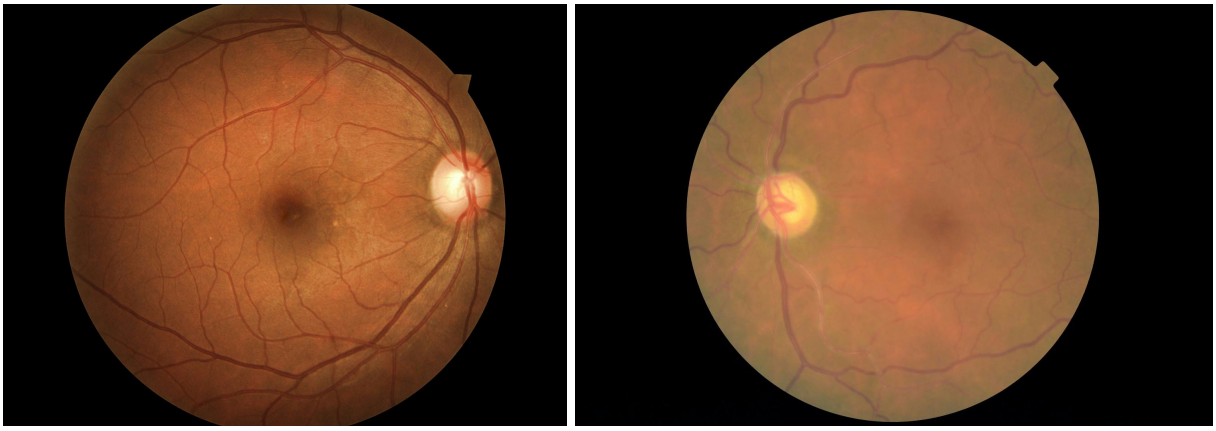
W251: Scaling Up! Really Big Data
Summer 2015



Diabetic Retinopathy in Eye Images

(A Kaggle Competition)

Team Members:
Katie Adams, Kevin Alen, Nate Black, Malini Mittal

Introduction

Diabetic Retinopathy (DR) is an eye disease associated with long-standing diabetes. Around 40% of the 29 million Americans with diabetes have some stage of the disease. Progression to vision impairment can be slowed or averted if DR is detected in time, however this can be difficult as the disease often shows few symptoms until it is too late to provide effective treatment. Kaggle, a website that promotes machine learning competitions, recently conducted a competition to promote the detection of DR. Kaggle provided users with training data in the form of JPEG images which were classified as one of the five categories below:

0 - No DR
1 - Mild
2 - Moderate
3 - Severe
4 - Proliferative DR

In addition to the training data, Kaggle also provided test data with unlabeled images. Contestants were asked to classify the test data and were then evaluated on a “weighted kappa score.” The size of the compressed training data was 33 GB while the test data was 50 GB.

This project explores two disparate methods of image classification: distributed computation using Spark’s MLlib framework and parallel computation using GPUs with the Python library Theano. A third approach using GPUs on Spark via the DeepLearning4J package was explored but the DeepLearning4J package was too unstable for reliable use. While classification using machine learning algorithms was part of the project, the focus was not on classification accuracy. Rather, the team focused on the *tools and setup* required to successfully implement a large-scale image classification problem. The group focused on automated cloud provisioning, large-scale cloud computing (e.g. GPUs and distributed frameworks), efficiently storing and retrieving data in the cloud, and maximizing resource use. This paper aims to outline the highlights of the project including key takeaways, pain points, and ideas for future exploration.

Data Acquisition

Training and test data were provided by Kaggle in the form of zipped JPEG files. The training data was divided into 5 segments, and the test data into 7. These segments were downloaded individually, and then concatenated to form the complete training and test data. The training data had 35,126 observations and the test data had 53,576, where each observation was provided as an individual JPEG file. Each image size was approximately about 3000 x 4000. The training data labels were provided in a separate comma-separated values (CSV) file.

Data was downloaded to a Spark cluster and preprocessed. SoftLayer’s Object Storage was used to store both the original as well as the preprocessed data. Object Storage provided a very convenient, low-cost, and robust means to centralized storage. It was convenient in that the data needed to be transformed only once, and could be transferred to the virtual machine clusters as needed for use; this was important as our project relied on spinning up clusters on demand and tearing them down after analysis. Had we been required to download and transform the raw data with every cluster, productivity would have diminished exponentially as preprocessing was a time-consuming task, requiring up to a day. We could have stored data locally and transferred it to the cloud as needed but this would have required users to store ~100GB (multiple datasets were used) of data on their local disk. Additionally, all users would need to store the data locally where SoftLayer Object Storage allowed the data to be stored in a central repository accessible to all users. The data replication provided robustness.

