Paper 3215-2019

Deep Learning with SAS® and Python: A Comparative Study

Linh Le, Institute of Analytics and Data Science; Ying Xie, Department of Information Technology;
Kennesaw State University

ABSTRACT

Deep learning has evolved into one of the most powerful techniques for analytics on both structured and unstructured data. As a well-adopted analytics system, SAS® has also integrated deep learning functionalities into its product family, such as SAS® Viya®, SAS® Cloud Analytic Services, and SAS® Visual Data Mining and Machine Learning. In this paper, we conduct an in-depth comparison between SAS and Python on their deep learning modeling with different types of data, including structured, images, text, and sequential data. We focus on using such deep learning frameworks in SAS environment, and highlight the main differences between SAS and Python on programming styles on deep learning along with each tool's advantages and disadvantages.

INTRODUCTION

In recent years, deep learning has evolved into one of the most powerful techniques for analytics on both structured and unstructured data. In general, a deep learning model utilizes a high number of parameters structured by layers of neural networks to map the data to a feature space on which a decision-making model is applied. This architecture of stacking parameters by layers allows a deep network to transform data into high-level and non-linear representations that boosts the quality of the decision-making process. Moreover, both the network's parameters and its decision-making model are trained with a learning objective that is closely tied to the data and the given task which overall enhances the capabilities of the network in solving complex problems. Finally, various types of networks are designed for different types of data, for example, deep feed-forward network (DNN) [1] for tabular data, convolutional neural networks (CNN) [2] for image data, recurrent neural network (RNN) [3] for sequential data, etc. All these facts make deep learning a powerful tool in analytics and artificial intelligence.

As a well-adopted analytical system, SAS® has also integrated deep learning functionalities into its product family, such as SAS® Viya®, SAS® Cloud Analytic Services (CAS), and SAS® Visual Data Mining and Machine Learning. The CAS environment has been developed to be utilized in different programming languages like Lua, Python, and R. In a pure SAS environment, deep learning can be done through the CAS language (CASL) that is available through the CASUTIL and CAS procedure in SAS Viya. In this paper, we focus on using CASL in SAS Viya for deep learning to showcase how a modeling task with deep learning can be done purely in SAS. We then conduct an in-depth comparison between SAS and Python on their deep learning modeling. Our comparison highlights the main differences between SAS and Python on programming styles. While many Python packages are available for deep learning, we mainly focus on TensorFlow [4] for high-level API, and Theano [5] for low-level API. Since model performances largely depends on the training algorithms rather than the platform or packages, we are not focusing on this criterion in this paper.

In the following sections, we first briefly introduce CASL, then we discuss different types of deep learning models for a specific type of data, namely DNN for tabular data, CNN for image data, and RNN for sequential data. We also provide example of how each network can be built and trained in SAS as well as Python (detailed examples will be focusing on SAS/CASL) and compare and their differences.

BASIC CAS LANGUAGE

The CAS Language is available through SAS Viya under the CAS and CASUTIL procedure. In a SAS Studio environment that is connected to SAS Viya, the users can initiate a CAS session simply with running:

```
cas;
```

The user can also give the CAS session a name, for example:

```
cas casauto;
```

begins a CAS session named "*casauto*". A successfully initiated session gives a message that is showed in Figure 1.

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
72
73         cas casauto;
NOTE: The session CASAUTO connected successfully to Cloud Analytic Services              .kennesaw.edu using port
      is                                  . The user is       and the active caslib is CASUSERHDFS(     ).
NOTE: The SAS option SESSREF was updated with the value CASAUTO.
NOTE: The SAS macro _SESSREF_ was updated with the value CASAUTO.
NOTE: The session is using 6 workers.
74
75         OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
88
```

Figure 1. Logs from a Successfully Initialized CAS Session

A simple way to load data into a CAS session is to create a CAS library (*caslib*) reference and use it to store data with a data step like a normal lib name. For example:

```
caslib _all_ assign;
libname inmem cas caslib=casuser;
data inmem.train;
    set dnntest.train;
data inmem.test;
    set dnntest.test;
```

In details, the piece of code above first makes all cas libraries available to the current SAS session. It then creates a library reference named *inmem* that links to the caslib *casuser*. Then, two data steps are used to save the *train* and *test* datasets from the *dnntest* library to the caslib *casuser*. Both library references and the datasets they contain can be seen in SAS Studio, as showed in Figure 2.

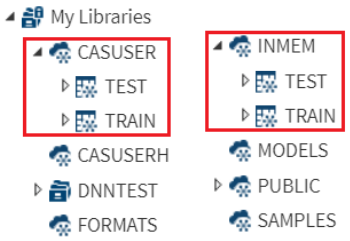

Figure 2. Datasets in a CAS Library and its Library Reference

Further steps of conducting analyses are done through "actions" in the CAS procedure. In short, an action is similar to a statement in other SAS procedures. CAS actions are divided into action sets of based on their use cases. The general syntax is to call the CAS procedure, then call the actions along with defining their required parameters. The following snippet of codes:

```
proc cas;
    session casauto;
    table.tableInfo /
          caslib = "casuser";
          name = "train";
    run;
quit;
```

invokes the $tableInfo$ action from the action set $table$ in the CAS procedure. The parameters of the action in this case set the target caslib location to $casuser$ and the target dataset to $train$. The result of this snippet can be seen in Figure 3.

Results from table.tableInfo

Table Information for Caslib CASUSER

Table Name	Number of Rows	Number of Columns	Number of Indexed Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Compressed
TRAIN	5773	8	0	utf-8	20Mar2019:20:54:59	20Mar2019:20:54:59	No	No	No	No
TEST	503	8	0	utf-8	20Mar2019:20:54:59	20Mar2019:20:54:59	No	No	No	No

Figure 3. Result of the $tableInfo$ Action

In the next sections, we review the theory if DNN, CNN, and RNN, then show how they can be built in CASL as well as Python.

DEEP LEARNING IN CASL

DEEP FEED-FORWARD NETWORK

The simplest form of a deep network is a deep feed-forward network or a deep neural network (DNN). Mathematically, let H_i, W_i, and b_i denote the output, the weight matrix, and the bias vector of hidden layer i respectively, then

$$H_i = \sigma(W_i \cdot H_i + b_i) \tag{1}$$

with $\sigma(\cdot)$ being an activation function. The most commonly used activation functions are sigmoid ($\sigma(x) = \frac{1}{1-\exp(-x)}$), hyperbolic tangent ($\sigma(x) = \tanh(x)$), or rectified linear function (ReLU) ($\sigma(x) = \max(0, x)$). The input layer can be denoted as $H_0 = X^{(j)}$ with $X^{(j)}$ being data instance j, and its output is $\hat{y} = s(W_k \cdot H_k + b_k)$ with k being the number of hidden layers, and $s(\cdot)$ being an output function. $s(\cdot)$ is selected based on the given task, for example, in a multilabel classification problem $s(\cdot)$ is often the SoftMax function which output a vector with the i^{th} dimension being $s_i(x) = \frac{\exp(x_i)}{\sum(\exp(x_j))}$. In a regression problem, $s(\cdot)$ is an identity function ($s(x) = x$). Overall, the computation from a data instance $X^{(i)}$ to its prediction $Y^{(i)}$ in a DNN of k hidden layers can be represented as follows

$$\begin{aligned} H_0 &= X^{(i)} \\ H_1 &= \sigma(W_0 \cdot H_0 + b_0) \\ H_2 &= \sigma(W_1 \cdot H_1 + b_1) \\ &\dots \\ H_k &= \sigma(W_{k-1} \cdot H_{k-1} + b_{k-1}) \\ y^{(i)} &= s(W_k \cdot H_k + b_k) \end{aligned} \tag{2}$$

DNNs are usually trained to minimize a predefined cost function L, varied by the tasks, using gradient descent. Parameters in layer i of the DNN are updated by

$$W_i \leftarrow W_i - \alpha * \frac{\partial L}{\partial W_i}$$
$$b_i \leftarrow b_i - \alpha * \frac{\partial L}{\partial b_i} \quad (3)$$

With α being a selected learning rate (a positive scalar, normally smaller than 1). L is selected based on the given task, for example Binary Cross-Entropy or Negative Log-Likelihood for classification, and Mean Squared Error for regression. In recent years, the ReLU activation function is preferred over others since it solves the gradient vanishing problem (gradients approach 0 when being passed to deeper layers with respect to the output layer).

In CAS, a DNN is built layer by layer. In other words, the users first generate an empty network where new layers are added sequentially. In general, the required parameters for each layer are its type, input, output, number of hidden neurons, and activation function. In CAS, the basic actions for building, training, and scoring a DNN are $buildModel$, $addLayer$, $dlTrain$, and $dlScore$, which are available in the $deepLearn$ action set. Additionally, the $modelInfo$ action in the same action set can be used to obtain information about an existing network. The code snippet below creates an empty DNN then adds an input layer, two hidden layers, and one output layer to it:

