AI-assisted Radiology Using Distributed Deep Learning on Apache Spark and Analytics Zoo

April 2019

H17686

White Paper

Abstract

This white paper describes building a distributed deep neural network with Apache Spark and Analytics Zoo to predict diseases from chest x-rays.

Dell EMC Solutions

Copyright

Contents

Executive summary

The health care industry is poised to reap the benefits of AI and deep learning to improve patient outcomes, reduce costs, and expediate diagnoses. Consequently, we have developed a deep learning model to predict pneumonia, emphysema, and other diseases from chest x-rays. Using the Stanford University CheXNet model as inspiration, we explore methods of developing an accurate model on a distributed Apache Spark cluster.

The model is built by using BigDL, a distributed deep learning library for Apache Spark and Analytics Zoo, a unified analytics and AI platform that seamlessly unites Spark, TensorFlow, Keras, and BigDL programs into an integrated pipeline. The solution is implemented on Dell EMC Ready Solution for AI: Machine Learning with Hadoop.

Using Analytics Zoo APIs, we built an integrated Spark ML pipeline that incorporates reading images as Spark Dataframes, feature engineering, transfer learning, and neural network training. We provide insights and observations about Spark parameters and model hyperparameter tuning, including optimizers and batch size, which lead to achieving over 80 percent average AUC for the 14 diseases in the x-rays.

Finally, we evaluate and present the results about how our model scales across an Apache Spark cluster that is powered by Dell EMC PowerEdge servers and Intel Xeon Scalable processors. By using a transfer learning approach, we can accurately train the model in approximately 2.5 hours on a 16-node Spark cluster. We show that we can achieve three times speedup scaling from four nodes to 16 nodes.

Document purpose

This white paper describes how we build a distributed deep neural network with Apache Spark and Analytics Zoo to predict diseases from chest X-rays.

We value your feedback

Dell EMC and the authors of this document welcome your feedback on the solution and the solution documentation. Contact the Dell EMC Solutions team by email or provide your comments by completing our documentation survey.

Authors: Bala Chandrasekaran (Dell EMC), Yuhao Yang (Intel), Sajan Govindan (Intel), Mehmood Abd (Dell EMC)

Acknowledgements: We want to acknowledge the following contributors to this study: Michael Bennett, Jenwei Hsieh, Phil Hummel, Andrew Kipp, Dharmesh Patel, Leela Uppuluri, Luke Wilson, and Penelope Howe-Mailly.

Note: The AI Info Hub for Ready Solutions on the Dell EMC Communities website provides links to additional documentation for this solution.

Introduction

Artificial intelligence is expected to revolutionize many industries. The health care industry is poised to realize the early benefits of AI for early detection of diseases, diagnosis, decision making, and treatment. Deep learning is the practice of training and deploying artificial neural networks models on various datasets that include images, videos, speech, and structured and unstructured text data. It is expected to be adopted widely as it can provide better prediction from massive amounts of data while automatically comprehending feature extraction.

Data scientists still spend an inordinate amount of time on data wrangling – the process of selecting and transforming raw data into a format for analysis and prediction. In today's enterprises, much of the data storage and data wrangling occur on big data systems that are running Hadoop and Spark ecosystem solutions. Having an integrated deep learning pipeline and framework on Spark significantly reduces the model development time. It also eliminates the complexities of operating a separate deep learning cluster and the need to migrate training data to it.

Analytics Zoo[1] is an open-source framework that unifies analytics and AI, integrating a deep learning framework on Apache Spark. It provides code samples, pretrained models, and reference use cases that can jump start any project seeking to unite Spark, TensorFlow, Keras, and BigDL[2] programs into an integrated pipeline. By using this tool set, data scientists can develop, train, tune hyperparameters, and deploy deep learning models. Existing Hadoop and Spark compute clusters or worker nodes can now be used for distributed training and inference.

In this white paper, we demonstrate how to build an integrated ML pipeline on Apache Spark to develop a deep neural model to predict diseases from chest X-rays using Analytics Zoo. The open-source software that is described in this white paper is available in GitHub at:

https://github.com/dell-ai-engineering/BigDL-ImageProcessing-Examples

Detecting diseases in chest x-rays

We used the chest x-ray dataset that was released by the National Institutes of Health (NIH)[3] of the United States to develop an AI model to diagnose pneumonia, emphysema, and other thoracic pathologies from chest x-rays. Based on the Stanford University CheXNet project[4], we explore ways to develop accurate models on a distributed Spark cluster. We explore various neural network topologies and hyperparameter tunings to gain insight into what types of models provide better accuracy and reduce training time.

The dataset contains more than 120,000 images of frontal chest x-rays, each potentially labeled with one or more of 14 different thoracic pathologies. The following table and figure list the diseases and the number of occurrences in the dataset.

Table 1. Occurrence of diseases in the ChestXray14 dataset

Disease	Images	Percentage
Atelectasis	11535	10.28
Consolidation	4667	4.16
Infiltration	19871	17.72
Pneumothorax	5298	4.72
Edema	2303	2.05
Emphysema	2516	2.24
Fibrosis	1686	1.5
Effusion	2516	2.24
Pneumonia	1353	1.2
Pleural Thickening	3385	3.01
Cardiomegaly	2772	2.47
Nodule	6323	5.64
Mass	5746	5.12
Hernia	227	0.2
No Findings	60412	53.88
Totals	112,120	100

