Lab Program - 12
Handwritten Digit Recognition with MNIST Using Dist-Keras

```
#Part - I
from distkeras.evaluators import *
from distkeras.predictors import *
from distkeras.trainers import *
from distkeras.transformers import *
from distkeras.utils import *
from keras.layers.convolutional import *
from keras.layers.core import *
from keras.models import Sequential
from keras.optimizers import *
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.feature import StandardScaler
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler
import pwd
import os

#Part – II
# First, set up the Spark variables. You can modify them to your needs.
application_name = "Distributed Keras MNIST Notebook"
using_spark_2 = False
local = False
path_train = "/FileStore/tables/mnist_train.csv"
path_test = "/FileStore/tables/mnist_test.csv"
if local:
   # Tell master to use local resources.
   master = "local[*]"
   num_processes = 3
   num_executors = 1
else:
   # Tell master to use YARN.
   master = "yarn-client"
   num_executors = 20
   num_processes = 1
```

```
# This variable is derived from the number of cores and executors and will be used
to assign the number of model trainers.
num_workers = num_executors * num_processes

#Part – III
print("Number of desired executors: " + 'num_executors')
print("Number of desired processes / executor: " + 'num_processes')
print("Total number of workers: " + 'num_workers')
# Use the Databricks CSV reader; this has some nice functionality regarding
invalid values.
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages com.databricks:sparkcsv_
2.10:1.4.0 pyspark-shell'
conf = SparkConf()
conf.set("spark.app.name", application_name)
conf.set("spark.master", master)
conf.set("spark.executor.cores", 'num_processes')
conf.set("spark.executor.instances", 'num_executors')
conf.set("spark.executor.memory", "4g")
conf.set("spark.locality.wait", "0")
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.local.dir", "/tmp/" + get_os_username() + "/dist-keras");

#Part – IV
# Check if the user is running Spark 2.0 +
if using_spark_2:
   sc = SparkSession.builder.config(conf=conf) \
   .appName(application_name) \
   .getOrCreate()
else:
   # Create the Spark context.
   #sc = SparkContext(conf=conf)
   # Add the missing imports.
   from pyspark import SQLContext
   sqlContext = SQLContext(sc)
# Check if we are using Spark 2.0.
if using_spark_2:
   reader = sc
else:
   reader = sqlContext
# Read the training dataset.
```

```
raw_dataset_train = reader.read.format('com.databricks.spark.csv') \
.options(header='true', inferSchema='true') \
.load(path_train)

#Part – V
# Read the testing dataset.
raw_dataset_test = reader.read.format('com.databricks.spark.csv') \
.options(header='true', inferSchema='true') \
.load(path_test)
# First, we would like to extract the desired features from the raw
# dataset. We do this by constructing a list with all desired columns.
# This is identical for the test set.
features = raw_dataset_train.columns
features.remove('label')
# Next, we use Spark's VectorAssembler to "assemble" (create) a vector of
# all desired features.
vector_assembler = VectorAssembler(inputCols=features, outputCol="features")

#Part – VI
# This transformer will take all columns specified in features and create an
additional column "features" which will contain all the desired
# features aggregated into a single vector.
dataset_train = vector_assembler.transform(raw_dataset_train)
dataset_test = vector_assembler.transform(raw_dataset_test)
# Define the number of output classes.
nb_classes = 10
encoder = OneHotTransformer(nb_classes, input_col="label",
output_col="label_encoded")
dataset_train = encoder.transform(dataset_train)
dataset_test = encoder.transform(dataset_test)
