Classifying Galaxy Morphologies
Using Deep Learning

Andrei Ignat, Ankit Gupta, Keshav Mathur, Ryan Hausen

June 7, 2016

Abstract

In this project, we try to replicate the results obtained in Huertas-Company et al. 2015 using the same dataset. The problem we face is of classifying galaxies based on their morphologies. The general idea is to narrow down the possibilities of types for the given galaxy instead of labeling the galaxy itself. We use deep convolutional neural networks using different optimization techniques and activation functions in order to do it. We have used the CANDLES GOODS-S dataset for this project.

1 Introduction

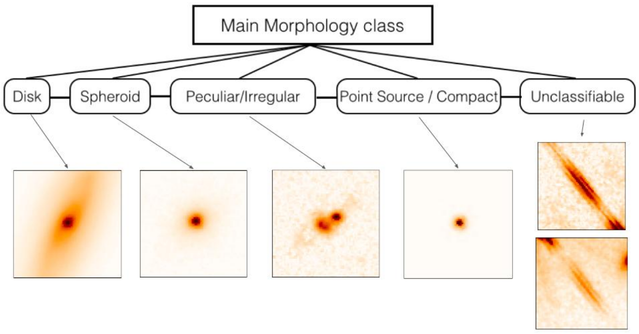
Today, galaxy classification is done primarily by astronomers by looking at the images on computers. This method is not much efficient when we realize that in the next decade when big astronomy projects like WFIRST and JSWT start collecting terabytes of data every night! This is overwhelmingly large amount of data to be processed by humans, but thankfully we think we have a solution. Machine Learning!

The general idea is to train a machine to classify galaxies based on input image. While some "algorithms" have been suggested to do this job, we take a machine learning approach. In this problem, we classify galaxies from one of the following category: disk, spheroid, peculiar/irregular, point source/compact, unclassifiable. A general visualization of each of these is pictured below:

2 Dataset

The dataset used is a subset of CANDLES dataset. It consists of images taken at different frequencies and we use only F160 images for our project. This is because it was discovered that comparable results can be achieved using just the F160 filter images. Each of these images is a jpeg with dimesnsions of 454x454px. We start off with three random rotations of each galaxy to artificially increase the dataset and avoid over fitting. There are 8000 labeled images that we can use for our learning and testing. However, we have augmened the available data by doing various pre-processing like cropping, rotation and translational changes in the image while keeping the object of concern in center.

Figure 1: Morphology classes in our dataset



3 Approach

3.1 Neural Network

A neural network is a network consisting of artificial neurons across different layers connected with each other. The artificial neurons we use are mostly perceptrons or sigmoids. Each neuron takes input from the previous layer of the network, applies its activation function and sends output to the neurons connected in the next layer. This way when we send an input to the input layer, after series of such function applications, we get the output at the output layer. All the layers except input and output layers form the "hidden" layers of the netowrk. This hidden layers is where the neural network "magic", or more formally, learning takes place.

The output of a neuron depends upon its weights and biases. Ideally, we want to have a subset of weights and biases for all the neurons in the network, which when applied to the input generates the desired output. The training or the learning phase of the neural network can be thought as actually learning these weights and biases. Once we have obtained this values, we can say that we have a network which functions correctly.

However, we need to a little bit careful during training. We can only use labeled data for training, but also for testing we need labeled data to check if our network is correct or not. Hence, we need to divide the available data into training and testing. The testing data shouldn't be used during trainig phase, which means we do not "learn" the weights and biases for test data. We learn them on training data and use it to classify the test data to check the performance of our network.

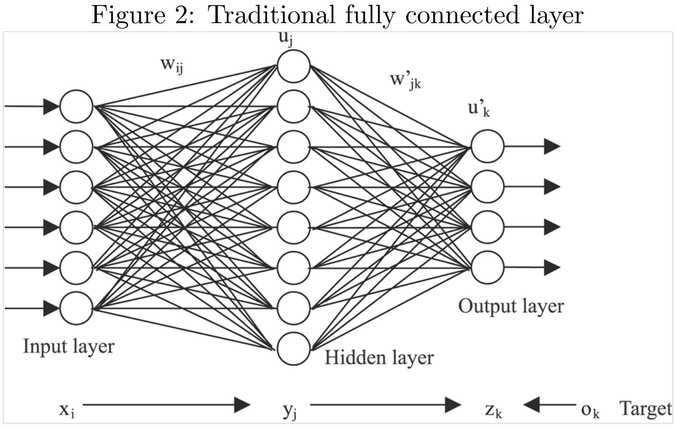
Also, during training we need to control hyper-parameters like learning rate in order to make sure that our network does not overfit. An overfit network performs well on the training data but poorly on test. Ideally, the test and training data should have equal proportion of all the classes.

