Demo Steps: Deep Learning Forecasting Using the R API

The National Centers for Environmental Information (NCEI) provides public access to environmental data archives. The data set **durham** contains hourly weather information for the city of Durham, North Carolina, from 2008 to 2017. There are approximately 90,000 observations. The goal is to use a recurrent neural network to forecast the maximum hourly temperature in Durham, North Carolina.

Name	Model Role	Measurement Level	Description
LST_DATE	Date	Date	Date of each observation (YYYYMMDD)
LST_TIME	Input	Nominal	Hour of the observation (0-2300 by 100)
T_MAX	Target	Interval	Maximum air temperature in degrees C during the hour
P_CALC	Input	Interval	Total amount of precipitation in mm during the hour

1. From Jupyter Lab, select **File Browser** > **Home** > **Courses** > **EVMLOPRC** > **Notebooks** and select the **R_Time_Series_Demo.ipynb** notebook. Scroll to part 2.

2. Load the SWAT and repr packages. Use the options function to change the size of the plots.
```
library(swat)
library(repr)
options(repr.plot.width=5.5, repr.plot.height=5.5)
```
3. Connect to CAS.
```
conn = CAS(Sys.getenv("CASHOST"), as.numeric(Sys.getenv("CASPORT")), password=Sys.getenv("SAS_VIYA_TOKEN"))
```
4. Load the durham.csv file on the server. Print the dimension and first few observations.
```
castbl = cas.read.csv(conn, file=
   paste0(Sys.getenv("HOME"),"/Courses/EVMLOPRC/DATA/durham.csv"),
   casOut = list(name="durham", replace=TRUE))

indata = 'durham'

dim(castbl)
head(castbl)
```
5. Use a DATA step to subset the first year of data (2008).
```
loadActionSet(conn, 'dataStep')

cas.dataStep.runCode(conn, code=
    "
    data sample;
        set durham;
        if lst_date < 20090000 then output sample;
    run;
    "
)
```
6. Download the first year of data to the client and plot the hourly maximum temperatures.
```
df_sample = defCasTable(conn, table='sample')
df_sample = to.casDataFrame(df_sample)
plot(df_sample$T_MAX, type="l", col="blue", main='1 Year Hourly
   Temperature Max', ylab='Celsius', xlab='Hour', ylim=c(-20,35))
```
Note: Missing data is coded as -99.

7. Use a DATA step to create lag variables for **t_max**, **lst_time**, and **p_calc** for a maximum of five hours.
```
cas.dataStep.runCode(conn, code=
    "
    data durham;
        set durham;
        t_max_1 = lag1(t_max);
        t_max_2 = lag2(t_max);
        t_max_3 = lag3(t_max);
        t_max_4 = lag4(t_max);
        t_max_5 = lag5(t_max);

        lst_time_1 = lag1(lst_time);
        lst_time_2 = lag2(lst_time);
        lst_time_3 = lag3(lst_time);
        lst_time_4 = lag4(lst_time);
        lst_time_5 = lag5(lst_time);

        p_calc_1 = lag1(p_calc);
        p_calc_2 = lag2(p_calc);
        p_calc_3 = lag3(p_calc);
        p_calc_4 = lag4(p_calc);
        p_calc_5 = lag5(p_calc);
    run;
    "
    ,single='YES'
)

cas.table.fetch(conn, table=indata, to=6)
```

Because lags are computed sequentially based on the time-ordered data, you must run the DATA step sequentially instead of in parallel.

Selected argument:

Argument	Description
single	specifies when to restrict execution to a single thread. ◦ NO is the default and runs the program in the number of threads that are specified by the nThreads parameter. ◦ NOINPUT runs the program in one thread when there are no input data sets. When there are input data sets, the nThreads parameter specifies the number of threads to use. For distributed servers, if the program has no input data sets, the program runs in one thread on one worker. Otherwise, the nThreads parameter specifies the number of threads to use. ◦ YES overrides the nThreads parameter and runs the program in one thread. For distributed servers, the program runs on one thread on one worker.

Notice the missing data for the lags until observation 6.