# Allocate a MinMaxTransformer from Distributed Keras to normalize
# the features.
# o_min -> original_minimum
# n_min -> new_minimum
transformer = MinMaxTransformer(n_min=0.0, n_max=1.0, o_min=0.0,
o_max=250.0, input_col="features", output_col="features_normalized")



#Part – VII
```

```
# Transform the dataset.
dataset_train = transformer.transform(dataset_train)
dataset_test = transformer.transform(dataset_test)
# Keras expects the vectors to be in a particular shape; we can reshape the vectors using Spark.
reshape_transformer = ReshapeTransformer("features_normalized", "matrix", (28, 28, 1))
dataset_train = reshape_transformer.transform(dataset_train)
dataset_test = reshape_transformer.transform(dataset_test)
# Now, create a Keras model.
# Taken from Keras MNIST example.
# Declare model parameters.
img_rows, img_cols = 28, 28
# Number of convolutional filters to use
nb_filters = 32
# Size of pooling area for max pooling
pool_size = (2, 2)
# Convolution kernel size
kernel_size = (3, 3)
input_shape = (img_rows, img_cols, 1)

#Part – VIII
# Construct the model.
convnet = Sequential()
convnet.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1], input_shape=input_shape))
convnet.add(Activation('relu'))
convnet.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
convnet.add(Activation('relu'))
convnet.add(MaxPooling2D(pool_size=pool_size))
convnet.add(Flatten())
convnet.add(Dense(225))
convnet.add(Activation('relu'))
convnet.add(Dense(nb_classes))
convnet.add(Activation('softmax'))
# Define the optimizer and the loss.
optimizer_convnet = 'adam'
loss_convnet = 'categorical_crossentropy'
# Print the summary.
convnet.summary()
```

```
#Part – IX
# We can also evaluate the dataset in a distributed manner.
# However, for this we need to specify a procedure on how to do this.
def evaluate_accuracy(model, test_set, features="matrix"):
    evaluator = AccuracyEvaluator(prediction_col="prediction_index",
    label_col="label")
    predictor = ModelPredictor(keras_model=model, features_col=features)
    transformer = LabelIndexTransformer(output_dim=nb_classes)
    test_set = test_set.select(features, "label")
    test_set = predictor.predict(test_set)
    test_set = transformer.transform(test_set)
    score = evaluator.evaluate(test_set)
    return score
```

```
Part – X
# Select the desired columns; this will reduce network usage.
dataset_train = dataset_train.select("features_normalized", "matrix","label",
"label_encoded")
dataset_test = dataset_test.select("features_normalized", "matrix","label",
"label_encoded")
# Keras expects DenseVectors.
dense_transformer = DenseTransformer(input_col="features_normalized",
output_col="features_normalized_dense")
dataset_train = dense_transformer.transform(dataset_train)
dataset_test = dense_transformer.transform(dataset_test)
dataset_train.repartition(num_workers)
dataset_test.repartition(num_workers)
# Assessing the training and test set.
training_set = dataset_train.repartition(num_workers)
test_set = dataset_test.repartition(num_workers)
# Cache them.
training_set.cache()
test_set.cache()
# Precache the training set on the nodes using a simple count.
print(training_set.count())
# Use the ADAG optimizer. You can also use a SingleWorker for testing
# purposes -> traditional nondistributed gradient descent.
trainer = ADAG(keras_model=convnet, worker_optimizer=optimizer_convnet,
loss=loss_convnet, num_workers=num_workers, batch_size=16,
communication_window=5, num_epoch=5, features_col="matrix",
label_col="label_encoded")
trained_model = trainer.train(training_set)
print("Training time: " + str(trainer.get_training_time()))
print("Accuracy: " + str(evaluate_accuracy(trained_model, test_set)))
print("Number of parameter server updates: " +
str(trainer.parameter_server.num_updates))
```

LAB Program 13

Program on spark with dl using Dogs and Cats Image Classification

```
#Part – I
import matplotlib.pyplot as plt
import numpy as np
import cv2
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing import image
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras import backend as K
# The image dimension is 150x150. RGB = 3.
if K.image_data_format() == 'channels_first':
   input_shape = (3, 150, 150)
else:
   input_shape = (150, 150, 3)
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape, activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

#Part – II
# Compile the model.
model.compile(loss='binary_crossentropy',
optimizer='rmsprop',
metrics=['accuracy'])
print(model.summary())

#Part – III
# We will use the following augmentation configuration for training.
train_datagen = ImageDataGenerator(
rescale=1. / 255,
width_shift_range=0.2,
height_shift_range=0.2,
horizontal_flip=True)

