Flower Taxonomic Classification using CNNs

Xavier Mignot
05788346
xmignot@stanford.edu

Maxwell Siegelman
05815725
maxsieg@stanford.edu

Abstract

We attempt to use a convolutional neural network to classify a 102 class image dataset of flowers. We resized the original images to be 32 by 32 pixels, and with a small convolution net and several data augmentation strategies (random cropping, mirroring) were able to achieve relatively high classification accuracies of 0.61. We also tested a pre-trained deeper architecture by fine-tuning the last layers but did not achieve high accuracies, leading us to conclude that the task of naive automated taxonomy is both difficult and different from more classic image recognition tasks.

1. Introduction

Convolutional Neural Nets (CNNs) have in recent years vastly increased computer accuracy in many image classification and processing tasks. However, while much work has been done on classification of broader groups of images with a variety of classes, classification in a narrower scope is less well researched.

Floral taxonomy is a prime example of such a problem - while flora is diverse and varied, the fundamental subject matter is the same so many of the images still share a large overlap in features (i.e., most flowers have petals and fall in to broadly the same shapes). And, flowers are non-rigid objects and so can deform in different ways from image to image. More broadly, taxonomy is an interesting image classification problem in its own right - it takes significant training for humans to correctly distinguish between some species, and often very particular morphological traits are the only thing separating closely related organisms. Moreover, these particular traits can be completely different from one particular species of plant to another (stem length vs petal shape, for example).

1.1. Previous Work

This dataset was originally used by Nilsback and Zisserman from the University of Oxford, who attempted to classify the flowers with an SVM. They noted many of the same challenges in classifying species of flowers as we encountered in our own work, such as the large amount of similarity between classes and the fact that features which distinguish two given classes are often much different from features which distinguish a different pair of classes. In particular, they noted that "what distinguishes one flower from another can sometimes be the colour, e.g. blue-bell vs sunflower, sometimes the shape, e.g. daffodil vs dandelion, and sometimes patterns on the petals, e.g. pansies vs tigerlilies etc. The difficulty lies in finding suitable features to represent colour, shape, patterns etc, and also for the classifier having the capacity to learn which feature or features to use." [4]

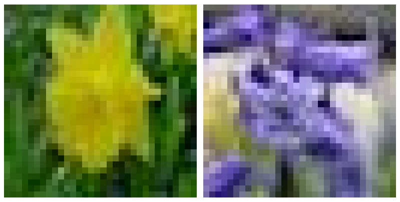
Using an SVM with hand designed features meant to described the shape, texture, boundary, distribution of petals and the color of the flowers, the team was able to achieve a 0.728 accuracy on the dataset. Some previous work has been done using Fully-Connected Nets to address taxonomic classification. Hernndez-Serna and Jimnez-Segura built a neural network to classify various fish, plant, and butterfly species and were able to achieve around 0.90 accuracy across classes. However, their methodology relied on the extraction of features from the image by a trained taxonomist, and so did not test the ability of a computer to extract these separating traits unaided [1]. Seung-Ho and Su-Hee took a similar approach to classifying just butterfly species. While they also used a neural net, it once again relied on extracted features such as wing length [2].

So, while significant work has been done on specific taxonomy, no work has yet attempted to use just a learned model with no a priori feature selection and preprocessing. Previous work has largely focused on training models learned with features extracted from images, rather than the images themselves. Furthermore, in both of the following experiments images were not taken in a natural setting - pictures were of collected samples in the lab, and not the organism in it's natural environment. An automated system that could recognize and classify at species in situ (say, by sending in a picture of something found in the field) would be far more useful as a taxonomic tool. Promising results with our specific dataset could indicate that our methods can be extrapolated to other species classification problems.

Beyond this, good results could indicate new planes of separation between the species we are looking at (overlooked trait differences) or even reveal new relationships between them (by looking at common misclassifications).

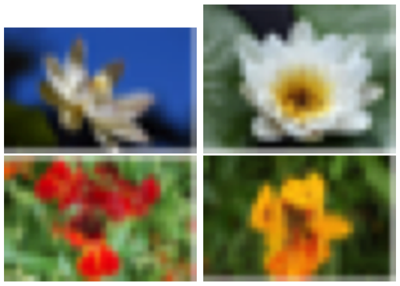
CNNs have proven to be quite effective at classification tasks over recent years, so these seem like a promising approach. The breakout example of this is of course the performance of the "AlexNet" on the ImageNet classification challenge [3]. While the classes in ImageNet have more broadly differentiable traits (container ships vs cherries, for example), hopefully a similar model is able to pick up on the more subtle differences between particular flower species.