8. Use a DATA step to remove missing observations and temperatures that are coded as less than -30 degrees Celsius.

```
cas.dataStep.runCode(conn, code=
    "
    data durham missing;
        set durham;
        if cmiss(of _all_) or t_max < -30 then output missing;
        else output durham;
    run;
    "
)
```

9. Use a DATA step to partition the data into training (before 2015), validation (2015 and 2016), and test (2017) sets.

```
cas.dataStep.runCode(conn, code=
    "
    data train validate test;
        set durham;
        if lst_date < 20150000 then output train;
        else if lst_date < 20170000 then output validate;
        else output test;
    run;
    "
)
```

10. Use the deepLearn action set to build a long short-term memory (LSTM) recurrent neural network with two hidden layers and 15 neurons in each.

```
loadActionSet(conn, 'deepLearn')

cas.deepLearn.buildModel(conn,
    model = list(name='lstm', replace=TRUE),
    type = 'RNN'
)

cas.deepLearn.addLayer(conn,
    model = 'lstm',
    layer = list(type='input', std='std'),
    replace = TRUE,
    name = 'data'
)

cas.deepLearn.addLayer(conn,
    model = 'lstm',
    layer = list(type='recurrent', n=15, init='xavier',
                 rnnType='LSTM', outputType='samelength'),
    srcLayers = 'data',
    replace = TRUE,
    name = 'rnn1'
)

cas.deepLearn.addLayer(conn,
    model = 'lstm',
    layer = list(type='recurrent', n=15, init='xavier',
                 rnnType='LSTM', outputType='encoding'),
    srcLayers = 'rnn1',
    replace = TRUE,
    name = 'rnn2'
)

cas.deepLearn.addLayer(conn,
    model = 'lstm',
    layer = list(type='output', act='identity', init='normal'),
    srcLayers = 'rnn2',
    replace = TRUE,
    name = 'output'
)

cas.deepLearn.modelInfo(conn,
    model='lstm'
)
```

Selected arguments for the buildModel action:

Argument	Description
model	specifies an in-memory table that is used to store the model.
type	specifies the model type. ◦ CNN creates an empty model for building a convolutional neural network. ◦ DNN creates an empty model for building a deep, fully connected neural network. ◦ RNN creates an empty model for building a recurrent neural network.

Selected arguments for the addLayer action:

Argument	Description
model	specifies the in-memory table that is the model.
name	specifies a unique name for the layer. As you add layers to the model, you specify this name in the srcLayers parameter. The name is case insensitive.
srcLayers	specifies the names of the source layers for this layer.
layer	specifies the layer type and related parameters for the layer. The value that you specify for **type** determines the other parameters that apply.

Selected layer arguments options:

Argument	Description
std	specifies how to standardize the variables in the input layer. ◦ MIDRANGE specifies to scale variables so that the midrange is 0 and the half-range is 1. The result is that variables have a minimum of -1 and a maximum of 1. This is the default standardization when the model type is DNN. ◦ NONE specifies that variables are not modified. This is the default standardization when the model type is CNN. ◦ STD specifies to scale variables so that the mean is 0 and the standard deviation is 1. This is the default standardization for other model types.
type	can be set to BATCHNORM, CONVO, FC, INPUT, OUTPUT, POOL, PROJECTION, RESIDUAL, and RECURRENT.
n	specifies the number of neurons for the layer.
act	specifies the activation function for the layer. The default AUTO specifies to assign the activation function automatically. For a convolutional layer, RECTIFIER is used. For a fully connected layer, TANH is used. For a pooling layer, IDENTITY is used. For an output layer, classification models use SOFTMAX and other models use IDENTITY. For a recurrent layer, LSTM and GRU types use IDENTITY, and RNN uses TANH.
init	specifies the initialization scheme for the neurons.
rnnType	specifies the type for the RNN layer. ◦ GRU specifies that the layer type is a gated recurrent unit. ◦ LSTM specifies that the layer type is a long short-term memory unit. ◦ RNN specifies that the layer type is a recurrent neural network.
outputType	specifies the output type for the RNN layer. ◦ ARBITRARYLENGTH specifies that a sequence with an arbitrary length should be generated. ◦ ENCODING specifies that a fixed-length vector should be generated. ◦ SAMELENGTH specifies that a sequence with the same length as the input should be generated.
error	specifies the error function. This function is also known as the loss function. AUTO specifies that the error function should be set according to the model. For a classification model, the error function is ENTROPY. For all other models, the error function is NORMAL. The error can be set to AUTO, CTC, ENTROPY, FCMPERR, GAMMA, NORMAL, and POISSON.

11. Create variables for the target and inputs to pass into subsequent actions.

```
castbl = defCasTable(conn, table=indata)
inputs = names(castbl)
inputs = inputs[c(-1,-3)]
target = 'T_MAX'

target
inputs
```

12. Use the dlTrain action to build the LSTM RNN.

```
cas.deepLearn.dlTrain(conn,
    table      = 'train',
    validTable = 'validate',
    target     = target,
    inputs     = inputs,
    sequenceOpts = list(timeStep=5),
    seed = '1234',
    modelTable = 'lstm',
    modelWeights = list(name='trained_weights', replace=TRUE),
    optimizer = list(miniBatchSize=4, maxEpochs=50,
                     algorithm=list(method='adam', gamma=0.2,
                     learningRate=0.01, clipGradMax=10000,
                     clipGradMin=-10000, stepSize=30,
                     lrPolicy='step'))
)
```

Selected arguments:

Argument	Description
table	specifies the settings for an input table.
validTable	specifies the table with the validation data. The validation table must have the same columns and data types as the training table.
sequenceOpts	specifies the settings for sequence data.
modelTable	specifies the in-memory table that is the model.
modelWeights	specifies an in-memory table that is used to store the model weights.
optimizer	specifies the settings for the optimization algorithm, optimization mode, and other settings such as a seed, the maximum number of epochs, and so on.

