TRAINING ON DEEP NEURAL NETWORK (DRAFT)

BY PUN WAI TONG

0. Notations on Deep Neural Network

All neural networks, e.g. convolution neural network and recurrent neural network, share the same training spirit. In order to keep the note simple and clear, let us consider training a fully connected deep neural network for MNIST classification (supervised learning) throughout the note.

Definition 0.1. Let d be a positive integer. Let $\{n_i\}_{i=1}^{d} \subseteq \mathbb{N}^+$ and $\{f_i\}_{i=2}^{d}$ is a set of activation functions, e.g. *relu*, sigmoid and *tanh*. A n -layer depth fully connected deep neural network is a directed weighted graph (see Figure 0.1) such that

(1) the graph consists of n layers (or called level) and the i^{th} layer contains n_i number of neurons (or called nodes) and an activation function f_i(if $i \geq 2$),
(2) two neurons are connected by an weighted edge only if the layer of the neuron is next to that of the other neuro,
(3) the first layer is called input layer whose neuron values are fed from the processed training data and the last layer is called the output layer whose neuron values are compared with actual labels in the training data. And all other layers are called hidden layers.
(4) By convention, x_j^i stands for j^{th} neuron value on the i^{th} layer and w_{jk}^i corresponds to a weight from j^{th} neuron on the i^{th} layer to the k^{th} neuron on the $(i+1)^{th}$ layer. Also, let b_j^i is a biases value of a j^{th}neuron on the i^{th} layer. The relationship among neuron values are as follows: for $d-2 \geq i \geq 1$, the neuron value is

$$x_k^{i+1} = f_{i+1}(\sum_{j=1}^{n_i} x_j^i w_{jk}^i + b_k^{i+1}). \tag{0.1}$$

for $i = d-1$,

$$\{x_k^d\}_{k=1}^{n_d} = f_d\left(\left\{\sum_{j=1}^{n_{d-1}} x_j^{d-1} w_{jk}^{d-1} + b_k^d\right\}_{k=1}^{n_d}\right) \tag{0.2}$$

1. Processing Training Data

In order to boost up the accuarcy performance of the neural network, raw training data are processed before being fed to the neural network. There are many ways to process raw training data depending on many factors, e.g. the nature of training data itself and the type of deep neural network. For example, raw data of MNIST

Date: 10/03/17.

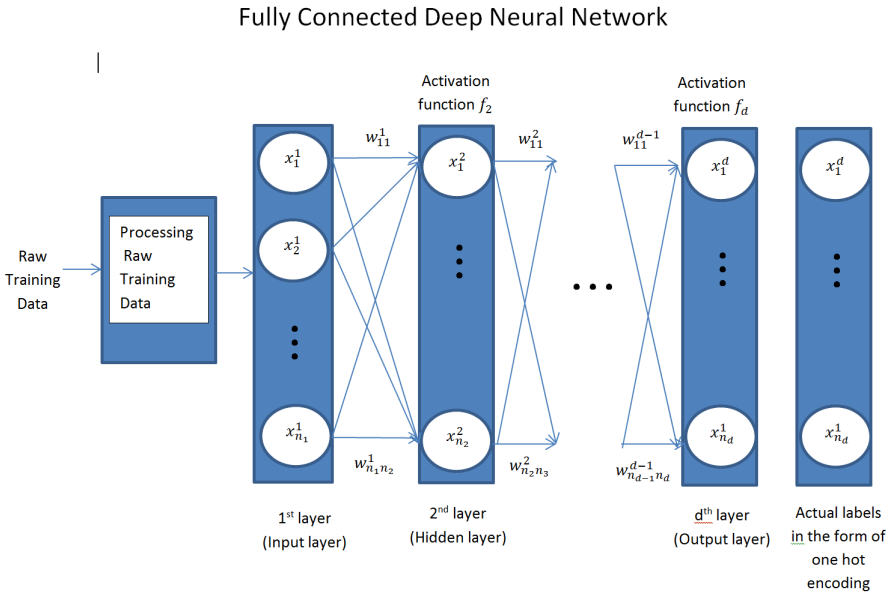

FIGURE 0.1. A d-layer depth fully connected deep neural network

data set are images which are represented by an array of pixels value (may transform from a matrix to an array if necessary) ranging from 0 to 255. If the first two layer of the deep neural network corresponding to restricted Boltzmann machine to learn features, then the raw data of MNIST data set is transformed to a binary values which is either 0 or 1. In our example of MNIST classification using a fully connected deep neural network, each pixel value x of an image in MNIST data set is proceeded by a transformation T

$$T(x) = \frac{x - \frac{255}{2}}{\frac{255}{2}}.$$

so that the value of each input data is symmetric about 0 and lies in $[-1, 1]$.

Remark 1.1. In our example of MNIST classification using a fully connected deep neural network, the size of an image is 28×28 pixels. The number of neurons in the 1^{st} layer (input layer), n_1 , is 28×28.

2. Representation of LAbels in supervised training

In the supervised training, each training example corresponds to a label. For example, in the MNIST data set, a training example is an image while the corresponding label is a digit shown in the image . The label is transformed to a representation so that conclusion, e.g. similarity can be drawn by comparing between the representation and the neuron values in the output layer. One common representation of the MNIST data set is one hot encoding. Suppose MNIST data set contains images of digits from 1 to 10. In the form of one hot encoding, the k^{th}

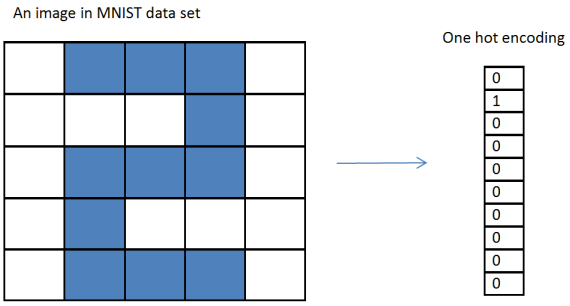

FIGURE 2.1. Training Example and Label of MNIST. The training example is the image of digit 2 and the label is 2. The column matrix on the right is the representation of a digit 2 by using one hot encoding.

digit is represented by a column vector with 10 entries such that the only k^{th} entry is 1 and all other entries are 0 (see Figure 2.1).