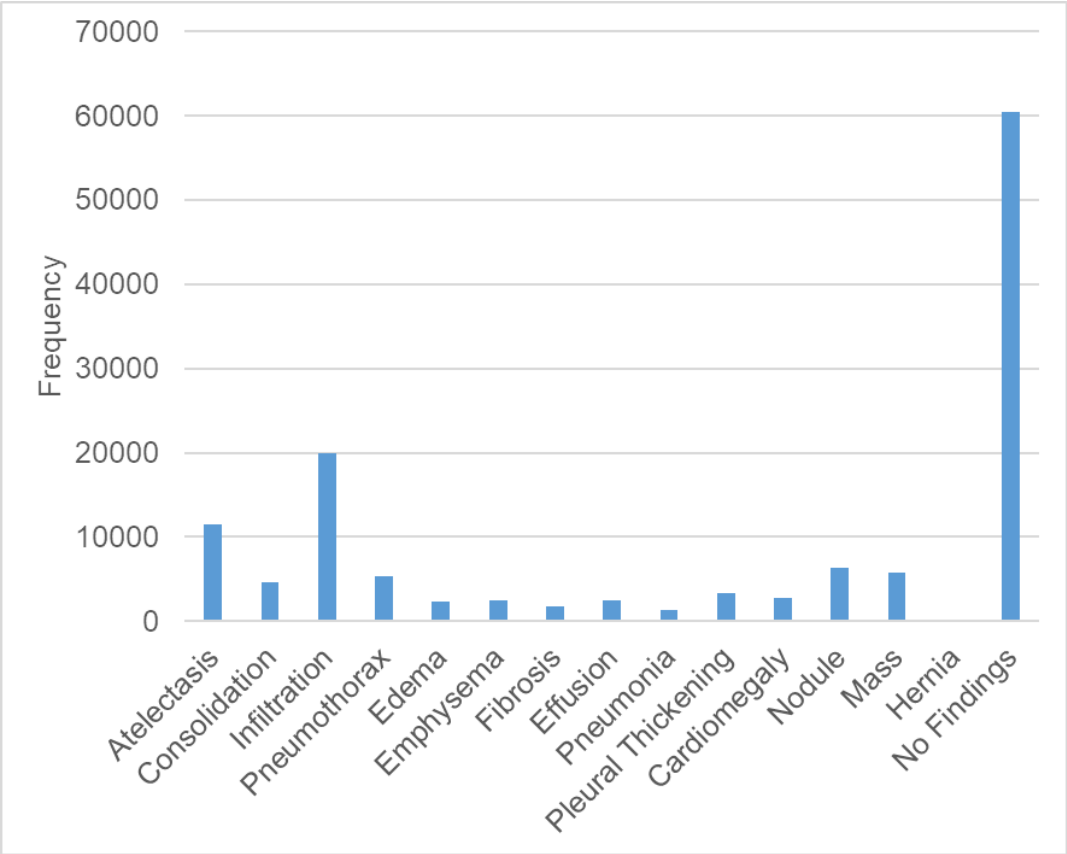

Figure 1. Occurrence of diseases in the ChestXray14 dataset

As the figure shows, the dataset is also unbalanced, with no findings for more than half of the dataset images. Also, a single chest x-ray image can indicate more than one disease. This requirement causes a multiclass, multilabel classification problem. Our goal is to develop a classification model that can predict the probability of diseases for a single, specific chest x-ray image. The model also must be able to correctly predict multiple diseases if they are present in the chest x-rays.

This deep learning model experimentation was developed and implemented on Dell EMC Ready Solution for AI: Machine Learning with Hadoop. The following sections provide an overview of the Ready Solution, an overview of Analytics Zoo, a description of the methodology that we used to develop the model, and observations and results.

Solution stack

Ready Solutions for AI: Machine Learning with Hadoop

The Ready Solution for Artificial Intelligence: Machine Learning with Hadoop architecture[5] addresses all aspects of running machine learning jobs on a production Cloudera Enterprise cluster. It addresses the physical server hardware, the network fabric, the software environment, and the interfaces between the core cluster and the machine learning environment.

The following figure shows the primary components in this Ready Solution.

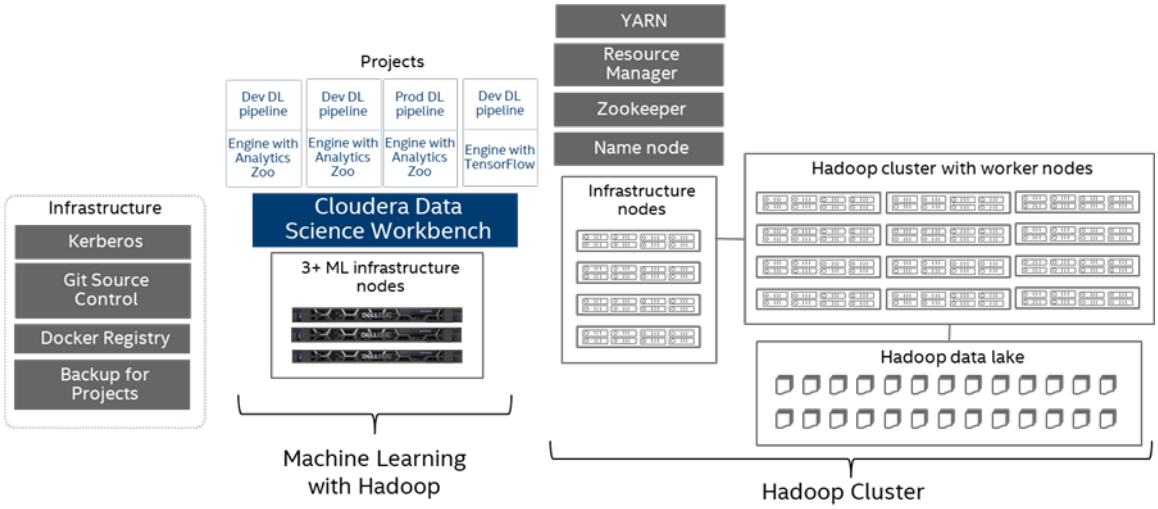

Figure 2. Primary components of the Ready Solution

The components include:

- **Cloudera Enterprise Hadoop Cluster**— Cloudera Enterprise Hadoop cluster powered by Dell EMC hardware infrastructure.
- **Apache Spark**—A unified analytics engine for large-scale data processing, which is a key component of Cloudera Hadoop. It runs on the cluster worker nodes and provides libraries that include SQL and DataFrames for data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.
- **Cloudera Data Science Workbench**—A multiuser platform for data scientists to develop and collaborate. Cloudera Data Science Workbench organizes data science activities into projects and provides isolated, containerized environments called engines for interactive sessions and production workflows. Cloudera Data Science Workbench is fully integrated into the security, management, and operations infrastructure of Cloudera Enterprise.
- **Cloudera Data Science Workbench Nodes**—Specialized edge nodes deployed on Dell EMC PowerEdge servers that run the data science workbench application and the supporting Kubernetes and Docker infrastructure. They also provide storage for projects and local data. These nodes have direct connections to both the cluster data network and the corporate network.
- **BigDL and Intel Analytics Zoo**—BigDL is an open-source distributed deep learning framework for Spark. Analytics Zoo simplifies building distributed deep learning applications on Spark that are based on TensorFlow, Keras, and BigDL by providing an end-to-end analytics and AI platform. An overview BigDL and Analytics Zoo are provided in the following sections.

- **Dell EMC Preconfigured Engines**[6]—Containerized environments with ready-to-run projects including examples that are based on common use cases and implemented with BigDL and Spark. These engines contain the necessary binaries, configuration, and libraries to use BigDL and Analytics Zoo with Cloudera Data Science Workbench.

In this study, we use 16 PowerEdge R740xd servers that are configured with Intel Xeon Platinum 8160 CPU @ 2.10GHz as a Cloudera Hadoop and Spark cluster. The chest x-ray images are stored in HDFS, which is distributed across the cluster. Model training is initiated on the Spark cluster by creating a project Cloudera Data Science Workbench and using the preconfigured engines mentioned previously.

Analytics Zoo

Analytics Zoo provides a unified analytics and AI platform that seamlessly unites Spark, TensorFlow, Keras, and BigDL programs into an integrated pipeline. The entire pipeline can then transparently scale out to a large Hadoop/Spark cluster for distributed training or inference. Key features of Analytics Zoo include:

- Data wrangling and analysis by using PySpark
- Deep learning model development by using TensorFlow or Keras
- Distributed TensorFlow, Keras, and BigDL training/inference on Spark
- High-level pipeline APIs with native support for Spark Dataframe, ML pipelines and transfer learning, and model serving APIs for inference pipelines

In addition, Analytics Zoo also provides a rich set of analytics and AI support for the end-to-end pipeline, including:

- Easy-to-use abstractions and APIs (for example, transfer learning support, autograd operations, Spark DataFrame and ML pipeline support, online model serving API, and so on.)
- Common feature engineering operations (for image, text, 3D image, time series, speech, and so on)
- Integrated deep learning models (for example, object detection, image classification, text classification, recommendation[7], anomaly detection, text matching, sequence-to-sequence, and so on)
- Reference use cases (for example, anomaly detection, sentiment analysis, fraud detection, image similarity, chatbot, and so on)

BigDL

BigDL is a distributed deep learning library for Spark. By using BigDL, users can write their deep learning applications as standard Spark programs, which can run directly on existing Spark or Hadoop clusters. BigDL takes advantage of Spark's distributed in-memory compute engine to enable efficient scale-out data analytics and deep learning workloads. It achieves high performance on the Intel Xeon processors in the Hadoop cluster by using Intel Math Kernel Library (Intel MKL) and multithreaded programming in each Spark task.

BigDL enables distributed training of a neural network. It is built from the ground up on Spark, thereby inheriting fault tolerance and other architectural benefits of Spark. To

enable efficient distributed training, BigDL uses a data-parallel approach to training with synchronous minibatch Stochastic Gradient Descent (SGD). Training data is partitioned into RDD samples, which are automatically partitioned and distributed to each worker. In addition, BigDL also constructs an RDD of models, each of which is a replica of the original neural network model. The model and sample RDDs are copartitioned and colocated across the cluster. Model training is performed in an iterative process. It first computes gradients locally on each worker by taking advantage of locally stored partitions of the training data and model to perform in-memory transformations. Then, an `AllReduce` function schedules workers with tasks to calculate and update weights. Finally, a broadcast synchronizes the distributed copies of the model with updated weights. The following figure shows the training approach.

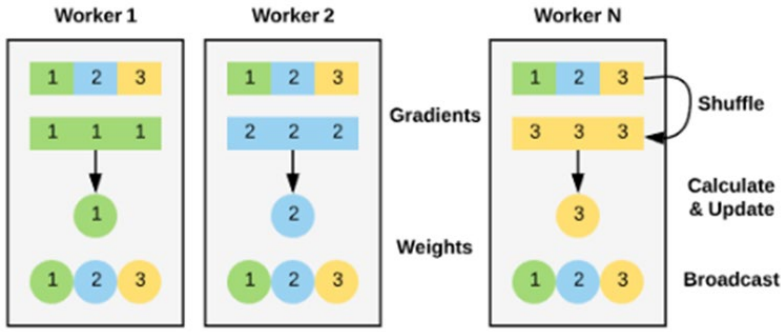

Figure 3. Distributed training in BigDL

A key criterion is that the number of data samples in the minibatch (batch size) must be a multiple of the number of Spark executors that are multiplied by the number of cores in each executor. This value ensures that the local gradients are computed on the same number of sample datasets by each worker.