2. Dataset and Features



Some example images from the dataset, resized. To train we resized to a lower resolution (32x32).

We will be working with the dataset developed for a publication by Nilsback and Zisserman consisting of 103 different flower species commonly occur in the United Kingdom country side. Each class contains between 40 and 258 individual images. Overall there are 8189 separate flower images over the 103 classes [4]. The images vary in position and lighting, as well as scale. Thus, we will need to size normalize before running the images through our model. This was done in one pass at the command line, resulting in a uniform sizing of 32 x 32. Additionally, several of the categories are quite similar to each other, consisting of closely related species of flower. Thus we anticipated that the task would be more difficult than broader classification challenges like ImageNet. Below, we've included two flowers in different classes (above) and two flowers in the same class (below) to illustrate this difficulty:



Images resized to 32x32, then magnified. Note how difficult the classification task is at this resolution.

Our data split was done over each individual class in order to ensure that the original distribution of examples was maintained. Our initial data split that we used for the baseline used 680 images in the training set and 340 in the validation set from the data with 17 classes. We then started working with our larger 102 class dataset - from this spread we withheld 15% as a final test set, and a further 15% as a validation set to tune our model hyperparameters. The data split was done over each individual class in order to ensure that the original distribution of examples was maintained. With the larger dataset we had 5762 training images, and 1210 validation images (1215 testing images).
We also built data augmentation into our models, with random cropping (to 28x28) and mirroring as well as more traditional overfitting defenses such as batch normalization and dropout. This should allow us to train for more iterations despite the relatively small size of the dataset. These methods were used to enhance the number of "novel" examples fed into the network. We also computed an image mean over the training set and subtracted it from each example, as neural nets work most effectively on data centered around the origin across dimensions.

3. Approaches and Techniques

Note: All models with exception of the baseline were trained locally using Caffe on OS X Yosemite MacBooks. The fine-tuned network was trained on the GPU with smaller batch-sizes, while the from-scratch networks were trained on CPU so as to have larger batches.

Initially, we implemented a simple 3 layer convolutional network similar to the one built in class. With minimal hyperparameter tuning this net was able to achieve 49% validation accuracy and 85% training accuracy. The high training accuracy indicated to us that with some hyperparameter tuning, and perhaps more data, we should be able to reduce our overfitting and achieve a solid validation accuracy. This was not a bad baseline, but we hoped to use deeper CNNs to improve our accuracy even further. These results led us to believe that a sufficiently deep network with a more optimal number of filters, especially one that is pretrained, which is run for more epochs with more carefully chosen hyperparameters could achieve very good results. Note however that this baseline was trained using the smaller 17 class dataset. On the larger dataset we trained a naive bayes model as an even simpler baseline, and were able to get 0.031538 accuracy. But we simply used the average color of each image as our feature, so we expected rather poor results. This doubled as a good way to test our intuition of how much the flower color really said about the class. This model does barely better than random chance ($\frac{1}{102}$ 0.0098), indicating that color is not a very powerful separating feature on it's own.

Note that in the original paper, the authors were able to get accuracies of 0.72 using hand-extracted features on an SVM with the larger dataset [8]. We expected that a trained neural net, while perhaps not outperforming this expert built model, would be able to seriously improve on our baseline.

First, we decided to train a shallower network from scratch. We chose to keep the network small in order to be able to compute all layer weights more quickly and iterate over hyper parameters. Our final architecture was:
Layer 1: A convolutional layer (with 96 outputs), a RELU activation, a max pool, and then a normalization layer.
Layer 2: A convolutional layer (with 256 outputs), a RELU activation, a max pool, and then a dropout layer.
Layer 3: A Fully-Connected inner product layer
Loss: Softmax

The softmax function takes a vector of K arbitrary real values and transforms it to a vector of values between 0 and 1 that collectively sum to 1:

$$\sigma(z)_j = \frac{e^{zj}}{\sum_{k=1}^{K} e^{zk}}$$

The loss function for the ith training example is then the log of this value for the correct label of that training example:

$$L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})$$

Here y_i is the correct label for the training example. In the context of training a neural network, the output values of the softmax function can be interpreted as the normalized probabilities of each class being correct for a given input. In turn, the loss function is expressing how much our probability estimations are off from the real probability, which is of course 1 for the correct class and 0 for all the other classes. This function is then minimized by repeatedly computing gradients relative to the loss for different subsets of the training examples (batch gradient descent). We tried varying learning rate decay policies in tandem with SGD as our solver.

