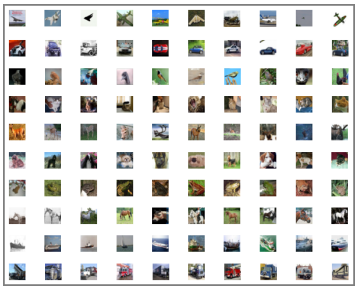
Demo Steps: Classifying Color Images Using the Python API

The CIFAR-10 data set consists of 10,000 32x32 color images with 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck). In this demonstration, a convolutional neural network is built and trained to classify images into the 10 possible classes.



1. From Jupyter Lab, select **File Browser** > **Home** > **Courses** > **EVMLOPRC** > **Notebooks** and select the **Python_Image_Classification_Demo.ipynb** notebook.

2. 2. Load the os, sys, SWAT, numpy, pandas, and matplotlib packages. Use the %matplotlib inline statement to create graphics within the notebook and set the CAS option to print CAS messages in the notebook.
```
import os
import sys
import swat
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
swat.options.cas.print_messages = True
```
3. Connect to CAS and create a connection object called **conn**.
```
conn = swat.CAS(os.environ.get("CASHOST"), os.environ.get("CASPORT"), None, os.environ.get("SAS_VIYA_TOKEN"))
```
4. Use the addCasLib action from the table action set to add a CAS library connecting to the course data directory where the cifar10 images are located.
```
conn.table.addCaslib(name="mycl", path=os.environ.get("HOME")+"/Courses/EVMLOPRC/DATA",
    subDirectories = True, dataSource="PATH",
    description="Personal File Save Location", activeOnAdd=False)
```
5. Load the image action set and then use the loadImages action to load all the images from the **cifar10** folder on the server.
```
conn.loadActionSet('image')
actions = conn.builtins.help(actionSet='image')

conn.image.loadImages(
    caslib='mycl',
    path='cifar10',
    recurse=True,
    decode=True,
    distribution="random",
    labelLevels=1,
    casout=dict(name='cifar10', blocksize='128',replace=True)
)

indata = 'cifar10'
```
Note: in the cifar10 folder there are ten sub folders. Each folder contains images of a specific image type. The recurse argument will read in all the images from the ten directories and the labelLevels argument will label the images according to each subdirectory to keep the data organized.

The loadImages action reads images and associated metadata into memory from a variety of sources, including these:
- photographic image files
- biomedical image files, including those in Digital Imaging and Communications in Medicine (DICOM) format
- ZIP files
- a directory that contains image files or ZIP files
- a directory that contains one or more series of photographic or biomedical images
- a directory tree that contains any number of the preceding items
- URLs
- lists that contain combinations of the sources in the preceding items

The output contains images in a CAS table in encoded or decoded format, depending on the value of the decode input parameter. When this parameter is set to False (default value), the output table has the following columns:
- **_id_** An integer that is unique for each image
- **_path_** The full pathname of the image
- **_image_** A binary large object that contains the entire image file
- **_size_** The byte length of the binary large object in the _image_ column
- **_type_** A three-character string that specifies the format of the image (for example, **jpg** for JPG images)

When the **decode** parameter is set to True, the _image_ column contains decompressed image data, and the output table has these additional columns:

- **_dimension_** Number of dimensions of the image (for example, 2 for two-dimensional images, 3 for three-dimensional images)
- **_resolution_** Size of the image in each dimension (for example, {100, 200} for an array for a two-dimensional image that is 100 pixels wide and 200 pixels high)
- **_imageFormat_** Integer that represents the organization of data in the _image_ column. The most common values are 0, 16, 2, and 18, corresponding to 8-bit gray scale, 8-bit RGB, 16-bit gray scale, and 16-bit RGB images, respectively.

Selected arguments:

Argument	Description
path	specifies the file or directory path to load images from the server.
recurse	when set to *True*, loads images in subdirectories recursively.
decode	when set to *True*, writes decompressed images and metadata to the output table.
distribution	specifies the algorithm and parameters to be used to distribute loaded images to worker nodes.
labelLevels	specifies the maximum number of directory levels to include in the label. If labelLevels is set to a nonzero integer *n*, the output table will include an additional column called _label_ whose type is string. If *n* is positive, each image in a directory will have a _label_ column that contains *n* top-level subdirectories of the root directory, separated by slashes.

