Communications for Statistical Applications and Methods
2019, Vol. 26, No. 6, 591–610

https://doi.org/10.29220/CSAM.2019.26.6.591
Print ISSN 2287-7843 / Online ISSN 2383-4757

Introduction to convolutional neural network using Keras; an understanding from a statistician

Hagyeong Lee[a], Jongwoo Song[1,a]

[a]Department of Statistics, Ewha University, Korea

Abstract

Deep Learning is one of the machine learning methods to find features from a huge data using non-linear transformation. It is now commonly used for supervised learning in many fields. In particular, Convolutional Neural Network (CNN) is the best technique for the image classification since 2012. For users who consider deep learning models for real-world applications, Keras is a popular API for neural networks written in Python and also can be used in R. We try examine the parameter estimation procedures of Deep Neural Network and structures of CNN models from basics to advanced techniques. We also try to figure out some crucial steps in CNN that can improve image classification performance in the CIFAR10 dataset using Keras. We found that several stacks of convolutional layers and batch normalization could improve prediction performance. We also compared image classification performances with other machine learning methods, including K-Nearest Neighbors (K-NN), Random Forest, and XGBoost, in both MNIST and CIFAR10 dataset.

Keywords: deep neural network, convolutional neural network, image classification, machine learning, MNIST, CIFAR10, Keras

1. Introduction

Various machine learning algorithms, as key parts of Artificial Intelligence (AI), have been further developed as data size and computer resources have increased. Statistical models and algorithms perform a specific task effectively depending on patterns and inference. Deep Neural Networks (DNN), decision tree based methods including Random Forest (RF) and Extreme Gradient Boosting (XGBoost), and Support Vector Machines (SVM) are remarkable for real-world regression and classification problems.

Here, we focus on deep neural network also known as ‘Deep Learning’. Deep learning is a combined approach to machine learning that has drawn on the knowledge of the human brain, statistics, and applied math (Goodfellow, *et al.*, 2015). It is not a newly proposed method. Deep Learning represent deeply stacked multiple layers of the Artificial Neural Network (ANN) that connects each node with artificial neurons like a biological brain. It can solve complex non-linear problems and find high-level features from input data by stacking a number of hidden layers between input and output layers versus a single layer network that cannot. This fundamental deep network is called multi-layer perceptron (MLP) and is a mathematical function that maps some set of input values to output values (Goodfellow *et al.*, 2015).

[1] Corresponding author: Department of Statistics, Ewha Womans University, 52, Ewhayeodae-gil, Seodaemun-gu, Seoul 03760, Korea. E-mail: josong@ewha.ac.kr

Published 30 November 2019 / journal homepage: http://csam.or.kr

The deep learning method was not preferred in the past because it didn't work well; however, it is very widely used now due to several breakthroughs that have grown and made deep learning popular. We summarize a brief history of deep learning, starting from the 1940s.

Deep Learning appeared for the first time in the 1940s, and was implemented in its basic form until the early 1980s.

- McCulloch and Pitts (1943) created a simple linear model based on a human brain and introduced the concept of ANN for the first time.
- Rosenblatt (1958) implemented a probabilistic model named Perceptron, training a single neuron to calculate linear combination of inputs and weights. However, the perceptron was not able to learn complex patterns.

From 1980s to 2006, the evolution of deep learning can be summarized as MLP and Back Propagation, ReLU activation and the introduction of Convolutional Neural Network (CNN).

- Rumelhart *et al.* (1986) introduced an innovative training algorithm named back-propagation that enabled multilayer perceptron training.
- LeCun *et al.* (1998) presented *LeNet-5*, the root of modern Convolutional Neural Networks, as well as introduced rectified linear unit ReLU as the activation function. ReLU was faster than other activations such as *sigmoid* and *tanh*, and did not suffer from vanishing gradient problem that as the network went deeper with most gradients becoming zero.

The true concept of 'deep learning' starting around 2006 by Hinton *et al.*, and has continued until today. Various ideas have been designed to solve overfitting, improve learning speed and accuracy. In the area of supervised learning, deep MLP, CNN, and RNN are representative networks. In particular, several CNN models won the title in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) competition in 2012 and achieved prime accuracies with regard to classifying 1,000 classes of images.

With its growing history, deep learning is commonly used for real-world applications such as Image Classification, Natural Language Processing (NLP), and other AI studies of computer vision.

In this paper, we consider deep learning algorithms as parts of supervised learning and try to explain them from a statistical viewpoint. It is not based on inferences, but it is a statistical learning method that predicts a function based on data. Many developers have shown achievements about prediction accuracy and efficient usage of computer resources, but it is hard for practitioners to fully understand how parameters are used mathematically to compute a loss function and why estimation works well. We sum up the topics on the basis of statistical learning theory and associate them to functions of Keras to help users become more familiar to use. The following descriptions and examples are based on statistical regression and classification models that learn features and patterns from input data and estimate corresponding targets.

The remaining of the paper is organized as follows. In Section 2, we first explain the basic structures and learning procedures of DNN, including CNN. We also describe how parameter estimation is done in detail. There are millions of parameters in a deep network and it can be difficult to estimate them appropriately. In Section 3, we explain some advanced techniques to prevent some problems and increase prediction accuracy. In Section 4, we discuss the results of applying advanced CNN models to image classification problems using two typical data sets; MNIST and CIFAR10. Section 5 provides the concluding remarks.

2. Deep learning procedures and basics of CNN

In this section, we describe overall learning procedures of the general deep neural network. First we investigate essential steps while training a model and explain the whole procedures systematically to understand what a model does to predict targets. We also explain how parameters are estimated. At the end of this section, we pay attention to a Convolutional Neural Network and describe the basic concepts.

2.1. Deep learning procedures

A network's learning can be expressed as finding an optimal set of weights (and biases) where each single weight and bias is used for computation between layers. The parameter set of a network is composed of all weights and biases in a model. Here "optimal" means that they have the minimum loss defined by a model and the algorithm tries to update the weights repeatedly.

The entire training procedure can be summarized to repeat following loop for the fixed number of iterations, defined by **epochs** and sample size.

i. Draw a fixed size of mini-batch training sample (*batch size*) and this mini-batch sample enters the first layer of the network.

ii. Along the entire network, transform input data to high-level features by multiplying weights and going through activation functions.

iii. Calculate the mean loss using true and estimated values of output on the batch.

iv. Compute a gradient of the loss to each weight, on the way back from output layer to input layer, and update the weights in the direction of decreasing the loss.

v. Draw a next mini-batch sample and iterate.

Usually, a loss is generally defined as a mean squared error for regression, and a categorical cross entropy for classification. We will explain it later in this section using a matrix form.

2.1.1. Step A. Parameter initialization

We can set initial random values at all parameters (weights or bias) in the network once we define our own network structure. These random values can be drawn from certain probability distributions, including normal and uniform. Setting the initial values should be considered carefully because it might have significant effects on the following learning procedures.