3.2 Convolutional Neural Networks

The recent success of neural networks in machine learning especially within problem settings such as image classification, has inspired many different fields to use the technology to automate some of their processes.

Neural networks are an algorithm that attempts to mimic the way that the brain learns using 'neurons'. In a traditional neural network, the neurons are setup in layers each only connected to the one preceding and the one succeeding it, visualized in the diagram below:

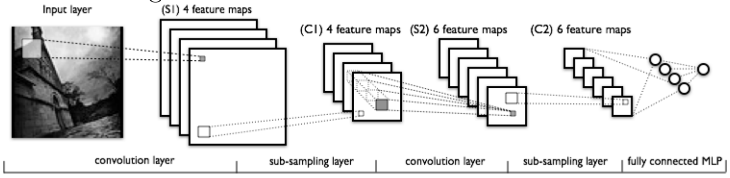
Figure 2: Traditional fully connected layer



Each input to a neuron is multiplied by a weight that is adjusted as the network learns. The result of that operation is passed through a non-linear transformation, traditionally a sigmoid, though other functions have become popular such as the tanh and the rectified linear unit(ReLU) functions.

Recently, a different architecture has been found to be very successful with image recognition, convolutional neural networks(CNN). A CNN works differently than a traditional neural network by convoluting over regions of the input rather than processing the entire input at once as a vector.

Figure 3: Convolutional Neural Networks



There are a couple observational advantages to this approach. First, we consider each pixel in the context of the pixels surrounding it, which helps us understand the structure of the data. Second, we limit the amount parameters we have to learn to the size of the kernel that moves over the image, because a single set of weights equal to the size of the kernel is used per filter. This allows CNNs to exploit the structure in images and hence make them the model of choice in many state-of-the-art vision problems.

3.3 Activation Functions

While CNNs are found to be more efficient in image processing realted problems than ANN, the performance of CNN varies a lot based on the activation function or the non-linearities used at different layers. An activation function is basically a way of treating inputs at different neurons and getting an output. Some of the popular activation functions are as follows:

- Sigmoid: A sigmoid function squashes a real-valued input into the range of 0 to 1.
- Relu: A rectified linear unit function outputs the maximum value between 0 and the input. So, if the input is negative, the output will be 0 and not a negative number. For positive input, the output value is linear.
- tanh: Just like a sigmoid, a tanh activation function also squashes the input value but the range in this case is [-1, 1]. The output of tanh is zero-centered, hence it is preferred over sigmoid.
- Maxout (Goodfellow et al. 2013): In a convolutional network, a maxout feature map can be constructed by taking the maximum across k affine feature maps. This activations has given state of the art performance results for some datasets like MNIST and CIFAR-10 when used in conjunction with dropout.