Integration with Spark ML pipelines

Spark enables ML pipelines[8], which provide a uniform set of high-level APIs built on Spark DataFrames that help users create and tune machine learning pipelines. ML pipelines enable users to chain together multiple transformers (data manipulations) and estimators (models) to specify an ML workflow. A benefit of this workflow is that all transformers and estimators share a common API for specifying parameters. For more information about Spark ML pipelines, including Transformers and Estimators go to Spark ML Pipelines. The following figure from the Spark documentation shows an example ML pipeline for a Logistic Regression model.

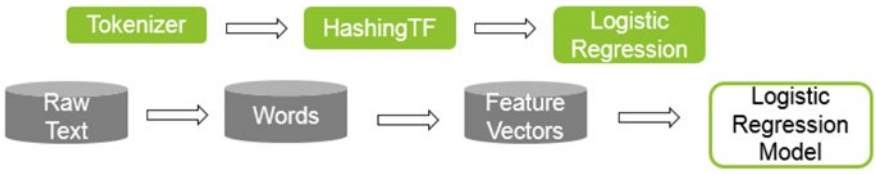

Figure 4. Spark ML Pipeline: `Pipeline.fit()` method

Analytics Zoo uses Spark ML pipeline concepts for deep learning and artificial neural networks. NNFrames is an Analytics Zoo package that provides high-level APIs that are built on Spark DataFrames to help Spark users create neural network pipelines. It supports native integration with Spark ML Pipelines, which enables users to combine the power of Analytics Zoo, BigDL, and Apache Spark MLlib. Key concepts of NNFrames and Analytics Zoo include:

- **PreProcessing and Chained PreProcessing**—Preprocessing defines data transforms during feature preprocessing. Multiple Preprocessing can be combined into a `ChainedPreprocessing` operation. For example, image rotation is a preprocessing transformation.
- **NNEstimator**—`NNEstimator` extends the Spark ML Pipeline `Estimator` and supports training a BigDL model with Spark DataFrame data. It can be integrated into a standard Spark ML Pipeline to enable users to combine the components of BigDL and Spark MLlib.
- **NNModel**—`NNModel` extends Spark's ML Transformer. Invoking a `.fit()` method in `NNEstimator` yields an `NNModel`. A pretrained BigDL Model can be loaded into a `NNModel` and used as a transformer in a Spark ML pipeline to perform transfer learning with new `DataFrames` or predict the results for a `DataFrame`.
- **NNClassifier and NNClassifierModel**—`NNClassifier` is a specialized `NNEstimator` that simplifies the data format for classification problems. Invoking a `.fit()` method in `NNClassifier` yields a `NNClassifierModel`, as shown in the following figure.

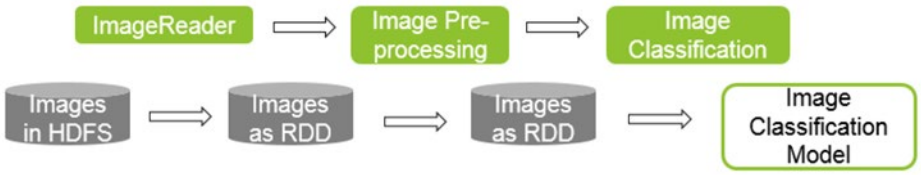

Figure 5. Deep learning pipeline with Analytics Zoo: `Classifier.fit()` method

In Model development and training, we demonstrate how we use the Analytics Zoo API to develop our model. We use the feature engineering APIs that are available in Analytics Zoo to process chest x-ray images and `NNEstimator` to build our model. We train the neural network to produce `NNModel`, which predicts the diseases.

Transfer learning

Transfer learning[9] is a machine learning method in which a model that is trained for a task is reused as the starting point for a model on a second related task. In deep learning, a neural network that is trained on a dataset is used as a starting point to train a new dataset. For example, a neural network that is trained to identify vehicles can be used as a starting point to identify specific vehicle models.

Analytics Zoo provides high-level APIs that can be used to apply transfer learning methodologies to train a neural network. The capabilities include:

- **Loading an existing pretrained model**—The `Net.load` API can be used to load existing models for BigDL, Torch, TensorFlow, and Caffe.

- **Remove the last few layers in the neural network**—The `newGraph` API can be used to remove the last few layers in the pretrained network. This API is typically used to remove the output layers. For example, output layers that are designed for the previous classification task must be removed as they are no longer applicable.
- **Adding new layers**—The `to_keras` API can be used to add new layers for the new task. For example, new output layers that are designed for the new classification task can be added.
- **Freeze the first few layers**—The `freeze_up_to` API can be used to freeze the first few layers. This API is typically used to preserve some of the training in the previous task.

We use transfer learning to develop our model for predicting diseases from chest x-rays. The following section shows how we use these APIs to load and configure a pretrained model to train with the chest x-ray dataset.

Model development and training

This section describes how we developed and trained the deep learning model for detecting diseases in chest X-rays by using Analytics Zoo API. The steps for developing the deep learning model include:

- Reading the chest x-ray images—Reading images from Hadoop Distributed File System (HDFS) as a Spark Dataframe
- Feature Engineering—Processing the chest x-ray images to improve the prediction of the model and decrease the training time
- Defining the model—Loading and configuring a pretrained model
- Defining the optimizer— Defining the Adam optimizer for use with the learning rate scheduler
- Building the model—Building the `NNModel` by using `NNEstimator`
- Measuring the accuracy—Measuring accuracy by using AUC-ROC

Reading the chest x-ray images

We use `NNImageReader` to read the images from HDFS and convert them to a Spark DataFrame. The images in the chest x-ray dataset are 1024 x1024 pixels and use three channels (RGB). We use one hot encoding to represent the 14 labels in the dataset. We encode the 14 diseases in the array into 14 binary elements. When the images are converted to Spark DataFrames, it is easy to perform image preprocessing operations. The operations are now transformations on the RDDs and Spark implements them on the distributed compute cluster.

