MiniPlaces Challenge
Team *Team2*

Martin Nisser
MIT CSAIL
6.869
nisser@mit.edu

Kenneth Friedman
MIT CSAIL
6.869
ksf@mit.edu

Abstract

For computers to understand the physical world, computer vision should be used to visual recognition tasks. In this MiniPlaces challenge, we researched and developed methods to improve scene recognition. First, we researched the fundamentals of deep learning and developed a plan of methods to examine. Then, we developed networks and trained them on virtual servers. In the end, we developed a network that was able to achieve a result of 76.1% accuracy on the Top5 scene classification test data.

1. Introduction

Over the last few years, visual recognition tasks such as image classification have seen a surge in performance with the rise of large datasets and computational power combined with deep learning techniques. Deep neural networks, and convolutional neural networks (CNN) in particular, have contributed greatly to the advances in the state of the art of computer vision tasks. In an effort to gain a deeper understanding of these techniques, this paper highlights the authors' efforts to improve upon some baseline CNN models. The particular challenge is that of the MiniPlaces Challenge, an image classification task based on a subset of the Places2 dataset [15], consisting of 100,000 training images spanning 100 different categories, with an additional 10,000 images apiece for validation and testing.

2. Methods of Testing

2.1. Experiment Hardware

We initially ran experiments on our personal laptops to confirm a proper installation of TensorFlow and an understanding of how to run the code. As soon as we successfully ran a training session, we realized that more performant hardware was necessary to run sessions in a reasonable amount of time.

Therefore, we began running experiments using GPU-optimized servers on Amazon Web Services (AWS). Specifically, we ran tests on various (virtual) hardware systems; AWS's EC2 instances. In the early stages of experimentation, we ran $g2.2xlarge$ and $g2.8xlarge$ servers. However we ran out of memory on larger batch sizes of 256 and longer iterations above 10,000. For the final period of testing, we moved to $g3.2xlarge$ and $g3.8xlarge$, which offer fewer overall features, but a significant increase in RAM. As described in later sections, besides computation time, RAM was often a limiting factor in experimentation.

2.2. Experiment OS/Software

On our EC2 instances, we ran a Ubuntu Linux Amazon Machine Image (AMI). In October, we selected the default "deep learning" Linux AMI which came with TensorFlow for the GPU pre-installed. In November, AWS switched the default deep learning instance to use the Anaconda python distribution. However, we were able to recover the AMI first used, which is now called *Deep Learning AMI Ubuntu Linux - 2.4_Oct2017* or more directly: $ami - 37bb714d$. This AMI enabled us to run our code (in addition to any TensorFlow code with GPU optimization) without downloading and compiling TensorFlow on every new instance.

We interacted with these virtual servers using SSH commands, and we used *Transmit.app* for file transfer on a Mac. We also used Putty SSH and WinSCP for interaction and file transfer on Windows. On the virtual servers themselves, we used the *Screen* application to be able to start training sessions, disconnect from SSH, and allow the python processes to continue running.

2.3. Custom Meta Scripts

Throughout the course of our testing, we also built some "metasystem" scripts for rapid prototyping and for viewing results faster. The first (and required) system was a script to convert the output of a given neural net to output the top 5 predictions for every image, and then write those results to an output file. This script is knonwn as *testr.py*.

In addition, we added a script to write the training and validation results to a log every 50 iterations in order to track the loss and prediction accuracy of the networks. We built this into our base code and is apparent in the training file of every network we tested. We built a companion script, known as *formattr.py*, that took the output logfile and converted the results into a CSV (comma separated value file) to easily graph the results at any given point, even if the network was still running.

Finally, we created a "ensembling" script (see section 3.5). This was by far the most complicated one. It requires outputing all 100 predictions of the network for every test image, with an extra softmax layer to convert weights into probabilities that sum to 1. This script is known as *ensemblr.py*. Then, for each image, each weight of the softmax output is averaged with the same image weight of the other output networks being "ensembled". Finally, those weights are re-sorted and converted to the top 5 predictions of the average of the probabilities. That script is called *mergr.py*.

2.4. Division of Work

Martin and Kenny both contributed to modifying the code, training sessions, and writing this report. Kenny had a stronger background in the use of Amazon AWS and Linux (ssh, screen, etc), while Martin had a stronger math/theoretical background. Therefore, Kenny created most of meta-scripts, and did early tests getting the code running on EC2, and various experiments with dropout, batch size, and data augmentation. Martin made major contributions to the biggest modifications of the architecture of our networks: changing the number of layers, size of convolutions, and general architectures (AlexNet, VGG, etc). Because our best result used ensembling, we both literally contributed to the best submission result. Martin and Kenny contributed to the written report equally.

3. Approach

We start with the base implementation of AlexNet [9] by Krizhevsky *et al*. in TensorFlow that was provided as part of the challenge. From there, we made modifications to a variety of parameters and metaparameters, and completed several major overhauls of the architecture. These modifications are described below.

3.1. Dropout

Hinton, et. al showed that "dropout" can be a useful technique to prevent overfitting [12]. Dropout is the concept of randomly omitting connections within a neural network in an attempt to prevent co-adaption (multiple neurons detecting the same feature).