Remark 2.1. Since MNIST data set is assumed to contain images of digits from 1 to 10. The number of neurons in the d^{st} layer (output layer), n_d , is 10.

3. Activation Function

Choices of activation functions $f(x)$ are keys to train a deep neural network successfully. Inapropriate activations function could make the deep neural network difficult to be trained. There are some requirements of activation functions. For example,

(1) the activation function $f(x)$ is easy to compute.
(2) the activation function $f(x)$ cannot grow too fast as $|x|$ grows otherwise overflow may occur for neuron values, especially if the neuron network is very huge (i.e., large number of neurons in each layer) and deep (i.e., many layers).
(3) the activation function $f(x)$ is differentiable and its derivatives is easy to compute.
(4) the derivative of activation function should not vanish when $|x|$grows, i.e. $|f^{'}(x)| \to 0$ as $|x| \to \infty$. An activation function having this property is called a non-saturated function.

In our example of MNIST classification, activation functions f_k^i from the 2^{nd} layer to the $(d-1)^{th}$ layer (for $2 \leq i \leq d-1$ and $1 \leq k \leq n_i$) are $Relu$ function which is defined as

$$Relu(x) := \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

It can be seen that

$$(Relu)^{'}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & x < 0 \end{cases}.$$

The activation function $Relu$ is differentiable everywhere except $x = 0$ and its derivative is easy to compute. Also, $Relu$ is non-saturated. Therefore, $Relu$ satisfies the 1^{st}, 3^{nd} and 4^{th} requirement of the activation function mentioned above. Owing to the unclear description of the 2^{nd} requirement of the activation function, it is hard to justify if $Relu$ satisfies the requirement. But practical experience shows that overflow of neuron values are rarely caused by $Relu$ function. All these reason makes $Relu$ one of the most popular activation functions nowaday.

The activation function $f_d(\cdot)$ in the output layer isvery special and different from all activation functions in other layers. First, the 1^{st}, 2^{nd} and 4^{th} activation function requirement are not valued too much for f_d. But the range of the activation function $f_d(\cdot)$ is required to be between 0 and 1 such that $\sum_{k=1}^{n_d} x_k^n = 1$ (normalization requirement). The reason for the normalization requirement is to view the value of the j^{th} neuron on the output layer as the probability of the training example belonging to the j^{th} class.By convention, let us define the notations of neuron value before activation.

Notation 3.1. Let $\tilde{x}_k^{i+1}$ is the k^{th} neuron value on the $i+1^{th}$ layer before activation, i.e. for $d-1 \geq i \geq 1$

$$\tilde{x}_k^{i+1} = \sum_{j=1}^{n_i} x_j^i w_{jk}^i + b_k^{i+1}. \tag{3.1}$$

and

$$x_k^{i+1} = f^{i+1}\left(\tilde{x}_k^{i+1}\right) \tag{3.2}$$

where f^{i+1} be an activation function on the $(i+1)^{th}$ layer.

In order to achieve the normalization requirement, $f_d(\cdot)$ rely on all neurons value before activation in the output layer, i.e.$f_d : \mathbb{R}^{n_d} \to [0,1]^{n_d}$ (c.f. $Relu : \mathbb{R} \to \mathbb{R}$). In our example of MNIST classification, the activation function in the output layer, f_d, is a $softmax$ function defined as

$$\begin{aligned} softmax : \mathbb{R}^{n_d} &\to [0,1]^{n_d} \\ (y_1, ..., y_{n_d}) &= \left(\frac{\exp(y_1)}{\sum_{j=1}^{n_d-1} \exp(y_j)}, ..., \frac{\exp(y_{n_d})}{\sum_{j=1}^{n_d-1} \exp(y_j)} \right). \end{aligned}$$

Note that $n_d = 10$ in our example and the k^{th} neuron value which is $x_k^d = j^{th}$ component of $softmax\left(\left\{\tilde{x}_j^d\right\}_{j=1}^{n_d}\right)$ is interpreted as the probability of the training example belonging to j^{th} digit.

Remark 3.2. A function is called saturated if the function is not non-saturated. A saturated function brings troubles to train a neural network because it is hard to learn. [Say it in training or appendix].

4. Loss function

Loss function is another important criteria for successful training of a deep neural network. Given a set of training examples, the loss function is a real-valued function of parameters (weights and biases in a deep neural network).The construction of a loss function depends on the task and the structure of a deep neural network model. For example, the loss function of the restricted Boltzmann machine for unsupervised learning is about the probability occurrence of some training examples. In our

example of classification on MNIST data set by fully connected deep neural network, the loss function is about the error difference between predicted digits and actual digits for some training examples. Before formulating a loss model, let us introduce some notations first.

Notation 4.1. Let T be a mini-batch which is a subset of the training data and

$$\theta = \left\{ w^i_{jk}, b^{i+1}_k \mid 0 \leq i \leq d-1 \text{ and } 1 \leq j \leq n_i \text{ and } 1 \leq k \leq n_{i+1} \right\} \tag{4.1}$$

be a set of training parameters. Given $t \in T$, we notate $x^i_j(t)$ as the j^{th} neuron value on the i^{th} layer by inputting training example t to the fully connected deep neural network. We also notate $\{y_j(t)\}_{j=1}^{n_d}$ to be the one hot encoding representation of the j^{th} label (it is a digit in our example).

IIn our example of MNIST classification, a cross entropy is used to measure the difference between the predicted label and actual label.