For example, suppose that we set them all zero or a certain identical constant. The initial learning step uses those parameters for computation in all layers from input to output. If they are all zeros, the initial outputs from the network become zeros and good-for-nothing for the following steps. If they are all same constants, they act like just one neuron regardless of the number of nodes and neurons in the network. Either way, learning cannot be improved properly. So we need to choose different values in a reasonable range. The choice of the initial values depends on activation functions in the network. Figure 1 and Table 1 represent three commonly used activation functions; sigmoid, tanh, and ReLU.

The function outputs for *sigmoid* and *tanh* converge in single constant such as 0, 1, or -1, resulting in 'vanishing gradients' if the absolute values of initial weights are too large. Going through ReLU activation, negative values are dead while large positive values result in 'exploding gradients'. If absolute values of initial weights are too small, some simple networks work. However, as the network

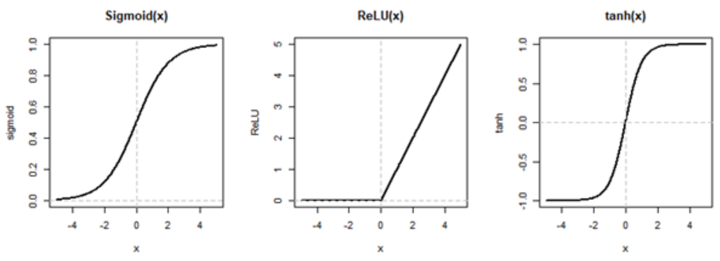

Figure 1: *Three widely used activation functions.*

Table 1: Three commonly used activation functions

Function	Equation
Sigmoid	$f(x) = \dfrac{1}{1 + \exp(-x)}$
ReLU	$f(x) = \max(0, x)$
tanh	$f(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

goes deeper, the weights lies in very small range of values, so that gradients become almost identical and learning becomes difficult.

We need to set a proper range of dynamic initial weights to prevent those potential problems. In general, the uniform or normal distributions with mean zero are the most popular probability distribution for initial values. The critical point is the variance of the distribution and there are some empirical initial distributions of parameters, which depends on variance. Xavier's initialization is now commonly used to generate random values setting the variance on the basis of the number of both input and output nodes in each two adjacent layers. They presented an idea that letting the Jacobian of weight matrix in each layer close to 1 can make training faster and easier (Glorot and Bengio, 2010). Xavier's normalized initialization consider the variance of weights from both forward and backward propagation perspectives; this let the variance of activation values and the variance of the gradients fixed to 1 avoid decreasing. Hence, it sets the variance of weights between i^{th} and $(i + 1)^{th}$ layer equals to $2/(n_i + n_{i+1})$.

He's initialization (He *et al.*, 2015) has been recently developed and is especially efficient and robust for ReLU activation than Xavier's. It differs from Xavier's distribution in terms of dealing with the ReLU non-linearities. The variance of weights in He's is $2/(n_i)$.

In Keras for the real-world applications, the default weight initializers (set by '**kernel initializer**' option) are set as 'Xavier's Uniform' in Dense layer, Convolutional layers, and Recurrent layers. Other choices are available including 'Xavier's Normal', 'He's Uniform', and 'He's Normal' as well as standard random distributions. Users are able to change initializers depending on the activation functions they use.

2.1.2. Step B. Forward pass

Given the initial weights, the batch size of training sample enters the network and the network calculates the outputs following sequential layers from beginning to the end. Each layer does output = $f(W \cdot \text{input} + b)$ where f is the activation function. Here, we illustrate procedures using a classification model that has a MLP structure with two hidden layers between the input and output layer.

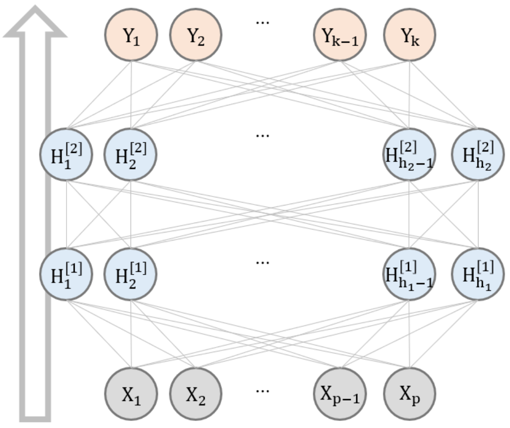

Figure 2: *A diagram of forward pass with two hidden layers.*

These are notations for our quick example in this section.

m : number of input samples used for one iteration (batch size*)
p : number of nodes in the input layer (number of input features)
h_1 : number of nodes in hidden layer [1]
h_2 : number of nodes in hidden layer [2]
k : number of nodes in the output layer (number of classes)

*One might consider using all training data at once for each iteration; however, we employed more common mini-batch training that uses m observations at one iteration. More details about the batch size are in Section 4.

i. INPUT → HIDDEN[1]

$$X_{\text{batch}} \cdot W^{[1]} + b^{[1]} = z^{[1]}, \quad a^{[1]} = \text{activation}(z^{[1]})$$

$$\underbrace{\begin{bmatrix} x_{1,1} & \cdots & x_{1,p} \\ \vdots & & \vdots \\ x_{m,1} & \cdots & x_{m,p} \end{bmatrix}}_{(m,p)} \cdot \underset{\times}{} \underbrace{\begin{bmatrix} w_{1,1} & \cdots & w_{1,h_1} \\ \vdots & & \vdots \\ w_{p,1} & \cdots & w_{p,h_1} \end{bmatrix}}_{(p,h_1)} + \underbrace{\begin{bmatrix} b_1 \\ \vdots \\ b_{h_1} \end{bmatrix}}_{(1,h_1)} = \underbrace{\begin{bmatrix} z_{1,1} & \cdots & z_{1,h_1} \\ \vdots & & \vdots \\ z_{m,1} & \cdots & z_{m,h_1} \end{bmatrix}}_{(m,h_1)} \xrightarrow{f} \underbrace{\begin{bmatrix} a_{1,1} & \cdots & a_{1,h_1} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,h_1} \end{bmatrix}}_{(m,h_1)}.$$

In each layer's linear combination, a bias vector has a length h_i, the number of nodes in layer i. The matrix $X_{\text{batch}} \cdot W^{[i]}$ has a shape of (m, h_1). To add those two terms, the transposition of bias vector is shared to each row of the matrix. Each row in the matrix is corresponded to the h_i computed values of each observation in a mini-batch sample and the bias are element-wisely added to all m rows. This computation is also called as 'Broadcasting', which means expanding one matrix to have a same dimension as the other. Note that broadcasting is feasible only when the length of an axis in two matrices are same.

ii. HIDDEN[1] → HIDDEN[2]