Cluster Provisioning and Bootstrapping

An important part of the project was creating an automated provisioning script that would allow the team to provision a cloud on demand with minimal manual setup. The distributed computation aspect of the project required the Hadoop Distributed File System (HDFS), Spark, and Softlayer's Object Storage (as well as other dependencies).

Depending on the size of the cluster, even experienced users can take 30 minutes or more to configure a cluster for HDFS and Spark. In addition to being time consuming, HDFS and Spark configurations are often error prone (e.g. forgetting a config file, typos, etc). Therefore, the team looked to two different solutions: SaltStack and Ansible. The team originally planned to only use SaltStack, but when difficulties caused the implementation to be delayed, other options were explored. In the end, the team created solutions with both technologies.

Ansible

Ansible is an IT automation tool that is quickly growing in popularity compared to rivals such as Chef, Puppet, and Salt. Written in Python, Ansible is a "push" system where commands are pushed from a host machine to "slave" machines. While critics argue that Ansible is less powerful than its competitors, Ansible shines in terms of ease of use. Commands are written in easy-to-read YAML files called "playbooks". Most common tasks are packaged in pre-built modules allowing for fast, efficient code (e.g. apt/yum commands, Github, Python, etc.). The Ansible configuration for this project consisted of three components: provisioning virtual machines, configuring the cluster with the needed software, and launching HDFS/Spark.

Provisioning of virtual machines was done via a Python script using the SoftLayer API. An Ansible script *start_cluster.yml* runs the Python provisioning script on the local host. The provisioning script looks to a configuration file that specifies machine configuration information. The provisioning script is robust in that machines are provisioned sequentially and the script will wait until the desired cluster is built. This is important as machines may not be available immediately. Assuming the user has Ansible installed and correctly configure the configuration files, the following command will provision a cluster of 3 SoftLayer nodes.

```
$ ansible-playbook --extra-vars '{"NUM_NODES":"3"}' -s start_cluster.yml -i
./hosts
```

The next Ansible script configures the cluster with the needed software for the project, namely HDFS and Spark. The script is encrypted to protect sensitive information. The localhost now has several files that will be piped to each node of the cluster such as */etc/hosts*, HDFS config files, and Spark config files. In addition to the HDFS and Spark set up, the script installs dependencies (e.g. SBT, NumPy, g++, etc.) and the Softlayer Object Storage training and test data sets specified in the playbook call. Upon completion the cluster will have all of the needed software for the Spark machine learning aspect of the project. The command below will download Spark, Hadoop, an image dataset (based on the size specified in the command) and install the needed dependencies; this command was encrypted as well and requires a password.

```
$ ansible-playbook --extra-vars '{"IMAGE_SIZE":"64"}' -s main.yml -i ./hosts
--ask-vault-pass
```

The final Ansible script is a playbook that starts HDFS, moves data to HDFS, and starts Spark. After the completion of this script, the user can log in to the master node which will have both HDFS and Spark running. The user can then build the project with SBT and run the logistic regression script with the desired parameters. Depending on the input parameters, a user can perform the aforementioned process in 10-20 minutes with minimal manual work and free of configuration errors. The command below starts Spark/Hadoop and moves the data to HDFS.

```
$ ansible-playbook -s hadoop.yml -i ./hosts
```

SaltStack

SaltStack is a tool for the orchestration and automation of ITOps (think data centers), CloudOps, and DevOps. For this project, we used it both to manage infrastructure in the cloud and to manage the configuration of our development environments.

The SaltStack deployment process consists of two scripts, the first to bootstrap Salt on a virtual server to serve as the “salt master”, and the second to configure and start both HDFS and Spark. The scripts were written making a few assumptions, including that the operating system wouldn’t be altered, and that the virtual servers for HDFS would have a second disk on the /dev/xvdc partition on which to run HDFS.

