUs), 3.7 GHz. Learning is performed with a parallel computation on a logical core for each CPU. For our experiments, we perform data acquisition that produces 30,000 samples of capital letters, partitioned into two groups: 22,000 samples as training and 8,000 samples as testing datasets. There are about 843 samples for each letter from A to Z in training dataset, whereas in testing dataset, each letter has about 307 samples. Every single data in the training dataset will be used as an input for the learning algorithm.

In order to assess the performance of our proposed approach, we not only conduct experiments on DBN with Rprop fine-tuning, but also on three other approaches for the purpose of comparison. The three approaches are MLP with Backpropagation (Bp), MLP with Rprop, and DBN with Backpropagation. Table 1 shows the experimental configuration we use for each of those approaches.

Table 1. Experimental configurations

Dataset	Architecture	Pre-training	Fine-tuning
Letter	784-400-100-26 (MLP)	-	Bp (Ep.: 300)
	784-400-100-26 (MLP)	-	Rprop (Ep.: 300)
	784-400-100-26 (DBN)	CD-1 (Ep.: 300)	Bp (Ep.: 300)
	784-400-100-26 (DBN)	CD-1 (Ep.: 300)	Rprop (Ep.: 300)
Digit (MNIST)	400-400-100-10 (MLP)	-	Bp (Ep.: 500)
	400-400-100-10 (MLP)	-	Rprop (Ep.: 500)
	400-400-100-10 (DBN)	CD-1 (Ep.: 300)	Bp (Ep.: 500)
	400-400-100-10 (DBN)	CD-1 (Ep.: 300)	Rprop (Ep.: 500)

Table 1 shows that the neural network architecture we use when dealing with the letter dataset is 784-400-100-26. In this architecture, the input is a 784-sized vector, which is a flattening of the 28×28 tile-map matrix for each letter sample. Both MLP and DBN use 400 and 100 neurons in the first and second hidden layers, respectively. The size of the output vector is adjusted to the number of letter labels, namely 26 classes with one-hot-encoding mechanism.

Beside using the letter dataset captured by our own data acquisition mechanism, we also make use of the digit dataset from MNIST database [27]. The MNIST digit dataset consists of 60,000 samples in training dataset and 10,000 samples in testing dataset. Each sample in the MNIST digit dataset is formed by a grayscale image. To use them, it is necessary first to perform preprocessing by transforming the grayscale into binary images. Afterwards, width normalization is carried out by changing the original size of 28×28 pixels to 20×20 pixels. The 20×20 tile-map matrix is then flattened into an input vector of size 400 as shown in the last four rows of Table 1. Similar to the letter configuration, in this digit configuration, both MLP and DBN use 400 neurons and 100 neurons in the first and second hidden layers, respectively. Because the digit labels range from 0 to 9, the size of the output vector is 10 with one-hot-coding mechanism.

Both for the letter and MNIST digit datasets, we use the same configuration on the pre-training phase of the DBN. There are two RBM architectures in letter configuration: the first RBM is 784-400 and the second RBM is 400-100. In MNIST digit configuration, the first RBM architecture is 400-400 and the second is 400-100. There is no pre-training phase in the MLP. Each RBM in the letter or MNIST digit configurations is trained with CD-1 algorithm. Based on our experiments, we select the same initialization throughout, namely learning-rate = 0.5, momentum = 0.5, and weight-decay = 0.001. The maximum epoch for each RBM is set to 300. The batch size parameter for each RBM is set to 100, which means that if there are 22,000 training samples in the dataset, they will be processed 100 at a time.

The Backpropagation for fine-tuning on DBN or training on MLP uses learning rate = 0.05, momentum = 0.5, and batch size = 100. Unlike Backpropagation, Rprop does not need learning rate and momentum, but instead needs constant values Δ_0, Δ_{max}, Δ_{min}, η^+, and η^- (cf. Section 3.3.3.). The initial epoch for Rprop is set to 300 for letter and 500 for MNIST digit configurations.

4.2. Testing and recognition

Table 2 shows accuracy and the computational times required in recognizing each letter (from testing dataset) by the four different methods. Recall that there are about 307 samples for each letter in testing dataset.