Selected optimizer argument options:

Argument	Description
algorithm	specifies the algorithm that is used to find the model weights. The available methods are ADAM, LBFGS, MOMENTUM, and VANILLA. The value that you specify for method determines the other parameters that apply.
miniBatchSize	specifies the number of observations per thread in a mini-batch. You can use this parameter to control the number of observations that the action uses on each worker for each thread to compute the gradient before updating the weights. Larger values use more memory. When synchronous SGD is used (the default), the total mini-batch size is equal to miniBatchSize * number of threads * number of workers. When asynchronous SGD is used (by specifying the elasticSyncFreq parameter), each worker trains its own local model. In this case, the total mini-batch size for each worker is miniBatchSize * number of threads.
maxEpochs	specifies the maximum number of epochs. For SGD with a single-machine server or a session that uses one worker on a distributed server, one epoch is reached when the action passes through the data one time. For a session that uses more than one worker, one epoch is reached when all the workers exchange the weights with the controller one time. The syncFreq parameter specifies the number of times each worker passes through the data before exchanging weights with the controller.
bet1	specifies the exponential decay rate for the first moment in the Adam learning algorithm.
beta2	specifies the exponential decay rate for the second moment in the Adam learning algorithm.
gamma	specifies the gamma for the learning rate policy.
learningRate	specifies the learning for stochastic gradient descent.
clipGradMax	specifies the maximum gradient value. All gradients that are greater than the specified value are set to the specified value.
clipGradMin	specifies the minimum gradient value. All gradients that are less than the specified value are set to the specified value.
stepSize	specifies the step size when the learning rate policy is set to STEP.
lrPolicy	specifies the learning rate policy. ○ FIXED specifies a fixed learning rate. ○ IIN specifies to set the learning rate after each epoch according to the initial learning rate, the value of the gamma parameter, and the value of the power parameter. The rate is calculated as the learningRate*(1+gamma*current epoch) ^(-power). ○ MULTISTEP specifies to set the learning rate after each of the epochs that are specified in the steps parameter. The learning rate is multiplied by the gamma parameter value. ○ POLY specifies to set the learning rate after each epoch according to the initial learning rate, the maximum number of epochs, and the value of the power parameter. The rate is calculated as the learningRate*(1-current epoch /maxEpochs)^power. ○ STEP specifies to set the learning rate to the current learning rate multiplied by the gamma parameter value. The number of steps is specified in the stepSize parameter. The rate is recalculated for each group of epochs, according to the step size.

Note: For time series analysis, it is easy to overfit the data by building too many hidden layers and neurons into the model. A general rule is to keep the number of weight parameters below the number of training observations. When the number of weight parameters exceeds the number of observations, the forecasts can be dramatically different from the actual observations.

13. Use the trained weights to score the test data set.

```
cas.deepLearn.dlScore(conn,
    table = 'test',
    model = 'lstm',
    initWeights = 'trained_weights',
    copyVars = c(target, 'LST_DATE', 'LST_TIME'),
    casout = list(name='lstm_scored', replace=TRUE)
)
```

Selected arguments:

Argument	Description
initWeights	specifies an in-memory table that contains the model weights. These weights are used to initialize the model.
copyVars	specifies the variables to transfer from the input table to the output table.

Note: You need to copy the data and time variables so that you can order the scored data.

14. Download the scored information to the client and order it by the **lst_date** and **lst_time** variables.

```
df = defCasTable(conn, table='lstm_scored')
df = to.casDataFrame(df)

df = df[order(df$LST_DATE,df$LST_TIME),]
head(df)
dim(df)
```

15. Print the average absolute forecast error for the test data set and plot the first 1000 hours of the test data along with the period's forecast.

```
paste0('Average absolute error = ',
        round(mean(abs((df$T_MAX-df$`_DL_Pred_`))),4))

plot(1:1000,df$T_MAX[1:1000], type="l", col="blue",
     main='A Thousand Hour Forecast', ylab='Celsius', xlab='Hour')
lines(1:1000,df$`_DL_Pred_`[1:1000], type="l", col="red")
legend(700,-8,legend=c('Observed','Predicted'),
       col=c('blue','red'),lwd=2, lty=c(1,1), bty='n')
```

16. End the CAS session.

```
cas.session.endSession(conn)
```