First, a “salt master” is created using a bash script.

```
$ ./saltcloud_init.sh -u SLxxxxxx -k … -h saltmaster -i mids
```

A SoftLayer username (-u) and API key (-k) can be supplied as command-line arguments or in ST_USER and ST_KEY environment variables. A hostname (-h) and a key identifier (-i) to use for SSH are also required. The script uses the command-line interface for the SoftLayer Python API to complete the provisioning, and then Salt is installed. During the provisioning, it is necessary to type *yes* (to add the VS to known_hosts) and enter the root password (printed to stdout after the VS is provisioned) twice. The script will error out if the provisioned virtual server has an IP that was previously added to the local known_hosts file.

Second, a cluster is configured by running the script hdfs_spark_init.v2.sh *on the salt master*.

```
$ git clone https://github.com/nathanieljblack/W251_Project.git
$ cd W251_Project/salt
$ chmod +x hdfs_spark_init.v2.sh
$ ./hdfs_spark_init.v2.sh
```

The script works by providing both the name of the master (for Hadoop NameNode and Spark Master) and a list of minions (DataNodes and Workers), along with YAML to describe the VS configuration. The cluster virtual servers are provisioned using Salt Cloud and the SoftLayer driver. There were two primary difficulties encountered with this process. The first was that the SoftLayer driver for Salt Cloud does not support provisioning disks with multiple drives. This issue was resolved by writing a simple patch that could be applied to the Salt v2015.8 codebase. The Salt codebase is very approachable, and the notion of contributing back to the open-source community was a key takeaway from this work. The other difficulty encountered was that Salt Cloud will bootstrap the develop version of Salt on all minions by default. The develop branch breaks and is repaired constantly, causing some peculiar side effects, until a script argument was passed in the cloud profile (/etc/salt/cloud.profiles.d/softlayer.conf).

After the machines are provisioned, HDFS and Spark are configured. Spark setup is very straightforward and easy. Setting up HDFS was initially attempted using a SaltStack Formula, which was pretty easy also, and allowed for extensive customizability and adaptability to different machine architectures thanks to Jinja templating. Salt Grains and Pillars are used to set up the necessary configuration files, but because this process is hard to untangle and manipulate, combining the two setups was more difficult than anticipated. Among other issues, the Hadoop formula requires that the Salt minion ID be set to the fully qualified domain name, but Spark looks for workers by hostname and SaltCloud exclusively uses hostname when provisioning servers. It is also difficult to set up multiple Hadoop clusters using the default targeting—inspecting the “roles” grain for either “hadoop_master” or “hadoop_slave” on a minion—in the Hadoop SaltStack Formula. Eventually, it was decided to forego the formula and manually maintain the configuration files necessary for HDFS. The last step of the script is to start the Hadoop NameNode and DataNode and Spark Master and Slave services. Other dependencies were not installed because this script was *not* used for managing clusters used for performance testing, but adding that additional configuration is trivial.

On the whole, after all issues were worked through, Salt became very easy to use and was robust to failure. If the script *hdfs_spark_init.v2.sh* fails due to a timeout or network issues, it can be rerun until it is successful. This script was written almost completely using Salt States, mostly as an exercise, but it also provided useful output when running the states and would simplify running multiple environments (e.g., dev, qa, prod).

Ansible vs. Salt

Because we automated provisioning using two independent tools, we were able to make a comparison of Ansible and SaltStack. In terms of ease of use, Ansible was the clear winner. Ansible's modules and simple YAML format allow users to get up and running in very little time. SaltStack outperforms Ansible in the realm of customization; while both Salt and Ansible can be used to automate virtually anything, SaltStack allows for easier customization. That being said, Ansible provides easy-to-use built-in modules for most configuration tasks (Git, shell commands, etc). Based on informal GitHub searches, SaltStack appears to have a more robust code base. Based on Google Trends, Ansible is clearly trending upward at a much faster rate compared to Chef, Puppet, and SaltStack. Both tools are written in the project's primary language, Python, so there is no advantage here. Finally, SaltStack has better SoftLayer integration.

The table below highlights some of the key areas of importance for this project and the better tool for each category.

	Ansible	**Salt**
Ease of Use	X	
Power		X
Built-in Functionality	X	
Code Base		X
Trending	X	
Python	X	X
SoftLayer Compatibility		X

Prediction - Logistic Regression

The first method of prediction was based around the Apache Spark framework. Data was stored in the Hadoop Distributed File System and machine learning was done via the MLLib library. Originally, the group planned to use PySpark as the majority of the project was coded in Python. After attempting to use MLLib, however, the group determined that the PySpark API did not support multi-class classification problems. Therefore, the classification was done in Spark's native language, Scala.