6. Use the summarize Images action to view attributes of the images and use the columnInfo action from the table action set to view the variable names in the **cifar10** data.
```
conn.image.summarizeImages(imagetable=indata)

conn.table.columnInfo(table=indata)
```
The _label_ variable contains the response level information for the image classification model.

7. Use the freq action from the simple action set to view the number of images in each level of the variable _label_ created by the loadImages action.
```
conn.loadActionSet('simple')
actions = conn.builtins.help(actionSet='simple')

conn.simple.freq(
    table = indata,
    inputs = '_label_'
)['Frequency']
```
8. Partition the data into 60% for training, 20% for validation, and 20% for testing using the srs action from the sampling action set.
```
conn.loadActionSet('sampling')
actions = conn.builtins.help(actionSet='sampling')

conn.sampling.srs(
    table = indata,
    samppct = 60,
    samppct2 = 20,
    seed = 802,
    partind = True,
    output  = dict(casOut = dict(name = indata, replace = True),
                   copyVars = 'ALL')
)
```
9. Use the shuffle action from the table action set to rearrange the data and avoid a potential ordering bias.
```
conn.table.shuffle(
    table = indata,
    casOut = dict(name=indata, replace=True)
)
```

10. Use the deepLearn action set to build a convolutional neural network with a convolution layer, a pooling layer, and a fully connected layer.
    - In the input layer, set nchannels to 3 because these are color images (red, green blue) and set the width and height to the image size of 32x32. Scale the image and standardize them as well.
    - In the convolutional layer use 10 5x5 filters with a stride of 1. Also let the activation function be relu and the weight initialization be Xavier.
    - In the pooling layer use tiles of 2x2 with a stride of 2 and summarize the localized regions with the maximum summary function.
    - In the fully connected layer use 100 neurons and a dropout rate of 0.4 to prevent overfitting. Also use relu and Xavier for the activation and initiation respectively.
    - In the output layer use softmax because there are ten unique target classes.

```
conn.loadActionSet('deepLearn')
actions = conn.builtins.help(actionSet='deepLearn')

conn.deepLearn.buildModel(
    model = dict(name='cnn',replace=True),
    type = 'CNN'
)

conn.deepLearn.addLayer(
    model = 'cnn',
    layer = dict(type='input', nchannels=3, width=32, height=32,
                 scale=0.004, std='STD'),
    replace=True,
    name = 'data'
)
```

```
conn.deepLearn.addLayer(
    model = 'cnn',
    layer = dict(type='convolution', act='relu', nFilters=10,
                 width=5, height=5, stride=1, init='xavier'),
    srcLayers = 'data',
    replace=True,
    name = 'cnn1'
)

conn.deepLearn.addLayer(
    model = 'cnn',
    layer = dict(type='pooling', width=2, height=2, stride=2,
                 pool='max'),
    srcLayers = 'cnn1',
    replace=True,
    name = 'pool1'
)

conn.deepLearn.addLayer(
    model = 'cnn',
    layer = dict(type='fullconnect', n=100, act='relu',
                 init='xavier', dropout = 0.4),
    srcLayers = 'pool1',
    replace=True,
    name = 'fc1'
)

conn.deepLearn.addLayer(
    model = 'cnn',
    layer = dict(type='output', act='softmax', init='xavier'),
    srcLayers = 'fc1',
    replace=True,
    name = 'output'
)

conn.deepLearn.modelInfo(
    model='cnn'
)
```

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 argument options:

Argument	Description
type	can be set to BATCHNORM, CONVO, FC, INPUT, OUTPUT, POOL, PROJECTION, RESIDUAL, and RECURRENT.
nchannels	specifies the number of channels in the input data. Common values are 1 for monochrome images and 3 for images with red, green, and blue components.
width	specifies the width of each image in pixels. By default, the width is determined automatically when the model training begins.
height	specifies the height of each image in pixels. By default, the height is determined automatically when the model training begins.
scale	specifies a scaling factor to apply to each image.
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.
stride	specifies the stride - both width (horizontal) and height (vertical).
pool	specifies the pool type for the pooling layer (AVERAGE, FIXED, MAX, MIN, RANDOM).

