Software Review

Journal of Educational and Behavioral Statistics
2020, Vol. 45, No. 2, pp. 227–248
DOI: 10.3102/1076998619872761
Article reuse guidelines: sagepub.com/journals-permissions

Deep Learning With TensorFlow: A Review

Bo Pang
Erik Nijkamp
Ying Nian Wu
UCLA

This review covers the core concepts and design decisions of TensorFlow. TensorFlow, originally created by researchers at Google, is the most popular one among the plethora of deep learning libraries. In the field of deep learning, neural networks have achieved tremendous success and gained wide popularity in various areas. This family of models also has tremendous potential to promote data analysis and modeling for various problems in educational and behavioral sciences given its flexibility and scalability. We give the reader an overview of the basics of neural network models such as the multilayer perceptron, the convolutional neural network, and stochastic gradient descent, the most commonly used optimization method for neural network models. However, the implementation of these models and optimization algorithms is time-consuming and error-prone. Fortunately, TensorFlow greatly eases and accelerates the research and application of neural network models. We review several core concepts of TensorFlow such as graph construction functions, graph execution tools, and TensorFlow's visualization tool, TensorBoard. Then, we apply these concepts to build and train a convolutional neural network model to classify handwritten digits. This review is concluded by a comparison of low- and high-level application programming interfaces and a discussion of graphical processing unit support, distributed training, and probabilistic modeling with TensorFlow Probability library.

Keywords: *adaptive testing; computation; modeling; neural network; program evaluation; statistics; technology*

1. Introduction

In the field of machine learning, neural networks have shown remarkable success in a wide range of areas such as computer vision (Krizhevsky, Sutskever, & Hinton, 2012), natural language processing (Collobert & Weston, 2008), and bioinformatics (Min, Lee, & Yoon, 2017). These models also have

enormous potential to promote data analysis and modeling for various problems in educational and behavioral sciences given their flexibility and scalability as universal function approximator. In recent years, a variety of software libraries have been released, which significantly ease and accelerate the research and application of neural network models. TensorFlow, originally created by Google researchers, is the most popular one among the plethora of these libraries. This article aims to provide an introduction to the basics of neural networks and apply the model as a computational tool. To benefit the most from this review, the reader should have a basic understanding of the programming language Python. In this article, we begin with setting up the notation and terminology in the context of linear and logistic regression and then introduce common neural network models and their optimization methods. We lay out the basic elements to build a TensorFlow program and illustrate these concepts with a deep convolutational neural network model that classifies handwritten digits. It is followed by a discussion of other features and tools available in TensorFlow such as high-level application programming interfaces (APIs), graphical processing unit (GPU) support, and distributed training.

2. Neural Network Basics

2.1. Background and Notation

Let us first consider a linear regression model. Suppose $X_i \in \mathbb{R}^p$ denotes the predictors or features, and $y_i \in \mathbb{R}$ is the outcome or target variable for the ith sample and $i \in \{1, \ldots, n\}$. Then, linear regression model is given by

$$y_i = X_i^T \boldsymbol{\beta} + \varepsilon_i, \tag{1}$$

where $\boldsymbol{\beta} \in \mathbb{R}^p$ collects the parameters and $\varepsilon_i \sim N(0, \sigma^2)$ independently. We can simply let X_1 to be all 1s to include an intercept term.

If $y_i \in \{0, 1\}$ representing the category of the ith sample, then the problem is often modeled with a logistic regression. It assumes that $y_i | X_i \sim \text{Bernoulli}(p_i)$ and

$$\begin{aligned} p_{\mathrm{b}}(y_i = 1 | X_i) &= \text{sigmoid}(X_i^T \boldsymbol{\beta}) = \frac{1}{1 + \exp(-X_i^T \mathrm{b})}, \\ p_{\mathrm{b}}(y_i | X_i) &= p_i^{y_i} (1 - p_i)^{1 - y_i} \end{aligned} \tag{2}$$

where $p_i = p_{\boldsymbol{\beta}}(y_i | X_i)$.

To fit the model or learn the parameters (in machine learning terminology), we need to define a *loss function* to penalize errors made by the model. For linear regression, the common and convenient one is the squared loss,

$$L(y_i, X_i; \boldsymbol{\beta}) = (y_i - X_i^T \boldsymbol{\beta})^2. \tag{3}$$

For logistic regression, we usually use the negative log likelihood as loss function, which is

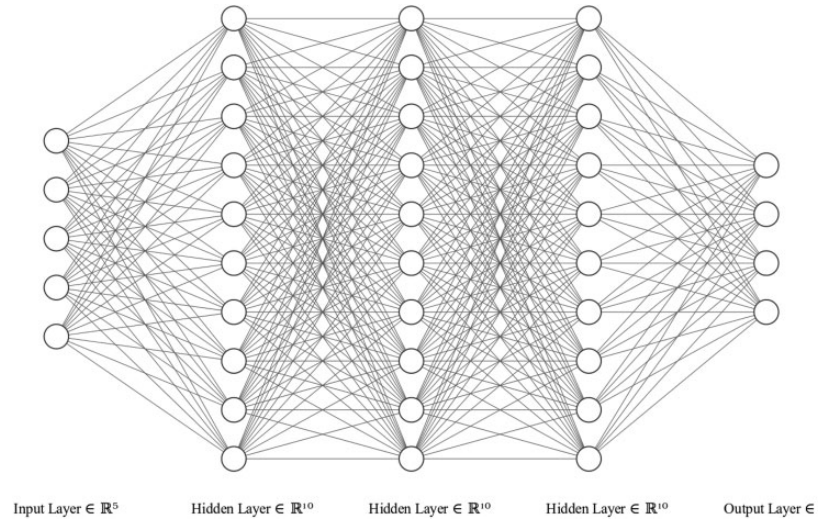

FIGURE 1. *Multilayer perceptron (three hidden layers here) or feedforward neural network. There is an input layer (or an input vector), an output layer, and multiple hidden layers (or hidden vectors). Each layer is a linear transformation of the layer beneath, followed by an elementwise nonlinear transformation. This figure was generated using LeNail (2019).*

$$L(y_i, X_i; \boldsymbol{\beta}) = -\log p_{\boldsymbol{\beta}}(y_i|X_i) = -[y_i \log p_i + (1 - y_i)\log(1 - p_i)]. \tag{4}$$

Minimizing the average loss incurred by all samples, $\frac{1}{n}\sum_{i=1}^{n} L(y_i, X_i; \boldsymbol{\beta})$, over all the parameter β yields the optimal parameter values. This optimization problem has an analytic solution for simple models such linear regression but requires iterative algorithms in most cases such as logistic regression.