```
proc cas;
   session casauto;
   deepLearn.buildModel /
      modelTable={name="DNN",
                  replace=TRUE
                 }
      type="DNN";
   run;
   deepLearn.addLayer /
      layer={type="INPUT"}
      modelTable={name="DNN"}
      name="data";
   run;
   deepLearn.addLayer /
      layer={type="FC"
             n=50
             act='relu'
             init='xavier'
            }
      modelTable={name="DNN"}
      name="dnn1"
      srcLayers={"data"};
   run;
   deepLearn.addLayer /
      layer={type="FC"
             n=50
             act='relu'
             init='xavier'
            }
      modelTable={name="DNN"}
      name="dnn2"
      srcLayers={"dnn1"};
   run;
   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
```

```
            init='xavier'
           }
      modelTable={name="DNN"}
      name="outlayer"
      srcLayers={"dnn2"};
   run;
quit;
```

First, the $buildModel$ action initialize an empty DNN and links it to a new CAS dataset named DNN. The $replace = TRUE$ argument specifies to overwrite the DNN dataset if it has already existed, and the $type = DNN$ specifies that the network is a deep feed-forward network. Next, the $addLayer$ action is used to add one input layer, two hidden layer, and one output layer to the empty network. The $layer =$ argument of $addLayer$ specifies the adding layer's parameters such as layer type (in this case, we have $input$, FC – fully connected, and $output$), number of hidden neurons in the layer ($n = 50$), activation function ($act = 'relu'$), and the weight initialization method ($init = 'xavier'$). Three other important arguments of the $addLayer$ action are $modelTable$, $name$ and $srcLayers$ which defines the target network, the name of the adding layer, and the layer that act as the input of the adding layer, respectively. In this case, the names of the four layers are "$data$", "$dnn1$", "$dnn2$", and "$outlayer$"; they are sequentially connected: $data \rightarrow dnn1 \rightarrow dnn2 \rightarrow outlayer$. The results from the $buildModel$ and $addLayer$ action are showed in Figure 4.

Results from deepLearn.buildModel

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	dnn	1	5

Results from deepLearn.addLayer

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	dnn	10	5

Figure 4. Results from Actions to Build a DNN

After adding layers to the network, the action $modelInfo$ can be used to show the network architecture:

```
proc cas;
   session casauto;
   deepLearn.modelInfo /
      modelTable={name="DNN"};
   run;
quit;
```

The result of the $modelInfo$ action can be seen in Figure 5.

Results from deepLearn.modelInfo

Model dnn Information Details	
Model Name	dnn
Model Type	Deep Neural Network
Number of Layers	4
Number of Input Layers	1
Number of Output Layers	1
Number of Fully Connected Layers	2

Figure 5. Result of the modelInfo Action

Finally, the network is trained and scored with the $dlTrain$ and $dlScore$ actions. The code snippet below trains the network using the Adam method [6] in 10 iterations (epochs) in the $train$ dataset then scores the trained network in the $test$ dataset.