To perform SGD, the gradients of all the weights in the network are calculated with respect to the softmax loss value, and then the weights are adjusted according to:

$$W = W - \lambda W'$$

where W is the weights, W' is the gradient of the weights with respect to the loss function, and λ (the learning rate) is a hyperparameter meant to control how much the network adjusts its parameters in response to each batch of training data. In this way we aim to minimize the value of the loss function, which has the effect of moving the outputs our network makes as close to the actual labels of the data as possible.

To tune these hyper parameters (λ and ρ the rate of learning rate decay), we tested validation accuracies at a variety of learning rates after 400 Iterations:

Learning Rate	Validation Accuracy	Test Loss
0.1	0.01785	4.451
0.01	0.10119	4.0513
0.005	0.172619	3.34423
0.001	0.3392	2.61414
0.0005	0.3392	2.95198
0.0001	0.22024	4.7147
0.00005	0.077381	4.17213
0.00001	0.017857	4.49171

We also wanted to test the ability of a deeper network to learn more effectively. In order to deal with issues of insufficient data and longer training times, we decided to use an already trained network (from the Caffe-Model Zoo) and fine-tune the last layers.

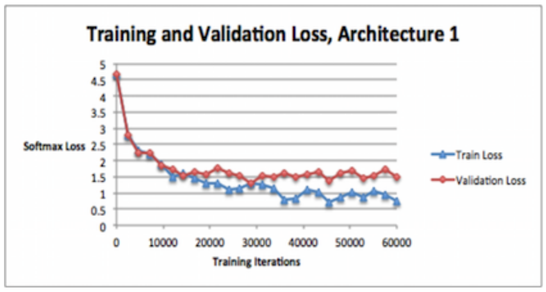
Normally all the parameters in a network are updated using this method. However, a network that has already been trained for one task can be fine-tuned to a different but related task by only updating the weights at the later layers and not the earlier ones. The intuition behind this practice is that the earlier layers will have been tuned to pick out certain low level features which will generalize across related problems while the higher level layers pick out more abstract features, so the earlier layers can be reused and the later layers can be updated for the new task. This reasoning makes particularly good sense in the context of convolutional neural networks for image recognition, since the lower level layers often learn to detect very basic and general features such as edges. The later layers represent combinations of these more general features and can be fine-tuned for the new image recognition task. The advantage this approach has over simply retraining the whole network is that it takes less time, since weight changes do not have to be calculated for the whole network, and it allows large networks to be used for tasks with small datasets without overfitting since only a portion of the network is trained on the small dataset.

Because of limits on GPU memory, we decided to fine tune the NIN-CIFAR10 model which achieved an error rate of 10.4% on CIFAR10. This net has 3 conv-relu layers followed by pooling and dropout, followed by 3 more conv-relu layers and another round of pooling and dropout, followed by 3 more conv-relu layers, another round of pooling and finally a softmax loss layer. Using the pre-trained weights, we altered the learning rate of the layers so that only the final four conv layers had non-zero learning rates, then set the base learning rate for the entire model fairly high in order to train the back end of the model. This choice was not based on a rigorous hyperparameter search given the constraints on our computation, but of the combinations of overall learning rate and layer learning rate combinations we attempted this was the most effective. A further investigation of what setup is optimal is an important part of our future plans, since we cannot think of a theoretical reason why a pretrained model should not be able to perform well on this task.

4. Results and Error Discussion

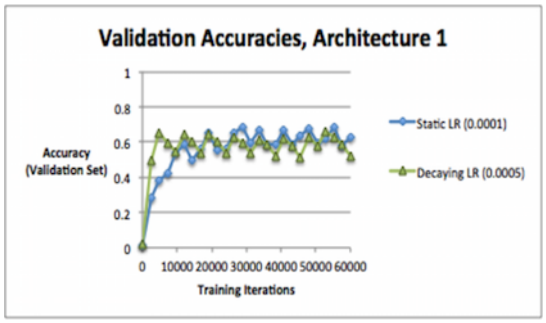
4.1. Architecture 1 - Small Net

For the first architecture, we tuned hyperparameters by training the model for 2000 iterations and periodically testing loss and validation loss to ensure learning was proceeding in the correct direction. The optimal learning rate settled in the range of 1e-4, with a momentum of 0.9. While we weren't seeing overfitting, the gains in accuracy and corresponding decrease in loss quickly plateaued (by iteration 20000 - see the first figure below).