Roughly speaking, most neural network models are designed to solve regression or classification (i.e., logistic regression) problems, but rather than utilizing a linear function, $X_i^T \boldsymbol{\beta}$, to model the mapping from X_i to y_i, neural networks employ a class of more sophisticated functions that have much richer and more flexible representations. Multilayer neural networks are deemed flexible and even universal functional approximators (Cybenko, 1989; Hornik, 1991). Due to the complexity of neural networks, their parameters are often optimized through an iterative method, mini-batch gradient descent.

2.2. Multilayer Perceptron (MLP)

A typical example of neural network is a feedforward neural network or an MLP. Suppose X_i and y_i have the same meaning as above, but y_i is allowed to be

either one dimensional as above or multidimensional like in multivariate regression to accommodate more general cases. Figure 1 provides an illustration. An MLP consists of an input layer, an output layer, and multiple hidden layers in between. Each layer can be represented by a vector, which is obtained by multiplying the layer below it by a weight matrix, plus a bias vector, and then transforming each element of the resulting vector by some nonlinear function such as *sigmoid*, $f(x) = 1/[1 + \exp(-x)]$; *rectified linear unit (ReLU)*, $f(x) = \max(0, x)$; and *softplus*, $f(x) = \log[1 + \exp(x)]$. ReLU is a default in current practice. More formally, the network computes a mapping, $y = f_\theta(X)$, with the following recursive structure:

$$\begin{array}{l} s_l = W_l h_{l-1} + b_l \\ h_l = f_l(s_l) \end{array}, \tag{5}$$

for $l = 1, \ldots, L$, where l denotes the layer, with $h_0 = X$, which is the input vector, and h_L is used to predict y, which is the output. W_l and b_l are the weight matrix and bias vector, respectively, and $\theta = \{W_l, b_l, l = 1, \ldots, L\}$ collects all parameters. f_l is an elementwise nonlinear transformation. In other words, each component of h_l is obtained by a nonlinear transformation of the corresponding component of s_l, and $h_{lk} = f_l(s_{lk})$, where k indicates the kth component of the vectors s_l and h_l.

2.3. Convolutional Neural Network (CNN)

In the neural network, $h_l = f_l(W_l h_{l-1} + b_l)$, the linear transformation $s_l = W_l h_{l-1} + b_l$ that maps h_{l-1} to s_l can be highly structured. One important structure is the CNN (LeCun, Bottou, Bengio, & Haffner, 1998), where W_l and b_l have a convolutional structure. Specifically, a convolutional layer h_l is organized into a number of feature maps. Each feature map is also called a channel and is obtained by a filter or a kernel operating on h_{l-1}, which is organized into a number of feature maps. Each filter is a local weighted summation, plus a bias, followed by a nonlinear transformation. Figure 2 illustrates the local weighted summation. In this example, we convolve an input feature map with a 3×3 filter to obtain an output feature map. The value of each pixel of the output feature map is obtained by a weighted summation of pixel values of the 3×3 patch of the input feature map around this pixel. We apply the weighted summation around each pixel with the same weights to obtain the output feature map. If there are multiple input feature maps, that is, multiple input channels, the weighted summation is also over the multiple feature maps.

We can apply different filters to the same set of input feature maps. Then, we will get different output feature maps. Each output feature map corresponds to one filter (see Figure 3 for an illustration). After the weighted summation, we may also add a bias and apply a nonlinear transformation such as ReLU. The filter becomes nonlinear. After obtaining the feature maps in h_l,

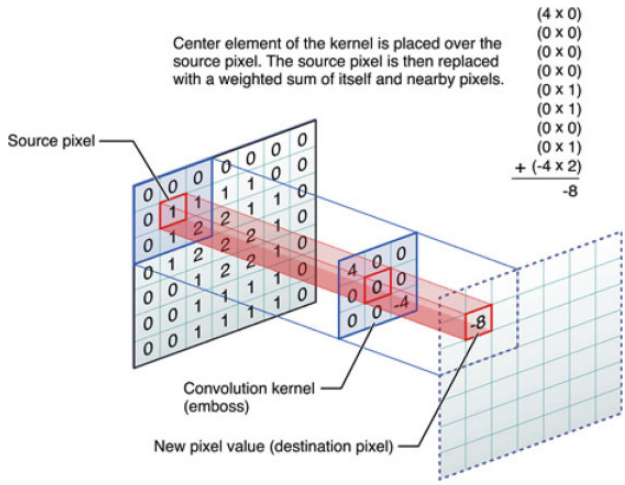

FIGURE 2. *Local weighted summation. Here, the filter or kernel is 3 × 3. It slides over the whole input image or feature map. At each pixel, we compute the weighted sum of the 3 × 3 patch of the input image, where the weights are given by the filter or kernel. This gives us an output image or filtered image or feature map. This figure was retrieved from https://developer.apple.com/library/archive/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html.*