#Part - IV
# The only augmentation for test data is rescaling.
```

```
test_datagen = ImageDataGenerator(rescale=1. / 255)
train_generator = train_datagen.flow_from_directory('/kaggle/input/cats-and-dogs-
small/cats_and_dogs_small/train',
target_size=(150, 150),
batch_size=16,
class_mode='binary')
#Found 4000 images belonging to 2 classes.

validation_generator = test_datagen.flow_from_directory('/kaggle/input/cats-and-dogs-
small/cats_and_dogs_small/test',
target_size=(150, 150),
batch_size=16,
class_mode='binary')
#Found 1000 images belonging to 2 classes.

#Part – V
# steps_per_epoch should be set to the total number of training sample,
# while validation_steps is set to the number of test samples. We set
# epoch to 15, steps_per_epoch and validation_steps to 100 to expedite
# model training.
model.fit_generator(
train_generator,
steps_per_epoch=100,
epochs=25,
validation_data=validation_generator,
validation_steps=100)

#Part – VI
# We get a 71% validation accuracy. To increase model accuracy, you can try several things such
as
#adding more training data and increasing the number of epochs.
model.save_weights('dogs_vs_cats.h5')
# Let's now use our model to classify a few images. dogs=1, cats=0
# Let's start with dogs.
img = cv2.imread("/kaggle/input/cats-and-dogs-
small/cats_and_dogs_small/test/dogs/dog.1503.jpg")
img = np.array(img).astype('float32')/255
img = cv2.resize(img, (150,150))
plt.imshow(img)
plt.show()

#Part – VII
img = img.reshape(1, 150, 150, 3)
print(model.predict(img))
print(np.round(model.predict(img)))
```

LAB Program 14

Program on Running a CNN for learning MNIST with DeepLearning4j over Spark

```
object CNN_MNIST {
def main(args:Array[String]): Unit ={
val nCores =2
val conf = new SparkConf()
.setMaster("spark://master:7077")
.setAppName("MNIST_CNN")
.set(SparkDl4jMultiLayer.AVERAGE_EACH_ITERATION,
String.valueOf(true))
val sc = new SparkContext(conf)
val nChannels = 1
val outputNum = 10
val numSamples = 60000
val nTrain = 50000
val nTest = 10000
val batchSize = 64
val iterations = 1
val seed = 123
val mnistIter = new MnistDataSetIterator(1,numSamples, true)
val allData = new ListBuffer[DataSet]()
while(mnistIter.hasNext) allData.+=(mnistIter.next)
new Random(12345).shuffle(allData)
val iter = allData.iterator
var c =0
val train = new ListBuffer[DataSet]()
val test = new ListBuffer[DataSet]()
while(iter.hasNext) {
if(c <= nTrain) {
train.+=(iter.next)
c +=1
}
else test.+=(iter.next)
}
val sparkDataTrain = sc.parallelize(train)
sparkDataTrain.persist(StorageLevel.MEMORY_ONLY)
println("Building model ....")