```
proc cas;
   session casauto;
   deepLearn.dlTrain /
      inputs={"open","close","high","low","adjclose","volume"}
      modelTable={name="DNN"}
      modelWeights={name="DNNWeights",
                    replace=TRUE
                   }
      nThreads=1
      optimizer={algorithm={method="ADAM",
                            lrPolicy='step',
                            gamma=0.5,
                            beta1=0.9,
                            beta2=0.999,
                            learningRate=0.1
                           },
                 maxEpochs=10,
                 miniBatchSize=1
                }
      seed=54321
      table={caslib="casuser",name="train"}
      target="y"
      nominal="y";
   run;
   deepLearn.dlScore /
      initWeights={name="DNNWeights"}
      modelTable={name="DNN"}
      table={caslib="casuser",name="test"};
   run;
quit;
```

Important arguments of the $dlTrain$ action include the input variables ($inputs =$), the network to train ($modelTable =$), the dataset to store the trained weights ($modelWeights =$) – which will be created during training, the training data ($table =$), and the target variable ($target =$). In this case, the network is trained for a classification task, so we add the $nominal =$ argument. In the $dlScore$ action, the users specify the weight table and network table, and the target scoring data. The result of the two actions can be seen in Figure 6.

Results from deepLearn.dlTrain

Model dnn Information Details	
Model Name	dnn
Model Type	Deep Neural Network
Number of Layers	4
Number of Input Layers	1
Number of Output Layers	1
Number of Fully Connected Layers	2
Number of Weight Parameters	2900
Number of Bias Parameters	102
Total Number of Model Parameters	3002
Approximate Memory Cost for Training (MB)	1

Optimization History of Deep Learning Model for TRAIN

Epoch	Learning Rate	Loss	Fit Error
0	0.1	10.684002959	0.481206
1	0.1	2.1650135031	0.467868
2	0.1	0.6949376032	0.527802
3	0.1	0.6916774632	0.479127
4	0.1	0.6907963548	0.464923
5	0.1	0.6912866069	0.464923
6	0.1	0.6910014063	0.464923
7	0.1	0.6907991905	0.464923
8	0.1	0.6907049727	0.464923
9	0.1	0.6908353689	0.464923

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSERHDFS( )	DNNWeights	3002	3

Results from deepLearn.dlScore

Score Information for TEST	
Number of Observations Read	503
Number of Observations Used	503
Misclassification Error (%)	44.13519
Loss Error	0.687353

Figure 6. Results from the dlTrain and dlScore Actions

CONVOLUTIONAL NEURAL NETWORK

The CNN architecture uses a set of filters that are slide through the pixels of each input image to generate feature maps, which allows features to be detected regardless of their locations in the image. The feature maps output by a convolutional layer are usually further subsampled to reduce their dimensionality and signify the major features in the maps. One among the common sub-sampling methods used in CNN is Max-Pooling, which returns the maximum value from a patch in the feature map. The convolutional/sub-sampling layer pair can be repeated as needed. Their final outputs are then typically connected to regular neural network layers then the output layer. Figure 7 illustrates a simple CNN of two convolutional/subsampling layers, one fully connected layer, and one output layer. CNN can be trained with gradient descent like a regular DNN.

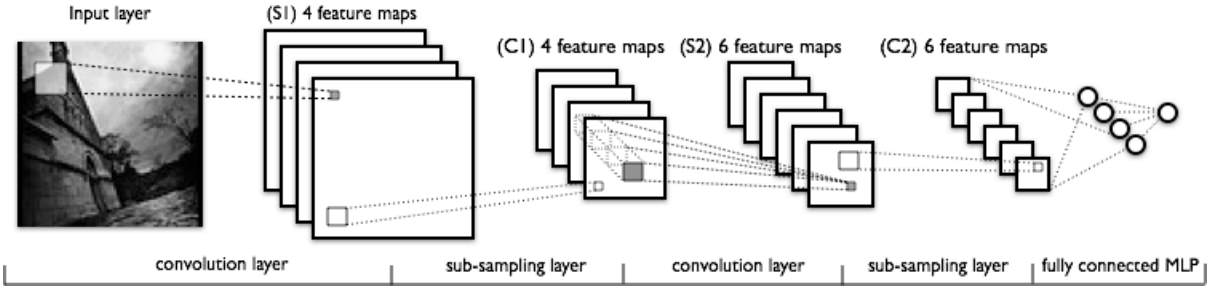

Figure 7. An Example of a Complete Convolutional Neural Network[1]

Recent successful architectures of CNN include AlexNet [7], VGG Net[8], ResNet [9], Google FaceNet [10], etc.

Building a CNN in CASL is similar to the process of building a regular DNN that is discussed in the previous section. The users first generate an empty network with the $modelBuild$ action, then add layer to it with the $addLayer$ action. There are two layer types used in a CNN beside the FC as in DNN, which are $CONVO$ (corresponding to the convolutional layer), and $POOL$ (corresponding to the pooling layer). The code snippet below generates an empty network, add one convolution/max-pooling layer pair and one fully connected layer to the network, besides the regular input and output layer:

```
proc cas;
   session casauto;
   deepLearn.buildModel /
      modelTable={name="CNN",
                  replace=TRUE
                 }
      type="CNN";
   run;
```

[1] . Image retrieved from http://deeplearning.net/tutorial/lenet.html

```
    deepLearn.addLayer /
       layer={type="INPUT"
              nchannels=1
              width=23
              height=28
             }
       modelTable={name="CNN"}
       name="data";
    run;
    deepLearn.addLayer /
       layer={type="CONVO"
              nFilters=20
              width=5
              height=5
              stride=1
             }
       modelTable={name="CNN"}
       name="conv1"
       srcLayers={"data"};
    run;
    deepLearn.addLayer /
       layer={type="POOL"
              width=2
              height=2
              stride=2
             }
       modelTable={name="CNN"}
       name="pool1"
       srcLayers={"conv1"}
       replace=TRUE;
    run;
    deepLearn.addLayer /
       layer={type="FC"
              n=500
             }
       modelTable={name="CNN"}
       name="dense"
       srcLayers={"pool1"};
    run;
    deepLearn.addLayer /
       layer={type="output"
              act='softmax'
              init='xavier'
             }
       modelTable={name="DNN"}
       name="outlayer"
       srcLayers={"dense"};
    run;
 quit;
```

As a CNN works with image data, its input layer has different arguments from the input layer of a DNN in the $addLayer$ action. More specifically, the users have to define the number of channels of the input images ($nchannels =$) (regular RGB images have three channels; grayscale images have one channel), the image's height ($height =$) and width ($width =$) in pixels. When adding layer of type $CONVO$, the required arguments are number of filters ($nFilters =$), the filters' size ($height =$ and $width =$), and the number of pixels to

slide the filters in each step ($stride =$). The $POOL$ layer has the same arguments as the CONVO layers, except for number of filters.

Similar to a DNN, the built CNN can be viewed with the $modelInfo$ action, and trained and scored with the $dlTrain$ and $dlScore$ actions, respectively.

RECURRENT NEURAL NETWORK

Recurrent Neural Networks (RNN) are specifically designed to handle temporal information in sequential data. Commonly used RNN types include vanilla RNN [11], Long Short-Term Memory (LSTM) [12], and Gated Recurrent Unit (GRU) [13]. In vanilla RNN's, the memory state of the current time point is computed from both the current input and its previous memory state. More formally, given a sequence $X = \{X_0, X_1, \dots, X_T\}$, the hidden state U_t of X_t (i.e. the state of X at time t) outputted by the network can be expressed as

$$U_t = \sigma(W \cdot X_t + R \cdot U_{t-1} + b) \quad (4)$$

where W and R are weight matrices of the network; b is the bias vector of the network; and $\sigma(\cdot)$ is a selected activation function.

Since its memory state is updated with the current input at every time point, vanilla RNN is typically unable to keep long-term memory. LSTM is an improved version of RNN with the design goal of learning to capture both long-term and short-term memories. A LSTM block uses gate functions, namely input gate, forget gate, and output gate, to control how much its long-term memory would be updated at each time point. The outputted short-term memory is then computed from the current input, the current long-term memory, and the previous short-term memory.

Compared with vanilla RNN, LSTM introduces a mechanism to learn to capture task-relevant long-term memory. However, the architecture of an LSTM block is relatively complex, which may cause training of a LSTM-based model difficult and time consuming. GRU can be viewed as an alternative to LSTM that can learn to capture task-relevant long-term memories with a simplified architecture. A GRU block contains only two gates and does not use long-term memory like in LST.

In CAS, building an RNN is relatively simple compared to a CNN. The $addLayer$ action can be used similarly as in building a DNN, except for the type of the recurrent layers. More specifically, the users need to set $type = 'recurrent'$, and add an argument $rnnType =< 'rnn','gru','lstm' >$ to specify the RNN type. Below is the code snippet to generate an empty RNN, then add an input layer, a GRU layer, and an output layer to it:

```
proc cas;
    session casauto;
    table.tableInfo /
          caslib = "casuser";
          name = "train";
    run;
quit;
proc cas;
   session casauto;
   deepLearn.buildModel /
      modelTable={name="GRU",
                  replace=TRUE
                 }
      type="RNN";
   run;
   deepLearn.addLayer /
      layer={type="INPUT"}
```

```
      modelTable={name="GRU"}
      name="data";
   run;
   deepLearn.addLayer /
      layer={type="recurrent"
             n=50
             act='tanh'
             init='xavier'
             rnnType='gru'
            }
      modelTable={name="GRU"}
      name="rnn1"
      srcLayers={"data"};
   run;
   deepLearn.addLayer /
      layer={type="output"
             act='softmax'
             init='xavier'
            }
      modelTable={name="GRU"}
      name="outlayer"
      srcLayers={"rnn1"};
   run;
quit;
```

Similar as DNN and CNN, the RNN can be trained with the $dlTrain$, and score with the $dlScore$ actions.

DEEP LEARNING WITH PYTHON

HIGH-LEVEL API

There are numerous deep learning packages available in Python, for example, TensorFlow [4], Theano [5], Keras [14], PyTorch [15], etc. The method of building a network by defining and connecting layers in CASL can be considered a high-level method that is similar to the high-level API in TensorFlow, Keras, or PyTorch. In this section, we focus on the high-level API in TensorFlow/Keras.

To use an external package in Python, it must first be imported into the current session. For example, the snippet below:

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import pandas as pd
```

loads the packages TensorFlow, Numpy [16], and Pandas [17], into the session, and aliases them as tf, np,and pd, respectively. In Python, aliasing a package allows user to call it during the session without having to refer to the package's full name. The $layers$ module is imported from $tensorflow.keras$ without being given any alias.

Assuming the users have already had the data loaded in the correct format for TensorFlow (training data and labels are $trainX$ and $trainY$, and testing data and labels are $testX$ and $testY$), the code snippet below generates a two hidden layer DNN with a SoftMax output layer:

```
model = tf.keras.Sequential()
model.add(layers.Dense(50, activation='relu'))
```

```
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dense(2, activation='softmax'))
```

In line-by-line order, an empty model is first created as the $model$ object. The $Sequential$ function allows layers to be added to the $model$ object one by one without specifying their inputs and outputs. Then, two fully-connected (dense layers) with 50 hidden neurons and using the ReLU activation function, and an output layer of two output neurons using SoftMax output function, are added to the network. As can be seen, besides syntax, this is similar to using the $buildModel$ and $addLayer$ actions in CASL. A difference is that, unless the input data is not ready for a network, an input layer is not necessary.

An initialized network must be compiled before training. A simple way is as below:

```
model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(trainX, trainY, epochs=10, batch_size=32)
```

the compile function set some important criteria such as optimizer (Adam in this case), loss function, and evaluation metric. The model then is trained with the fit function. Finally, a trained model can be scored with the evaluation function:

```
model.evaluate(testX, testY, batch_size=32)
```

Similar to in CASL, the layer type can be changed to convolutional, pooling, recurrent, etc. to accommodate the deep architecture that is needed. For example, the snippet below:

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=2, padding='same',
    activation='relu', input_shape=(28,28,1)))
model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=2, padding='same',
    activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

generates an empty model, then add two pairs of convolutional/pooling layers, and two fully-connected layers to the empty model. As mentioned previously, the network must be compiled and trained before using.

LOW-LEVEL API

A more complicated but more flexible way (depending on the use case) to build a deep learning model is to define its computational map. This method is usually referred to as low-level API in deep learning packages such as TensorFlow and Theano. Refer to equation (2), the computational flow of a two-hidden-layer DNN with binary output can be as follows

$$\begin{aligned} H_1 &= relu(W_0 \cdot X + b_0) \\ H_2 &= relu(W_1 \cdot H_1 + b_1) \\ \hat{y} &= sigmoid(W_{out} \cdot H_2 + b_2) \end{aligned} \tag{5}$$