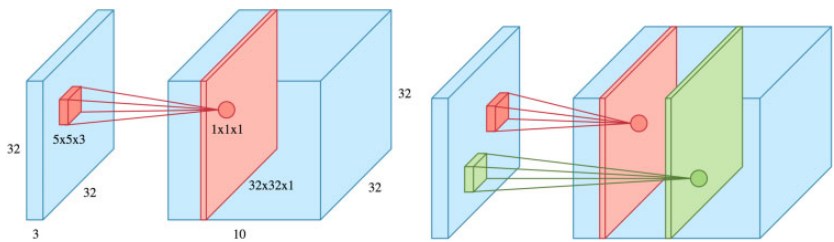

FIGURE 3. *Convolution. The input may consist of multiple channels, illustrated by a rectangle box. Each input feature map is a slice of the box. The local weighted summation in convolution is also over the channels. If the spatial range of the filter is 3 × 3 and the input has three channels, then the filter or kernel is 3 × 3 × 3 box. The spatial range can also be 1 × 1, then the filter involves weighted summation of the three channels at the same pixel. For the input image, there are three channels corresponding to three colors: red, green, black. For the hidden layers, each layer may consist of hundreds of channels. Each channel is obtained by a filter. For instance, in the figure on the right, the red feature map is obtained by the red filter, and the green feature map is obtained by the green filter. This figure was retrieved from https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.*

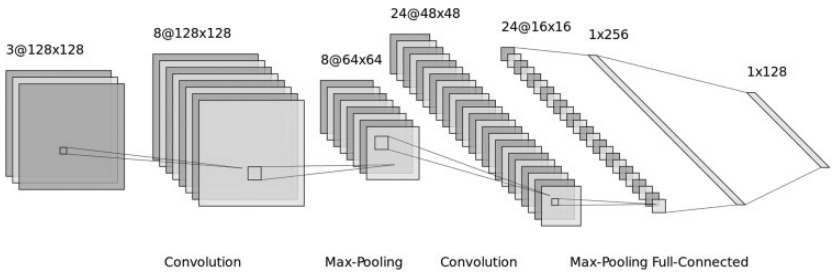

FIGURE 4. *Convolutional neural network. The input image has three channels (R, G, B). Each subsequent layer consists of multiple channels (e.g., eight channels after the first convolutional layer) and is illustrated by multiple squares with one square representing one feature map or channel. Maximum pooling is performed after each convolutional layer so that the feature maps at the higher layers are smaller than those at the lower layer. Meanwhile, the feature maps at the higher layer may have more channels than those at the lower layer, illustrated by the fact that there are more squares at the higher layer than at the lower layer. A fully connected layer consists of* 1×1 *feature maps and is illustrated by a flat rectangle. This figure was generated using LeNail (2019).*

we may perform max pooling. For instance, for each feature map, at each pixel, we replace the value of this pixel by the maximum of the 2×2 patch around this pixel. We may also do average pooling. That is, at each pixel, we replace the value of this pixel by the average of the 2×2 patch around this pixel. Both max pooling and average pooling reduce the size of features maps. For example, max pooling or average pooling with a filter size of 2×2 reduces the width and height of the feature map by half. h_l may have more channels than h_{l-1} because each element of h_l covers a bigger spatial extent than each element of h_{l-1}, and there are more patterns of bigger spatial extent. The output feature map may also be 1×1, whose value is a weighted sum of all the elements in h_{l-1}. h_l may consist of a number of such 1×1 maps. It is called the fully connected layer because each element of h_l is connected to all the elements in h_{l-1}. A CNN consists of multiple layers of convolutional and fully connected layers, with max pooling and average pooling between the layers. Figure 4 offers an example. The input image has three channels (R, G, B). Each subsequent layer consists of multiple channels (e.g., eight channels after the first convolutional layer) and is illustrated by multiple squares with one square representing one feature map or channel. Max pooling is performed after each convolutional layer, so that the feature maps at the higher layers are smaller than those at the lower layer. Meanwhile, the feature maps at the higher layer may have more channels than those at the lower layer, illustrated by the fact that there are more squares at the higher layer than at the lower

layer. A fully connected layer consists of 1×1 feature maps and is illustrated by a flat rectangle.

2.4. Softmax Layer for Classification

Suppose we want to classify the input X_i into one of K categories. We can build a network whose top layer is a K dimensional vector $h_i = (h_{ik}, k = 1, \ldots, K)$. Then, the probability that the input X_i belongs to category k is simply the normalized value:

$$p_{ik} = \frac{e^{h_{ik}}}{\sum_{k'=1}^{K} e^{h_{ik'}}}. \tag{6}$$

We use the notation k' to avoid confusion with the index k in the numerator. This is called softmax probability. In prediction, we can also simply classify X_i to category k whose h_{ik} is the maximum in h_i. This is hard max. Suppose X_i is the input image, and y_i is the output category. Let $p(y|X, \theta)$ be the probability that the input image X belongs to the category y according to the above softmax probability. Then, we can estimate $\theta = \{W_l, b_l, l = 1, \ldots, L\}$ by minimizing the negative log likelihood, $-\sum_{i=1}^{n} \log p(y_i|X_i, \theta)$.

2.5. Stochastic Gradient Descent

Let the training data be (X_i, y_i), $i = 1, \ldots, n$, and $L_i(\theta) = L(y_i, X_i; \theta)$ be the loss caused by (X_i, y_i). For regression, $L(y_i, X_i; \theta) = [y_i - f_\theta(X_i)]^2$, where $f_\theta(x_i)$ is parametrized by a neural network with parameters θ. For classification, $L(y_i, X_i; \theta) = -\log p_\theta(y_i|X_i)$ where $p_\theta(y_i|X_i)$ is modeled by a neural network with parameters θ, with a softmax layer at the top. You may notice the remarkable similarity between loss functions of the neural network models and those of the linear and logistic regression (Equations 3 and 4). Let $L(\theta) = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta)$ be the overall loss averaged over the whole training data set. The gradient descent updates θ at time step $t + 1$ with the rule

$$\theta_{t+1} = \theta_t - \eta_t L'(\theta_t), \tag{7}$$

where η_t is the step size or learning rate, and $L'(\theta_t)$ is the gradient evaluated at θ_t. The gradients are computed with backpropagation (Rumelhart, Hinton, & Williams, 1988), which is essentially applying the chain rule of differentiation. As you will see soon, gradient computation is greatly eased by TensorFlow.

