Big Data Analytics - SAT5165
Report for Small Project 4
By: Olivia Gette

Title: Flower Image Classification using Spark-Based Convolutional Neural Networks
GitHub Link: https://github.com/omgette/smallProject4

The goal of this project is to explore the use of distributed deep learning with Apache Spark to classify flower images using Convolutional Neural Networks (CNNs). Image classification is an important task in computer vision, and CNNs are a good approach for learning hierarchical features directly from raw image data. This is what makes them good for classifying images with structured patterns such as petals, stems, and leaves. By applying Spark to this workflow, I want to demonstrate how distributed computing can support large-scale image preprocessing and model training across multiple virtual machines.

For this small project, I am using the publicly available Kaggle Flowers Recognition Dataset, which contains thousands of images across five categories: daisy, dandelion, rose, sunflower, and tulip. The project highlights the integration of Spark with deep learning libraries to handle big image data efficiently, while also serving as a stepping stone toward my final project on pneumonia detection from X-ray images.

To manage the dataset and training process, I will use Apache Spark across two virtual machines (VMs), one Master Node and one Worker Node. I expect that using two VMs will reduce preprocessing and training time compared to using just one VM. This would show the importance of high-dimensional computing when working with large amounts of data like I am here. Ultimately, the goal of this project is to show how Spark-based CNNs can be applied to real-world image classification problems, while also preparing the foundation for more complex medical imaging tasks in my final project.

Code Explanation and Method

Python Code

Load Data and Preprocess:

The first stage of the flower classification pipeline uses the Kaggle dataset “Flowers Recognition”, which contains thousands of images across five categories: daisy, dandelion, rose, sunflower, and tulip. Since the dataset was downloaded directly onto my virtual machine from Kaggle, the images were already available in my local file system, and I didn’t need to upload them separately to HDFS.

The dataset was read into Spark using PySpark’s image loading utilities, which allowed me to create a Spark DataFrame with image paths and labels. Basic preprocessing steps were applied to ensure compatibility with the CNN training pipeline:

- Images were resized to a consistent dimension (128x128 pixels) to standardize input size.

- Pixel values were normalized to the range [0,1] for stable training.
- The dataset was split into training and test sets using an 80/20 ratio to allow performance evaluation.

Unlike datasets I've worked with in the past, this image dataset required preprocessing tailored to computer vision tasks. These steps ensured that the data was compatible with Spark's distributed deep learning workflow and optimized for training a Convolutional Neural Network classifier.

VM Configuration

I configured two VMs: one master node and one worker node. I accessed the Michigan Tech vSphere client using the BIG-IP Edge VPN and used the VMware Remote Console (VMRC) to launch and manage the VMs.

The IP addresses assigned were:

- Master: 192.168.13.122
- Worker1: 192.168.13.123

Network & Hostname Configuration:

I configured two virtual machines (VMs) for distributed data preprocessing: one master node and one worker node. Hostnames were assigned to each VM using /etc/hosts to simplify communication, and connectivity between nodes was verified using ping. This allowed the VMs to reference each other by hostname instead of IP address.

Here is the ping verification:

```
[sat3812@hadoop2 ~]$ ping master
PING master (192.168.13.122) 56(84) bytes of data.
64 bytes from master (192.168.13.122): icmp_seq=1 ttl=64 time=0.098 ms
64 bytes from master (192.168.13.122): icmp_seq=2 ttl=64 time=0.301 ms
64 bytes from master (192.168.13.122): icmp_seq=3 ttl=64 time=0.311 ms
64 bytes from master (192.168.13.122): icmp_seq=4 ttl=64 time=0.306 ms
64 bytes from master (192.168.13.122): icmp_seq=5 ttl=64 time=0.299 ms
64 bytes from master (192.168.13.122): icmp_seq=6 ttl=64 time=0.314 ms
^C
--- master ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5104ms
rtt min/avg/max/mdev = 0.098/0.271/0.314/0.077 ms
```

```
[sat3812@hadoop1 /]$ ping worker1
PING worker1 (192.168.13.123) 56(84) bytes of data.
64 bytes from worker1 (192.168.13.123): icmp_seq=1 ttl=64 time=0.460 ms
64 bytes from worker1 (192.168.13.123): icmp_seq=2 ttl=64 time=0.304 ms
64 bytes from worker1 (192.168.13.123): icmp_seq=3 ttl=64 time=0.290 ms
64 bytes from worker1 (192.168.13.123): icmp_seq=4 ttl=64 time=0.288 ms
64 bytes from worker1 (192.168.13.123): icmp_seq=5 ttl=64 time=0.294 ms
64 bytes from worker1 (192.168.13.123): icmp_seq=6 ttl=64 time=0.189 ms
^C
--- worker1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5108ms
rtt min/avg/max/mdev = 0.189/0.304/0.460/0.079 ms
```