Where H_1 and H_2 are the output of the hidden layers, $\hat{y}$ is the output of the DNN, and W_* and b_* are the weights and bias vectors of the layers. In building a network with low-level API, X, H_1, H_2, and $\hat{y}$ are considered variables that are input or computed on fitting; whereas all W's and b's are considered trainable parameters that have to be initialized (e.g. randomly

initialized) before training. The training process then updates the values of W's and b's to optimize a certain loss function. After training, the values of all W's and b's are fixed. The code snippet below realizes the computational map in (5) with Theano:

```
from theano import *
import theano.tensor as T
from numpy.random import normal

x = T.matrix('x')
y = T.matrix('y')

#1st hidden layer
W0 = theano.shared(normal(loc=0,scale=0.001,size=(8,50)),name='W0')
b0 = theano.shared(np.zeros(50),name='b0')
H1 = T.nnet.relu(T.dot(x,W0) + b0)

#2nd hidden layer
W1 = theano.shared(normal(loc=0,scale=0.001,size=(50,50)),name='W1')
b1 = theano.shared(np.zeros(50),name='b1')
H2 = T.nnet.relu(T.dot(H1,W1) + b1)

#output layer
Wout = theano.shared(np.zeros((50,1)),name='Wout')
bout = theano.shared(np.zeros(1),name='b1')
Yhat = T.nnet.sigmoid(T.dot(H2,Wout) + bout)
```

First, used packages are first imported and aliased (if necessary). We also import the $normal$ function from Numpy to initialize the weights of the DNN with a normal distribution (with mean of 0 and scale of 0.001, as seen later in the code).

x and y are then generated as variables that will be used in training and testing the network; outsides of such cases, they are symbolic and carry no actual values. With the input (x) defined, we begin to generate the weights and biases of each layer, as well as define the computations as needed (i.e. the computational sequence $x \to H_1 \to H_2 \to \hat{y}$).

To train the DNN, Theano provides certain modules for computing gradients and updating parameters. For example, the generated DNN can be trained with stochastic Gradient Descent as follows:

```
#loss function
L = T.nnet.binary_crossentropy(Yhat,y).mean()

#select trainable parameters and compute gradients w.r.t. L
params = [W0,b0,W1,b1,Wout,bout]
gparams = [T.grad(L, param) for param in params]
learning_rate = T.scalar('learning_rate')
updates = [
    (param, param - learning_rate * gparam)
    for param, gparam in zip(params, gparams)
]

#functions to train and predict
train_model = theano.function(
    inputs=[x,y,learning_rate],
    outputs=L,
    updates=updates,
)
```

```
predict = theano.function(
    inputs=[x],
    outputs=Yhat
)
```

In this case, the loss function is binary cross entropy that is computed based on the true label y and the output label $\hat{y}$. After defining the gradients and updating rules, the *train_model* function can be called iteratively to train the selected parameters:

```
for epoch in range(10):
    print("Epoch %d, cost: %f" %
          (epoch,train_model(trainX,trainY,np.float32(0.1))))
```

Finally, we can make predictions with the *predict* function:

```
Y_pred = (predict(testX) > 0.5) * 1
```

Since the raw output of the predict function is the probability of $\hat{y} = 1$ (since the output layer use sigmoid function), we can compare it to 0.5 and convert the Boolean values to integer to have the final prediction. As seen from the sample codes, building a deep model with low-level API is more complicated compared to high-level tools like CASL.

DISCUSSION

Previously, we show the programming styles for deep learning in CASL and two representative Python packages – TensorFlow and Python. As can be seen, the steps to build, train, and use, a deep learning model in SAS/CASL is relatively similar to the high-level API of deep learning packages in Python. There are a few notable differences, however:

1. CASL has all the dataset-centric characteristics of SAS. More specifically, components (i.e. layers and parameters) of a deep network are stored in SAS datasets. In Python, parameters in layers are typically stored as tensors, matrices, or vectors, that are connected by the network's computational map.
2. Similar to the network's components, training and testing data are also stored in SAS datasets. In Python, data can be stored as different types of objects. In the simplest case, datasets are also tensors, matrices, or vectors.

The advantages of CASL is that its syntax and usages is similar to other SAS procedures, and thus being friendlier to SAS users. Moreover, the users can utilize other powerful tools like SAS data steps and procedures to manipulate the data in the same sessions. However, in certain cases, this data-centric characteristic of SAS may cause some disadvantages to user.