Feature Engineering

We perform the following preprocessing tasks on the images by using the `ChainedPreprocessing` and other feature engineering APIs in Analytics Zoo:

- Resize the images to 224 x 224 pixels. As noted previously, the original images are 1024 x 1024 pixels. We find that reducing the image size has little impact on accuracy while significantly improving the training time[10].
- Horizontally flip half the images, which are chosen randomly.

- Adjust the brightness for half the images, which are chosen randomly.
- Normalize the images by subtracting the mean of ImageNet dataset.

The following code snippet shows the image prepreocessing operations:

```
transformer = ChainedPreprocessing( RowToImageFeature(),
                  ImageCenterCrop(224, 224),
                  ImageRandomPreprocessing(ImageHFlip(), 0.5),
                  ImageRandomPreprocessing(ImageBrightness(0.0,
                  32.0), 0.5),
                  ImageChannelNormalize(123.68, 116.779, 103.939),
                  ImageMatToTensor(), ImageFeatureToTensor()])
```

Defining the Model

As with the Stanford approach, we use transfer learning to train our model. We use a pretrained ResNet-50 architecture, which has been previously trained on the ImageNet[11] dataset. The pretrained model provides a better-than-random starting point for training to identify features in chest x-ray images. It reduces the number of epochs of training that is required to converge to a functional model.

The following code snippet shows loading the pretrained model:

```
def get_resnet_model(model_path, label_length):
    full_model = Net.load_bigdl(model_path)
    model = full_model.new_graph(["pool5"])
    inputNode = Input(name="input", shape=(3, 224, 224))
    resnet = model.to_keras()(inputNode)
    flatten = GlobalAveragePooling2D(dim_ordering='th')(resnet)
    dropout = Dropout(0.2)(flatten)
    logits = Dense(label_length, W_regularizer=L2Regularizer(1e-
1), b_regularizer=L2Regularizer(1e-1), activation="sigmoid")
(dropout)
    lrModel = Model(inputNode, logits)
    return lrModel
```

Using Analytics Zoo APIs, we remove the final layer from the pretrained ResNet-50 model. This layer is a softmax layer that is specific to the ImageNet classification. We add a new last layer to predict the 14 diseases. We use Sigmoid for the activation function and a dropout rate of 0.25. To avoid overfitting, we use L2 regularization (Lasso Regression) for both the input weight and the bias. We also add a new input layer for the resized chest x-rays images (224 x 244 pixels with three channels).

We also evaluated the performance of Inception and DenseNet topology. For the results, see Model optimization and results.

Defining the optimizer

We use Adam optimizer[12] with the learning rate scheduler. The learning rate scheduler is implemented in two phases. First, we gradually increase the learning rate (warm up) for some epochs and the maximum learning rate. Then, we cool down the learning rate until we reach the required accuracy. For the results, see Model optimization and results.

The following code snippet shows how we defined the optimizer:

```
def get_adam_optimMethod(num_epoch, trainingCount, batchSize):
    iterationPerEpoch = int(ceil(float(trainingCount)batchSize))
    warmupEpoch = 5
    warmup_iteration = warmupEpoch * iterationPerEpoch
    initlr = 1e-7
    maxlr = 0.0001
    warmupDelta = (maxlr - initlr) / warmup_iteration
    cooldownIteration = (num_epoch - warmupEpoch) *
      iterationPerEpoch
    lrSchedule = SequentialSchedule(iterationPerEpoch)
    lrSchedule.add(Warmup(warmupDelta), warmup_iteration)
    lrSchedule.add(Plateau("Loss", factor=0.1, patience=1,
      mode="min", epsilon=0.01, cooldown=0, min_lr=1e-15 ),
      cooldownIteration)
    optim = Adam(lr=initlr, schedule=lrSchedule)
    return optim
```

Building the model with NNEstimator

We use the Analytics Zoo `NNEstimator` API to build the model. `BinaryCrossEntropy` is used as the loss function. By calling the `.fit()` method, we train the model. `NNEstimator` extracts feature and label data from the input DataFrame and uses the `ChainedPreprocessing` API to convert data for the model, including converting the feature and label to tensors. The model is then trained.

The following code snippet shows how we build the model:

```
estimator = NNEstimator(xray_model, BinaryCrossEntropy(),
transformer)
            .setBatchSize(batch_size)
            .setMaxEpoch(num_epoch)
            .setFeaturesCol("image")
            .setCachingSample(False)
            .setValidation(EveryEpoch(), validationDF, [AUC()],
            batch_size)
            .setTrainSummary(train_summary)
            .setValidationSummary(val_summary)
            .setOptimMethod(optim_method)

nnModel = estimator.fit(trainingDF)
```

Measuring the accuracy of the model using AUC-ROC

We use area under the curve (AUC) Receiver Operating Characteristics (RoC) to measure the accuracy of the model. We use the Spark ML pipeline `BinaryClassificationEvaluator` API to determine the AUC-ROC for each disease. We also calculate the average AUC-ROC for all 14 diseases. These values determine the accuracy of the model.

Model optimizations, recommendations, and results

This section presents the results of our model training, how our model scales, and recommendations that are based on our observations from tuning the model.