$a^{[1]} \cdot W^{[2]} + b^{[2]} = z^{[2]}, \quad a^{[2]} = \text{activation}(z^{[2]})$

$$\underset{(m,h_1)}{\begin{bmatrix} a_{1,1} & \cdots & a_{1,h_1} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,h_1} \end{bmatrix}^{[1]}} \underset{\times}{\cdot} \underset{(h_1,h_2)}{\begin{bmatrix} w_{1,1} & \cdots & w_{1,h_2} \\ \vdots & & \vdots \\ w_{h_1,1} & \cdots & w_{h_1,h_2} \end{bmatrix}^{[2]}} + \underset{(1,h_2)}{\begin{bmatrix} b_1 \\ \vdots \\ b_{h_2} \end{bmatrix}^{[2]\prime}} = \underset{(m,h_2)}{\begin{bmatrix} z_{1,1} & \cdots & z_{1,h_2} \\ \vdots & & \vdots \\ z_{m,1} & \cdots & z_{m,h_2} \end{bmatrix}^{[2]}} \xrightarrow{f} \underset{(m,h_2)}{\begin{bmatrix} a_{1,1} & \cdots & a_{1,h_2} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,h_2} \end{bmatrix}^{[2]}}.$$

iii. HIDDEN[2] → OUTPUT

$a^{[2]} \cdot W^{[3]} + b^{[3]} = z^{[3]}, \quad \hat{p} = \text{softmax}(z^{[3]})$

$$\underset{(m,h_2)}{\begin{bmatrix} a_{1,1} & \cdots & a_{1,h_2} \\ \vdots & & \vdots \\ a_{m,1} & \cdots & a_{m,h_2} \end{bmatrix}^{[2]}} \underset{\times}{\cdot} \underset{(h_2,k)}{\begin{bmatrix} w_{1,1} & \cdots & w_{1,k} \\ \vdots & & \vdots \\ w_{h_2,1} & \cdots & w_{h_1,k} \end{bmatrix}^{[3]}} + \underset{(1,k)}{\begin{bmatrix} b_1 \\ \vdots \\ b_k \end{bmatrix}^{[3]\prime}} = \underset{(m,h_2)}{\begin{bmatrix} z_{1,1} & \cdots & z_{1,k} \\ \vdots & & \vdots \\ z_{m,1} & \cdots & z_{m,k} \end{bmatrix}^{[3]}}$$

$$\xrightarrow{\text{softmax}} \underset{(m,h_2)}{\begin{bmatrix} c & \frac{\exp(z_{1,1})}{\Sigma_i \exp(z_{1,i})} & \cdots & \frac{\exp(z_{1,k})}{\Sigma_i \exp(z_{1,i})} \\ & \vdots & & \vdots \\ & \frac{\exp(z_{m,1})}{\Sigma_i \exp(z_{m,i})} & \cdots & \frac{\exp(z_{m,k})}{\Sigma_i \exp(z_{m,i})} \end{bmatrix}^{[3]}}$$

iv. LOSS

Calculate a loss for each observation i, and take an average of m losses in the current mini-batch. y_k^i indicates 1 if observation i is in Class$_k$ and 0 if not. Then, each loss is a negative log likelihood given the observed vector y^i and estimated vector of probabilities $\hat{p}^i$.

$$\text{LOSS}^i = -\sum_k y_k^i \cdot \log \hat{p}_k^i, \quad E = \frac{1}{m}\sum_{i=1}^{m} \text{LOSS}^i \quad \text{(Categorical crossentropy).}$$

2.1.3. Step C. Backward pass

Many machine learning algorithms use Gradient Descent (GD) methods to update parameters to gradually reduce the objective function value and reach the optimum in the end. DNN also uses this powerful method. Loss function (E) defined as an objective in the deep network is a differential function for a parameter set of individual weights (W). Therefore, it is possible to compute the differentials of the loss, or gradient vector for weights in each layer. The model can update the weight in the descent direction of the loss after obtaining a gradient for a certain weight making up the loss as follows.

$$w_{ij}^* = w_{ij} - \eta \cdot \frac{\partial E}{\partial w_{ij}}$$

η is a hyperparameter that defines learning rate. It is hard to converge if the learning rate is too large; however, the algorithm has to go through a lot of iterations to converge, which takes a long time if the learning rate is too small.

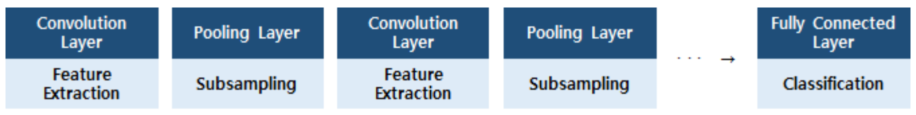

Figure 3: *A typical structure of deep CNN between input and output layer.*

The model has to calculate millions of gradients and update all parameters using GD per one iteration. Back Propagation makes this computation very effective. Right after a forward pass is done, starting from the output layer's loss, the model passes on each gradient loss to the previous layer that is toward input layer. It greatly decreases the learning time.

Depending on the number of observations using at a time, backward passing method can be distinguished as follows.

- **Batch Gradient Descent** uses all observations in the training dataset at once. It gets inefficient for very large training data or deep complex models because it makes the computation slow. However, it is stable and guarantees convergence to a local minimum.

- **Stochastic Gradient Descent (SGD)**, on the contrary, uses just one sample for calculating loss and gradient. It is very fast and carries out a small computation at each iteration, which is efficient when observations in the dataset are homogeneous. For many deep networks, which the loss has many local minima, it gives possibilities to escape local values at random. However, it is unstable and the loss decreases on average while fluctuating. Generally, SGD repeats for the total sample size per epoch. Some observations can be included more than once for an epoch, others might not be included at all since it randomly selects a sample. So we can shuffle the entire training set and select then systematically.

- **Mini-Batch Gradient Descent**, as a combination of the Batch and Stochastic GD, draw m random observations for a small sample for each iteration. It takes advantages of improvements in computation speed from the GPU. It is most commonly used method for efficient and stable learning. Keras usually uses 32 as the default batch size.

2.1.4. Step D. Iteration

After updating all weights in the network, the network inputs 'next' mini-batch sample and repeat Step B–C. In each iteration, forward step draws a fixed size of observations and computes output values for those observations and the following mean loss using the most recently updated weights. Backward step computes gradients to that loss for the weights and propagate them back as well as updates all weights. The entire training procedure for a fixed size of epochs is to adjust weights repeatedly by performing forward and backward propagation while expecting loss decreases.

2.2. Basics of convolutional neural network

2.2.1. Basic concept

CNN, as a specifically defined structure of DNN, uses conceptually identical training and parameter estimation steps. The difference lies in what happens in 'Convolution' and 'Pooling' layers. Each layer of CNNs conducts different tasks.

Suppose an input of CNN is an image with a fixed size.

- **Convolutional Layer** has two key hyperparameters. One is *kernel size*, and the other is the number of *filters*. The layer first separates the input image into some fixed size of local patches, which have the same size in all filters (set by *kernel size*) that the current convolutional layer uses. Each filter of the current convolutional layer has the same depth (number of channels) with current inputs. There are many filters in a convolutional layer that consists of various weight values so that each filter can learn as many different features as possible. The elements in both a filter and a patch are multiplied in pairs and added to a single value. The network then applies an activation function to this value. The network itself extracts important features from the original image. Therefore, the convolutional layer outputs a series of feature maps that enter the next convolutional or pooling layer.