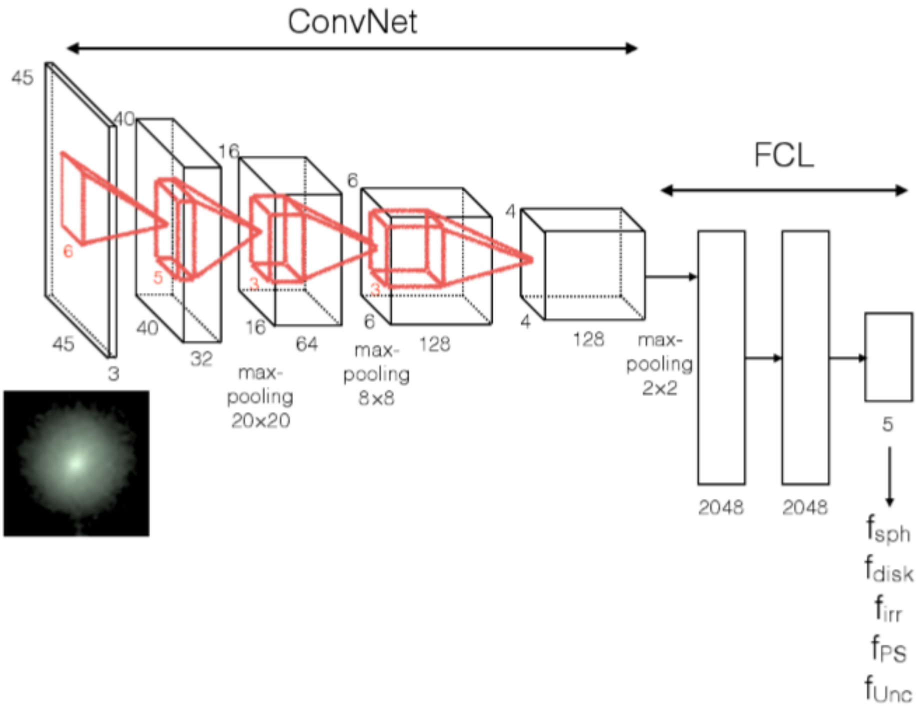
3.4 Architecture

The architecture used in our paper tried to very closely to mirror the architecture originally described in (Dieleman, Willett, and Dambre 2015) and used in (Huertas-Company et al. 2015)

The architecture consists of an input layer that is 45x45x3 which represents the layout of the image and the three color channels (RGB). This is passed through four convolutional layers three of which have max-pooling layers(first, second, fourth). The convolution layers have the kernel sizes: six, five, three, and three, respectively. These are followed by two fully connected layers that use the ReLU function as they're non linearity function. This is the on contrast in our architecture to the one originally implemented in the paper. The architecture in the paper uses the maxout rather than ReLU, however the framework we used for our implementation, TensorFlow (Abadi et al. 2016), doesn't have native support for maxout.

Another difference in our implementation is the model averaging. The original paper has 17 different variations of the above described architecture each of which individually close to as good. They average over the predictions of each model for a given image and hence get better performance.

Figure 4: Architecture



Our model uses AdamOptimizer (Kingma and Ba 2014) for finding the parameters. The final learning rate used was 0.001. To avoid over fitting we use dropout (Srivastava et al. 2014) and do real time data augmentation as explained in (Dieleman, Willett, and Dambre 2015).

4 Installing Tensorflow on Hyades

Here we give a basic record of how we got tensorflow working on Hyades. You can easily install tensorflow using pip:
pip install `https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.9.0rc0-cp27-none-linux_x86_64.whl` *–user*
But using this will give a libc error. For this we followed the answer at (`http://stackoverflow.com/questions/33655731/error-while-importing-tensorflow-in-python2-7-in-ubu` `34897674#34897674`) and download local copies of the required libc in local folders. Next we will get a libcuda error. For this we basically need cuDNN. First we copy the Hyades Cuda 7.5 into a local directory. then download and unpack cuDNN v4 into local directories. Then we copy files from this into the local Cuda 7.5 folder.

cp local/cudnn/cuda/include/cudnn.h local/cuda/include/

cp local/cudnn/cuda/lib64/libcudnn local/cuda/lib64*

Now we have every thing in place. Now we just need to export the path to the local cuda:

Figure 5: Parameters in the original Dieleman, Willett, and Dambre 2015 paper. We do not use maxout and our output nodes use sigmoid activations.

	type	# features	filter size	non-linearity	initial biases	initial weights
1	convolutional	32	6×6	ReLU	0.1	$\mathcal{N}(0, 0.01)$
2	convolutional	64	5×5	ReLU	0.1	$\mathcal{N}(0, 0.01)$
3	convolutional	128	3×3	ReLU	0.1	$\mathcal{N}(0, 0.01)$
4	convolutional	128	3×3	ReLU	0.1	$\mathcal{N}(0, 0.1)$
5	dense	2048	–	maxout (2)	0.01	$\mathcal{N}(0, 0.001)$
6	dense	2048	–	maxout (2)	0.01	$\mathcal{N}(0, 0.001)$
7	dense	37	–	constraints	0.1	$\mathcal{N}(0, 0.01)$

export LD_LIBRARY_PATH=" /local/cuda/lib64/ /$LD_LIBRARY_PATH"
And run Python using the new libc.