Spark parameters

The following Spark parameters must be determined before training the model:

- **Number of executors and cores**—The number of executors and cores depends on the number of compute resources that are available for training. We recommend one Spark executor per available compute node. The number of executor cores determines the number of Spark tasks that can run in parallel. If the entire worker node is dedicated for training, we recommend that the number of cores per executor equals the number of physical cores in that server minus two (two cores can be allocated for Spark overhead).
- **Executor memory**—The following formula can determine the memory that is allocated to an executor:

  Executor memory = Number of cores per executor * (Memory required for each core + data partition)

  The memory that the executors use depends on the size of the dataset and size of the neural network model. In our chest x-ray model training using ResNet-50, we used 300 GB memory for each executor.

Neural Network Topology

We evaluated three topologies to train our model: ResNet-50, DenseNet, and Inception. We trained the model on four executors for 15 epochs. We used the Adam optimizer with the learning rate scheduler for all three topologies.

The following table and figure shows the results.

Table 2. Results on different topologies

Disease	ResNet-50	DenseNet	Inception
Atelectasis	0.789	0.764	0.770
Consolidation	0.873	0.915	0.844
Infiltration	0.865	0.826	0.855
Pneumothorax	0.689	0.683	0.688
Edema	0.807	0.778	0.764
Emphysema	0.729	0.762	0.684
Fibrosis	0.731	0.592	0.696
Effusion	0.870	0.753	0.841
Pneumonia	0.794	0.696	0.775
Pleural Thickening	0.886	0.867	0.865
Cardiomegaly	0.887	0.796	0.825
Nodule	0.762	0.722	0.725

Disease	ResNet-50	DenseNet	Inception
Mass	0.768	0.709	0.738
Hernia	0.759	0.796	0.565
Average AUC	0.801	0.761	0.760

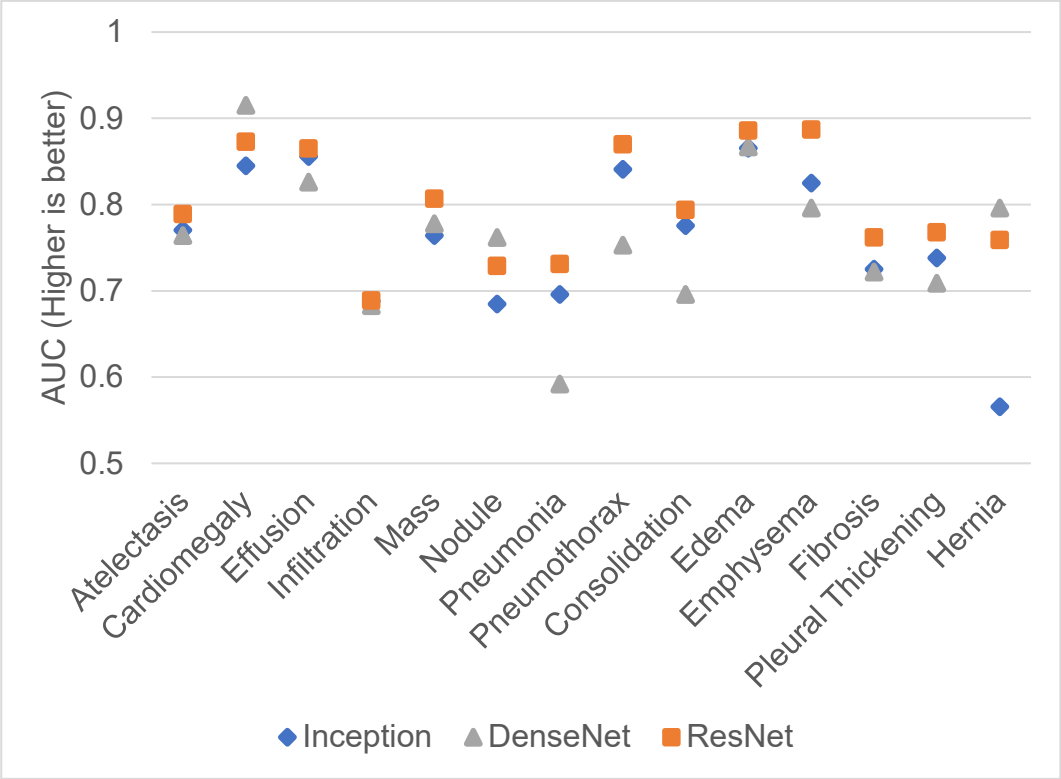

Figure 6. Results on different topologies

As noted, ResNet-50 yields the best results. Other experiments that are conducted on a distrusted TensorFlow framework and found that ResNet-50 scales effectively on distributed training when compared to DenseNet[13] and Inception. DenseNet consists of many repeating blocks of convolutional and batch normalization layers. The prevalence of batch normalization layers might be the limiting factor in scaling DenseNet topology for large-scale distributed training. ResNet-50 slightly outperforms Inception because we believe that Inception is impacted by vanishing gradient[14].

We chose ResNet-50, pretrained with an ImageNet dataset, as the primary model for our training and to determine how well our model scales.

Optimizers and the learning rate scheduler

We investigated two optimizers with adaptive learning rates: SGD and Adam. SGD with minibatch gradient decent is a simple optimizer that computes the gradient for every batch. The Adam optimizer improves on SGD by adding several decay components. In our case, we found that the Adam optimizer converges faster and provides the best model in seven to eight epochs, while SGD does not yield as much accuracy, even with 50 or more epochs.

The following figure shows the impact on AUC by learning rate scheduler for Adam optimizer.