Preprocessing

Each image was preprocessed in an attempt to reduce its size. It was cropped at the edges to remove empty areas, converted to a square size, and then resized to a smaller size using the Python Imaging Library (PIL) module. The images were resized to various smaller sizes - 16x16, 32x32, 64x64, 128x128, and 256x256, in an effort to see whether increasing the size would result in a better prediction. The reduced images were then flattened into arrays, and linked to their corresponding data label for training data and their IDs for the testing data. The new datasets were stored as text files, one each for training and testing, for each size, in Object Storage. The preprocessing was done on a Spark cluster, using the PySpark module, to parallelize the mapping process.

Baseline

Prior to running the classification on Spark, the team attempted to run logistic regression on a single virtual machine using Python's Scikit-Learn machine learning package. Using SoftLayer's most powerful spot instance (64GB RAM, 16 cores), the team was able to run logistic regression in 13 hours using a reduced dataset (64x64 pixel size). Larger datasets failed due to memory errors leading us to believe that Spark was appropriate for the task at hand. In comparison, runs for the same 64x64 dataset on Spark took 2-5 minutes on average.

Setup

During the provisioning process, data is downloaded from SoftLayer's Object Storage to the local disk of the master node and sent to HDFS. When the user logs in to the master node for the first time, training and test data will be

available in the form of CSV files on HDFS. Scala code to run logistic regression will also be present in the *hadoop* user's home directory. The user needs to build the project using the Simple Build Tool (SBT) and execute *spark-submit* to run the process. The Spark cluster can be monitored at http://<MASTER IP>:8080 while the Spark job can be monitored at http://<MASTER IP>:4040.

Cluster Optimization

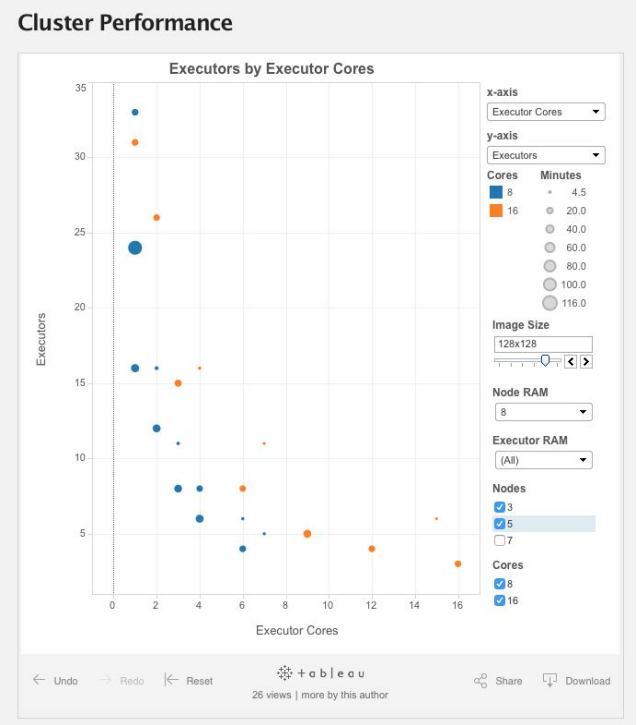
In order to answer the question - "What is the optimal configuration to classify large data?", we needed to run the prediction algorithm on various clusters with different configurations. The number of nodes used were 3, 5, or 7. The number of cores per machine was either 8 or 16. And the RAM per machine was chosen to be 8, 12, or 16 GB. This resulted in 18 combinations for each image size. After some quick proof of concept runs with the smaller images, we moved on to running the complete set for the two biggest sizes in our data - 128x128 and 256x256.