The above gradient descent may be time-consuming because we need to compute $L(\theta)$ by summing over all the examples. If the number of examples is large, the computation can be time-consuming. We may use the following

stochastic gradient descent algorithm (Robbins & Monro, 1951). At each step, we randomly select i from $\{1, 2, \ldots, n\}$. Then, we update θ by

$$\theta_{t+1} = \theta_t - \eta_t L_{i'}(\theta_t), \tag{8}$$

where $L_{i'}(\theta_t)$ is the gradient only for the ith example. Because i is randomly selected, the above algorithm is called the stochastic gradient descent algorithm. Instead of randomly selecting a single example, we may randomly select a mini-batch that consists of a certain number of samples (e.g., 200) and replace $L_{i'}(\theta_t)$ by the average of the mini-batch. The mini-batch gradient descent is the commonly used approach in current practice.

3. TensorFlow Review

TensorFlow is a flexible and scalable software library for numerical computations using dataflow graphs. This library and related tools enable users to efficiently program and train neural network and other machine learning models and deploy them to production. Core algorithms of TensorFlow is written in highly optimized C++ and CUDA (Compute Unified Device Architecture), a parallel computing platform and API created by NVIDIA. It has APIs available in several languages. The Python API is the most complete and stable one. Other officially supported languages include JavaScript, C++, Java, Go, and Swift. Third-party packages are available for more languages such as C# and Ruby. Examples given in this review are all based on Python since it is the easiest to read and use.

In the following sections, we first present the core elements of a TensorFlow program and illustrate them through an example of classifying handwritten digits with a CNN, after which high-level APIs, distribute training and more tools available in TensorFlow are introduced.

3.1. Core Elements

Generally, a TensorFlow program consists of two sections: building a graph of computations (construction phase) and running the computational graph (execution phase). In the construction phase, we often utilize TensorFlow functions to build a computational graph that represents a machine learning model. The graph is composed of edges and nodes. The edges represent data in the form of tensor (e.g., vector, matrix, or higher dimensional data array) that will flow through the graph, which gives the name of the library, TensorFlow. The nodes are called operations that represent computations (e.g., addition, multiplication) on tensors. An operation takes zero or more tensor as input and produces zero or more tensor output. TensorFlow provides basic building blocks such as fully connected layer, convolutional layer, recurrent neural network module, and nonlinear activation functions. Operations created by these functions form the major components of a computational graph that execute the computation from model input to output.

TensorFlow also offers various loss functions such as cross-entropy and mean squared error (MSE). Adding loss computation operations on top of the output tensor completes the forward pass (from input to loss) of a neural network model.

To finish a neural network computational graph, we then need to construct operations that compute the gradients of loss with respect to model parameters (the gradient computation process is also called backward pass since it starts from loss and moves back toward input). Due to the model complexity and the enormous number of parameters, mathematically deriving the gradients from a loss function for all parameters is error-prone and sometimes computationally inefficient. Thus, a central feature of TensorFlow is auto-differentiation (autodiff). It automatically computes gradients, with which gradient-based optimization methods such as stochastic gradient descent (see Equation 8) can be applied to optimize the model parameters. TensorFlow functions like `tf.gradients()` can be utilized to compute the gradients of loss with respect to model parameters. Essentially, `tf.gradients()` adds a series of operations to the model computational graph automatically, which return the gradients desired if executed. Besides autodiff, TensorFlow also offers a variety of off-the-shelf optimizer such as `AdamOptimizer` and `AdagradOptimizer` to make optimization even simpler. Basically, it combines the steps of computing gradients and parameter updating. In short, we use basic modules such as layer construction functions and loss functions to construct the forward pass of a computational graph and autodiff functions or optimizers construct backward pass operations automatically.

In the execution phase, we run the computational graph over some number of updating steps to train the model and improve the model parameters. TensorFlow provides `tf.Session` as a mechanism to run through the graph and execute computations. A `tf.Session` object interacts with physical computational resources such as local CPUs and GPUs or remote devices and places operations in the graph on devices. A `with` block is often used to create a session, which also automatically takes care of closing the session and thus freeing up resources after computations. The `tf.Session.run()` method is the main mechanism for executing operations and evaluating tensors. After receiving a list of operations or tensors (or tensorlike objects), called a fetch list, it automatically backtracks from the list and determines a subgraph that consists of all operations whose outputs are employed to compute the value of the fetches.

3.2. TensorBoard

Neural network models are often large, convoluted structures and may lead to confusion. TensorBoard is a visualization tool that makes it easier to understand and debug TensorFlow programs. Users can use TensorBoard to easily visualize the computation graph of a model, training metrics, and parameter values. A