- **Pooling Layer** reduces the size of a feature map. Once the feature maps from the previous convolutional layer enter a pooling layer, the network separates the input feature map into some fixed size of regions (set by *pool size*) and summarizes the pixel values in each region into one maximum value or one average value. It is called max-pooling or average-pooling. These procedures can reduce model complexity and prevent overfitting while allowing some loss of information.

- **Fully-connected Layer** also known as a **Dense Layer** is located at the last part of the network. After all convolution-pooling computations are done, the network arranges the values of final feature maps in a row. Then, all the pixels are fully-connected with all nodes of the next dense layer. The computation of this part is identical to general MLP. The final dense layer outputs the probabilities for classification using a softmax method.

A typical CNN structure for a classification stacks one (or more) convolutional layers and pooling layers alternately which conduct feature extraction and subsampling. The final output features from those stacked blocks are flattened and fully-connected (FC) with all nodes of the output layer and conduct classification. One can consider adding more than one FC hidden layers right before the final output layer in order to improve classification performance. Figure 3 simplifies this structure. This typical structure is based on *LeNet-5* (LeCun *et al.*, 1998) and used for recognizing MNIST handwriting ten digits, 0–9.

LeCun *et al.* (1998) claimed that "Convolutional networks combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance: *local receptive fields*, *shared weights* (or weight replication), and spatial or temporal *sub-sampling*."

The goal of training CNN is to extract important features from input images that decide a unique class of each image by learning all weights in the filters. If we use a large filter, the model tries to find features in a large area of the input at each computation, whereas setting the filter size small makes the network view a small area. This is one of the tuning parameters that enables setting the different filter size depending on what we expect from the model.

CNN has a great advantage distinct from MLP. It can drastically reduce the number of parameters to estimate by sharing filters at different local regions of the input. A filter visits all parts of an image and performs identical computation. Moreover, it regards several features (pixels) as one feature of a local region, rather than considering each feature different.

2.2.2. An example of basic convolutional neural network structure

In this section, we illustrate an example of 2D CNN for image classification using the Keras context to see the structure of CNN. Suppose each input is a gray-scaled image with the height and width of 28 pixels each. Each pixel has one single channel and the input has a shape (28, 28, 1) that is the input shape of MNIST images. The number of channels are 3 if we consider color images with RGB values. Let the number of unique classes 10. We consider a CNN structure with two convolution-pooling layers repeated.

Table 2: Example: Hyperparameters setting in each conv2D layer

Convolutional layer	Number of filters	Kernel size
1	32	(3, 3)
2	64	(3, 3)

Table 3: Example: Summary of 2D CNN model structure

Layer(options)	Output shape	Number of parameters
Input	(None, 28, 28, 1)	-
Conv2D(filters=32, kernel_size=(3, 3), padding='valid', strides=1, activation='relu')	(None, 26, 26, 32)	320
MaxPooling2D(pool_size=(2, 2), padding='valid', strides=2)	(None, 13, 13, 32)	0
Conv2D(filters=64, kernel_size=(3, 3), padding='valid', strides=1, activation='relu')	(None, 11, 11, 64)	18496
MaxPooling2D(pool_size=(2, 2), padding='valid', strides=2)	(None, 5, 5, 64)	0
Flatten	(None, 1600)	0
Dense(units=10, activation='softmax')	(None, 10)	16010

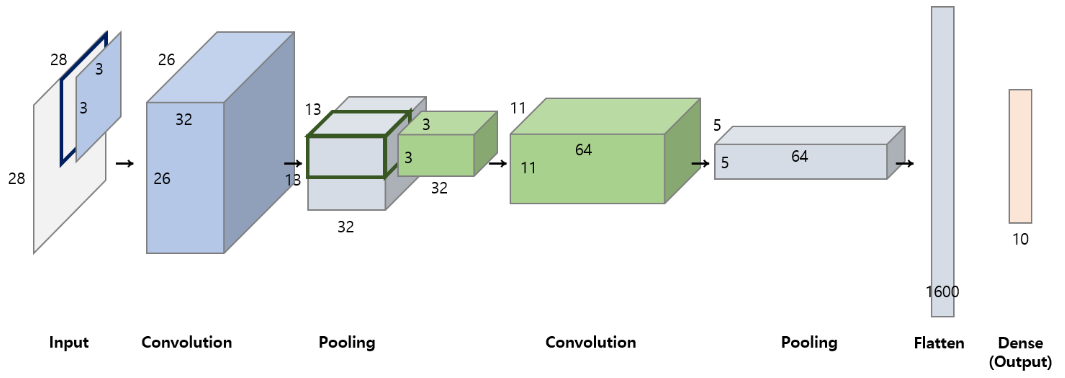

Figure 4: *Example: An illustration of a computation from an input image to output.*

With some hyperparameters defined in Table 2, Table 3 and Figure 4 summarize the model structure of our example. We suppose that each activation function of convolutional layer is *ReLU* and the final activation function is *softmax*. The first dimension of each output shape in Table 3 are defined by batch size. It can be any value from one to the number of all observations in the training sample and does not influence defining the model.

We will explain several options that make the structure diverse in detail.

(Zero) '**Padding**' means to fill four edges of each layer's inputs by zero to keep the size of inputs and outputs same. To apply this technique in a convolution or pooling layer, we can set *padding = 'same'* in the context of the layer function. Otherwise, we set *padding = 'valid'* not to pad and the size of feature maps will gradually decrease as they go through convolutional layers. It is default setting option in Keras. If the size of inputs and filters in a convolutional layer are (n, n) and (l, l) respectively, then, the resulting single feature map have a size $(n - l + 1, n - l + 1)$.

'**Stride**' is the number of pixels that a filter (or kernel) moves by at once inside an input. If it is one, the filter moves right one pixel at a time. In the default setting, we usually set strides equal to one for convolutional layers, and same value as pool size for pooling layers. If the value of stride and pooling kernel size are the same, it prevents each kernel from being overlapped.

The **number of weights** in each convolution (2D) layer is equal to (number of input channels) $*$

Table 4: Example; Hyperparameters setting in each conv1D layer

Convolutional layer	Number of filters	Kernel size
1	32	3
2	64	3

Table 5: Example; Summary of 1D CNN model structure

Layer(options)	Output shape	Number of parameters
Input	(None, 200, 1)	-
Conv1D(filters=32, kernel_size=3, padding='valid', strides=1, activation='relu')	(None, 198, 32)	128
MaxPooling1D(pool_size=2, padding='valid', strides=2)	(None, 99, 32)	0
Conv1D(filters=64, kernel_size=3, padding='valid', strides=1, activation='relu')	(None, 97, 64)	6208
MaxPooling1D(pool_size=2, padding='valid', strides=2)	(None, 48, 64)	0
Flatten	(None, 3072)	0
Dense(units=1, activation='sigmoid')	(None, 1)	3073

(filter width) $*$ (filter height) $*$ (number of output channels), and bias exist as much as the number of output channels. Total parameters are the sum of all weights and bias in each conv2D layer. For example, above the first conv2D layer has 320 parameters that consist of $1 * 3 * 3 * 32 = 288$ weights and 32 bias. The number of weights in a dense layer is equal to (number of input channels) $*$ (number of output channels). The last dense layer of our example has 16,010 parameters that consists of $1600 * 10 = 16000$ weights and 10 bias.