Apache Spark Configuration

Apache Spark was set up in standalone cluster mode to enable distributed processing of the image dataset. This allowed preprocessing steps, such as reading and resizing images, to be parallelized across the two VMs. Spark handled the data efficiently, while the CNN model was trained separately using PyTorch

Here is the check to confirm that the passwordless SSH works correctly:

```
[sat3812@hadoop1 /]$ ssh worker1
Last login: Mon Nov 17 18:53:09 2025
```

I changed some of the configurations in the hadoop files. First, I started by updating /opt/hadoop/etc/hadoop/hadoop-env.sh with:

```
export JAVA_HOME=/opt/jdk
export HDFS_NAMENODE_USER="sat3812"
export HDFS_DATANODE_USER="sat3812"
export HDFS_SECONDARYNAMENODE_USER="sat3812"
export YARN_RESOURCEMANAGER_USER="sat3812"
export YARN_NODEMANAGER_USER="sat3812"
export HADOOP_CONF_DIR=/opt/hadoop/etc/hadoop
```

I then checked the following files to make sure they contained the correct information:

```
core-site.xml
hdfs-site.xml
yarn-site.xml
mapred-site.xml
```

On master nodes I listed the worker host name using /opt/spark/conf/workers:

```
worker1
```

Performance and Model Comparison:
Across the four iterations of my training pipeline, I made improvements to both data loading and model architecture. These changes were motivated by concepts from class related to regularization, data augmentation, model capacity, and overall training stability. Below is a summary of what changed from script to script and why.

Script 1 → Script 2
Main Issues in Script 1

- Minimal CNN architecture with only two convolutional layers.
- No normalization or augmentation.
- Used raw directory walking instead of preprocessed CSVs.
- No regularization, so the model overfit quickly.
    - Seen by the almost perfect training accuracy (99.7%) and the poor testing accuracy (65.36%)

Results:

```
Number of flower classes: 5
Using device: cpu
Epoch 1/10 - Loss: 1.2554, Acc: 0.4816
Epoch 2/10 - Loss: 0.9567, Acc: 0.6366
Epoch 3/10 - Loss: 0.7707, Acc: 0.7094
Epoch 4/10 - Loss: 0.5303, Acc: 0.8127
Epoch 5/10 - Loss: 0.3227, Acc: 0.8921
Epoch 6/10 - Loss: 0.1503, Acc: 0.9575
Epoch 7/10 - Loss: 0.0881, Acc: 0.9754
Epoch 8/10 - Loss: 0.0396, Acc: 0.9926
Epoch 9/10 - Loss: 0.0384, Acc: 0.9906
Epoch 10/10 - Loss: 0.0200, Acc: 0.9969
Test Accuracy: 0.6536
Saved model as flower_cnn.pth
```

Key Changes in Script 2

- Added dropout layer (0.5) in the classifier
    - Reduced overfitting and improved generalization.
- Introduced data augmentation (horizontal flip, rotation, color jitter)
    - Forced the model to learn more robust features.
- Added normalization to stabilize learning.
- Slightly deeper classifier (Linear → 128 → output).
- Reduced epoch number to hopefully avoid the model memorizing the data.
- Reduced learning rate (5e-4) to help training stability.