val builder = new NeuralNetConfiguration.Builder()
```

```
.seed(seed)
.iterations(iterations)
.regularization(true).l2(0.0005)
.learningRate(0.01)
.optimizationAlgo(OptimizationAlgorithm
.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.ADAGRAD)
.list(6)
.layer(0, new ConvolutionLayer.Builder(5, 5)
.nIn(nChannels)
.stride(1, 1)
.nOut(20)
.weightInit(WeightInit.XAVIER)
.activation("relu")
.build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer
.PoolingType.MAX, Array(2, 2))
.build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
.nIn(20)
.nOut(50)
.stride(2,2)
.weightInit(WeightInit.XAVIER)
.activation("relu")
.build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX,
Array(2, 2))
.build())
.layer(4, new DenseLayer.Builder().activation("relu")
.weightInit(WeightInit.XAVIER)
.nOut(200).build())
.layer(5, new OutputLayer.Builder
(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
.nOut(outputNum)
.weightInit(WeightInit.XAVIER)
.activation("softmax")
.build())
.backprop(true).pretrain(false);
new ConvolutionLayerSetup(builder,28,28,1);
val multiLayerConf:MultiLayerConfiguration= builder.build()
val net:MultiLayerNetwork = new
MultiLayerNetwork(multiLayerConf)
net.init()
net.setUpdater(null)
val sparkNetwork = new SparkDl4jMultiLayer(sc, net)
val nEpochs = 5
(0 until nEpochs).foreach{i => val network =
sparkNetwork.fitDataSet(sparkDataTrain, nCores*batchSize)

//Evaluate the model
val eval = new Evaluation()
for(ds <- test)
{
val output = network.output(ds.getFeatureMatrix)
eval.eval(ds.getLabels, output)
}
println("Statistics..."+eval.stats())
}
}
}
```

Lab Program - 15

Working with Caffe On Spark - Running a feed-forward neural network with Deep Learning 4j over Spark

Here is the Spark application which uses CaffeOnSpark to train a dataset on HDFS and MLlib to perform non-deep learning, that is, logistic regression for classification:

```
val conf = new SparkConf()
.setMaster("spark://master:7077")
.setAppName("Caffe_Spark_Application")
val sc = new SparkContext(conf)
val cos = new CaffeOnSpark(sc)
val config = new Config(sc,args)
val dl_train_source = DataSource.getSource(config, true)
cos.train(dl_train_source)
val lr_raw_source = DataSource.getSource(config, false)
val extracted_df = cos.features(lr_raw_source)
val lr_input_df = extracted_df.withColumn("Label",
cos.floatarray2doubleUDF(extracted_df(config.label)))
.withColumn("Feature",
cos.floatarray2doublevectorUDF(extracted_df(config.features(0))))
//Learn a LogisticRegression model via Apache MLlib
val lr = new LogisticRegression()
.setLabelCol("Label")
.setFeaturesCol("Feature")
val lr_model = lr.fit(lr_input_df)
//save the LogisticRegression classification model onto HDFS
lr_model.write.overwrite().save(config.outputPath)
```

The preceding code is submitted to the Spark cluster as follows:

```
spark-submit \ -files
caffenet_train_solver.prototxt,caffenet_train_net.prototxt \ -
num-executors 2 \ -class
com.yahoo.ml.caffe.examples.MyMLPipeline \
caffe-grid-0.1-SNAPSHOT-jar-with-dependencies.jar \
-features fc8 \ -label label \ -conf
caffenet_train_solver.prototxt \ -model
hdfs:///sample_images.model \ -output
hdfs:///image_classifier_model \ -devices 2
```

Running a feed-forward neural network with DeepLearning 4j over Spark

```
import scala.collection.mutable.ListBuffer
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.canova.api.records.reader.RecordReader
import org.canova.api.records.reader.impl.CSVRecordReader
import org.deeplearning4j.nn.api.OptimizationAlgorithm
import org.deeplearning4j.nn.conf.MultiLayerConfiguration
import org.deeplearning4j.nn.conf.NeuralNetConfiguration
import org.deeplearning4j.nn.conf.layers.DenseLayer
import org.deeplearning4j.nn.conf.layers.OutputLayer
import org.deeplearning4j.nn.multilayer.MultiLayerNetwork
import org.deeplearning4j.nn.weights.WeightInit
import org.deeplearning4j.spark.impl.multilayer.