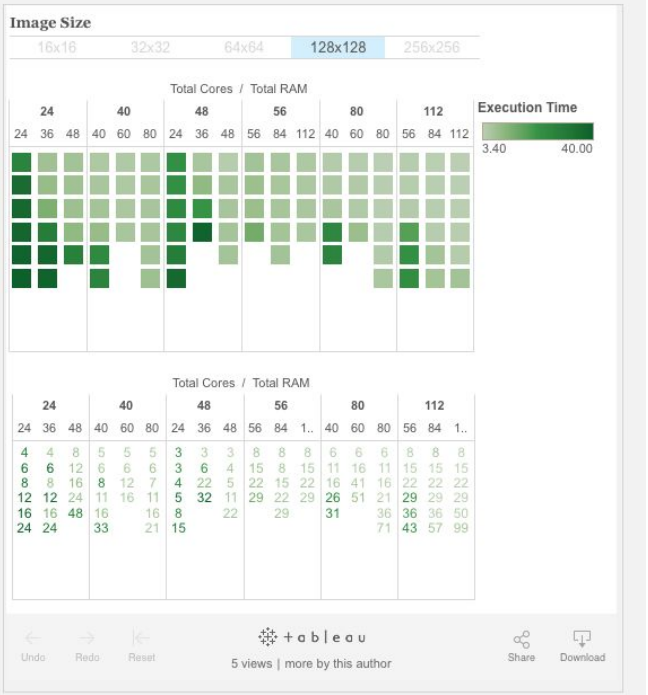
The group quickly realized the importance of maximizing the cluster resources when submitting Spark jobs. Tuning of the various Spark runtime parameters was essential. Of the various tunable parameters, we focused on number of executors, amount of executor RAM, and the number of executor cores. Tuning these parameters turned out to be more of an art than a science but several heuristics were derived based on empirical evidence. These heuristics are based on our specific algorithm (i.e. logistic regression on image arrays) and may vary with other algorithms that require more/less RAM, are more/less parallelizable, etc.

- Maximizing usage of available resources is the key to fast run times
- Spark will override configurations if resources are over-allocated
- Given N nodes, the number of executors will be a multiple of N plus an additional executor for the driver.
    - Specifying a different number of executors may result in Spark overriding the configuration and fewer executors than expected
    - E.g. in a three node cluster, specifying 10 executors will result in 10 executors while specifying 11 executors will result in 10 executors
- The total available RAM is roughly 60% of the total cluster RAM
    - The easiest way to determine the total RAM is to visit the Spark user interface
- Given N nodes with M cores each, the number of cores can be specified as:
    - The total number of cores divided by the non-driver executors rounded to the nearest integer, minus one core for the driver.

$$floor((N \cdot M)/(N-1)) - 1$$

Visualization





Results

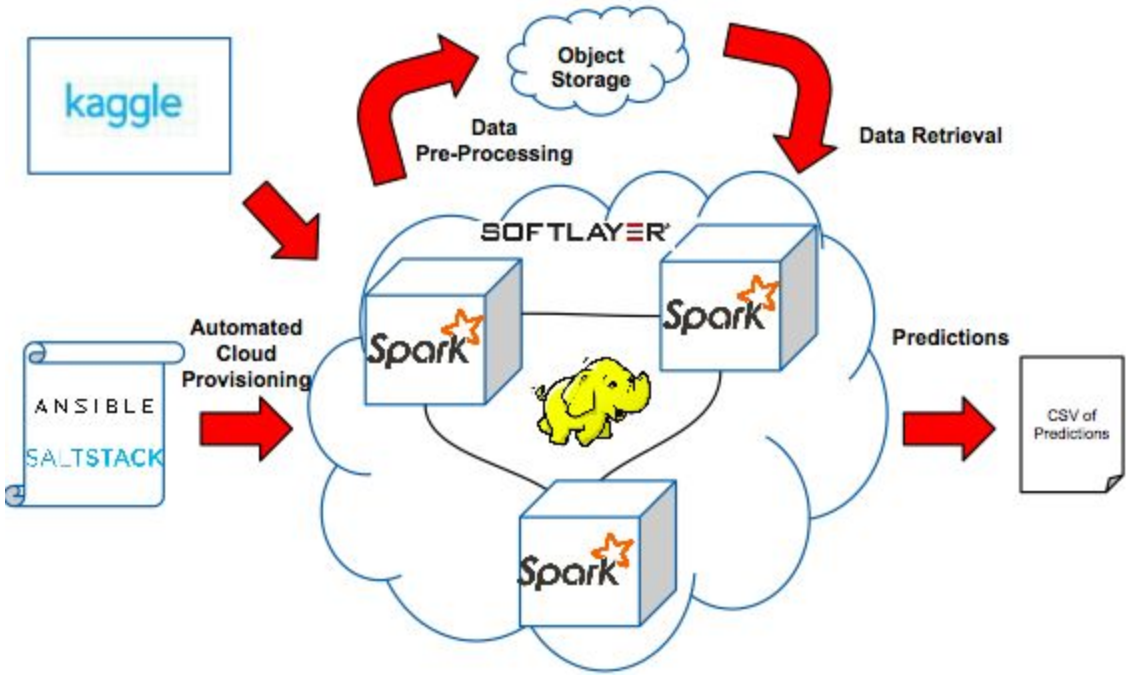
The following table is a snapshot of the complete results from our performance testing:

Image size	Nodes	Cores	RAM (GB)	Num. Executors	Executor RAM (GB)	Executor cores	Time (min)
128x128	3	8	8	4	6	6	27
128x128	3	8	8	16	1	1	40
128x128	3	16	16	3	14	15	4.3
128x128	5	8	8	5	6	7	5.5
128x128	5	8	8	33	1	1	27
128x128	7	16	12	15	4	7	3.7
256x256	3	8	8	4	6	6	150
256x256	3	8	8	6	4	4	204

The table tries to illustrate the findings that proper settings for various parameters are required for optimal execution of the algorithm. Higher number of cores and higher RAM in the machines can results in a better performance. For the same cluster configuration, the settings for the number of executors, executor RAM, etc. can have a huge influence on the run time.

One of our observations was that for the larger image (256x256), while we were able to run the algorithm successfully with a 3-node cluster, the majority of the runs for 5 and 7-node clusters resulted in an out-of-memory error. While this seems counter-intuitive, we surmise that the RDDs created are not split optimally, which results in

more shuffling of data than necessary, resulting in either a worse run time than the smaller cluster or running out of memory.



Where Can We Use This?

This framework to run a simple logistic regression on large data can be used for any classification problem where the data can be represented in CSV format, as was done for this project. For example, it could be used to predict whether a patient has a given disease (e.g. hypertension, coronary heart disease, etc.), based on observed characteristics of the patient (age, sex, body mass index, results of various blood tests, etc.). Another example of a large dataset is data related to American voters like age, income, sex, race, state of residence, votes in previous elections, etc., which can be used to classify whether a person will vote Democratic or Republican.

Prediction - Convolutional Neural Network

Deep learning and Convolutional Neural Networks (CNNs) are the state of the art for image recognition and classification. Many of the winners of the popular Kaggle image classification competition use CNNs[1]. In deep learning, “neurons” specialize to activate on certain features that arise through model training, for example edge detectors or different textures. CNNs have the added constraint that the same feature map is convolved over the entire image to detect that feature anywhere in the image. Therefore, CNNs are invariant to translations of the image [2].

Preprocessing

Preprocessing included centering and trimming blank space around the eye images, shrinking the sizes, and normalizing colors. The preprocessing of each image is independent so the process is trivially parallelizable and was conducted over four VS’s. For the purposes of quickly, getting inputs for developing a CNN, preprocessing was done by simply copying images to separate VS’s based on the mod of the image’s subject number, but a more scalable solution would include distributed file storage and mapreduce processing in Spark, which were used for the Spark solution above.

---

[1] http://benanne.github.io/2015/03/17/plankton.html, https://www.kaggle.com/c/diabetic-retinopathy-detection/forums/t/15807/team-o-o-competition-report-and-code, https://github.com/benanne/kaggle-galaxies/blob/master/doc/documentation.pdf

[2] http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf

The first step of preprocessing the images was to center the input images on the eye part of the image and trim the surrounding blank space. This was accomplished by padding each image with blank space to make a square image, then identifying the eye part of the image as the pixels that were above some threshold for black (blank). The center and radius of the part of the image that was not very close to black was calculated, and the image was trimmed to a square that was tight around this circle. The image quality varied greatly and among the images and some images were very dark, and others very light, so this threshold for identifying black blank space had to be varied some to process all of the images so as not to crop to within the eye or far from it. For future work there may be better procedures from the image processing and computer vision fields for this problem, but this approach worked well enough for this project. After trimming to a square around the eye, the images were reduced to 1024x1024 to speed up transfers and further preprocessing and saved as jpeg image files. This first step took the longest of the preprocessing steps, 12-15 hours for each VS.

The preprocessing procedure was designed to keep the image in an image format until the last steps just before the CNN input, after which the image was stored as an array. This was not the case in earlier versions of preprocessing and there was information loss in converting back and forth between arrays and image formats. Saving a slightly larger image than we planned to use for training the model, gave us flexibility with later processing without having to redo these initial steps.

Data augmentation is often successfully used to improve CNN performance, by increasing the amount of data to train on with synthetic data. We used a few simple transformations for creating synthetic data for this project. Three rotations of the images were created and saved, 90-, 180-, and 270-degrees. The images were then reversed and rotations of the mirror images were saved as well. These transformations increased our training data set size to 8x the original amount. Future work could include other transformations such as translations and scaling. These transformations were also saved as 1024x1024 JPEGs.