Table 2. Accuracy and computational times for the letter (testing) dataset

Lab	MLP-Bp			MLP-Rprop			DBN-Bp			DBN-Rprop		
	Acc.	$\bar{t}$ (μs)	stdev	Acc.	$\bar{t}$ (μs)	stdev	Acc.	$\bar{t}$ (μs)	stdev	Acc.	$\bar{t}$ (μs)	stdev
A	99.35	4,572.2	1,081.2	100	5,305.1	1,718.6	100	5,117.5	1,392.5	100	3,278.2	811.7
B	92.86	5,740.0	1,534.4	99.68	4,622.1	918.5	99.03	5,369.4	1,416.5	100	5,235.6	2,074.0
C	94.81	3,727.2	912.2	100	5,337.8	1,117.6	99.35	5,126.9	1,684.3	100	3,922.0	2,140.9
D	92.88	2,972.8	628.1	100	4,991.5	1,229.1	98.71	4,124.2	996.1	100	5,668.1	1,935.6
E	92.51	3,151.1	751.4	100	4,428.3	968.1	99.35	3,718.4	941.1	100	5,212.0	1,156.5
F	97.73	3,265.4	724.0	99.68	5,219.6	1,224.0	99.03	4,776.6	1,213.6	99.35	4,602.7	1,556.9
G	77.67	4,862.7	1,100.5	100	5,068.3	1,262.4	98.38	3,558.6	958.8	99.68	5,459.3	1,452.2
H	90.29	4,893.7	1,163.0	99.68	4,205.8	935.3	99.68	3,139.1	819.7	99.68	4,549.9	1,634.3
I	90.58	5,055.4	1,231.4	97.40	3,573.4	1,095.7	95.45	3,714.9	1,205.8	98.38	5,856.4	3,309.5
J	96.74	5,439.8	1,390.0	100	4,720.8	1,233.2	100	5,064.3	1,357.8	100	4,439.5	1,304.9
K	94.48	4,619.5	1,006.0	99.03	4,766.1	1,270.2	98.70	5,416.3	1,433.2	100	5,325.4	1,096.8
L	99.68	3,705.6	913.6	100	4,446.1	1,392.5	100	5,260.5	1,268.8	100	5,545.3	1,287.1
M	91.53	3,833.7	907.5	99.67	5,013.1	1,151.6	99.02	4,462.5	1,136.8	99.67	3,833.9	896.3
N	98.38	4,287.0	1,097.0	99.35	4,819.3	1,019.2	99.35	3,699.3	914.0	99.35	4,909.1	863.1
O	95.45	3,338.1	816.7	99.03	5,237.3	1,000.0	98.38	4,846.7	1,167.2	99.35	4,849.6	1,563.8
P	95.44	5,606.4	1,390.2	99.67	5,486.6	1,502.8	99.67	3,633.0	880.7	100	5,707.3	2,559.3
Q	91.53	4,486.3	903.3	99.67	5,036.0	1,219.2	99.35	3,387.2	788.2	99.67	4,680.0	1,310.3
R	91.86	4,513.4	1,194.6	99.67	5,515.9	1,335.5	95.44	5,289.7	1,185.9	99.67	4,567.4	1,063.3
S	93.16	4,300.1	1,176.0	99.35	3,967.6	890.0	99.35	5,619.3	1,385.8	100	3,560.6	970.2
T	93.18	5,536.6	1,321.9	100	4,180.4	1,104.3	99.03	4,618.1	1,199.2	100	5,156.8	1,118.2
U	94.16	4,861.5	1,147.3	100	3,639.5	864.8	100	5,545.0	1,445.5	100	4,873.8	1,197.9
V	99.03	4,755.6	1,467.6	100	5,169.7	1,482.1	100	4,198.5	1,255.0	100	3,876.2	1,317.2
W	95.44	4,117.0	1,094.3	99.35	4,782.4	1,781.9	100	4,723.2	1,347.3	100	4,161.0	773.4
X	95.11	4,788.1	1,129.6	99.35	4,025.6	1,605.4	100	3,967.8	893.0	100	4,158.1	1,044.3
Y	98.05	3,699.5	986.6	99.35	5,022.5	1,330.0	99.67	3,239.5	777.4	99.67	5,238.1	1,067.9
Z	77.20	4,996.6	1,105.0	99.35	4,571.7	1,473.3	98.37	4,002.9	973.7	98.05	5,230.8	1,210.8
All	93.43	4,427.9	1,083.6	99.59	4,736.6	1,235.6	99.05	4,446.9	1,155.3	99.71	4,765.3	1,412.2

The first four columns of the first row of Table 2 indicate that out of 307 samples of the letter A, 99.35% (Acc.) of them are recognized and the computational time spent until the point when the letters are successfully or unsuccessfully recognized is on average 4,572.2 microseconds ($\bar{t}$), with a standard deviation of 1,081.2 (stdev). The last row of the table provides the overall statistics of accuracy and computational times for the whole letter training dataset produced by the four different methods. Table 3 shows similar results for the recognition of the MNIST digit testing dataset.