The illustration of computation in Figure 4 represents the matrix transformation from an input image array with single channel to high-level feature maps with 64 channels. Table 3 shows that the number of parameters in conv2D and dense layers varies from layer to layer. However, the number of parameters in pooling and flatten layers are zero. It is because pooling layers and flatten layer does not contain any parameters to estimate. These layers just reduce and summarize features, or rearrange them in a row.

CNNs are designed to deal with 2D images; however, they can also accept 1D shaped input data such as a time-series, a text that includes a series of words, and other sequential data. The algorithm is identical to 2D CNN except that the filters also have 1D shapes. The filters in 1D CNN slide an input sequence in one direction toward right, and compute an activated value for each local subsequence.

Table 4 and Table 5 summarize an example of 1D CNN for a binary classification with two classes. It can be used for a case of text sentiment classification where 0 and 1 indicates negative and positive respectively. You can see that the only difference between 2D CNN and 1D CNN are the shapes of inputs and filters. Note that the objective loss function for this example is binary cross-entropy, and the last activation function is set to *sigmoid*.

3. Advanced techniques for training DNN

In a large deep neural network for a supervised learning, the model contains a huge number of parameters and is often very complex. The performance sometimes becomes poor on a validation or a hold-out test set if the model is too complex. This 'overfitting' problem frequently occurs in many machine learning models. Figure 5 shows that the loss of the training dataset decreases continuously while the loss of the validation set decreases at early steps and starts to increase after a certain point.

The parameter estimation procedure relies on initial weights; consequently, the hyperparameters should be tuned by using some grid search or random search such as Bayesian optimization. When

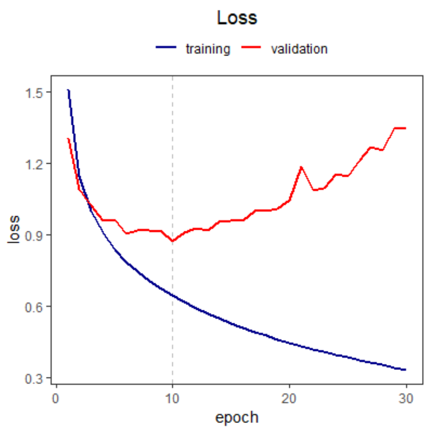

Figure 5: *Example: Evaluated losses by epoch on both a training and validation set in a CNN model.*

training a large network, the learning procedure can easily go the wrong way.

Recently, some advanced techniques have been developed for training a large neural network to overcome difficulties and improve prediction performance. We can use those techniques by adding an extra step between some training steps in the layers. In this section, we describe two main techniques that greatly contribute to the development of deep learning; Dropout and Batch Normalization.

3.1. Dropout

Dropout was presented in 2012, by Hinton *et al.* research group. The goal of dropout method is to prevent a overfitting problem on the small training dataset when training a deep multilayer network with many hidden layers of non-linear unit nodes. Srivastava *et al.* (2014) claimed that "Dropout prevents overfitting and provides a method of approximately combining exponentially many different neural network architectures effectively."

A dropout procedure randomly omits (or sets to 0) some nodes in a certain hidden layer at each update on each training epoch. The choice of which node to drop out is randomly decided by the probability p (set by *rate*), which is a hyperparameter that can be tuned using cross-validation. At each iteration for updating parameters, removed nodes are not included in any computation on the current step, but can be used for the next step. These dropped nodes do not contribute to the forward pass and are not used in the back propagation (Krizhevsky *et al.*, 2012).

Suggesting this designed technique, Hinton research group reported some experimental results on seven datasets that included image data sets such as MNIST, CIFAR10, and ImageNet. They found that dropout improved performances on all data sets compared to neural networks without dropout (Srivastava *et al.*, 2014). They also applied dropout on AlexNet (Krizhevsky *et al.*, 2012) and it outperformed previous competitive models in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in 2012. ImageNet is a large image database designed for a visual detection of objects, which contains more than 14 million high-level real world images with more than 20,000 categories. The competition has been held annually since 2012, where many software development groups compete for image classification or detection in 1,000 subcategories.

It can be generally used in a deep neural network, for various applications including image classification, speech recognition, and document classification. In the following section 4, we apply the dropout technique for a case study of 2D CNN on MNIST dataset to see how well it improves the classification performance.

3.2. Batch normalization

Many applications of deep learning in various domains use generalized stochastic gradient descent with a batch size of sample at a time, to reduce the computation complexity and obtain an effective learning. However, training those deep networks is usually difficult because the loss tends to fluctuate for every update. Ioffe and Szegedy (2015) claimed that "While stochastic gradient is simple and effective, it requires careful tuning of the model hyperparameters, specifically the learning rate used in optimization, as well as the initial values for the model parameters". They developed the Batch Normalization in 2015 and achieved more than 95% of the test accuracy in ImageNet classification. It is widely used now in many advanced CNN architectures, and Keras provides some applications of those architectures such as ResNet50, Inception V3, and Xception (Chollet, 2017).

As we described in the matrix form in Section 2, all the inputs of each layer except for the first input layer are the activated values (notated as $a_{i,j}$) that passed from the previous layer. The input distributions of hidden layers change as the network update the weights during training with the current mini-batch because the activation values change differently in every update. Batch normalization simply standardizes the mini-batch of linearly computed values (notated as $z_{i,j}$ in the matrices) in each layer to mean 0 and variance 1 before activation. It then adjusts them by a new scale parameter γ and location parameter β different from layer to layer and from channel to channel. It applies optimal fixed γ and β instead of the original mean and scale of each mini-batch. This technique stabilizes the training procedure and reduced the risk that the weights are updated in a wrong direction.

$$z_{BN} = \gamma \frac{z - \mu_B}{\sigma_B} + \beta, \quad a_{BN} = \text{activation}(z_{BN})$$

In Keras, we can separate an activation layer from a convolutional layer or a fully-connected layer and put a batch normalization layer between them.

4. Convolutional neural network case study: MNIST and CIFAR10 dataset

This section aggregates main concepts and techniques in CNN models and apply them to two widely used datasets for image classification study. We first use a standard MNIST hand-written digits digits (LeCun *et al.*) data to build and train a CNN model using one of Keras Examples in the R interface to 'Keras' (Falbel *et al.*). The entire dataset has 60,000 training images and 10,000 test images. We next use CIFAR10 tiny images for more advanced experiments. We build several models by gradually increasing the complexity in order to capture some crucial steps in CNN that can improve the performance on test data. We also compare performances with other machine learning methods such as K-Nearest Neighbors (KNN), Random Forest, and Extreme Gradient Boosting (XGBoost) in both data to ensure that CNNs perform better than others in the image classification problem.