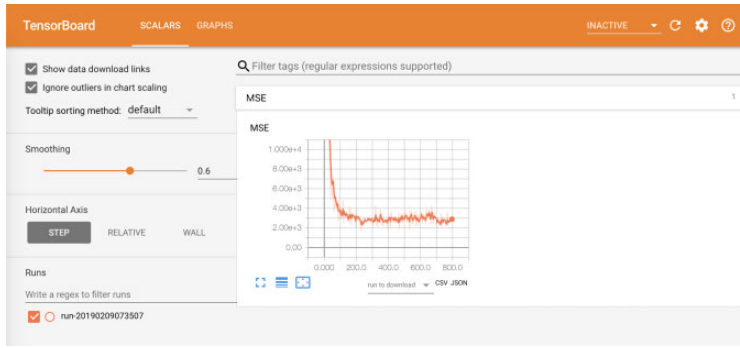

FIGURE 5. *Training statistics visualization with TensorBoard.*

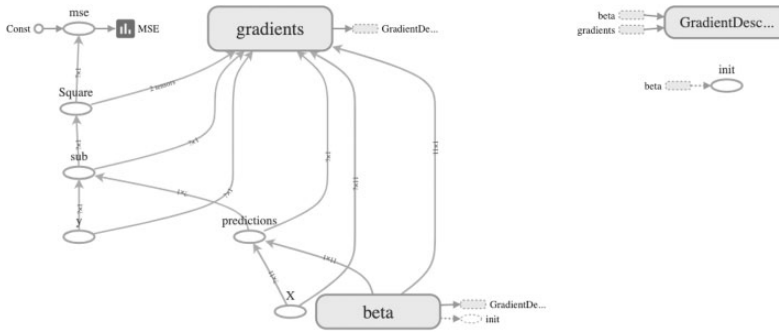

FIGURE 6. *Computational graph of linear regression.*

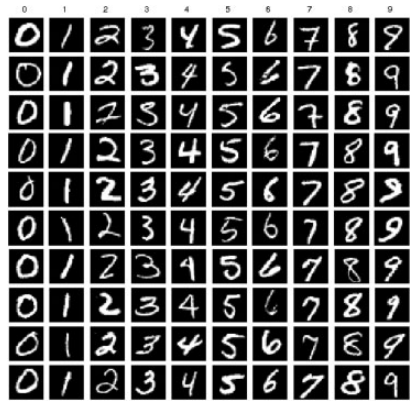

FIGURE 7. *Examples of handwritten digits in MNIST.*

TensorFlow module, `tf.summary`, creates operations to save graph definition and record training statistics. It also writes them as an events file to hard disk in a format that TensorBoard can read. These files will be visualized in a browser when TensorBoard is launched. One example for MSE is shown in Figure 5. It displays MSE as a function of training steps. Figure 6 shows a computational graph that represents a linear model.

3.3. Handwritten Digits Classification With Neural Network in TensorFlow

This section illustrates the core elements introduced above by applying them to build a deep CNN model for classifying handwritten digits. The MNIST database of handwritten digits consists of a training set of 60,000 examples and a test set of 10,000 examples. Figure 7 shows some examples for each digit. Each example is an image of a handwritten digit.

Code Example 1 gives the entire code with comments. At the very beginning, some modules are imported. We then create some directories to store events files for TensorBoard visualization. To start constructing a graph, we create two placeholders, `x` and `y`, to accept feeds of images and labels. Then, the images are reshaped to a four-dimensioanl tensor in the format of `[batch, in_height, in_width, in_channels]`, which is required by a convolutional layer function in TensorFlow. We next construct the first convolutional layer that maps one grayscale image to 32 feature maps. A filter of shape `[filter_height, filter_width, in_channels, out_channels]` and a bias term of shape `[out_channels]` are created with `tf.get_variable()`. The reshaped image and the filter are input to the convolutional function `tf.nn.conv2d()` with specified strides and padding. The stride number indicates the number of pixels that the filters move at a time as we slide them around. Stride of 1 means that the filters move one pixel at a time. `strides=[1, 1, 1, 1]` denotes the strides along the four dimensions of the image tensor. `padding='SAME'` pads 0s around the original feature maps and results in a output size same as the input size if the stride is 1. `tf.nn.conv2d()` returns 32 feature maps for each input image. The bias term is then added to them. It is followed by applying elementwise nonlinearity with `tf.nn.relu()`. This completes defining a convolutional layer. Notice that we put everything under the variable scope `conv1` to group operations that define the first convolutional layer and take a similar strategy to group operations in the following sections. This makes the graph more clear, which is especially important for visualizing a complicated model.

We then use `tf.nn.max_pool()` to create a pooling layer that downsamples the output from the first convolutional layer. It requires specification of the filter shape, strides, and padding. It is followed with another convolutional layer and a second pooling layer. The features of each sample are then put together as a

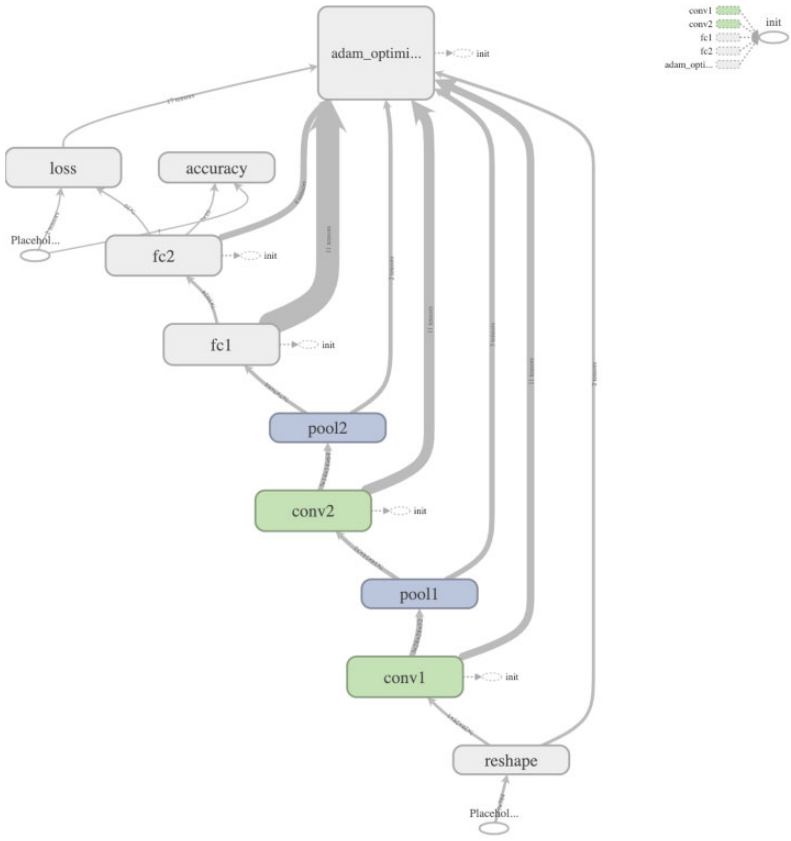

FIGURE 8. *Computational graph of a convolutional neural network model.*

single vector by `tf.reshape()` so that we can apply two fully connected layers. For each layer, a weight matrix and a bias term are created and then applied to the feature maps, followed by nonlinearity. The output of the second fully connected layer for each sample is a vector of length 10. They are employed as logits that can be input to a softmax layer and normalized as a valid probability distribution, and in turn, the negative log likelihood can be computed as the model loss. A loss function `tf.losses.sparse_softmax_cross_entropy()` simplifies the process and takes the logits and observed labels, `y`, directly and produces negative log likelihood or cross-entropy for each sample. Averaging over all samples in a batch (samples will be fed to the model in batch during graph execution) yields the model loss, `cross_entropy`. We then create a node to minimize the loss by an optimizer function, `f.train.Adam`