LD_LIBRARY_PATH=~/local/libc/libc6_2.17/lib/x86_64linuxgnu/:~/local/libc/libc6_2.17/usr/lib64/:$LD_L
/local/libc/libc6_2.17/lib/x86_64-linux-gnu/ld-2.17.so /pfs/sw/python/Python-2.7.8/bin/python

5 Evaluation

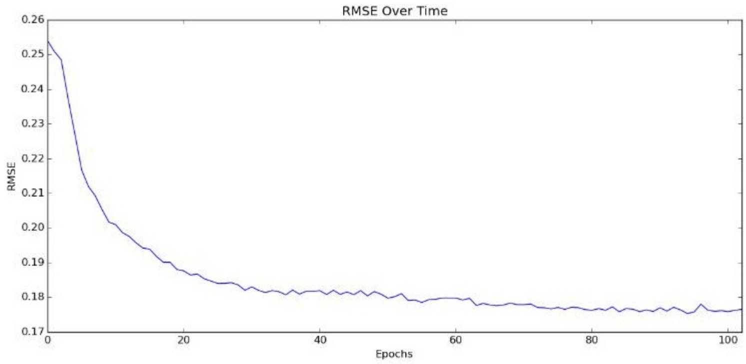
5.1 Results

To evaluate the accuracy of our models used root mean square error (RMSE):

$$RMSE = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

We stopped training models at around 100 epochs because that is when we noticed a that test error began to increase while training error continued decrease. This is typical of a model that is over fitting the training set. At this point our models had average RMSE of .17.

Figure 6: RMSE on test set



Although this is a little higher than the average RMSE for the trained model in the Huertas-Company et al. 2015 paper which was ~ .13. We think that the slight differences between our experiments can account for the small difference in our results. The differences were:

1. **Preprocessing**

   The method by which the Huertas-Compnay paper preprocessed the images wasn't exactly clear so we implemented a slightly different form of preprocessing than the paper.

2. **Real time data augmentation**

   In our implementation of the real time augmentation of the images, we excluded the brightness adjustment.

3. **Model Averaging**

   The best results in the Huertas-Company were achieved when they averaged the predictions of multiple trained models. However for this report we only report the model that had the best results.

4. **Maxout layers**

   As a result of the TensorFlow framework not having native support for the Maxout layer we instead opted to only use the ReLU activation function.

5.2 Performance

We trained the network on Hyades' GPU node (Tesla K20) and a PC (GeForce GTX 650). At Hyades it takes an average of < 50 seconds per epoch (without real time data augmentation). Each epoch involves learning from 19,000 images. Before doing pre processing (rescaling to 45x45) it took 8 minutes per epoch on Hyades compared to just just 108 seconds on the PC. A possible cause could be IO latency. This aspect needs inspection before we can do real time augmentation on Hyades. But Hyades has the benefit of higher memory and can be trained on bigger batches and can do predictions in bulk instead of batches.

6 Conclusions

In this paper we trained a convolutional neural network model based on the the Huertas-Company et al. 2015 paper to classify galaxy morphologies from the CANDELS dataset with promising results. The performance differences can hopefully be covered by implementing the features mentioned in section 5.1.

7 Future Work

We still have some ideas which we wish to implement in future in order to make more accurate network. Some of them are:

- The network we have implemented is basically a blatant copy of network used in D15 paper to classify galaxies. However, there network was for classifying 32 different galaxy classes while we only have 5, so we can try different network architecture.

- Maxout non-linearities have been shown to give good results on a number of classic datasets when used in conjunction with dropout, we will try to implement it in tensorflow.
- Do better pre-processing to augment training data.
- We need to explore the IO latency faced on Hyades' GPU node when running image operations during run time.
- Do model averaging over multiple networks.

8 Acknowledgments

I, in no way, claim that I should get all the credits for this project. All the authors of this project discussed different approaches among themselves and implemented it and shared their results with each other. In fact, I would say that I contributed least amount of fresh code in this project but it has been a good learning experience for me since this is my first time working on deep neural nets, using GPU and using tensorflow as well. Personally, I implemented the whole network by myself, fixed errors in it after suggestions from other team members, randomized the batch image selection and tried various network configurations and hyperparameters to check the performance of network.

References

Abadi, Martın et al. (2016). "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467*.

Dieleman, Sander, Kyle W Willett, and Joni Dambre (2015). "Rotation-invariant convolutional neural networks for galaxy morphology prediction". In: *Monthly Notices of the Royal Astronomical Society* 450.2, pp. 1441–1459.

Goodfellow, Ian J et al. (2013). "Maxout networks". In: *arXiv preprint arXiv:1302.4389*.

Huertas-Company, M et al. (2015). "A catalog of visual-like morphologies in the 5 CANDELS fields using deep-learning". In: *arXiv preprint arXiv:1509.05429*.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Srivastava, Nitish et al. (2014). "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.