SparkDl4jMultiLayer
import org.nd4j.linalg.io.ClassPathResource
import org.nd4j.linalg.lossfunctions.LossFunctions
object FeedForwardNetworkWithSpark {
def main(args:Array[String]): Unit ={
val recordReader:RecordReader = new CSVRecordReader(0,",")
val conf = new SparkConf()
.setMaster("spark://master:7077")
.setAppName("FeedForwardNetwork-Iris")
val sc = new SparkContext(conf)
val numInputs:Int = 4
val outputNum = 3
val iterations =1
val multiLayerConfig:MultiLayerConfiguration = new
NeuralNetConfiguration.Builder()
.seed(12345)
.iterations(iterations)
.optimizationAlgo(OptimizationAlgorithm
.STOCHASTIC_GRADIENT_DESCENT)
.learningRate(1e-1)
.l1(0.01).regularization(true).l2(1e-3)
.list(3)
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(3)
.activation("tanh")
.weightInit(WeightInit.XAVIER)
.build())
.layer(1, new DenseLayer.Builder().nIn(3).nOut(2)
.activation("tanh")
.weightInit(WeightInit.XAVIER)
.build())
.layer(2, new
OutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
.weightInit(WeightInit.XAVIER)
.activation("softmax")
.nIn(2).nOut(outputNum).build())
.backprop(true).pretrain(false)
.build
val network:MultiLayerNetwork = new
MultiLayerNetwork(multiLayerConfig)
network.init
network.setUpdater(null)
val sparkNetwork:SparkDl4jMultiLayer = new
SparkDl4jMultiLayer(sc,network)
val nEpochs:Int = 6
val listBuffer = new ListBuffer[Array[Float]]()
(0 until nEpochs).foreach{i =>
val net:MultiLayerNetwork =
sparkNetwork.fit("file:///<path>/
iris_shuffled_normalized_csv.txt",4,recordReader)
listBuffer +=(net.params.data.asFloat().clone())
}
println("Parameters vs. iteration Output: ")
(0 until listBuffer.size).foreach{i =>
println(i+"\t"+listBuffer(i).mkString)}
}
}
```

LAB Program 16
Installing TensorFlow - Working with Spark TensorFlow

```
import numpy as np
import tensorflow as tf
import os
from tensorflow.python.platform import gfile
import os.path
import re
import sys
import tarfile
from subprocess import Popen, PIPE, STDOUT
def run(cmd):
  p = Popen(cmd, shell=True, stdin=PIPE, stdout=PIPE, stderr=STDOUT,
close_fds=True)
  return p.stdout.read()
model_dir = '/tmp/imagenet'
image_file = ""
num_top_predictions = 5
DATA_URL = 'http://download.tensorflow.org/models/image/imagenet/inception-
2015-12-05.tgz'

IMAGES_INDEX_URL = 'http://image-
net.org/imagenet_data/urls/imagenet_fall11_urls.tgz'
# The number of images to process.
image_batch_size = 3
max_content = 1000L

def read_file_index():
  from six.moves import urllib
  content = urllib.request.urlopen(IMAGES_INDEX_URL)
  data = content.read(max_content)
  tmpfile = "/tmp/imagenet.tgz"
  with open(tmpfile, 'wb') as f:
    f.write(data)
  run("tar -xOzf %s > /tmp/imagenet.txt" % tmpfile)
  with open("/tmp/imagenet.txt", 'r') as f:
    lines = [l.split() for l in f]
    input_data = [tuple(elts) for elts in lines if len(elts) == 2]
    return [input_data[i:i+image_batch_size] for i in range(0,len(input_data),
image_batch_size)]
```

```
class NodeLookup(object):
  """Converts integer node ID's to human readable labels."""