Definition 4.2. The loss function L in our example is defined as

$$L(\theta|T) = \frac{1}{|T|} \sum_{t \in T} \sum_{k=1}^{n_d} y_k(t) log\left(x^d_k(t)\right). \tag{4.2}$$

Just a reminder that x^d_i does not mean x_i to the power d but the k^{th} neuron value in the d^{th} layer. Note that there is only one non-zero term in the inner summation in Eq. (4.2).

5. Initialization of training parameters

Values of training parameters which are weights and biases are initialized in a deep neural network. Owing to being huge and deep in the neural network model, an inappriporiate initialization leads to many problems in the training, e.g. overflow in the neuron values occurs and neuron values lie in the saturated region (see the Appendix). At the end of the day, the neural network cannot learn from training data. Some schemes are introduced for the initialization of training parameters. One common scheme is Xavier initialization which are designed to apply the deep neural network if activation functions on all but the last layer are *tanh* or sigmoid.

Definition 5.1. Let $\mu \in \mathbb{R}$ and $\sigma > 0$ and $N(\mu, \sigma)$ be a Gaussian distribution with mean μ and standard deviation σ. Xavier Initiailization initializes weights and biases in the following way for $0 \leq i \leq d-1$.

(1) If the activation function is *tanh* and sigmoid, weights w^i_{jk} are initialized by a random number drawn from $N(0, \frac{1}{\sqrt{n_i + n_{i+1}}})$.
(2) Biases b^{i+1}_k is initialized to be zero.

Another common initialization scheme for *Relu* activation function is studied by [1] written by K. He, X, Zhang, S. Ren and J. Sun. We will use the scheme in our MNIST classification example and the scheme is defined as follows:

Definition 5.2. He, Zhan, Ren and Sun's initialization scheme initializes weights and biases in the following way for $0 \leq i \leq d-1$.

(1) If the activation function is *Relu* on all but the last layer, weights w^i_{jk} are initialized by a random number drawn from $N(0, \frac{2}{\sqrt{n_i}})$.
(2) Biases b^{i+1}_k is initialized to be zero.

Loosely speaking, a main spirit of both schemes is to keep the variance of neuron values before activation in all layers almost the same. To illustrate the spirit, let us show the argument in He, Zhan, Ren and Sun's initialization scheme.

Theorem 5.3. *Assume x_j^0 from Notation 3.1 follows the same symmetric probability density distribution (aka pdf) around 0 for $0 \leq i \leq d-2$ and $0 \leq j \leq n_i$. Suppose weights w_{jk}^i and neuron values x_j^0 are all independent random variables. If*

$$var\left[w_{jk}^i\right] = \frac{2}{n_i} \tag{5.1}$$

hold, then, under He, Zhan, Ren and Sun's initialization scheme, there exists a constant $c \geq 0$ such that the neuron relationship from Eq.(0.1) leads to

$$Var\left[\tilde{x}_j^i\right] = c. \tag{5.2}$$

for $1 \leq i \leq d-1$ and $1 \leq j \leq n_i$.

Proof. From Eq.(0.1), $\tilde{x}_k^1$ is a sum of a product of independent weights w_{jk}^0 and neuron values x_j^0 which are assumed to be independent and follow a symmetric pdf around 0. Therefore, $\tilde{x}_k^1$ follows the symmetric pdf around 0 and same the same variance among all k. Therefore, using Eq.(0.1) where $i = 1$, we learn $\tilde{x}_k^2$ share the same symmetric pdf near 0 and

$$\begin{aligned} Var\left[\tilde{x}_k^2\right] &= \sum_{j=1}^{n_1} Var\left(x_j^1 w_{jk}^1\right) = \sum_{j=1}^{n_1} Var\left(w_{jk}^1\right) E\left[\left(x_j^1\right)^2\right] \\ &= n_1 Var\left[w_{11}^1\right] E\left[\left(x_1^1\right)^2\right]. \end{aligned} \tag{5.3}$$

By using Eq. (5.1), we can conclude that and an update function $g : \mathbb{R} \times \mathbb{R} \times S \to \mathbb{R}$ where S could be an empty space.

$$Var\left[\tilde{x}_k^2\right] = E\left[\left(x_1^1\right)^2\right]. \tag{5.4}$$

Let dp be a probability measure for $\tilde{x}_1^1$. Since $x_1^1 = Relu\left(\tilde{x}_1^1\right)$, we have

$$E\left[\left(x_1^1\right)^2\right] = \int_{\tilde{x}_k^1 > 0} \left(\tilde{x}_1^1\right)^2 dp = \frac{1}{2} \int_{\mathbb{R}} \left(\tilde{x}_1^1\right)^2 dp = \frac{1}{2} E\left[\left(\tilde{x}_1^1\right)^2\right]. \tag{5.5}$$

The second equality of Eq. (5.5) is because of $\tilde{x}_1^1$ having the symmetric pdf around 0. Since $\tilde{x}_k^1$ has the mean zero, we have

$$E\left[\left(\tilde{x}_k^1\right)^2\right] = Var\left[\tilde{x}_1^1\right]. \tag{5.6}$$

Combining all Eqs (5.3), (5.4), (5.5) and (5.6), we can conclude that for all $1 \leq k \leq n_2$

$$Var\left[\tilde{x}_k^2\right] = Var\left[\tilde{x}_1^1\right].$$

By induction, Eq. (5.6) is asserted.

■

6. Training in Deep neural Netowrk

In order to compute update all weights w_{jk}^i and biases b_k^{i+1} of the loss function L in Eq. (4.2) efficiently in terms of speed and memory, there are two steps in the training of deep neural network: forward propagation and back propagation.

6.1. **Forward Propagation.** Let T be a mini-batch. Suppose values of

$$\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1} \text{ and } \left\{b_j^i\right\}_{1\le j\le n_i,2\le i\le d}$$

are given. Forward propgation is to compute the neuron values of $x_k^i(t)$ layer by layer from left to right in Figure 0.1 through Eqs. (0.1) and (0.2) for each $t \in T$, . For example, we first compute $\left\{x_k^2(t)\right\}_{k=1}^{n_2}$ in the 2^{nd} layer by pluging neuron value $\left\{x_j^1(t)\right\}_{j=1}^{n_1}$ in the first layer (input layer) through Eq. (0.1). Therefore, values of neurons in all layers are known after forward propgation. The implementation of forward propagation is summarized in the following algorithm.