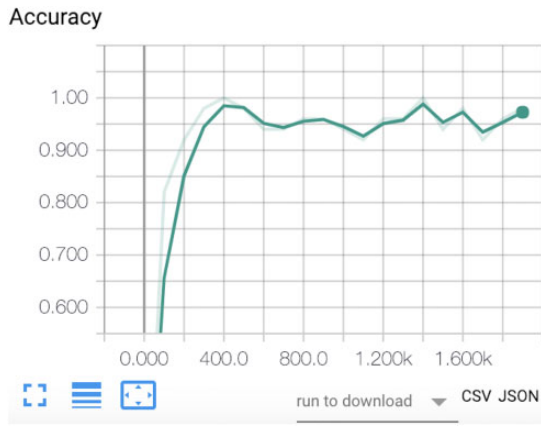

FIGURE 9. *Training accuracy over steps.*

`Optimizer`. We also add an operation to compute prediction accuracy, which can be employed to monitor the training process and to measure the model performance during testing. Finally, initialization and summary operations are added, which completes the construction phase. The graph visualized with TensorBoard is shown in Figure 8.

In the execution phase, we first create a session and initialize all variables. TensorFlow provides functions to download and read MNIST in batch of a given size. The images are stored as a matrix where each row of length 784 represents the pixel values of an image. The labels are stored as a vector with each element identifying one image. The data are read in with the `input_data()` function. We obtain the data in batch, say 50 samples per batch, by the method, `next_batch()`. TensorFlow provides an easy access to some commonly used computer vision data sets. It would also be straightforward to write some Python function to produce features and labels in batch for your own data set. For every other 10 batches, the accuracy is evaluated and added to the events file for TensorBoard visualization and printed out directly in the console. In every batch, we run the `train_step` operation that determines a subgraph that contains nodes associated with this operation, and a batch of `(x, y)` is fed to the graph. This subgraph runs through the model, computes the logits and loss, calculates the gradients, and updates the parameters once. For illustration, the model is trained with 2,000 batches of data. During and after training, we can check the training `accuracy` in TensorBoard. Figure 9 is a screenshot from the TensorBoard, displaying the accuracy curve. The final training accuracy is around .98. After training, we evaluate testing accuracy by running the `accuracy` operation and feeding the testing data to the `run()` method. The testing accuracy is around .97.

```
from TensorFlow.examples.tutorials.mnist import input_data
import TensorFlow as tf
from datetime import datetime

# create directory to store events file
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "Documents/mnist/tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)

#######################################################################
#################### build the graph ##################################
#######################################################################
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.int64, [None])

# Reshape to use within a convolutional neural net.
# Last dimension is for "features" - there is only one here, since images
   are
# grayscale -- it would be 3 for an RGB image, 4 for RGBA, etc.
with tf.name_scope('reshape'):
    x_image = tf.reshape(x, [-1, 28, 28, 1])

# First convolutional layer - maps one grayscale image to 32 feature maps.
with tf.variable_scope('conv1'):
    W_conv1 = tf.get_variable(name='weight_variable',
                          shape=[5, 5, 1, 32],
                          initializer=tf.initializers.truncated_normal(
                             stddev=0.1))
    b_conv1 = tf.get_variable(name='bias_variable',
                          initializer=tf.constant(0.1, shape=[32]))
    h_conv1 = tf.nn.relu(tf.nn.conv2d(x_image, W_conv1, strides=[1, 1, 1,
       1], padding='SAME') + b_conv1)

# Pooling layer - downsamples by 2X.
with tf.name_scope('pool1'):
    h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1, 2, 2, 1],
                      strides=[1, 2, 2, 1], padding='SAME')

# Second convolutional layer -- maps 32 feature maps to 64.
with tf.variable_scope('conv2'):
    W_conv2 = tf.get_variable(name='weight_variable',
                          shape=[5, 5, 32, 64],
                          initializer=tf.initializers.truncated_normal(
                             stddev=0.1))
    b_conv2 = tf.get_variable(name='bias_variable',
                          initializer=tf.constant(0.1, shape=[64]))
    h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, W_conv2, strides=[1, 1, 1,
       1], padding='SAME') + b_conv2)

# Second pooling layer.
with tf.name_scope('pool2'):
    h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1, 2, 2, 1],
                      strides=[1, 2, 2, 1], padding='SAME')
```

```
# Fully connected layer 1 -- after 2 round of downsampling, our 28x28
    image
# is down to 7x7x64 feature maps -- maps this to 1024 features.
with tf.variable_scope('fc1'):
    W_fc1 = tf.get_variable(name='weight_variable',
                            shape=[7 * 7 * 64, 1024],
                            initializer=tf.initializers.truncated_normal(
                               stddev=0.1))
    b_fc1 = tf.get_variable(name='bias_variable',
                            initializer=tf.constant(0.1, shape=[1024]))

    h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
    h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# # Dropout - controls the complexity of the model, prevents co-adaptation
     of
# # features.
# with tf.name_scope('dropout'):
# keep_prob = tf.placeholder(tf.float32)
# h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Map the 1024 features to 10 classes, one for each digit
with tf.variable_scope('fc2'):
    W_fc2 = tf.get_variable(name='weight_variable',
                            shape=[1024, 10],
                            initializer=tf.initializers.truncated_normal(
                               stddev=0.1))
    b_fc2 = tf.get_variable(name='bias_variable',
                            initializer=tf.constant(0.1, shape=[10]))

    y_conv = tf.matmul(h_fc1, W_fc2) + b_fc2

with tf.name_scope('loss'):
    cross_entropy = tf.losses.sparse_softmax_cross_entropy(
        labels=y, logits=y_conv)
    cross_entropy = tf.reduce_mean(cross_entropy)

with tf.name_scope('adam_optimizer'):
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

with tf.name_scope('accuracy'):
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), y)
    correct_prediction = tf.cast(correct_prediction, tf.float32)
    accuracy = tf.reduce_mean(correct_prediction)

# create an initilization operation
init_op = tf.global_variables_initializer()

accuracy_summary = tf.summary.scalar('Accuracy', accuracy)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