Table 3. Accuracy and computational times for the MNIST digit (testing) dataset

Lab	MLP-Bp			MLP-Rprop			DBN-Bp			DBN-Rprop		
	Acc.	$\bar{t}$ (μs)	stdev	Acc.	$\bar{t}$ (μs)	stdev	Acc.	$\bar{t}$ (μs)	stdev	Acc.	$\bar{t}$ (μs)	stdev
1	97.62	3,012.4	575.6	98.15	3,062.5	688.6	97.71	2,747.4	595.3	98.59	2,844.7	1019.6
2	86.92	3,028.2	890.5	94.67	2,268.8	562.4	88.76	2,703.7	482.7	95.83	3,017.7	721.4
3	75.54	2,294.3	673.0	95.84	2,783.3	354.4	91.98	2,623.2	802.8	96.14	2,425.2	589.1
4	88.70	3,059.5	699.7	95.52	2,400.4	621.4	88.90	2,616.5	307.7	96.74	2,525.1	819.4
5	90.92	3,041.4	763.8	96.08	2,795.0	377.6	85.76	2,720.6	295.3	95.52	2,452.3	742.4
6	89.56	3,211.7	795.0	96.45	2,774.5	331.8	95.51	2,656.7	834.6	95.93	2,408.7	556.1
7	88.62	2,811.0	736.3	94.94	2,732.5	661.6	90.66	2,759.4	589.3	95.04	3,069.1	777.3
8	85.01	2,920.0	710.8	94.25	2,002.9	506.3	88.19	2,327.9	594.7	95.69	2,429.5	567.5
9	88.01	1,740.0	556.3	93.56	2,835.8	629.6	87.22	2,804.7	790.1	95.44	3,017.0	665.7
0	95.61	3,031.5	561.9	98.06	2,404.7	641.0	95.41	2,773.7	819.0	98.06	2,805.1	379.9
All	88.65	2,815.0	696.3	95.75	2,606.0	537.5	91.01	2,673.4	611.1	96.30	2,699.4	683.9

Table 4 summarizes the results for recognizing both letter and MNIST digit testing datasets. The best accuracy (column Acc.) for both testing datasets is DBN with Rprop fine-tuning with accuracy of 99.71% and 96.30%, respectively. This is better than the accuracy of MLP with Rprop fine-tuning, which achieves 99.59% and 95.75%, respectively. These two approaches evidently are much more superior than both DBN and MLP with Backpropagation training. It seems that fine-tuning with Rprop contributes significantly to the accuracy of both MLP and DBN.

Table 4. Summary of accuracy and computational times

Algorithm (Dataset)	Acc.	Epoch	Training	Recognition	
			t (hours)	$\bar{t}$ (μs)	stdev
MLP-Bp (Letter)	93.43%	300	> 5	4,427.9	1,083.6
MLP-Rprop (Letter)	99.59%	35	> 1	4,736.6	1,235.6
DBN-Bp (Letter)	99.05%	300	> 10	4,446.9	1,155.3
DBN-Rprop (Letter)	99.71%	35	> 6	4,765.3	1,412.2
MLP-Bp (MNIST)	88.65%	500	> 12	2,815.0	696.3
MLP-Rporp (MNIST)	95.75%	70	> 2	2,606.0	537.5
DBN-Bp (MNIST)	91.01%	500	> 24	2,673.4	611.1
DBN-Rprop (MNIST)	96.30%	70	> 12	2,699.4	683.9

The last two columns of Table 4 ($\bar{t}$ and stdev) show that the recognition times of the four approaches are comparable. It seems that using Rprop results in longer recognition times both for MLP and DBN. Out of the four approaches, the recognition time of DBN fine-tuned with Rprop is the longest, namely on average 4,765.3 microseconds with a standard deviation of 1,412.2 for the letter dataset. For the MNIST digit dataset, the recognition times of the four approaches are much more similar to each other, and are a bit more than a half of those for the letter dataset. This is because the resulting neural network for the digit dataset has smaller input as well as output vectors, and thus is quicker to process. The time needed to recognize a letter or a digit is below ten milliseconds, which is excellent even for online gaming experience [28].

4.3. Training and fine-tuning

Prior to recognition, the neural networks, both DBN and MLP, must first be trained or fine-tuned using training dataset. The fourth column of Table 4 provides the computational times spent by the four different methods to complete training and pre-training (if any). Because MLP does not have pre-training phase, its training time is much shorter than DBN. The time spent by DBN to complete pre-training is around 5 hours for the letter dataset and 10 hours for the MNIST digit dataset. It is evident that this pre-training phase takes most of the training computational time of DBN and Rprop is really helpful in reducing the training (but outside of pre-training) time.