Algorithm 6.1. Forward Propagation:

- Inputs: a mini batch set T , $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$
- Outputs: $\left\{x_j^i(t)\right\}_{1\le j\le n_i,1\le i\le d,t\in T}$
- Algorithm:
  (1) Set $i = 2$
  (2) If $i = d$, then compute $x_j^i(t)$ by Eq. (0.2) for all $1 \le j \le n_i$ and for $t \in T$. Otherwise, compute $x_j^i(t)$ by Eq. (0.1) for all $1 \le j \le n_i$ and for $t \in T$.
  (3) Update i by $i + 1$
  (4) Go to Step 2 until $i = d$

6.2. **Back Propagation.** Given all neuron values, back progation is to compute the derivatives of the loss function L in Eq. (4.2) with respect to each weight w_{jk}^i and biases b_k^{i+1} layer by layer from right to left in Figure 0.1 to update

$$\text{weight } \left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1} \text{ and biases } \left\{b_j^i\right\}_{1\le j\le n_i,2\le i\le d}$$

How the back propgation is implemented? Let us start a new notation for illustration of back propagation.

Definition 6.2. Let $\tilde{x}_j^i$ be as in Notation (3.1) and L be a loss function in Eq. (4.2). Let T be a mini-batch. For $t \in T$, an error derivative

$$E_j^i(t) := \frac{\partial L}{\partial \tilde{x}_j^i}(\theta|\{t\}) \tag{6.1}$$

is defined as a derivative of L with respect to $\tilde{x}_j^i$ evaluated at $\tilde{x}_j^i(t)$ for $2 \le i \le d$ and $1 \le j \le n_i$.

The main idea of the back propgation is to derive a recursive relationship between $\left\{E_j^i(t)\right\}_{j=1}^{n_i}$ and $\left\{E_k^{i+1}(t)\right\}_{k=1}^{n_{i+1}}$ by using a chain rule and compute $\frac{\partial L(\theta|T)}{\partial w_{jk}^i}$ and $\frac{\partial L(\theta|T)}{\partial b_k^{i+1}}$ based on $\left\{E_k^{i+1}(t)\right\}_{k=1,t\in T}^{n_{i+1}}$.

First, let us derive the recursive relationship between $\left\{E_j^i(t)\right\}_{j=1}^{n_i}$ and $\left\{E_k^{i+1}(t)\right\}_{k=1}^{n_{i+1}}$ in the following lemma.

Lemma 6.3. *All notations are as above. For $2 \le i \le d-1$, $1 \le j \le n_i$ and $1 \le k \le n_{i+1}$, we have*

$$E_j^i(t) = \sum_{k=1}^{n_{i+1}} E_k^{i+1}(t) w_{jk}^i (f^i)'|_{\tilde{x}_j^i(t)} \tag{6.2}$$

where $(f^i)'|_{\tilde{x}_j^i(t)}$ *is the derivative of an activation function* f^i *(see Section 3) on the* i^{th} *layer evaluated at* $\tilde{x}_j^i(t)$. *[Note* $L(\theta|\{t\}) = y_i(t)log\left(x_i^d(t)\right)$ *from Eq. (4.2)]*

Proof. The proof is by a direct computation.

$$\begin{aligned} E_j^i(t) &= \frac{\partial L}{\partial \tilde{x}_j^i}(\theta|\{t\}) = \frac{\partial L}{\partial \tilde{x}_j^i}|_{\tilde{x}_j^i(t)} \\ &= \sum_{k=1}^{n_{i+1}} \frac{\partial L}{\partial \tilde{x}_k^{i+1}}|_{\tilde{x}_k^{i+1}(t)} \frac{\partial \tilde{x}_k^{i+1}}{\partial x_j^i}|_{x_j^i(t)} \frac{\partial x_j^i}{\partial \tilde{x}_j^i}|_{\tilde{x}_j^i(t)} \\ &= \sum_{k=1}^{n_{i+1}} E_k^{i+1} w_{jk}^i (f^i)'|_{\tilde{x}_j^i(t)}. \end{aligned}$$

The last equality is because of Eqs. (3.1), (3.2) and (6.1). ■

The following lemma illustrates how to compute $\frac{\partial L}{\partial w_{jk}^i}$ and $\frac{\partial L}{\partial b_k^{i+1}}$ by using $\left\{E_k^{i+1}\right\}_{k=1}^{n_{i+1}}$.

Definition 6.4. Let $k, \ell \in \mathbb{N}$. Then, a kronecker delta $\delta_{k,\ell}$ is defined as

$$\delta_{k,\ell} := \begin{cases} 1 & \text{if } k = \ell \\ 0 & \text{otherwise} \end{cases}$$

Lemma 6.5. *All notations are as above. Given the mini batch set* T, *for* $1 \le i \le d-1$, $1 \le j \le n_i$ *and* $1 \le k \le n_{i+1}$, *the derivative of the loss function* $L(\theta|T)$ *with respect to its trainining parameters* θ *becomes*

$$\frac{\partial L}{\partial w_{jk}^i}(\theta|T) = \sum_{t\in T} E_k^{i+1}(t) x_j^i(t) \text{ and } \frac{\partial L}{\partial b_k^{i+1}}(\theta|T) = \sum_{t\in T} E_k^{i+1}(t). \tag{6.3}$$

Proof. From Eq. (4.2), we can learn that

$$L(\theta|\{t\}) = y_i(t)log\left(x_i^d(t)\right) \text{ and } L(\theta|T) = \frac{1}{T}\sum_{t\in T} L(\theta|\{t\}).$$