Train and Test Losses for architecture 1 with tuned hyper parameters

We decided to experiment further and determine if we were seeing initial signs of overfitting or if perhaps the lack of learning rate decay was causing our model to bounce around the optimal parameters instead of continuing the optimization process. Thus we modified our solver to use a stepwise learning rate decay at the point where we began to see plateauing (15000-20000 iterations), but this unfortunately did not change the results very much.



Validation Accuracy, architecture 1 with stepwise decaying learning rate vs static learning rate

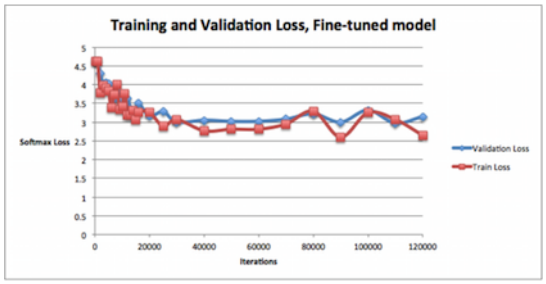
The accuracy and loss both plateau around the same place, indicating that these hyperparameters are fairly good even with a constant learning rate. We were surprised by how effectively a relatively small net was able to classify these images. The trained model achieved accuracy of 0.628 on the validation set and 0.619 on the test set. This falls short of the mark of 0.728 achieved by Nilsback and Zisserman, and it seems likely that even with a larger model and more hyperparameter optimization hand designed features would retain a lead of at least a few percent over a deep learning approach. Nonetheless, the approach clearly has good predictive power as it outperformed both our baseline and our fine-tuned network significantly.

The errors the network made seemed to most often have to do with small, subtle differences between flowers. It seemed common for misclassified examples to have the same color as the class they were mistaken for, as well as similar general petal structure. It was relatively uncommon for a flower to be mistaken for a class that was a completely different color. We were unable to detect any common pattern in what could have been confusing the network on examples that it misclassified. Of course, it is possible that our lack of knowledge about flowers is contributing to these errors - it would be interesting to have a botanist look at the mistakes.

4.2. Architecture 2 - Fine-tune Net

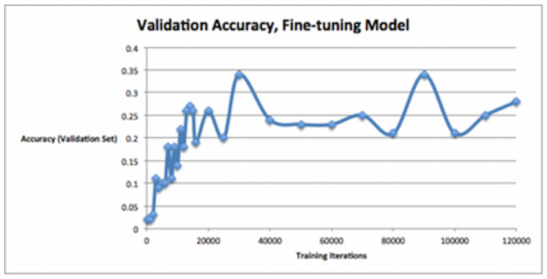
Once we achieved good results on our first net, we wanted to see if we could fine-tune a pre-trained net and get accuracy gains without having to commit large amounts of computational power to training a very deep network.

It was much more difficult to find suitable hyper parameters to train this model with, and we would often get divergent or static loss over many iterations. We eventually picked hyper parameters that gave us good loss decreases, but even then the loss plateaued relatively quickly.



Train and Validation Loss over 120000 Iterations

The final model tested at 0.294 validation accuracy and 0.291 testing accuracy, sadly not performing as well as we had expected.



Train and Validation Loss over 120000 Iterations

Note that in both of these graphs the values fluctuate more seriously than in architecture 1 because we trained locally on a GPU, so the testing batch size was smaller. However final accuracies reported are over the entire validation and withheld test sets.

The accuracy of the fine-tuned pretrained model did not get as high as the model trained from scratch. Loss could not be driven below 2.5 for either training or validation data. This does not appear to be a problem of overfitting, since training loss was equally resistant to improvement after about 20000 iterations. We hypothesized that the plateau could be broken by lowering our learning rate at sufficiently high iteration counts, but our experiments did not bear this out. Our next hypothesis for why this was the case is that flower classification requires very specific features to be extracted, and the low level features that were learned by the network may not be appropriate for the task of flower classification. As discussed previously, the distinguishing features for flower species are fairly tricky and the low level features learned from the CIFAR-10 dataset may lack some low level features that would be important to learn for a flower classification task. This could explain why the loss was plateauing, since a problem in the lower layers would not be optimized away because the learning rates for those layers was set to zero. We also discussed the idea that CIFAR-10 only having 10 classes was contributing to problems with having insufficiently rich low level features, but we don?t have any good evidence that a higher number of classes in the original dataset would have made a difference - there is no reason to think that in general a higher number of classes leads to more diverse low level features. Another potential reason for the plateaus we observed is that we were forced to train with a batch size of only 20 because of GPU memory constraints. This could have caused problems with parameter convergence, since larger batches would have done a better job approximating the entire training set.