In Figure 9 and Figure 10, we show how the accuracy of each method changes as the number of epochs increases in the training process. In order to produce these figures, right after each epoch in the training process (with training datasets), each method is paused, and the resulting neural network is then used to recognize letters or digits in the testing datasets. We observe these changes until 300 and 500 epochs for the letter and digit datasets, respectively.

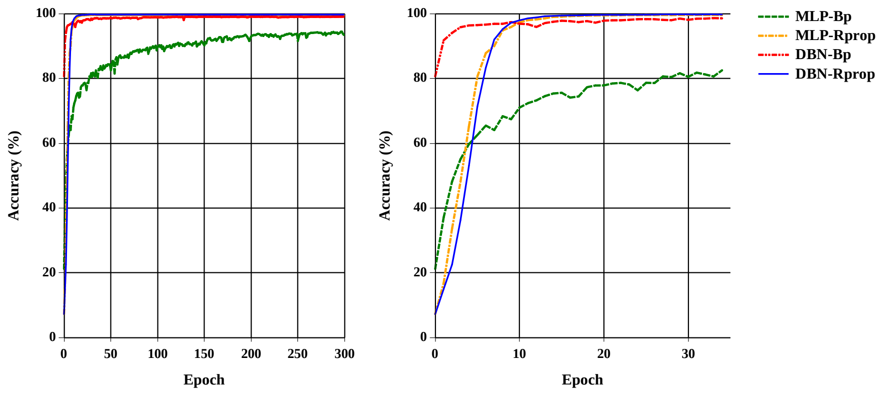


Figure 9. Accuracy vs. epoch for the letter (testing) dataset

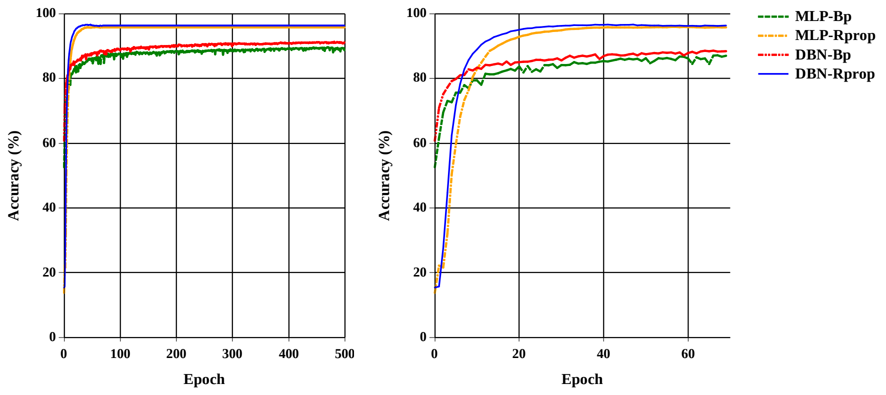


Figure 10. Accuracy vs. epoch for the MNIST digit (testing) dataset

In Figure 9(right), both MLP and DBN fine-tuned with Rprop require 35 epochs (cf. Table 4 column 3) to reach the highest accuracy (cf. Table 4 column 2) in the letter dataset. Out of 8,000 letter samples in the testing dataset, DBN fine-tuned with Rprop misrecognizes only 23 samples at epoch 35. In Figure 10(right), MLP and DBN fine-tuned with Rprop require 70 epochs to reach the best accuracy in the MNIST digit dataset. From 10,000-digit samples, DBN with Rprop misrecognizes 367 samples at epoch 70.

5. CONCLUSION

We have proposed an approach to capture and recognize which letter has been drawn in the air by the user with Leap Motion controller; an approach based on DBN fine-tuned with Rprop. We have also conducted experiments to find out the performance of our proposed approach. Our experiments indicated that, compared to MLP or DBN with Backpropagation, DBN fine-tuned with Rprop achieves the best accuracy of 99.71% for the letter dataset and of 96.30% for the digit dataset. This accuracy can be reached in only 35 and 70 epochs, respectively. The recognition time for letter is in the order of 5,000 microseconds, which is responsive enough even for online gaming experience. Rprop has been shown to be significant in making fine-tuning process faster and in producing a more accurate recognition.

We have selected neural network architectures 784-400-100-26 and 400-400-100-10 because they are still computationally feasible in a multi-core (typical gaming) computer and still manage to achieve an excellent accuracy. We are now working on lowering the computational requirements while still maintaining the overall accuracy.