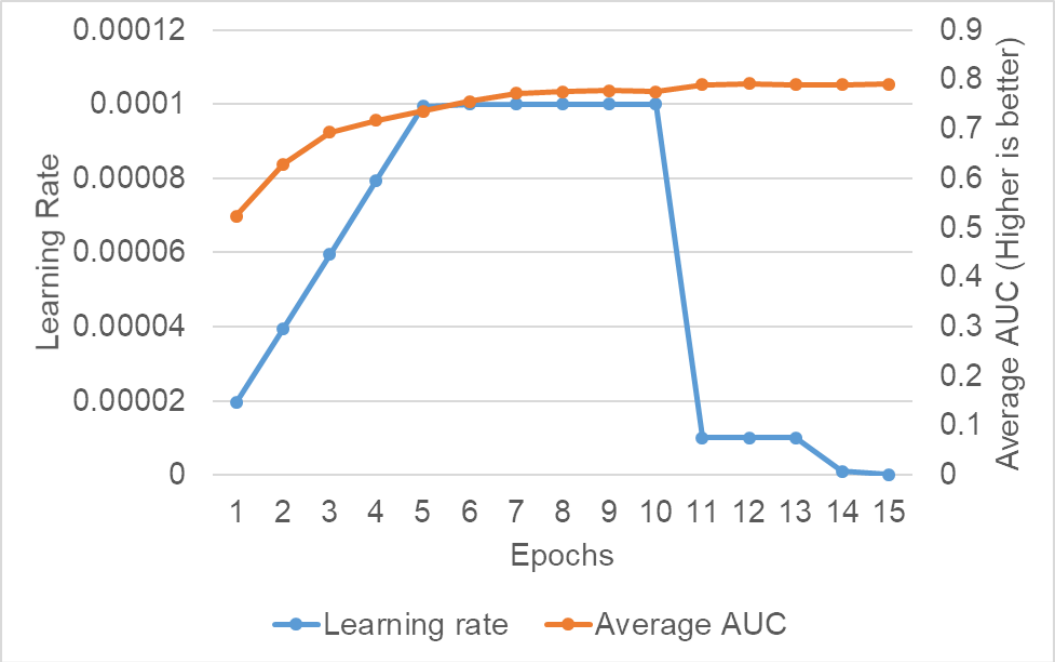

Figure 7. Impact on AUC by learning rate scheduler for Adam optimizer

Batch size

The batch size parameter is for model accuracy and for scalability. Artificial neural networks are trained by minimizing a loss function that measures the deviation of the output of the network from the wanted output. The input data is divided into batches and fed into the neural network to calculate the loss function. The weights of the network are adjusted at the end of each batch to optimize the loss function.

For a smaller batch size, the gradients are calculated based on a smaller sample of data. Therefore, it is reasonable to expect that the model converges faster for a larger batch size. Increasing the batch size also increases the parallelism.

While this result is true for smaller batch sizes, it is not always the case. Increasing the batch size can lead to a loss in generalization performance[15]. In other words, the performance of the model on testing datasets is often worse when trained with large-batch methods as compared to small-batch methods.

As explained in BigDL on page 9, the batch size must be a multiple of the number of executors that are multiplied by the number of cores in each executor. A lower batch size might yield better accuracy because the gradients are calculated for a smaller data sample. To gain more parallelism, you must increase the number of executors, potentially increasing the batch size. However, increasing the batch size might negatively affect the generalization performance of the model because the gradients are calculated on a larger data sample.

We evaluated the model by varying the number of images in a batch per thread. The following figure shows the results.

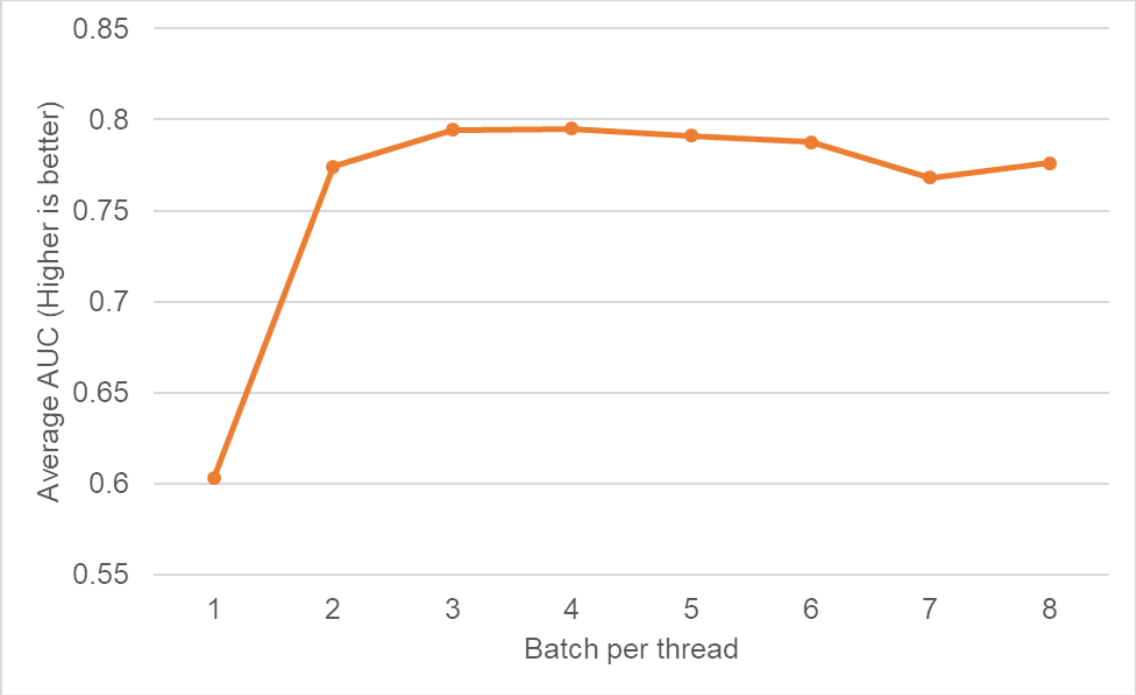

Figure 8. Results of varied number of images in a batch per thread

As expected, the average AUC increases as we increase the number of images in a batch per thread. After a certain point, due to a generalization issue, the average AUC continues to decrease. In our case, the average AUC is the highest when the batch per thread is four.