To remedy these problems, we tried fine-tuning even more layers of the net. Our hope was that doing this would allow the earlier layers to be optimized for our problem, potentially capturing low level features that are important for flower classification. This approach also unfortunately did not seem to work and yielded even lower accuracy values. One reason for this failure could be that our hyperparameters were wrong, an explanation which is plausible because loss jumped around a lot even as the learning rate dropped (although this could be an artifact of using such small batches of training data). Another possible reason for this failure is that we still did not train enough layers of the network. Even in our experiment which allowed the largest number of layers to be adjusted, the bottom two layers were frozen. We decided not to experiment with changing all the layers since it seemed to defeat the purpose of trying to fine-tune a net at all rather than simply starting from scratch.

It was difficult to detect much of a pattern in the mistakes the network was making since it made so many. The difficulty distinguishing flowers with similar petal shapes and colors we observed with the previous model no longer appeared, and no classes appeared significantly harder for the net to predict.

Ultimately, these results were disappointing since it seemed reasonable that a pretrained model should be able to at least come close to equaling the performance of our model built from scratch - especially since it has many more layers.

5. Conclusions and Future Work

Classifying flowers turned out to be a fairly tricky task for a convolutional neural network to pull off. Training a convolutional neural network from scratch was clearly more effective than fine-tuning one trained on another dataset, possibly because the low level features required to differentiate flowers are different from those needed for other image classification tasks. While our best model did not beat out hand designed features from previous work, it still performed admirably given the number of classes and the complexity of the task. It suggests that a larger network trained from scratch could potentially come quite close to equaling a model trained with hand designed features.

However, our largest regret from this project was that we could not achieve better performance with a fine-tuned model, and in future work we would definitely want to improve on this. There are a number of options we could pursue on this front. First off, it would be interesting to experiment with more hyperparameter combinations and larger batch sizes on a faster machine. Secondly, it would be worthwhile to fine-tune a larger model than the one we used and perhaps also to allow even more layers of that model to be adjusted. If our suspicions about the lower level features being the main obstacle to achieving better performance are correct then using a model which is larger could provide us with a richer set of low level features. We would have liked to investigate whether our hypothesis that the low level features learned for a flower classification task are significantly different from the low level features learned from a dataset like CIFAR-10 is in fact correct. This sort of investigation could help tell us whether our attempt to fine-tune a larger net trained on a different dataset were doomed from the start or if we simply made mistakes tuning our model?s hyperparameters.

Another experiment we would like to perform is training a larger model from scratch. Our results from the first model we trained from scratch are encouraging, but the plateaus we hit as the number of iterations grew indicate that perhaps a more powerful model is needed. However, the fact that we are already seeing signs of overfitting on this smaller model means that either a larger dataset or a higher dropout parameter would have to be used if a larger model was employed (or more aggressive data augmentation).

There are a number of other goals aside from increasing classification accuracy that we think would be interesting. We are very interested to see if a successful model for this classification task could be built into a successful model for a localization or detection task.This would require some work to modify our dataset, since it is currently not designed for a localization task, but we are curious because our intuitive notion is that it should be very easy to localize a flower from other objects. We wonder if the distinctiveness between flowers and their backgrounds is as stark for a computer vision algorithm as it is for the human visual system.

Finally, our primary motivation for picking this topic was that we wanted to be able to take a picture of a random flower in the wild and have our phone tell us what kind of flower it was. Building an application to do this, along with generalizing the model to work with many more species than the ones in our dataset, would be a somewhat ambitious but fun future undertaking.

6. References

1. Hernndez-Serna A, Jimnez-Segura LF. (2014) Automatic identification of species with neural networks. `https://doi.org/10.7717/peerj.563`
2. Seung-Ho Kang, Su-Hee Song, Sang-Hee Lee, Identification of butterfly species with a single neural network system, Journal of Asia-Pacific Entomology, Volume 15, Issue 3, September 2012, Pages 431-435, ISSN 1226-8615, `http://dx.doi.org/10.1016/j.aspen.2012.03.006`.
3. A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In NIPS, 2012.
4. Nilsback, M-E. and Zisserman, A. Automated Flower Classification over a Large Number of Classes Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing (2008)