By linearity of differentiation, it suffice for us to show

$$\frac{\partial L}{\partial w_{jk}^i}(\theta|\{t\}) = E_k^{i+1}(t) x_j^i(t) \text{ and } \frac{\partial L}{\partial b_k^{i+1}}(\theta|\{t\}) = E_k^{i+1}(t). \tag{6.4}$$

By using a chain rule and Eq. (3.1), Eq.(6.4) becomes

$$\frac{\partial L}{\partial w_{jk}^i}(\theta|\{t\}) = \sum_{\ell=1}^{n_{i+1}} \frac{\partial L}{\partial \tilde{x}_\ell^{i+1}} \frac{\partial \tilde{x}_\ell^{i+1}}{\partial w_{jk}^i} = \sum_{\ell=1}^{n_{i+1}} E_\ell^{i+1} x_j^i \delta_{\ell,k} = E_k^{i+1}(t) x_j^i(t)$$

and

$$= \frac{\partial L}{\partial b_k^{i+1}}(\theta|\{t\}) = \sum_{\ell=1}^{n_{i+1}} \frac{\partial L}{\partial \tilde{x}_\ell^{i+1}} \frac{\partial \tilde{x}_\ell^{i+1}}{\partial w_{jk}^i} = \sum_{\ell=1}^{n_{i+1}} E_\ell^{i+1} \delta_{\ell,k} = E_k^{i+1}(t).$$

The lemma follows.

■

From Lemma 6.3 and 6.5, the back propagation is implemented as follows.

Definition 6.6. An update scheme G is a scheme to update weights $\left\{w_{jk}^i\right\}_{1\le j\le n_i, 1\le k\le n_{i+1}, 1\le i\le d-1}$ and biases $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1}, 1\le i\le d-1}$ in a deep neural network by using current weights and derivatives as inputs (they may not be the only inputs in the scheme).

Algorithm 6.7. Back Propagation Algorithm:

- Inputs: $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$, $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$, neuron values $\left\{x_j^i(t)\right\}_{1\le j\le n_i,1\le i\le d,t\in T}$, actual labels of training data T $\{y_i(t)\}_{1\le i\le n_d,t\in T}$ and an update scheme G in Definition 6.6.
- Outputs: $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$
- Algorithm:
  (1) Set $i=d-1$
  (2) If $i=d-1$, compute $E_k^{i+1}(t)$ for $1\le k\le n_{i-1}$ by Eq. (6.1). Otherwise, compute $E_k^{i+1}(t)$ for $1\le k\le n_{i-1}$ by Eq. (6.2).
  (3) Compute $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)$ for $1\le j\le n_i$ and $1\le k\le n_{i+1}$ by using 6.3 where T is $\left\{x_j^1(t)\right\}_{1\le j\le n_1,t\in T}$
  (4) For $1\le j\le n_i$ and $1\le k\le n_{i+1}$, update weights w_{jk}^i and biases b_k^{i+1} by using the update scheme G with inputs $\{y_i(t)\}_{1\le i\le n_d,t\in T}$, w_{jk}^i and b_k^{i+1} and $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)$ (may have more inputs if necessary).
  (5) Delete $E_k^{i+2}(t)$ for $1\le k\le n_{i+2}$,if exists
  (6) Delete $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)$ for $1\le j\le n_i$ and $1\le k\le n_{i+1}$
  (7) Update i by $i-1$
  (8) Go to Step 2 until $i=0$.

Remark 6.8. Step 4 in Algorithm 6.7 still looks vague and confusing. Readers will have a more clear idea of the implementation of Aglorithm 6.7 after some examples of updates cheme G are introduced. There are different methods to train the deep neural network because different update schemes G are applied in Step 4 in Algorithm 6.7 , e.g. stochastic gradient descent, momentum method and ADAM method (See in later subsections).

By using forward and back propagation repeatedly on different mini batch data set T, an optimal training parameter set θ in Eq. (4.1) is found to minimize the loss function L in Eq. (4.2). [Note, in some other examples, we may need to maximize the loss function, e.g. if the loss function is a maximum likelihood function.] The algorithm of a training fully-connected deep neural network is summized as follows:

Definition 6.9. Let T be a mini batch set which is a subset of training data. Suppose I be the set of classes of labels in T, e.g. in example of the MNIST data set, $I=\{1,2,...,10\}$. Let D_i be the number of training examples in T corresponding to a label $i\in I$. If there is a constant $c\ge 0$ such that $D_i\approx c$ for all $i\in I$. Then, T is said to be balanced.

Algorithm 6.10. Training in a fully-connected deep neural network:

- Inputs: a total training set $\mathscr{T}$ and m which the size of mini batch set T and an update function g and $itr\in\mathbb{N}$
- Outputs: $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$
- Algorithm:

(1) Initialization of $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$ (e.g. see He, Zhan, Ren and Sun's initialization scheme in Definition 5.2)
(2) for $i = 1, ..., itr$ Do
  (a) Randomly partition $\mathscr{T}$ into mini batch sets $\{T_\alpha\}_{\alpha\in A}$ such that $|T_\alpha| \approx m$ for $\alpha \in A$, $\coprod_{\alpha\in A} T_\alpha = \mathscr{T}$ and T_α are balanced for $\alpha \in A$.
  (b) For α in A Do
    (i) forward propagation with three inputs T_α, $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$. Then $\left\{x_j^i(t)\right\}_{1\le j\le n_i,1\le i\le d,t\in T}$ is outputted.
    (ii) back propagation with four inputs $\left\{x_j^i(t)\right\}_{1\le j\le n_i,1\le i\le d,t\in T}$, $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$, $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$ and an update function g. Then $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$ are outputted.

Remark 6.11. In Step 2a in Algorithm 6.10, balanced T_α is hard to achieve sometimes. If the balanced condition of T_α fails, we may use other techniques to train the deep neural network which are not covered in this lecture note.