  def __init__(self,
               label_lookup_path=None,
               uid_lookup_path=None):
    if not label_lookup_path:
      label_lookup_path = os.path.join(
          model_dir, 'imagenet_2012_challenge_label_map_proto.pbtxt')
    if not uid_lookup_path:
      uid_lookup_path = os.path.join(
          model_dir, 'imagenet_synset_to_human_label_map.txt')
    self.node_lookup = self.load(label_lookup_path, uid_lookup_path)

  def load(self, label_lookup_path, uid_lookup_path):
    """Loads a human readable English name for each softmax node.

    Args:
      label_lookup_path: string UID to integer node ID.
      uid_lookup_path: string UID to human-readable string.

    Returns:
      dict from integer node ID to human-readable string.
    """
    if not gfile.Exists(uid_lookup_path):
      tf.logging.fatal('File does not exist %s', uid_lookup_path)
    if not gfile.Exists(label_lookup_path):
      tf.logging.fatal('File does not exist %s', label_lookup_path)

    # Loads mapping from string UID to human-readable string
    proto_as_ascii_lines = gfile.GFile(uid_lookup_path).readlines()
    uid_to_human = {}
    p = re.compile(r'[n\d]*[ \S,]*')
    for line in proto_as_ascii_lines:
      parsed_items = p.findall(line)
      uid = parsed_items[0]
      human_string = parsed_items[2]
      uid_to_human[uid] = human_string
    # Loads mapping from string UID to integer node ID.
    node_id_to_uid = {}
    proto_as_ascii = gfile.GFile(label_lookup_path).readlines()
    for line in proto_as_ascii:
      if line.startswith('  target_class:'):
        target_class = int(line.split(': ')[1])
      if line.startswith('  target_class_string:'):
        target_class_string = line.split(': ')[1]
        node_id_to_uid[target_class] = target_class_string[1:-2]

    # Loads the final mapping of integer node ID to human-readable string
    node_id_to_name = {}
    for key, val in node_id_to_uid.items():
      if val not in uid_to_human:
        tf.logging.fatal('Failed to locate: %s', val)
      name = uid_to_human[val]
      node_id_to_name[key] = name

    return node_id_to_name

  def id_to_string(self, node_id):
    if node_id not in self.node_lookup:
      return ''
    return self.node_lookup[node_id]


def create_graph():
  """"Creates a graph from saved GraphDef file and returns a saver."""
  # Creates graph from saved graph_def.pb.
  with gfile.FastGFile(os.path.join(
      model_dir, 'classify_image_graph_def.pb'), 'rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
    _ = tf.import_graph_def(graph_def, name='')

def run_inference_on_image(image):
  """Runs inference on an image.

  Args:
    image: Image file name.
  Returns:
    Nothing
  """
  if not gfile.Exists(image):
    tf.logging.fatal('File does not exist %s', image)
  image_data = gfile.FastGFile(image, 'rb').read()

  # Creates graph from saved GraphDef.
  create_graph()

  gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.15)
  with tf.Session(config=tf.ConfigProto(log_device_placement=True,
gpu_options=gpu_options)) as sess:
    # Some useful tensors:
    # 'softmax:0': A tensor containing the normalized prediction across
    #   1000 labels.
    # 'pool_3:0': A tensor containing the next-to-last layer containing 2048
    #   float description of the image.
    # 'DecodeJpeg/contents:0': A tensor containing a string providing JPEG
    #   encoding of the image.
    # Runs the softmax tensor by feeding the image_data as input to the graph.
    softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')
    predictions = sess.run(softmax_tensor,
                           {'DecodeJpeg/contents:0': image_data})
    predictions = np.squeeze(predictions)

    # Creates node ID --> English string lookup.
    node_lookup = NodeLookup()

    top_k = predictions.argsort()[-num_top_predictions:][::-1]
    for node_id in top_k:
      human_string = node_lookup.id_to_string(node_id)
      score = predictions[node_id]
      print('%s (score = %.5f)' % (human_string, score))


def maybe_download_and_extract():
  """Download and extract model tar file."""