4.1. MNIST

4.1.1. Build and train a convolutional neural network model without tuning

Keras interface provides some examples for training neural networks that include MLP, CNN, and RNN on real datasets that demonstrate codes in R. Among them, we made a convolutional network following a simple CNN implementation example on the MNIST dataset. Table 6 shows that the model has two conv2D layers and one max-pooling layer successively. It then contains an additional hidden dense layer right after the values are flattened. Both before the flatten layer and after the hidden

Table 6: MNIST; Summary of the CNN model structure

Layer(options)	Output shape	Number of parameters
Input	(None, 28, 28, 1)	
Conv2D(filters=32, kernel_size=(3, 3), activation='relu')	(None, 26, 26, 32)	320
Conv2D(filters=64, kernel_size=(3, 3), activation='relu')	(None, 24, 24, 64)	18496
MaxPooling2D(pool_size=c(2, 2))	(None, 12, 12, 64)	0
Dropout(rate=0.25)	(None, 12, 12, 64)	0
Flatten	(None, 9216)	0
Dense(units=128, activation='relu')	(None, 128)	1179776
Dropout(rate=0.5)	(None, 128)	0
Dense(units=10, activation='softmax')	(None, 10)	1290

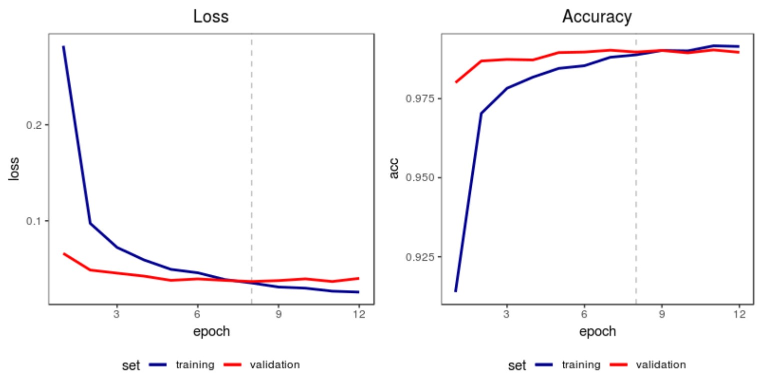

Figure 6: *MNIST; Training history of the model in Table 6.*

layer, the model also has drop-out steps to avoid potential overfitting. All images have an identical shape of (28, 28, 1). The first two dimensions is the size of the images, and the last dimension is one because they are gray-scaled. The digits are located in the middle of the images. The shape of outputs for each convolutional layer become gradually smaller and deeper because we did not pad the images. We trained the model for 12 epochs, using batch size 128 as it is in the example, and saved the last model after the final epoch is ended. We randomly splitted 10,000 validation set from 60,000 training set by using an option '*validation_split=0.2*' in the fitting context in order to compare loss and accuracy at each end of epoch. There is one more way to carry out validation during training, by using '*validation_data*' option instead of '*validation_split*'. Before using this option, the user splits the training and validation data set and fixes the validation data as a list. Then, the model uses the validation set to evaluate the loss at the end of each epoch while updating parameters using only training set. We did not tune any hyperparameters including the number of filters, kernel size, or batch size in this case for a simple implementation. Figure 6 shows evaluated loss and accuracy by epoch. Training accuracy ended with 99.15%, while validation accuracy ended with 98.97%. The model performs very well in the validation data as well despite being slightly lower than the training accuracy.

We trained another model that has an identical structure just without all dropout layers. We also splitted 20% of the original training set at random in the fitting context. Table 7 summarizes evaluated statistics for each data set. The model with dropout layers shows similar accuracies on the training and test set. However, the training accuracy of the model without dropout layers is close to one while the test accuracy is 98.88%. We could expect that those dropout layers helped the network not to overfit on the training set.

Table 7: MNIST; Evalutation for each set in two CNN models

Set	Model with dropout		Model without dropout	
	Loss	Accuracy	Loss	Accuracy
Training	0.0258	0.9915	0.0045	0.9986
Validation	0.0401	0.9897	0.0560	0.9876
Test	0.0302	0.9920	0.0465	0.9888

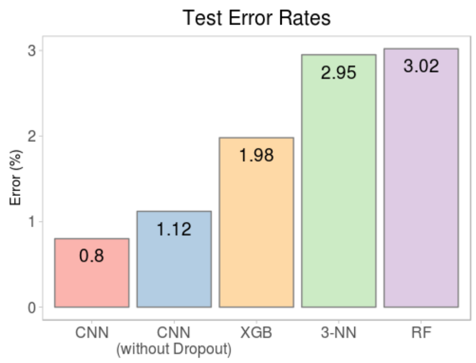

Figure 7: *MNIST; Test error performances of five different machine learning models.*

4.1.2. Compare the performance with other machine learning methods

We also applied several machine learning models on the same data set in order to verify high performances of the trained CNNs above; however, we have to use a different data structure from CNNs when fitting these models. Each pixel value becomes an independent feature of an image. Therefore, we used 786 predictor variables including 784 different pixel values and two summary statistics - mean and standard deviation of all pixels. We tuned all three models using cross-validation. We adjusted some parameters in the Random Forest that included the number of randomly sampled as candidates at each split and selected the best values by comparing OOB error rates. In XGBoost and K-NN, we used 2-fold CV.

Figure 7 illustrates error rates of five different models on the same test images. Even the CNN without drop-out procedures performs better than other three models.

4.2. CIFAR10

CIFAR10 (Krizhevsky *et al.*) contains 50,000 color images for training and 10,000 for test in ten different classes. There are same number of images of each class. The images are tiny and low-leveled, as they have 32 pixels in height and width each. All images were labeled in person. Figure 8 shows a name and sample image of each class.

4.2.1. Build and train advanced convolutional neural network models with tuning

In this case, we started from a basic structure that has three conv2D-pooling blocks and tried to find the optimal structure that can perform well with unknown test images. Each block has one or more convolutional layers and one max-pooling layer successively. We randomly divided 40,000 training images and 10,000 validation images from the original training images. Then monitoring the validation loss and accuracy, we adjusted the structure of the network in various ways; adding more