Scalability

We demonstrate how the model scales as we increase the number of servers and scale the batch size accordingly. The following calculation determines global batch size:

Global batch size = number of executors * number of cores * batch per thread

We change the number of executors (and thus the number of compute nodes) for four, eight, 12, and 16 executors. We choose four as the batch per thread and 32 as the number of cores. We measure the time for 15 epochs.

The following figure shows the throughput.

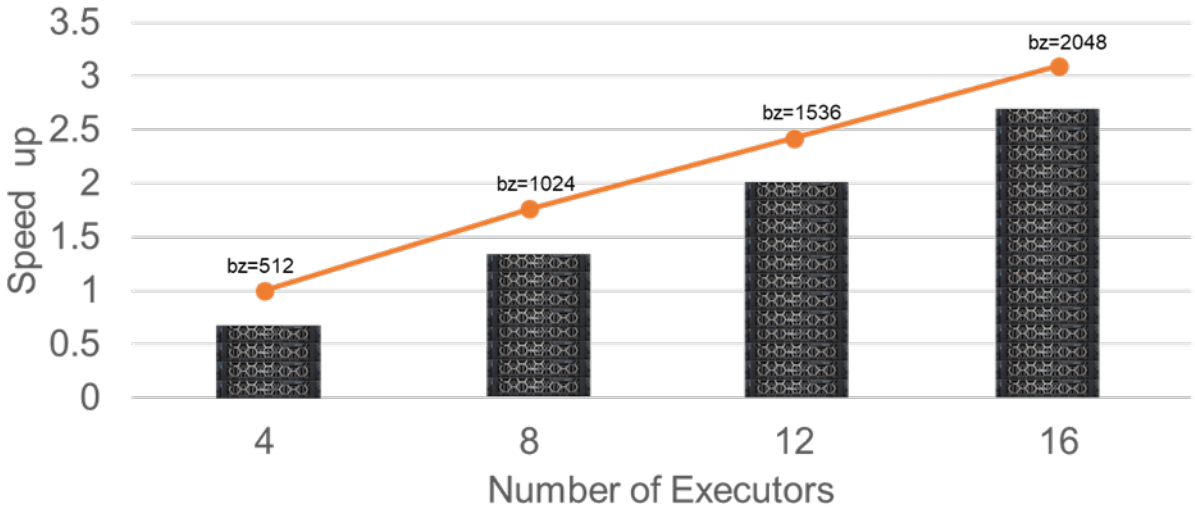

Figure 9. Results of scaling the number of executors on speedup

We achieve three times the speedup when scaling the number of Spark workers from four to 16 workers. We can accurately train the model in approximately 2.5 hours. The average AUC for 14 diseases is within a percentage variation between the data points. We believe that the overhead of the Spark executors and the tasks prevents us from achieving linear scalability.

Conclusion

This white paper describes how we built an end-to-end ML pipeline in Apache Spark to train a neural network to detect diseases from chest x-rays accurately. We show how to build a deep learning pipeline by using Analytics Zoo APIs to read images as Spark Dataframes, feature engineering, and neural network training. We describe how we used transfer learning to predict diseases accurately in chest x-rays by using a ResNet-50 model trained on an ImageNet dataset. We provide guidance for Spark parameters. Our work proves that it is important to consider batch size and the adaptive learning rate scheduler to improve accuracy of the model when training to scale.

We also demonstrate how deep learning applications can be developed and deployed at scale on an existing Hadoop and Spark cluster. This approach avoids the need to move data to a different deep learning cluster and eliminates the operational complexities of provisioning and maintaining yet another distributed compute environment.

In the future, we intend to implement learning rate schedule options such as LARS to determine how well the model can train on large-scale clusters. We also plan to investigate performance characteristics for image classification and develop guidance for sizing the Spark worker nodes.

References

[1]Analytics Zoo

[2]BigDL

[3]X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. M. Summers, "Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases," in 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017, pp. 3462–3471.

[4]P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Ding, A. Bagul, C. Langlotz, K. Shpanskaya et al., "Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning," arXiv preprint arXiv:1711.05225, 2017.

[5]Dell EMC Ready Solution for Artificial Intelligence Machine Learning with Hadoop: Design Guide

[6]BigDL and Analytics Zoo Engine for Cloudera Data Science Workbench

[7]Yang, Yuhao, Jiao(Jennie) Wang. Deep Learning with Analytic Zoo Optimizes Mastercard* Recommender AI Service, published on March 4, 2019.

[8]Spark ML Pipelines

[9]Y. Bengio, "Deep learning of representations for unsupervised and transfer learning," in Proceedings of ICML Workshop on Unsupervised and Transfer Learning, 2012, pp. 17–36.

[10]Lucas A. Wilson, Vineet Gundecha, Srinivas Varadharajan, Alex Filby and Quy Ta, "Fast and Accurate Training of an AI Radiologist on Intel Xeon-based Dell EMC Supercomputers".

[11]J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009, pp. 248–255.

[12]J. B. DP Kingma, "Adam: A method for stochastic optimization," in Proceedings of the 3rd International Conference on Learning Representations, 2014.

[13]Lucas A. Wilson, Vineet Gundecha, Srinivas Varadharajan, Alex Filby, Pei Yang, Quy Ta, Valeriu Codreanu, Damian Podareanu, Vikram Saletore, SC 2018. Fast and Accurate Training of an AI Radiologist

[14]Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Recognition", 2015, arXiv:1512.03385.

[15]Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, Ping Tak Peter Tang, "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima", in ICLR 2017, arXiv:1609.04836.