  from six.moves import urllib
  dest_directory = model_dir
  if not os.path.exists(dest_directory):
    os.makedirs(dest_directory)
  filename = DATA_URL.split('/')[-1]
  filepath = os.path.join(dest_directory, filename)
  if not os.path.exists(filepath):
    filepath2, _ = urllib.request.urlretrieve(DATA_URL, filepath)
    print("filepath2", filepath2)
    statinfo = os.stat(filepath)
    print('Succesfully downloaded', filename, statinfo.st_size, 'bytes.')
    tarfile.open(filepath, 'r:gz').extractall(dest_directory)
  else:
      print('Data already downloaded:', filepath, os.stat(filepath))

maybe_download_and_extract()
batched_data = read_file_index()
print "There are %d batches" % len(batched_data)

label_lookup_path = os.path.join(model_dir,
'imagenet_2012_challenge_label_map_proto.pbtxt')
uid_lookup_path = os.path.join(model_dir,
'imagenet_synset_to_human_label_map.txt')
def load_lookup():
  """Loads a human readable English name for each softmax node.

  Args:
    label_lookup_path: string UID to integer node ID.
    uid_lookup_path: string UID to human-readable string.

  Returns:
    dict from integer node ID to human-readable string.
  """
  if not gfile.Exists(uid_lookup_path):
    tf.logging.fatal('File does not exist %s', uid_lookup_path)
  if not gfile.Exists(label_lookup_path):
    tf.logging.fatal('File does not exist %s', label_lookup_path)

  # Loads mapping from string UID to human-readable string
  proto_as_ascii_lines = gfile.GFile(uid_lookup_path).readlines()
  uid_to_human = {}
  p = re.compile(r'[n\d]*[ \S,]*')
  for line in proto_as_ascii_lines:
    parsed_items = p.findall(line)
    uid = parsed_items[0]
    human_string = parsed_items[2]
    uid_to_human[uid] = human_string

  # Loads mapping from string UID to integer node ID.
  node_id_to_uid = {}
  proto_as_ascii = gfile.GFile(label_lookup_path).readlines()
  for line in proto_as_ascii:
    if line.startswith('  target_class:'):
      target_class = int(line.split(': ')[1])
    if line.startswith('  target_class_string:'):
      target_class_string = line.split(': ')[1]
      node_id_to_uid[target_class] = target_class_string[1:-2]

  # Loads the final mapping of integer node ID to human-readable string
  node_id_to_name = {}
  for key, val in node_id_to_uid.items():
    if val not in uid_to_human:
      tf.logging.fatal('Failed to locate: %s', val)
    name = uid_to_human[val]
    node_id_to_name[key] = name

  return node_id_to_name

node_lookup = load_lookup()
node_lookup_bc = sc.broadcast(node_lookup)

model_path = os.path.join(model_dir, 'classify_image_graph_def.pb')
with gfile.FastGFile(model_path, 'rb') as f:
  model_data = f.read()

model_data_bc = sc.broadcast(model_data)

def run_image(sess, img_id, img_url, node_lookup):
  from six.moves import urllib
  from urllib2 import HTTPError
  try:
    image_data = urllib.request.urlopen(img_url, timeout=1.0).read()
  except HTTPError:
    return (img_id, img_url, None)
  except:
    return (img_id, img_url, None)
  scores = []
  softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')
  try:
    predictions = sess.run(softmax_tensor,
                           {'DecodeJpeg/contents:0': image_data})
  except:
    return (img_id, img_url, None)
  predictions = np.squeeze(predictions)
  top_k = predictions.argsort()[-num_top_predictions:][::-1]
  scores = []
  for node_id in top_k:
    if node_id not in node_lookup:
      human_string = ''
    else:
      human_string = node_lookup[node_id]
    score = predictions[node_id]
    scores.append((human_string, score))
  return (img_id, img_url, scores)