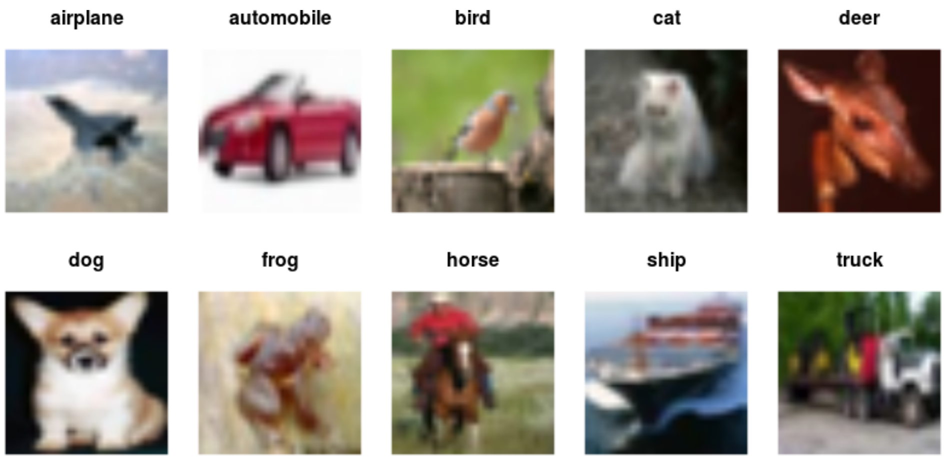

Figure 8: *CIFAR10; Sampled images of ten different classes.*

Table 8: CIFAR10; Overall structure of all CNN models

Layer	Output shape	Kernel size
Input	(None, 32, 32, 3)	-
Convolution Block 1	(None, 16, 16, 16)	(3, 3)
Convolution Block 2	(None, 8, 8, 32)	(3, 3)
Convolution Block 3	(None, 4, 4, 64)	(3, 3)
Flatten	(None, 1024)	-
(Dense)	(None, 128)	-
Output	(None, 10)	-

convolutional layers in each block, adding fully-connected hidden layers before the output layer, or applying batch normalization technique on each convolutional layer and dropout after a hidden dense layer to prevent overfitting. We fixed all dropout rates to 0.1.

We fixed other hyperparameters as described in Table 8. In each convolutional layer, we used zero-padding to maintain output shapes regardless of the number of convolutional layers in a block, and used ReLU activation. We also used 'He Uniform' to draw initial weights in each convolution and dense layer. It can be adjusted in Keras layers using '*kernel_initializer*'. We set 32 for batch size. The network updates weights by repeating a number of mini-batch for each epoch. The network carries out 40000/32 = 1250 mini-batches per epoch.

There are several useful callbacks during training to save the best model which has the optimal monitoring statistics and stop training at certain early epoch.

- ***model_checkpoint*** *(filepath, monitor='val_loss', save_best_only=TRUE, period=1)*

- ***early_stopping*** *(monitor='val_loss', patience=20)*

Above example functions specify a filepath and monitor the validation loss at each end of epoch. If current validation loss decreased from previous epoch, current model is saved to the filepath. Otherwise, the next epoch begins. The last saved point would be 10th epoch if the validation loss is minimum at 10th epoch and does not decrease for the next 20 epochs. Therefore, the entire learning is stopped at 30th epoch by the early stopping criteria. In our experiment, we trained eight differ-

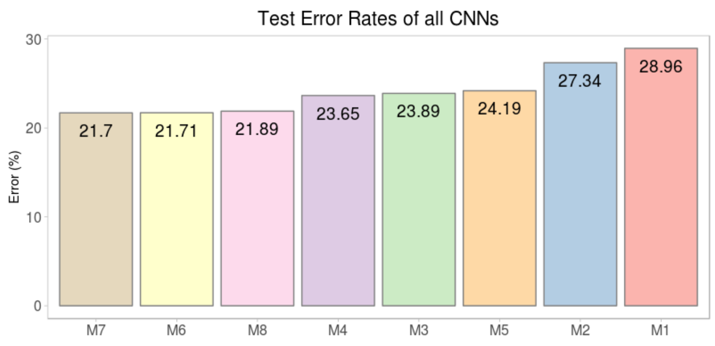

Figure 9: *CIFAR10; Test error rates of all eight CNN models.*

Table 9: CIFAR10; Summary of model structures and test errors of all CNN models

Model	Number of Conv. Layers	BN	Number of FC Layers	Error
M1	1	NO	1	0.2896
M2	2	NO	1	0.2734
M3	2	YES	1	0.2389
M4	2	YES	2	0.2365
M5	2	YES	3	0.2419
M6	3	YES	1	0.2171
M7	3	YES	2	0.2170
M8	3	YES	3	0.2189

ent models changing structures and saved each model by monitoring the validation accuracy for 20 epochs as patience (Table 9).

We usually set 100 epochs for total and 20 epochs for patience, and used 300 epochs and 50 epochs each for deeper networks that contained fully-connected hidden layers.

Figure 9 and Table 9 shows the results of error rates on the 10,000 test images. We using the validation images to select the best model in each structure and compared the performances on the test images. In summary, test error decreases as the number of convolutional layers per block increases. The batch normalization techniques in convolutional layers are also helpful. However, additional deep hidden dense layers do not improve the test performance consistently.

Model M6 and M7 both seem to be the best model. The only difference in two models is if the model contains an additional hidden layer (and following dropout layer). Because test performances are very similar, we selected the simpler model M6 for our final model. It has three repeated blocks of three convolutional layer with one max-pooling layer and each convolutional layer conducts batch normalization before activates the values.

4.2.2. Compare the performance with other machine learning methods

We also compared our final CNN model with other machine learning methods, Random Forest, XG-Boost, K-NN. For those methods, we used $32 * 32 = 1024$ pixel values and their mean, standard deviation for each color channel. Therefore, an image has $(1024 + 2) * 3 = 3078$ predictor variables that make a significantly large dimension of features. It also involved a lengthy training time. It took about half a day to train the XGBoost model with tuning, while the CNN models took less than half an hour in GPU on average (For the most complex CNN, we needed about two minutes per epoch in CPU and less than 30 seconds in GPU). In each machine learning method, we tuned the models using

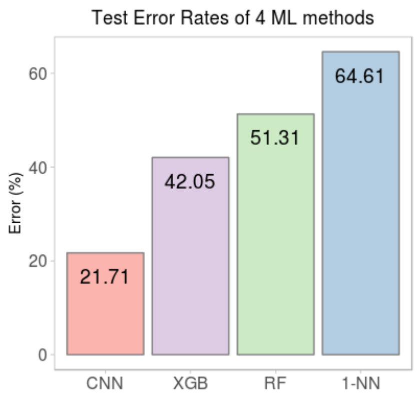

Figure 10: *CIFAR10; Test error performances of four different machine learning models.*

the same validation set as our CNNs.

Figure 10 compares the performances on the same 10,000 test images. Unlike MNIST, the gap between the CNN and other methods is large. Color images have very complex data structures, and thus CNN gets more advantages of reducing the number of parameters and capturing local regions at once.

5. Conclusion

Increases in computer resources and data size have resulted in more people paying attention to deep learning. Keras helps those people quickly get used to deep learning methods by using some core functions. Deep neural network is useful for supervised learning with unconventional data like images and texts. We organized the concepts in deep learning for statisticians in terms of parameter estimation procedures and advanced techniques that make a model performs better. We describe the computation of a deep network in the matrix form to understand its procedure more clearly. We also focused on the convolutional neural network which has top honor in dealing with image data and found that it performs better than other machine learning methods that cannot learn local features.