The last preprocessing steps were to convert the images to arrays, normalize the colors, and reduce the sizes further for use in the model. The color normalization was done by converting each channel (red, green, and blue) of each image to zero mean and unit variance. A relatively small image size, 128x128, was used for developing and training the CNN model. Because training on the GPU was relatively quick, results may be improved later by increasing the input image size.

Training and Testing Data

The data split into training and testing sets by subject number, with 20% of subjects randomly chosen for the test set. All eye images associated with a subject would be placed in the training or testing set, including left/right images and synthetic data generated from those images, so that there was no leakage of testing data into the training set.

CNN

Training a CNN that performed well on this data proved using Theano proved to be a very difficult task. We began with a relatively small network that worked well on the MNIST handwritten digits data set, with all of the training data as 64x64 images and the preprocessing procedure used above in the Spark experiments with an additional conversion to grayscale. The results were all zeros, which is the largest category by a lot. Thereafter, many ideas were explored to improve the model, and results were still all zeros, over and over and over again. A logistic regression model using Theano, ran in just a few minutes over all of the data, and also resulted in all zeros. Eventually, we switched to using nolearn and Lasagne, which are Python packages on top of Theano that make it much easier to build and train a CNN model, and the first attempt worked. We finally had a model that was clearly learning from the data.

The imbalance of the categories was one major problem for training the model. To address this problem we subsampled from the data to get a more even distribution. When this did not fix the problem of predicting all one category, we simplified further to using only images with a zero label, eyes without signs of diabetic retinopathy, and

those with a label of four, the most diseased eyes. This turned the problem into a binary classification problem, healthy or diseased. The preprocessing procedure was also improved and RGB colors were used instead of grayscale. While subsampling simplified the problem and should have made classification easier, it also greatly reduced the size of the data set from 35,216 images to 1,416 images because there are only 708 level four images in the training set. Even with data augmentation, that left us with just only 11,328 images.

One of the consequences of less data, is that the initial values of the CNN parameters are more important. There are many different approaches for parameter initialization for deep learning, some are quite involved, such as pretraining of the network. This parameter initialization is likely the reason that our Theano CNN did not learn well, and converged extremely quickly to predicting only one label. In nolearn, parameters are initialized from a uniform distribution within a range that is chosen to lead to convergence quickly[3]. With enough data and computation time, however, initialization becomes less important.

Deep learning has experienced a resurgence in recent years, in large part, due to being able to perform the needed computations very quickly on GPUs, so we trained our CNNs on AWS EC2 instances with GPUs. We used AWS instead of Softlayer because hourly billing is available on AWS, whereas GPU VS's are only available with monthly billing on Softlayer and the cost of using these machines is relatively high. For a relatively small four-layer CNN in Theano running on all of the data, we experienced a roughly 6x speed-up from running on a GPU over a CPU, from about one hour per training epoch to about 10 minutes.

The GPU has less available memory than the CPU however, and we did experience out of memory errors. Theano handles a separate memory space from the rest of the Python code and sends data to the GPU only when needed there. These errors primarily occurred when calculating predictions for the test data because all of the test data was being sent to the GPU at once. Training data was only loaded onto the GPU in the mini-batch sizes for the stochastic gradient descent. Therefore, we could control these errors by shrinking the batch sizes and splitting up the test data for predictions. These operations are handled by nolearn so we did not run into memory errors using the Theano extensions. With enough data, however, we would not be able to provision a machine large enough to load the data into the CPU. In this case, we could simply load a chunk of data and proceed through an epoch of stochastic gradient descent, then load the next chunk for the next epoch, so that we proceed through all of the data in these mini-epochs. We did not need to do that for this problem with the 128x128 input images.