```
#########################################################################
#################### create a session ####################################
#########################################################################
with tf.Session() as sess:
    sess.run(init_op)
    mnist = input_data.read_data_sets('/tmp/TensorFlow/mnist/input_data')
    for i in range(2000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:
            train_accuracy = accuracy.eval(feed_dict={
                x: batch[0], y: batch[1]})
            print('step␣%d,␣training␣accuracy␣%g' % (i, train_accuracy))
            summary_str = accuracy_summary.eval(feed_dict={x: batch[0], y:
               batch[1]})
            file_writer.add_summary(summary_str, i)
        train_step.run(feed_dict={x: batch[0], y: batch[1]})

    print('test␣accuracy␣%g' % accuracy.eval(feed_dict={
        x: mnist.test.images, y: mnist.test.labels}))
```

Code Example 1: CNN

3.4. High-Level APIs

The deep CNN model above is written in TensorFlow's low-level APIs. TensorFlow also provides two high-level APIs, Keras and Estimators. Keras has a simple, consistent interface and gives clear and actionable error messages. It offers a variety of modules and stacking them together forms a model. Users can also write custom building blocks to implement new ideas. You do not need to handle graph construction or execution like programming in low-level APIs. Instead you simply need to construct a model by stacking predefined or custom layers and call `compile` and `fit` methods to compile and train the model. Code Example 2 demonstrates this simple process.

```
model = tf.keras.Sequential([
layers.Dense(64, activation='relu', input_shape=(32,)),
layers.Dense(64, activation='relu'),
layers.Dense(10, activation='softmax')])

model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.fit(data, labels, epochs=10, batch_size=32)
```

Code Example 2: Keras Example

Estimators also highly simplifies the programming. Predefined estimators include many models such as `tf.estimator.DNNClassifier`, `tf.estimator.LinearClassifier`, and `tf.estimator.LinearRegressor`. Users only need to specify hyperparameters of these models. Custome estimators can be built by using Keras layers to construct a model and then applying the `tf.keras.estimator.model_to_estimator` method to convert the Keras model into a custom estimator. Code Example 3 illustrates the use of a premade linear classifier estimator. Three predictors `(population, crime_rate, median_education)` are passed to `feature_columns`, and `my_training_set` is a function that returns predictor and target values.

```
estimator = tf.estimator.LinearClassifier(
    feature_columns=[population, crime_rate, median_education])
estimator.train(input_fn=my_training_set, steps=2000)
```

Code Example 3: Estimator Example

The usage of low-level APIs facilitates understanding the internals of TensorFlow and eases debugging, while high-level APIs like Keras and Estimators enable fast experimentation and reduce the delay from idea to result. TensorFlow allows users to choose the one that is most appropriate to the task at hand.

3.5. Models Available in TensorFlow

TensorFlow is highly flexible and users can build a variety of neural network models using the aforementioned basic elements. Moreover, implementations of major neural network models are provided on TensorFlow's official website (https://www.tensorflow.org/). There are examples of supervised learning models for computer vision such as image classification and segmentation and for sequence data such as neural machine translation and audio recognition. Implementations for popular unsupervised learning models such as variational autoencoder and generative adversarial network are also provided. Furthermore, the code for these models is offered in Google Colab (https://colab.research.google.com/notebooks/welcome.ipynb), which is a free Jupyter notebook environment that runs completely in the cloud. Hence, users can run these models or modify them for their own interest in a Web browser. More advanced models such as Transformer (Vaswani et al., 2017) and BERT (Devlin, Chang, Lee, & Toutanova, 2018) can also be found in TensorFlow's GitHub repository (https://github.com/tensorflow/models). Since TensorFlow is by far the most popular deep learning library, many recently published models are implemented in TensorFlow, and the code is released by the authors on GitHub or reimplemented and

released by other researchers or practitioners. Therefore, the code for most published neural network models is available in TensorFlow.

Although it was designed to train neural networks, TensorFlow can also be utilized to train other machine learning models such as linear regression, gradient boosting model (`tf.estimator.BoostedTreesClassifier`), and support vector machine (`tf.contrib.learn.SVM`). While these models are also available in other machine learning libraries like scikit-learn or XGBoost, training these models with TensorFlow enjoys the benefits from tools and features like TensorBoard, GPU support, and data parallelism (which will be introduced in the next section).

3.6. GPU Acceleration and Data Parallelism

Neural network models and some other machine learning models heavily involve matrix multiplication that are simple computations and highly parallelizable. The architecture of GPU is ideal for this type of computation. GPU can be several 100 times (or even more, depending on the particular hardware) faster than CPU on neural network model training. TensorFlow code is optimized to run on GPU by utilizing CUDA and cuDNN (CUDA Deep Neural Network library), a deep neural network library based on CUDA.

Furthermore, models can also be easily trained across multiple GPUs in parallel with TensorFlow, which further accelerates training and allows for larger models by leveraging memory of multiple GPUs. `tf.distribute.Strategy` is a TensorFlow API to distribute training. With minimal code changes, researchers can distribute existing models. Various distribution strategies are available. For instance, `tf.distribute.MirroredStrategy` creates one replica on each GPU, and every variable is mirrored across all the replicas. `tf.distribute.experimental.CentralStorageStrategy` place all variables on the CPU, and operations are replicated across all GPUs. `tf.distribute.experimental.ParameterServerStrategy` designate some machines as workers and some as parameter severs. Computation is replicated across all workers, while each variable is placed on one parameter server.

Code Example 4 illustrates applying the mirrored strategy to a model construed with Keras API. Essentially, users simply need to create an instance of the appropriate `tf.distribute.Strategy` (mirrored strategy in this example) and build and compile the model inside `strategy.scope`. It is similar to distribute training with Estimator API. Distributing model training in low-level APIs is slightly more complicated but a great tutorial on this is given in the official TensorFlow documentation (https://www.tensorflow.org/guide/distribute_strategy).