def apply_batch(batch):
  with tf.Graph().as_default() as g:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(model_data_bc.value)
    tf.import_graph_def(graph_def, name='')
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.15)
    with tf.Session(config=tf.ConfigProto(log_device_placement=True,
gpu_options=gpu_options)) as sess:
      labelled = [run_image(sess, img_id, img_url, node_lookup_bc.value) for
(img_id, img_url) in batch]
      return [tup for tup in labelled if tup[2] is not None]

urls = sc.parallelize(batched_data)
labelled_images = urls.flatMap(apply_batch)
local_labelled_images = labelled_images.collect()
print(local_labelled_images)
```

Lab Program – 17
Implementing k-means using H2O over Spark - Running SVM with H2O over Spark

Here is the code to run k-means using H2O over Spark

```
import org.apache.spark._
import org.apache.spark.sql._
import org.apache.spark.h2o._
import hex.kmeans.KMeansModel.KMeansParameters
import hex.kmeans.{KMeans, KMeansModel}
import water._
import water.support.SparkContextSupport
object H2O_KmeansDemo {
def main(args:Array[String]): Unit = {
val conf = new SparkConf()
.setMaster("spark://master:7077")
.setAppName("H2O_KmeansDemo")
val sc = new SparkContext(conf)
val sqlContext = new SQLContext(sc)
val h2oContext = H2OContext.getOrCreate(sc)
import h2oContext._
import h2oContext.implicits._
import sqlContext.implicits._
val prostateDf = sqlContext.read.format("com.databricks.spark.csv")
.option("header", "true")
.option("inferSchema",
"true").load("hdfs://namenode:9000/prostate.csv")
prostateDf.registerTempTable("prostate_table")
val result = sqlContext.sql("SELECT * FROM prostate_table
WHERE CAPSULE=1")
val h2oFrame = h2oContext.asH2OFrame(result)
/* Build a KMeans model, setting model parameters via a
Properties */
val model = runKmeans(h2oFrame)
println(model)
// Shutdown Spark cluster and H2O
h2oContext.stop(stopSparkContext = true) }
def runKmeans[T](trainDataFrame: H2OFrame): KMeansModel = {
val params = new KMeansParameters
params._train = trainDataFrame._key
params._k = 3
// Create a builder
val job = new KMeans(params)
// Launch a job and wait for the end.
val kmm = job.trainModel.get
// Print the JSON model
println(new String(kmm._output.writeJSON(new
AutoBuffer()).buf()))
// Return a model
kmm
}
}
```

The code that runs SVM on the data is as follows:

```
import java.io._
import org.apache.spark.ml.spark.models.svm._
import org.apache.spark.h2o.H2OContext
import org.apache.spark.sql.SQLContext
import org.apache.spark.{SparkConf, SparkContext, SparkFiles}
import water.fvec.H2OFrame
import water.support.SparkContextSupport

object H2O_SVM {
def main(args: Array[String]): Unit = {
val conf = new SparkConf()
.setMaster("spark://master:7077")
.setAppName("H2O_SVMDemo")
val sc = new SparkContext(conf)
implicit val h2oContext = H2OContext.getOrCreate(sc)
implicit val sqlContext = new SQLContext(sc)
val breastCancerData = new H2OFrame(new File)
// Training data
breastCancerData.replace(breastCancerData.numCols()-1,
breastCancerData.lastVec().toCategoricalVec)
breastCancerData.update()
// Configure DeepLearning Algorithm
val parms = new SVMParameters
parms._train = breastCancerData.key
parms._response_column = "label"
val svm = new SVM(parms, h2oContext)
val svmModel = svm.trainModel.get
// Use model for scoring
val predictionH2OFrame = svmModel.score(breastCancerData)
val predictionsFromModel =
h2oContext.asDataFrame(predictionH2OFrame).collect
println(predictionsFromModel.mkString("\n===> Model predictions: ",
",\n", ", ...\n"))
h2oContext.stop(stopSparkContext = true)
}
}
```