7. Update Scheme G in Definition 6.9

This lecture note just focus on three update schemes—stochastic gradient descent, Nesterov momentum and ADAM method.

7.1. Stochastic gradient descent.

If an update scheme G is a stochastic gradient descent, then the implementation of G to update weights and biases is as follows:

Algorithm 7.1. Stochastic gradient descent:

- Inputs: w_{jk}^i, b_k^{i+1}, $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)$ and actual labels of training data $\{y_i(t)\}_{1\le i\le n_d,t\in T}$ from Algorithm 6.7 and learning rate η
- Outputs: w_{jk}^i and b_k^{i+1}
- Algorithm:
  (1) $w_{jk}^i \leftarrow w_{jk}^i - \eta\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $b_k^{i+1} \leftarrow b_k^{i+1} - \eta\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)$.

The stochastic of the update scheme comes from a mini batch set T. From Algorithm 6.10, the mini batch set T is different in each iteration. Therefore the loss function L in Algorithm 7.1 is different in each iteration. This is the stochastic part. Despite loss functions being different in each iteration, training examples in each mini-batch set T should follows the same probability distribution because we randomly divide the total training set $\mathscr{T}$ without any biases into mini-batch sets T such that each T is balanced (see Definition 6.9) and the size of each T is almost the same. Therefore, statistically speaking, noise in each mini batch set T should be cancel out during training and the update scheme can output optimal weights and biases which can minimize the actual loss function which is $L(\theta|\mathscr{T})$. Moreover, the learning rate η is set to be 0.01 or 0.001 is good enough in practice.

Remark 7.2. Actual labels of training data $\{y_i(t)\}_{1\leq i\leq n_d, t\in T}$ is required to compute $\frac{\partial L}{\partial w^i_{jk}}(\theta|T)$ and $\frac{\partial L}{\partial b^{i+1}_k}(\theta|T)$ (see Eq. 4.2).

7.2. **Momentum method.** In the author's point of view, the momentum (which is also known as the first momentum) of a variable in the loss function is considered as a discounted rate of accumlated derivatives of the loss function with respect to the variable.

Notation 7.3. Let mw^i_{jk} and mb^{i+1}_k be momentums of the weight w^i_{jk} and b^{i+1}_kin a neural network respectively.

If an update scheme G is momentum method, then the implementation of G to update weights and biases is as follows:

First, the step 1 of the Training in a fully-connected deep neural network (see Algorithm 6.10) is modified as follows:

Algorithm 7.4. Modified Step 1 of training in a fully-connected deep neural network (Algorithm 6.10)

(1) Set $\left\{mw^i_{jk}\right\}_{1\leq j\leq n_i, 1\leq k\leq n_{i+1}, 1\leq i\leq d-1} = 0$ and $\left\{mb^{i+1}_k\right\}_{1\leq k\leq n_{i+1}, 1\leq i\leq d-1} = 0$ and initialization of $\left\{w^i_{jk}\right\}_{1\leq j\leq n_i, 1\leq k\leq n_{i+1}, 1\leq i\leq d-1}$ and $\left\{b^{i+1}_k\right\}_{1\leq k\leq n_{i+1}, 1\leq i\leq d-1}$ (e.g. see He, Zhan, Ren and Sun's initialization scheme in Definition 5.2)

Algorithm 7.5. Momentum method:

- Inputs: w^i_{jk}, b^{i+1}_k, mw^i_{jk}, mb^{i+1}_k, $\frac{\partial L}{\partial w^i_{jk}}(\theta|T)$ and $\frac{\partial L}{\partial b^{i+1}_k}(\theta|T)$ and actual labels of training data $\{y_i(t)\}_{1\leq i\leq n_d, t\in T}$ from Algorithm 6.7 and learning rates η and γ
- Outputs: w^i_{jk}, b^{i+1}_k mw^i_{jk} and mw^{i+1}_{jk}
- Algorithm:
  (1) $mw^i_{jk} \leftarrow \gamma mw^i_{jk} - \eta\frac{\partial L}{\partial w^i_{jk}}(\theta|T)$ and $mb^{i+1}_k \leftarrow \gamma mb^{i+1}_k - \eta\frac{\partial L}{\partial w^i_{jk}}(\theta|T)$
  (2) $w^i_{jk} \leftarrow w^i_{jk} + mw^i_{jk}$ and $b^{i+1}_k \leftarrow b^{i+1}_k + mb^{i+1}_k$.

The learning rate γ and η lie in $(0, 1]$. The assignment of learning rate η is same as that in stochastic gradient descent (see Subsection 7.1). Another learning rate γ is usually set to be 0.5 initially and then gradually and slowly increase up to 0.9. There are many studies why the momentum method can improve training in a neural network. One of my favorite motivation of momentum method comes from a PDE decribing the dynmaics of an object in a viscous medium. Suppose a point mass m moves in a viscous medium with friction coefficient μ under the influence of a conservative force field with potential energy $E(w)$. Let the configuration space is a 1 dimensional real line and w stands for a position of the point mass m in the configuration space. The Netwonian equation is as follows:

$$m\frac{d^2w}{dt^2} + \mu\frac{dw}{dt} = -\nabla_w E(w). \tag{7.1}$$

Suppose we discretize the above Eq. (7.1) by substituting

$$\frac{dw}{dt} = \frac{w_{t+\triangle t} - w_t}{\triangle t} \text{ and } \frac{d^2w}{dt^2} = \frac{w_{t+\triangle t} - 2w_t + w_{t-\triangle t}}{(\triangle t)^2}.$$

We can learn that

$$m\left(\frac{w_{t+\triangle t}-w_t}{\triangle t}\right)=-\frac{m\left(\triangle t\right)}{m+\mu\triangle t}\nabla_w E\left(w\right)+\frac{m}{m+\mu\triangle t}\left[m\left(\frac{w_t-w_{t-\triangle t}}{\triangle t}\right)\right]. \quad (7.2)$$

If $m\left(\frac{w_{t+\triangle t}-w_t}{\triangle t}\right)$ is considered as the momentum term in the next step p_{t+1} and $m\left(\frac{w_t-w_{t-\triangle t}}{\triangle t}\right)$ is consider the current momentum term p_t, then Eq. (7.2) is analogous to the momentum update formula in Step 1 in Algorithm 7.5 with $\gamma=\frac{m}{m+\mu\triangle t}$ and $\eta=\frac{m(\triangle t)}{m+\mu\triangle t}$.