```
mirrored_strategy = tf.distribute.MirroredStrategy()
with mirrored_strategy.scope():
    model = tf.keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(32,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')])

    model.compile(optimizer=tf.train.AdamOptimizer(0.001),
                loss='categorical_crossentropy',
                metrics=['accuracy'])

model.fit(data, labels, epochs=10, batch_size=32)
```

Code Example 4: Distributed Training Example

3.7. TensorFlow Probability (TFP)

While TensorFlow focuses on neural network models, TFP, a Python library that is built on TensorFlow, enables users to easily combine probabilistic programming and deep learning models. This might be of great interest to many behavioral and educational researchers who have a background or an interest in traditional statistical methods and Bayesian statistics. Code Example 5 illustrates how to sample from probability distributions and fit a generalized linear model.

```
import tensorflow as tf
import tensorflow_probability as tfp

# Pretend to load synthetic data set.
features = tfp.distributions.Normal(loc=0., scale=1.).sample(int(100e3))
labels = tfp.distributions.Bernoulli(logits=1.618 * features).sample()

# Specify model.
model = tfp.glm.Bernoulli()

# Fit model given data.
coeffs, linear_response, is_converged, num_iter = tfp.glm.fit(
    model_matrix=features[:, tf.newaxis],
    response=tf.cast(labels, dtype=tf.float32),
    model=model)
# ==> coeffs is approximately [1.618]
```

Code Example 5: TFP Illustration

TFP includes a wide range of probability distributions and bijectors. Bijectors can be used transform samples from a probability distribution to random outcomes from a different distribution and thus create flexible distributions. The transformation is differentiable and injective (one to one). One application is to

specify flexible, arbitrarily complex and scalable approximate posterior distributions in Bayesian statistics (Papamakarios, Pavlakou, & Murray, 2017; Rezende & Mohamed, 2015). TFP also provides tools to fit traditional generalized and hierarchical linear models and to build and train deep probabilistic models. Users can optimize these models with varational inference, Markov chain Monte Carlo methods, or optimizers such as Nelder–Mead, BFGS (Broyden-Fletcher-Goldfarb-Shanno algorithm) built in TFP.

3.8. Documentation and Resources

It is our hope that this review helps readers to gain a clear understanding of the basics of neural networks and the core elements of TensorFlow. Readers are referred to some further resources for more details. Goodfellow, Bengio, and Courville (2016), who are major contributors in this field, provide a clear and comprehensive introduction to deep learning models and related conceptual and mathematical background in their book, *Deep Learning*. It offers excellent resources for those who would like to systemically study deep learning.

Google provides clear and extensive official documentation and various tutorials for TensorFlow, which can be found at `https://www.TensorFlow.org/`. Géron (2017) authored a practical introduction book on machine learning and deep learning with extensive Python and TensorFlow codes, *Hands-On Machine Learning With Scikit-Learn & TensorFlow*. The Stanford course, *CS 20: TensorFlow for Deep Learning Research* designed by Chip Huyen, whose materials are posted online, also offers a detailed introduction to TensorFlow and how to use TensorFlow to implement popular deep learning models. Due to the popularity of TensorFlow, there are also abundant other tutorials online and discussions on specific TensorFlow topics on various Question and Answer websites. Users can easily obtain help from these resources.

4. Conclusion

TensorFlow is a popular and flexible machine learning library. It has both low-level APIs and high-level APIs (Keras and Estimators) and supports multiple languages interfaces besides Python. Users can easily visualize model and training metrics with TensorBoard, distribute training with `tf.distribute.Strategy`, and combine probabilistic methods with deep neural networks using TFP library. We believe that behavioral and educational researchers can greatly benefit from neural network modeling and adopting TensorFlow in their research.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

References

Collobert, R., & Weston, J. (2008). A unified architecture for natural language processing. In W. Cohen, A. McCallum, & S. Roweis (Eds.), *Proceedings of the 25th international conference on machine learning—ICML '08* (pp. 160–167). New York, NY: ACM.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*, 303–314.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. Retrieved from https://arxiv.org/pdf/1810.04805

Géron, A. (2017). *Hands-on machine learning with scikit-learn and tensorflow: Concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA: MIT Press.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, *4*, 251–257.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, *25*, 1–9.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 2278–2324.

LeNail, A. (2019). Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, *4*, 747.

Min, S., Lee, B., & Yoon, S. (2017). Deep learning in bioinformatics. *Briefings in Bioinformatics*, *18*, 851–869.

Papamakarios, G., Pavlakou, T., & Murray, I. (2017). Masked autoregressive flow for density estimation. *Advances in Neural Information Processing Systems*, *30*, 2338–2347.

Rezende, D. J., & Mohamed, S. (2015). Variational inference with normalizing flows. Retrieved from https://arxiv.org/pdf/1505.05770

Robbins, H., & Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*, *22*, 400–407.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive Modeling*, *5*, 1.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*, 5998–6008.

Authors

BO PANG is a PhD student in the Department of Statistics at UCLA, 520 Portola Plaza, Los Angeles, CA 90095; email: bopang@g.ucla.edu. His research interests are

unsupervised learning and generative models in computer vision and natural language processing.

ERIK NIJKAMP is a PhD student in the Department of Statistics at UCLA, 520 Portola Plaza, Los Angeles, CA 90095; email: enijkamp@ucla.edu. His research interests are stochastic optimization in deep discriminative and generative models.

YING NIAN WU is a professor in the Department of Statistics at UCLA, 520 Portola Plaza, Los Angeles, CA 90095; email: ywu@stat.ucla.edu. His research interests are representational learning, unsupervised learning, generative models, computer vision, computational neuroscience, and bioinformatics.

Manuscript received March 11, 2019
Revision received June 20, 2019
Accepted August 2, 2019