The final network architecture for our nolearn CNN implementation is shown below:

Layer	Type	Size
0	Input	3 x 128 x 128
1	Convolution, 3x3 filter	32 x 126 x 126
2	Convolution, 3x3 filter	32 x 124 x 124
3	Convolution, 3x3 filter	32 x 122 x 122
4	Max Pool, 2x2	32 x 61 x 61
5	Convolution, 3x3 filter	64 x 59 x 59
6	Convolution, 3x3 filter	64 x 57 x 57
7	Max Pool, 2x2	64 x 29 x 29
8	Convolution, 3x3 filter	128 x 27 x 27
9	Convolution, 3x3 filter	128 x 25 x 25

[3] http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/

10	Max Pool, 2x2	128 x 13 x 13
11	Fully-connected dense	600
12	Dropout	----
13	Fully-connected dense	600
14	Output	2

This CNN took just over two minutes per epoch and ran for 410 epochs for a total time of 15 hours. Test set accuracy was relatively flat after only 60 epochs, however, while training loss continued to decrease. Test accuracy did not show a decrease, suggesting that we may not have been severely overfitting and could potentially increase the network size with more data. After training the network, we can save the fitted parameters and quickly make predictions on new images.

Perhaps the biggest lesson learned from this effort was to use the tools that are designed to make deep learning easier. Theano has a lot of flexibility but requires everything to be provided, such as the shape of weights arrays and the update functions. Lasagne and nolearn, as well as other libraries, include functionality that will specify these for you and greatly increases development time.

In future work, we would want to increase the size of the data used for training the network and would, therefore, need to adapt the methods to that increased data size. Additional data augmentation including scaling, more rotations, variations in color saturation, and small translations, could increase network performance. With this additional data, we would want to store the data into a distributed file system. Also, instead of subsampling to deal with the imbalance of categories, we could weight examples from different classes differently in the loss calculation so that the model could learn from the additional data that we have on healthy eyes. We would also like to improve the model by predicting ordinal instead of categorial labels[4]. Finally while the GPU was the best approach for this problem, there is growing interest in distributing deep neural networks over clusters of CPUs, such as with DistBelief, and could be a potential area of future work.

DeepLearning4J

DeepLearning4J is a much-hyped Java library for deep learning. The library allows for deep learning frameworks to be built on top of modern big data tools such as Hadoop and Spark while using GPUs on the backend for computation. Currently, other popular deep learning libraries such as Caffe, Torch, and Theano are not built on the Java Virtual Machine (JVM) which makes Hadoop and Spark integration (Hadoop and Spark are built on JVM) difficult. DeepLearning4J has much promise but the project is still very young. During the course of the project, the key functionality for image classification (convolutional and recurrent networks) was not working. The team was able to successfully implement DeepLearning4J on top of Spark but the functionality was extremely limited so the group abandoned the library in favor of more mature libraries such as MLLib and Theano. As DeepLearning4J matures, it will be very interesting to see how complex neural networks can be implemented using both the power of Spark as well as the speed of GPU computation.

Tools

The following list of tools were used as part of the project.

-Softlayer
-Python
-PySpark
-AWS
-Scala
-SBT

[4] https://web.missouri.edu/~zwyw6/files/rank.pdf

-Salt	-Ansible
-Spark	-MLLib
-NumPy	-Pandas
-PIL/Pillow	-Scikit-Image
-Scikit-Learn	-Theano
-nolearn	-HDFS
-Tableau	-Object Storage
-DeepLearning4J	-Lasagne

Issues

As with most projects, the team encountered numerous issues during the course of the semester. The following list highlights some of the project challenges.

- Downloading the data files was not straightforward, as the data was split into multiple files, and needed to be concatenated into one big zip file before it could be unzipped.
- Figuring out the various aspects of auto-provisioning was too time consuming. This resulted in a lot of time spent on provisioning and bootstrapping the clusters manually.
- MLlib on PySpark does not support multi-class logistic regression.
- The classification of eye images is a difficult problem.
- Inefficient allocation of Spark resources - how to maximize the use of the resources?
- Downloading larger data, which was segmented during storage, using the Object Store URL results in an error. This prevents the generic use of ‘wget’ to download the data.
- Neural network
    - Losing information through image format conversions during first preprocessing implementation
    - Difficulty using Theano without extensions that facilitate easier network building and training
    - Memory management on a single machine in order to use GPUs
- DeepLearning4J library was too unstable. Convolutional and Recurrent Networks were not functional during the project.

Future Work

Future work may include improving prediction accuracy via new algorithms and/or improved preprocessing, more efficient tuning of the Spark runtime parameters, better partitioning of the Spark data, and possibly using a distributed deep learning framework such as DeepLearning4J or DistBelief. Undoubtedly, the team will rely on the real-world experience gained from this project. Automated provisioning, HDFS, Spark, Theano, and deep learning are all skills that can and will be translated to other domains as we continue our careers in data science.

Github location (private)

https://github.com/nathanieljblack/W251_Project.git