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.
outputType	specifies the output type for the RNN layer. ◦ ARBITRARYLENGTH specifies to generate a sequence with arbitrary length. ◦ ENCODING specifies to generate a fixed-length vector. ◦ SAMELENGTH specifies to generate a sequence with the same length as the input.
error	specifies the error function. This function is also known as the loss function. Auto specifies to set the error function 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.
dropout	specifies the dropout rate for the layer.

11. Use the dlTrain action to train a convolutional neural network on the **cifar10** training data set (and validation for tuning) where the target is the variable _label_ and the input are the images represented by the _image_ variable. Train for 100 epochs using the momentum algorithm and use a learnging rate of 0.01. Save the trained weights to score the test data after.

```
conn.deepLearn.dlTrain(
    table    = dict(name=indata, where='_PartInd_ = 1'),
    validTable = dict(name=indata, where='_PartInd_ = 2'),
    target = '_label_',
    inputs = '_image_',
    seed = '649',
    modelTable = 'cnn',
    modelWeights = dict(name='trained_weights', replace=True),
    optimizer = dict(miniBatchSize=50, maxEpochs=100,  loglevel=1,
             algorithm=dict(method='momentum', learningRate=0.01))
)
```

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.
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 optinmizer argument options:

Argument	Description
algorithm	specifies the algorithm 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 prior to 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.
learningRate	specifies the learning for stochastic gradient descent.
loglevel	specifies how progress messages are sent to the client. The default value, 0, indicates that no messages are sent. Specify 1 to receive start and end messages. Specify 2 to include the iteration history.

12. Use the dlScore action to score the test data with the trained weights. Save the scored information and also the target and input in the scored output CAS table to analyze the results.

```
conn.deepLearn.dlScore(
    table = dict(name=indata, where='_PartInd_ = 0'),
    model = 'cnn',
    initWeights = 'trained_weights',
    copyVars = ['_label_','_image_'],
    layerImageType='jpg',
    casout = dict(name='cnn_scored', replace=True)
)
```

Here the misclassification is near 50%. Although this is a large error in general, a random guess model would have a 90% misclassification error rate. We could try reducing the error further by running the model longer, using more hidden layers, and changing the hyperparameters of the optimization routine.

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.
layerImageType	specifies the image type to store in the output layers table (JPG or WIDE).

13. Use the crosstab action to print the actual versus predicted image classes for the test data.
```
crosstab = conn.simple.crossTab(
    table = 'cnn_scored',
    row = '_label_',
    col = '_DL_PredName_'
)['Crosstab']

crosstab = crosstab.drop('_label_',1)
crosstab
```
The row represents the actual response level and the column represents the prediction. Ideally the diagonal values are as large as possible, indicating good model fit.

Note that using the assignment statement with the freq action effectively pulls the table to client as a data frame. We can than operate on it using open source functionality.

14. Use the crosstab action results to create a data frame of the proportion of correct and misclassified results for each image type.

```
correct = pd.DataFrame(np.diagonal(crosstab)) /
          pd.DataFrame(crosstab.sum(axis=1))
miss = 1-correct
classes = pd.DataFrame(['airplane','automobile','bird','cat',
                        'deer','dog','frog','horse','ship','truck'])

df = pd.concat([classes, correct, miss], axis=1)
df.columns = ['Label','Correct','Misclassified']
df
```
15. Create a bar chart for the proportion of misclassified image types.
```
MisPlot = df.plot(x='Label', y='Misclassified', kind='bar',
                  figsize=(8,8), fontsize=15, color='blue')
MisPlot.set_xlabel('Image', fontsize=15)
MisPlot.set_ylabel('Percent Misclassified', fontsize=15)
MisPlot.legend_.remove()
plt.show()
```
It appears that animal images in general (other than frogs) are more challenging to classify than mechanical images such as airplanes, automobiles, ships, and trucks using this five-layer convolutional neural network.

16. End the CAS session.
```
conn.session.endSession()
```