Remark 7.6. Readers may read another implementation of updates in the momentum method as follows:

- Algorithm (another implementation of updates in the momentum method):
  (1) $mw_{jk}^i\leftarrow\gamma mw_{jk}^i+\eta\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $mb_k^{i+1}\leftarrow\gamma mb_k^{i+1}+\eta\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$
  (2) $w_{jk}^i\leftarrow w_{jk}^i-mw_{jk}^i$ and $b_k^{i+1}\leftarrow b_k^{i+1}-mb_k^{i+1}$.

It is an exercise to show that Steps 1 and 2 in Algorithm 7.5 are equivalent to Steps 1 and 2 in Algorithm 7.6

Remark 7.7. Nesterov accelerated gradient method is very similar to the momentum method. The only difference is the evaluation of a derivative of the loss function in the Step 1 in Algorithm 7.5. $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ are evaluated at $w_{jk}^i-\gamma mw_{jk}^i$ and $b_k^{i+1}-\gamma mb_k^{i+1}$ respectively in Nesterov accerlated gradient method while $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ are evaluated at w_{jk}^i and b_k^{i+1} respectively in momentum method.

7.3. **Adam method .** The momentum in the momentum method (see Subsection 7.2) is called the first momentum since the first momentum only capture the information of a derivative of loss function to the power 1. The square of a derivative of loss function also helps to speed up the convergence of weight w_{jk}^i and biases b_k^{i+1}. Similarly to the first momentum, the second momentum of a variable in the loss function is considered as a discounted rate of accumlated square of derivatives of the loss function with respect to the variable. Adam method uses both the first and second momentum to train the neural network.

Notation 7.8. Let vw_{jk}^i and vb_k^{i+1} be the second momentums of the weight w_{jk}^i and b_k^{i+1}in a neural network respectively.

If an update scheme G is Adam method, then the implementation of G to update weights and biases is as follows:

First, the step 1 of the Training in a fully-connected deep neural network (see Algorithm 6.10) is modified as follows:

Algorithm 7.9. Modified Step 1 of training in a fully-connected deep neural network (Algorithm 6.10)

(1) Set $\left\{mw_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}=0$ and $\left\{mb_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}=0$ and $\left\{vw_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}=0$ and $\left\{vb_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}=0$ and initialization of $\left\{w_{jk}^i\right\}_{1\le j\le n_i,1\le k\le n_{i+1},1\le i\le d-1}$ and $\left\{b_k^{i+1}\right\}_{1\le k\le n_{i+1},1\le i\le d-1}$ (e.g. see He, Zhan, Ren and Sun's initialization scheme in Definition 5.2)

Algorithm 7.10. Adam method:

- Inputs: w_{jk}^i, b_k^{i+1}, mw_{jk}^i, mb_k^{i+1}, vw_{jk}^i, vb_k^{i+1}, $\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)$ and actual labels of training data $\{y_i(t)\}_{1\leq i\leq n_d, t\in T}$ from Algorithm 6.7, learning rates η, γ_1 and γ_2 and iteration itr from Algorithm 6.10
- Outputs: w_{jk}^i, b_k^{i+1} mw_{jk}^i , mw_{jk}^{i+1}, mb_k^{i+1}, vw_{jk}^i and vb_k^{i+1}
- Algorithm:
  (1) Set a stablizing constant $\epsilon = 10^{-8}$
  (2) $mw_{jk}^i \leftarrow \gamma_1 mw_{jk}^i + (1-\gamma_1)\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ and $mb_k^{i+1} \leftarrow \gamma_1 mb_k^{i+1} + (1-\gamma_1)\frac{\partial L}{\partial w_{jk}^i}(\theta|T)$ (Update the biased first momentum)
  (3) $mw_{jk}^i \leftarrow \gamma_2 mw_{jk}^i + (1-\gamma_2)\left(\frac{\partial L}{\partial w_{jk}^i}(\theta|T)\right)^2$ and $mb_k^{i+1} \leftarrow \gamma_2 mb_k^{i+1} - (1-\gamma_2)\left(\frac{\partial L}{\partial w_{jk}^i}(\theta|T)\right)^2$ (Update the biased second momentum)
  (4) $\hat{mw}_{jk}^i = \frac{mw_{jk}^i}{(1-\gamma_1^{itr})}$ and $\hat{mb}_k^{i+1} = \frac{mb_k^{i+1}}{(1-\gamma_1^{itr})}$ (Convert to the unbiased first momentum)e a
  (5) $\hat{vw}_{jk}^i = \frac{vw_{jk}^i}{(1-\gamma_2^{itr})}$ and $\hat{vb}_k^{i+1} = \frac{vb_k^{i+1}}{(1-\gamma_2^{itr})}$ (Convert to the unbiased second momentum)
  (6) $w_{jk}^i \leftarrow w_{jk}^i - \eta\frac{\hat{mw}_{jk}^i}{\sqrt{\hat{vw}_{jk}^i}+\epsilon}$ and $b_k^{i+1} \leftarrow b_k^{i+1} - \eta\frac{\hat{mb}_k^{i+1}}{\sqrt{\hat{vb}_k^{i+1}}+\epsilon}$.