REFERENCES

[1] C. Qu, D. Zhang, and J. Tian, “Online Kinect handwritten digit recognition based on dynamic time warping and support vector machine,” *Journal of Information and Computational Science*, vol. 12, no. 1, pp. 413–422, 2015.
[2] F. A. Huang, C. Y. Su, and T. T. Chu, “Kinect-based mid-air handwritten digit recognition using multiple seg-

ments and scaled coding," in *2013 International Symposium on Intelligent Signal Processing and Communication Systems*, 2013, pp. 694–697.

[3] Z. Ye, X. Zhang, L. Jin, Z. Feng, and S. Xu, "Finger-writing-in-the-air system using Kinect sensor," in *2013 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, 2013, pp. 1–4.

[4] S. Vikram, L. Li, and S. Russell, "Handwriting and gestures in the air, recognizing on the fly," in *The ACM SIGCHI Conference on Human Factors in Computing Systems (CHI 2013)*, 2013, pp. 1179–1184.

[5] C. Agarwal, D. P. Dogra, R. Saini, and P. P. Roy, "Segmentation and recognition of text written in 3D using Leap Motion interface," in *3rd IAPR Asian Conference on Pattern Recognition (ACPR 2015)*, 2016, pp. 539–543.

[6] Y. Chen, Z. Ding, Y. L. Chen, and X. Wu, "Rapid recognition of dynamic hand gestures using Leap Motion," in *2015 IEEE International Conference on Information and Automation (ICIA 2015)*, 2015, pp. 1419–1424.

[7] M. Abdar, S. R. N. Kalhori, T. Sutikno, I. M. I. Subroto, and G. Arji, "Comparing performance of data mining algorithms in prediction heart diseases," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 5, no. 6, pp. 1569–1576, 2015.

[8] W. Wiharto, H. Kusnanto, and H. Herianto, "Hybrid system of tiered multivariate analysis and artificial neural network for coronary heart disease diagnosis," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 7, no. 2, p. 1023, 2017.

[9] L. Li, H. Xia, L. Li, and Q. Wang, "Traffic prediction based on svm training sample divided by time," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 11, no. 12, pp. 7446–7452, 2013.

[10] L. Deng and D. Yu, *Deep Learning: Methods and Applications*. NOW Publishers, 2014.

[11] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural Computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

[12] R. McCartney, J. Yuan, and H.-P. Bischof, "Gesture recognition with the Leap Motion controller," in *Proceedings of the Int. Conf. on Image Processing, Computer Vision, and Pattern Recognition (IPCV)*, 2015, pp. 3–9.

[13] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[14] R. Salakhutdinov, "Learning deep generative models," *Annual Review of Statistics and Its Application*, vol. 2, pp. 361–385, 2015.

[15] Y. Hua, J. Guo, and H. Zhao, "Deep belief networks and deep learning," in *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things (ICIT)*, 2015, pp. 1–4.

[16] F. Weichert, D. Bachmann, B. Rudak, and D. Fisseler, "Analysis of the accuracy and robustness of the leap motion controller," *Sensors*, vol. 13, no. 5, pp. 6380–6393, 2013.

[17] Leap Motion Inc., "API overview," https://developer.leapmotion.com/documentation/csharp/devguide/Leap_Overview.html, 2017.

[18] G. E. Hinton, "Training products of experts by minimizing contrastive divergence," *Neural Computation*, vol. 14, no. 8, pp. 1771–1800, 2002.

[19] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proceedings of the 19th International Conference on Neural Information Processing Systems (NIPS'06)*. MIT Press, 2006, pp. 153–160.

[20] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[21] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *Journal of Machine Learning Research*, vol. 11, pp. 625–660, 2010.

[22] A. Setiawan, "Pengenalan karakter menggunakan DBN untuk tulisan tangan di udara yang ditangkap oleh Leap Motion controller," Master thesis, Universitas Gadjah Mada, Yogyakarta, Indonesia, 2016.

[23] M. F. Cohen, J. Shade, S. Hiller, and O. Deussen, "Wang tiles for image and texture generation," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 287–294, 2003.

[24] W. Dong, N. Zhou, and J.-C. Paul, "Tile-based interactive texture design," in *Third International Conference on Technologies for E-Learning and Digital Entertainment*. Springer, 2008, pp. 675–686.

[25] G. E. Hinton, "A practical guide to training restricted Boltzmann machines," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Springer, 2012, pp. 599–619.

[26] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: the Rprop algorithm," in *IEEE International Conference on Neural Networks*, 1993, pp. 586–591.

[27] Y. Lecun and C. Cortes, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/, 2017.

[28] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communication of ACM*, vol. 49, no. 11, pp. 40–45, 2006.