We distinguished parameters to be estimated and some main hyperparameters to be predefined before fitting a model in Table 10, particularly in the convolutional, pooling, and dense layer. Among them, the number of units (or filter) and kernel size decide the number of parameters and their shapes in the layer, while others such as dropout rate, initial values, optimizers, epochs, batch size, and patience of early stopping affect the progress of training.

Users can vary their models from three main perspectives.

- Hyperparameters in each layer that contains parameters to estimate
- Initializers (distributions) for sampling random values of parameters before begin training
- Optimizers (algorithms) for updating parameters and their options such as a learning rate

In our experiments in CIFAR10, we varied the number of layers concentrating on the depth of CNN models. VGGNet (Simonyan and Zisserman, 2014) also emphasizes the depth of CNN as an important designed aspect with other parameters fixed. Zhang and Wallace (2015) varied the number and size of filters, fixing the number of convolutional layers to 1 and applied models on seven datasets

Table 10: Parameters and mainly used Hyperparameters with their default values in Keras

Parameters

Layer	Variable	Default value in Keras
Dense, Convolutional	weights	`kernel_initializer='glorot_uniform'`
	bias	`bias_initializer='zeros'`

Hyperparameters

Layer or other	Variable with default value in Keras	Options
Dense	`n_dense`	number of fully-connected layers to stack
	`units`	number of hidden nodes
	`activation`	activation function of dense layer
		if not defined, output is just linear combination $X \cdot W + b$.
Convolutional	`n_conv`	number of convolutional layers to stack
	`filters`	number of output filters
	`kernel_size`	length of the window of each filter (1D)
		width and height of each filter (2D)
	`activation='relu'`	activation function of convolutional layer
	`padding='valid'`	valid' (no padding)
		same' (padding to make the output's shape same as the input's)
	`strides=1`	steps of the convolutional filter moves at once
Pooling	`pool_size=2 or (2,2)`	size of pooling,
		an integer (1D) or a list of two integers (2D)
	`padding='valid'`	
	`strides`	steps of pooling. if not defined, set to `pool_size`.
Dropout	`rate`	the sampling rate of the input units to drop
Global	`epochs`	number of entire training epoch
	`batch_size=32`	number of observations using at once per update
	`patience`	number of epochs with no improvement after which training will be stopped
Optimizers (ex. RMSprop)	`lr=0.001`	learning rate of an optimizer
	`decay=0`	amount of learning rate decline over each update

for Sentence Classification. They found the number and size of filters could have important effects and should be tuned. Users can construct their CNN models with some experiments on the depth of layers, number of filters and their size.

We used `'he_uniform'` for the weights initializer based on the empirical result that it is robust for ReLU. We used RMSProp (Hinton *et al.*, 2014) optimizer with learning rate 1e-3, decay 1e-6. Keras provides various optimization algorithms that have been developed recently as well as the naive SGD. Sebastian Ruder (2017) reviewed the existing commonly used optimization algorithms and their architectures.

We found that CNN learns high-level features as the network has more stacked convolutional layers. Therefore, we might stack more convolutional layers to improve the prediction accuracy. We also found that a network with more stacked convolutional layers learns high-level features from input images and estimates output values on CIFAR10 images more accurately. To improve the model performance, we might stack more convolutional layers deeper rather than add fully-connected layers at the end of the network. Our objective is to understand the network more clearly rather than obtain the best accuracy; however, there are lots of large convolutional networks that have achieved outstanding accuracy in image classification on MNIST and CIFAR10. For example, *LeNet-5*, as the beginning of CNN, achieved 0.8% test error rate on MNIST and Cireşan *et al.* (2012) reached 0.23%. On CIFAR10, *AlexNet* as the winner of ILSVRC in 2012 achieved then 11% error rate. In Huang *et al.* (2018) took the No.1 performance with 1% error rate.

We also verified the advantages of two advanced techniques. In detail, dropout helped the model estimate parameters not excessively focusing on the training set and batch normalization helped the model achieve better prediction accuracy. All the R codes of our experiments are written using Keras and accessible on the web (http://home.ewha.ac.kr/~josong/CNNCSAM.html). It covers defining, training, and evaluating the models, as well as transforming data shapes to fit each model we used in this paper. It would be helpful for someone who wants to fit a CNN for their own study.

References

Cireşan D, Meier U, and Schmidhuber J (2012). Multi-column deep neural networks for image classification, arXiv:1202.2745.

Chollet F (2017). *Deep Learning with Python*, Manning, New York.

Falbel D, Allaire JJ, and Chollet F. R interface to 'Keras'. https://keras.rstudio.com/index.html

Glorot X and Bengio Y (2010). Understanding the difficulty of training deep feedforward neural networks, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, **9**, 249–256.

Goodfellow I, Bengio Y, and Courville A (2015). *Deep Learning*, MIT Press, Cambridge.

He K, Zhang X, Ren S, and Sun J (2015). Delving deep into rectifiers: surpassing human-level performance on ImageNet classification, arXiv:1502.01852.

Hinton GE, Srivastava N, Krizhevsky A, Sutskever I, and Salakhutdinov RR (2012). Improving neural networks by preventing co-adaptation of feature detectors, arXiv:1207.0580.

Hinton G, Srivastava N, and Swersky K (2014). RMSprop: Divide the gradient by a running average of its recent magnitude. Available from: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Huang Y, Cheng Y, Bapna A, *et al.* (2018). GPipe: efficient training of giant neural networks using pipeline parallelism, arXiv:1811.06965.

Ioffe S and Szegedy C (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, **37**, 448–456, arXiv:1502.03167.

Krizhevsky A, Ilya Sutskever, Geoffrey Hinton. (2012). ImageNet classification with deep convolutional neural networks, *Advances in Neural Information Processing Systems*, **25**.

Krizhevsky A, Nair V, Hinton G. CIFAR10 and CIFAR100 datasets. Available from: https://www.cs.toronto.edu/ kriz/cifar.html

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

LeCun Y, Cortes C, and Burges CJC. MNIST handwritten digit database. Available from: http://yann.lecun.com/exdb/mnist/

McCulloch WS and Pitts WH (1943). A logical calculus of the ideas immanent in nervous activity, *The Bulletin of Mathematical Biophysics*, **5**, 115–133.

Rosenblatt F (1958). The perceptron: a probabilistic model for information storage and organization in the brain, *Psychological Review*, **65**, 386–408.

Ruder S (2017). An overview of gradient descent optimization algorithms, arXiv:1609.04747.

Rumelhart D, Hinton G, and Williams RJ (1986). Learning representations by back-propagating errors, *Nature*, **323**, 533–536.

Simonyan K and Zisserman A (2014). Very deep convolutional networks for large-scale image recognition, arXiv:1409.1556.

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, and Salakhutdinov R (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting, *Journal of Machine Learning Research*, **15**, 1929–1958.

Zhang Y and Wallace B (2015). A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. arXiv:1510.03820.

Received July 25, 2019; Revised September 11, 2019; Accepted October 21, 2019