In practice, γ_1 is set to be 0.9 and γ_2 is set to be 0.999 and η is set to be 0.001. Also, let us clarify the unbiased first and second momentums. In formally speaking and making abuse of notations, the first momentum is said to be unbiased if $E\left[\hat{mw}_{jk}^i\right] = E\left[\frac{\partial L}{\partial w_{jk}^i}(\theta|T)\right]$ and $E\left[\hat{mb}_k^{i+1}\right] = E\left[\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)\right]$ while the second momentum is said to be unbiased if $E\left[\hat{vw}_{jk}^i\right] = E\left[\left(\frac{\partial L}{\partial w_{jk}^i}(\theta|T)\right)^2\right]$ and $E\left[\hat{vb}_k^{i+1}\right] = E\left[\left(\frac{\partial L}{\partial b_k^{i+1}}(\theta|T)\right)^2\right]$.

8. Accuracy

After training the neural network, an optimal parameters, $\left\{w_{jk}^i\right\}_{1\leq j\leq n_i, 1\leq k\leq n_{i+1}, 1\leq i\leq d-1}$ and $\left\{b_k^{i+1}\right\}_{1\leq k\leq n_{i+1}, 1\leq i\leq d-1}$, in the neural network (See Eq. (4.1)) is obtained to minimize the loss function. It should be noted that although the optimal parameters are usually local minimium point, the optimal parameters are already good enough for the neural network to do many practical tasks. Although loss function is minimized, the value of loss function is not a good indicator to measure the accuracy performance of the nerual netwokr after training. One reason is that the definition of loss function (see Eq. (4.2)) is not directly related to the accuracy predictions by the neural network, hence it is not easy to judge the accuracy performance by the loss function. A more simple and straight forward metric is required to measure the accuracy performance of predictions by the neural network.

Definition 8.1. Let $\{y_j\}_{j=1}^{n_d}$ be a vector of n_d entries whose values are non negative numbers. An convertor h is defined to convert from one hot encoding representation

to the actual label (scalar value) and its formula is as follows:

$$h\left(\{y_j\}_{j=1}^{n_d}\right) := min\left(argmax_{1\leq j\leq n_d} y_j\right).$$

In our example of MNIST classification , $n_d = 10$. Note that, to make things simple, if $\{y_j\}_{j=1}^{n_d}$ has more than one maximum values. Smaller index is outputted.

Definition 8.2. Let $\mathscr{T}$ be a set of testing data. Let $\left\{x_j^1(t)\right\}_{j=1}^{n_1}$ and $\left\{x_j^d(t)\right\}_{j=1}^{n_d}$ be the column of neuron values in the first and last layer of the neural network respectively by inputting the t^{th} data example. The predictor function $p : \mathscr{T} \to \{1, 2, .., 10\}$ in our example is defined as follows:

$$p\left(\left\{x_j^1(t)\right\}_{j=1}^{n_1}\right) := h\left(\left\{x_j^{n_d}(t)\right\}_{j=1}^{n_d}\right)$$

In our example of MNIST classification , $n_d = 10$.

Definition 8.3. Let $\{y_j(t)\}_{j=1}^{n_d}$ be the actual labels of the t^{th} testing example in $\mathscr{T}$ in one hot encoding representation. The accuracy indicator ACC is an indicator to measure the accuracy performance of prediction by the neural network on testing data $\mathscr{T}$ and the formula is defined as follows:

$$ACC(\mathscr{T}) := \frac{1}{|\mathscr{T}|}\sum_{t\in\mathscr{T}} \delta_{p\left(\left\{x_j^1(t)\right\}_{j=1}^{n_1}\right),h\left(\{y_j(t)\}_{j=1}^{n_d}\right)} \tag{8.1}$$

where kronecker delta $\delta_{k,\ell}$ is defined in Notation 6.4.

The accuracy indicator $ACC(\mathscr{T})$ lies between 0 and 1. The $ACC(\mathscr{T})$ is getting to 1 if the neural network can predict labels very accurately. Also, there is a very common misunderstanding that the loss function L and the accuracy indicator $ACC(\mathscr{T})$ is always strictly correlated, i.e. lower the loss function L always results in higher the accuracy $ACC(\mathscr{T})$ or higher the loss function always results in lower the accuracy $ACC(\mathscr{T})$. Sometimes the decrease in the loss function L does not increase the accuracy indicator $ACC(\mathscr{T})$. One reason is that the loss function L only depends on training dataset only and there is no testing data information in the loss function. Over training can cause an overfitting problem which will be discussed later. Another reason is that the decrease in the loss function could make the prediction of the testing examples which were already correctly predicted before more confidently and could make the prediction of testing example which were incorrectly predicted before less confident but still be incorrect. The following numerical example can explain the second reason more clearly.

Example 8.4. In the MNIST dataset, the neural network is tested by two testing data. The label of the first testing data is digit 1 and the label of the second testing data is digit 2 respectively. Suppose the loss function L is currently 0.5. The prediction of the neural network is summarized in Table 1. If the neural network is trained a little bit more, the loss function L drops further to 0.2 and the prediction result is summarized in Table 2. After comparing Tables 1 and 2, we can see that lower the loss function L does not always result in higher the accuracy indicator $ACC(\mathscr{T})$.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. URL `http://arxiv.org/abs/1502.01852`.

	prediction of digit 1	prediction of digit 2
1st testing example	0.75	0.25
2nd testing example	0.7	0.3

TABLE 1. The table is the prediction result of the neural network when the loss function L is 0.5. The values inside the table is the probability of the testing example being the digit in the row header. For example, in the first row, the neural network predicts the 1st testing example has 75% being digit 1 and 25% being digit 2. The $ACC(\mathscr{T})$ is 0.5 (see Eq. (8.1)).

	prediction of digit 1	prediction of digit 2
1st testing example	0.9	0.1
2nd testing example	0.55	0.45

TABLE 2. The table is the prediction result of the neural network after further training. In this case, the loss function L is 0.2 but the $ACC(\mathscr{T})$ is still 0.5 (see Eq. (8.1)).

Under progress

E-mail address: punwai.tong@gmail.com