First, storing parameters in a dataset is not desirable in really big network, as one parameter takes place as one row with additional information like layer ID and weight ID (we show this architecture in Figure 8). Consequently, a network of millions of parameters would result in the same number of rows with considerably more information to be processed, which may cause more overhead when the network is first accessed.

Obs	_LayerID_	_WeightID_	_Weight_
1	1	0	-1.050011039
2	1	6	-0.408701062
3	1	12	-0.015391403
4	1	18	-0.163090229
5	1	24	1.2083156109
6	1	30	-0.236806735
7	1	36	-1.381471276
8	1	42	-0.150624439
9	1	48	0.2562239766
10	1	54	-0.702641487

Figure 8. Stored Parameters of a Deep Model in CAS

Second, storing data as SAS dataset is also not desirable in certain cases, for example, image processing. The wide image format in SAS converts each pixel in one channel to one column in the storing dataset. Therefore, a 28×28 RGB image is converted to a SAS dataset of $28 \times 28 \times 3 = 2352$ variables, or a 128×128 grayscale image results in a dataset of over 16,000 variables. Overall, this method requires more storage and processing power than the Python method, which represent images through 3D or 4D tensors.

Compared to low-level deep learning package like Theano, CASL is certainly simpler to use. These packages are usually used as backends for other high-level packages like Keras, or when users need more controls in the implementation process (e.g. when designing new types of deep models). Consequently, the low-level tools are arguably more flexible than high-level tools like CASL. However, this is, however, not necessary and may be over-complicated for new users or tasks that focus more on application of common deep architectures.

Outside of the disadvantages, however, we believe CASL/SAS is a powerful tool for SAS users to utilize deep learning architectures in their tasks without the needs of learning or integrating new tools in their SAS sessions.

CONCLUSION

In this paper, we showcase the uses of three toolboxes, namely SAS Viya/CASL, Python-TensorFlow, and Python-Theano, in modeling with deep learning. We highlight the main differences between CASL and the Python packages, and show that for the general purpose of using common deep learning models, CASL is sufficient and arguably more powerful for SAS users.

REFERENCES

[1] Schmidhuber, Jürgen (2015). "Deep learning in neural networks: An overview". In: Neural networks 61, pp. 85–117.

[2] LeCun, Yann, Yoshua Bengio, et al. (1995). "Convolutional networks for images, speech, and time series". In: The handbook of brain theory and neural networks 3361.10, p. 1995.

[3] Funahashi, Ken-ichi and Yuichi Nakamura (1993). "Approximation of dynamical systems by continuous time recurrent neural networks". In: Neural networks 6.6, pp. 801–806.

[4] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16) (pp. 265-283).

[5] Bergstra, James et al. (2010). "Theano:ACPU and GPU math compiler in Python". In: Proc. 9th Python in Science Conf, pp. 1–7.

[6] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In NIPS, 2012.

[8] Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: arXiv preprint arXiv:1409.1556.

[9] He, Kaiming et al. (2016). "Deep residual learning for image recognition". In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778.

[10] Schroff, Florian, Dmitry Kalenichenko, and James Philbin (2015). "Facenet: A unified embedding for face recognition and clustering". In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 815–823.

[11] Funahashi, Ken-ichi and Yuichi Nakamura (1993). "Approximation of dynamical systems by continuous time recurrent neural networks". In: Neural networks 6.6, pp. 801–806.

[12] Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: Neural computation 9.8, pp. 1735–1780.

[13] Chung, Junyoung et al. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: arXiv preprint arXiv:1412.3555.

[14] Gulli, A., & Pal, S. (2017). Deep Learning with Keras. Packt Publishing Ltd.

[15] Ketkar, N. (2017). Introduction to pytorch. In Deep learning with python (pp. 195-208). Apress, Berkeley, CA.

[16] Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. Computing in Science & Engineering, 13(2), 22.

[17] McKinney, W. (2011). pandas: a foundational Python library for data analysis and statistics. Python for High Performance and Scientific Computing, 14.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Linh Le
Institute of Analytics and Data Science
lle12@students.kennesaw.edu