Outcome:
The model became noticeably more stable and generalized better, but still struggled with complex flower classes due to limited network capacity.
Results:

```
Number of flower classes: 5
Using device: cpu
Epoch 1/8 - Loss: 1.3799, Acc: 0.3974
Epoch 2/8 - Loss: 1.1873, Acc: 0.5110
Epoch 3/8 - Loss: 1.1218, Acc: 0.5415
Epoch 4/8 - Loss: 1.0816, Acc: 0.5575
Epoch 5/8 - Loss: 1.0438, Acc: 0.5826
Epoch 6/8 - Loss: 0.9967, Acc: 0.5975
Epoch 7/8 - Loss: 0.9883, Acc: 0.6143
Epoch 8/8 - Loss: 0.9541, Acc: 0.6220
Test Accuracy: 0.6486
Saved model as flower_cnn.pth
```

Script 2 → Script 3
After Script 2, the model wasn't overfitting anymore, it actually seemed to be underfitting. Its accuracy plateaued early. I needed a deeper architecture.
Key Changes in Script 3

- Added a third convolutional layer, increasing representational power.
- Inserted BatchNorm after every convolution
    - Reduced internal covariate shift and improved gradient flow.
- Increased classifier size (Linear → 256 → output).
- Simplified augmentation slightly to reduce excessive noise.
- Increased total number of training epochs (8 → 15).

Outcome:
The model trained more effectively and extracted better features. BatchNorm also stabilized training, giving smoother loss curves.
Results:

```
Number of flower classes: 5
Using device: cpu
Epoch 1/15 - Loss: 1.4921, Acc: 0.4559
Epoch 2/15 - Loss: 1.0885, Acc: 0.5524
Epoch 3/15 - Loss: 1.0367, Acc: 0.5812
Epoch 4/15 - Loss: 0.9664, Acc: 0.6160
Epoch 5/15 - Loss: 0.9269, Acc: 0.6360
Epoch 6/15 - Loss: 0.8682, Acc: 0.6674
Epoch 7/15 - Loss: 0.8373, Acc: 0.6711
Epoch 8/15 - Loss: 0.8034, Acc: 0.6940
Epoch 9/15 - Loss: 0.7929, Acc: 0.6908
Epoch 10/15 - Loss: 0.7567, Acc: 0.7000
Epoch 11/15 - Loss: 0.7048, Acc: 0.7297
Epoch 12/15 - Loss: 0.6907, Acc: 0.7331
Epoch 13/15 - Loss: 0.6725, Acc: 0.7405
Epoch 14/15 - Loss: 0.6680, Acc: 0.7399
Epoch 15/15 - Loss: 0.6257, Acc: 0.7599
Test Accuracy: 0.7174
Saved model as flower_cnn.pth
```

Script 3 → Script 4

Script 4 had the most significant improvement and included many good CNN training practices.

Major Improvements

1. Added a fourth convolutional layer (128 channels)
    - allows the model to detect higher-level textures and shapes.
2. Added AdaptiveAvgPool2d

This automatically compresses the spatial dimensions to 1x1:

- makes feature size independent of input resolution,
- improves stability,
- greatly reduces the number of parameters in the classifier.

3. Improved classifier
    - Input dim shrank to 128 (due to adaptive pooling).
    - Added 256-unit hidden layer with ReLU
    - Increased dropout layer back to 0.5 for stronger regularization
4. Added Weight Decay to Optimizer
    - optimizer = Adam(..., weight_decay=1e-4)
        - combats overfitting by penalizing large weights.
5. Added Learning Rate Scheduler
    - StepLR(step_size = 5, gamma = 0.5)
        - gradually reduces learning rate to refine training and prevent the model from getting stuck.
6. More controlled augmentation

Kept augmentation but made it less aggressive than in Script 2 to reduce label noise.

Outcome:
Script 4 had the strongest training behavior, least overfitting, and the best test accuracy. The deeper architecture plus combined regularization techniques significantly improved performance.

Results:

```
Number of flower classes: 5
Using device: cpu
Starting training loop..
Epoch 1/15 - Loss: 1.2292, Acc: 0.4836
Epoch 2/15 - Loss: 1.0230, Acc: 0.5966
Epoch 3/15 - Loss: 0.9544, Acc: 0.6263
Epoch 4/15 - Loss: 0.8983, Acc: 0.6517
Epoch 5/15 - Loss: 0.8563, Acc: 0.6777
Epoch 6/15 - Loss: 0.8061, Acc: 0.6960
Epoch 7/15 - Loss: 0.7632, Acc: 0.7071
Epoch 8/15 - Loss: 0.7434, Acc: 0.7200
Epoch 9/15 - Loss: 0.7466, Acc: 0.7214
Epoch 10/15 - Loss: 0.7331, Acc: 0.7262
Epoch 11/15 - Loss: 0.6831, Acc: 0.7476
Epoch 12/15 - Loss: 0.6671, Acc: 0.7474
Epoch 13/15 - Loss: 0.6581, Acc: 0.7582
Epoch 14/15 - Loss: 0.6573, Acc: 0.7542
Epoch 15/15 - Loss: 0.6506, Acc: 0.7608
Test Accuracy: 0.7752
Saved model as flower_cnn.pth
```

This table summarizes the main hyperparameters and architectural changes made across the four scripts, highlighting how tuning these parameters contributed to improved training stability, feature extraction, and test accuracy.

Hyperparameter	**Script 1**	**Script 4**	**Notes**
Learning Rate	0.001	0.005	More stable learning
Optimizer	Adam	Adam	Added weight decay (1e-4) to reduce overfitting
Number of Epochs	10	15	Increased to allow deeper network to learn
Batch Size	32	32	Kept constant
Dropout	None	0.5	Added to reduce

			overfitting
Convolutional Layers	2	4	Increased depth to capture more features
Batch Normalization	No	Yes	Added to stabilize learning and improve gradient flow
Classifier Hidden Dim	128	256	Increased to improve representational capacity
Data Augmentation	None	Flip, rotation, color jitter	Added to improve generalization
Learning Rate Scheduler	None	StepLR(step = 5, gamma = 0.5)	Helps refine training and avoid getting stuck

Performance Summary:
The final CNN model achieved 77.5% test accuracy. Training accuracy increased steadily across epochs, and the loss decreased smoothly, showing that the model learned effectively. The improvements made across the four scripts, including deeper convolutional layers, batch normalization, data augmentation, dropout, and a learning rate scheduler, all contributed to better feature extraction, reduced overfitting, and improved generalization. Overall, the model provides a good baseline for classifying flower images and demonstrates how PyTorch CNNs can handle moderately sized image datasets well.

Discussion and Reflection:
Over the four scripts, I gradually improved the CNN for flower classification. The first script used a very simple CNN with two convolutional layers and minimal image transformations. In the second script, I added dropout and data augmentation to reduce overfitting and improve generalization. The third script introduced batch normalization and a deeper network with three convolutional layers, which further improved accuracy. In the fourth script, I added a fourth convolutional layer, adaptive pooling, and a learning rate scheduler, resulting in the highest test accuracy of around 77.5%.
Each change helped the model learn better features from the images and increased its performance. Using PyTorch throughout allowed for easy experimentation with network depth, regularization, and training techniques. Overall, the gradual improvements across scripts show the impact of deeper architectures, normalization, and augmentation on model performance.

Conclusion and Future Directions:
The final PyTorch model trained successfully and achieved strong performance. The combination of deeper CNN architecture, data augmentation, normalization, dropout, and a learning rate scheduler helped stabilize training and improve generalization.
For my final project, I will adapt this idea to pneumonia X-ray classification, which involves grayscale images, class imbalance between normal, bacterial, and viral cases, and likely a deeper CNN or transfer learning to capture complex medical features.
In the future, I could explore hyperparameter tuning, GPU acceleration, and distributed deep learning frameworks to further improve model accuracy and reduce training time. These steps will allow me to scale this project to larger or more complex image classification tasks.

Dataset Used:
Kaggle. (2021). Flowers Recognition Dataset [Dataset]. Retrieved from https://www.kaggle.com/datasets/alxmamaev/flowers-recognition?select=flowers.