Dropout adds a single hyperparameter to the system: p, the probability of retaining a unit in the network. When $p = 1$, all units will always remain in the system, and dropout will not exist. When $p = 0$, all units will dropout, and there will be no connections to learn from data. In between those extremes, a fractional portion of the units are dropped. Hinton, as well as Baldi and Sadowski [1] recommend a dropout of $p = 0.5$. We experimented with this dropout, along with other p values (which can be seen in our Experiments section).

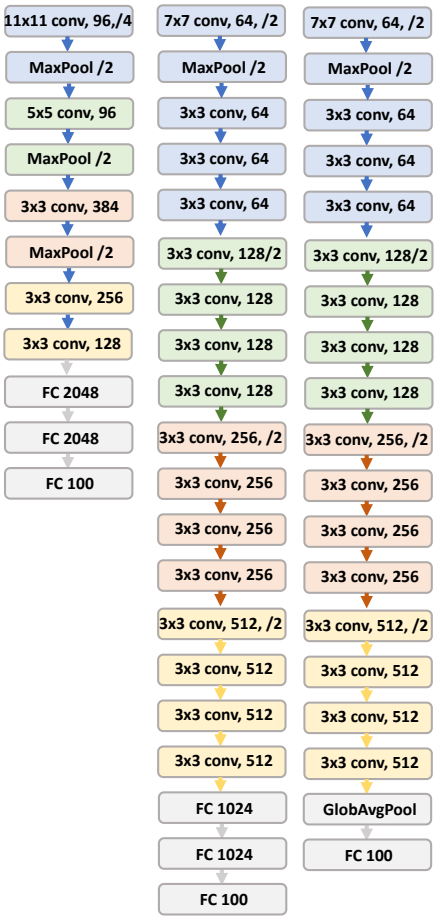

Figure 1. Best three network architectures. Each convolutional layer (conv) is followed by batch normalization and a ReLU activation. (Left) A scaled-down permutation of AlexNet: AlexBsmall. (Middle) A VGG-inspired model: VGGhyb. (Right) A fully convolutional VGG-inspired network: VGGconv.

3.2. Learning Rate

We experimented with various starting learning rates. Near the end of training, we had determined that the seemingly optimal path was to start at at learning rate of 0.001, and then descrease it by a factor of 10 when the derivative of the loss function approached zero leading to a plateauing of the training and validation accuracies. That is, when it stopped "learning" with back propagation, the parameter was adjusted to make smaller steps. In addition to this, we

used the Adam optimizer [8], which changes the learning rate adaptively during testing.

3.3. Architecture

A number of network architectures were experimented with in the course of the MiniPlaces Challenge. To begin with, the baseline AlexNet model in TensorFlow was trialled, before rapidly progressing to AlexB the Batch Normalization implementation. Next, The AlexB code was modified to give a network dubbed AlexBDeep: this network increased the depth of AlexB at the expense of size of the kernels of the convolutions; Specifically, two convolutional layers were added and all layers were given a kernel size of 3x3. In an effort to further reduce the dimensionality of the network to speed up training, AlexBDeep1 was then created from AlexBDeep by injecting 4 additional convolutional layers at even intervals of AlexBDeeps convolutional layers; these 4 layers had kernels of size 1x1 with an output half the size of the input, reducing dimensionality of the network by combining feature maps [2]. AlexBsmall was then designed in an attempt to reduce the overfitting of the previously implemented permutations of AlexNet; this network halved the number of parameters of earlier networks by decreasing the size of the fully connected layers. Following the stagnation of accuracy gains from modifications to AlexNet, the code was augmented to produce VGGhyb; an architecture modelled on the VGG [11] architecture. This network has 16 convolutional layers (first filter size 7x7, all subsequent size 3x3), ending with two fully connected layers of size 1024. Downsampling of the input image down to final size of 7x7 is performed by halving the feature map size using strides of 2 every 4 convolutions; this is done at the same time as the number of filters are doubled, which preserves time complexity at each layer and retains same number of filters for a given feature map size. Finally, a fully convolutional version of VGGhyb was designed called VGGconv, using a global average pooling in place of the fully connected layers which is then connected to the output. In addition to its computational efficiency [10] and advantageous spatial encoding, this architecture is furthermore useful for its imperviousness to the size of the input image.

Minibatch sizes for all implementations were set between a minimum of 32 and a maximum of 256, opting for the largest minibatch size that the available server would allow. The number of training iterations was set to 50,000 and terminated manually once top5 validation accuracy plateaued. The Rectified linear unit (ReLU) was used as an activation function and Adam [8] was chosen as a first-order gradient-based optimization, using cross-entropy loss and finally a Softmax function to calculate the final probabilities.

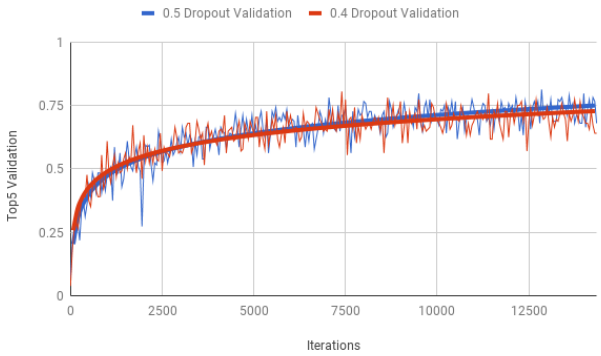

Figure 2. In this final test, the validation results of Top5 are compared between a 0.5 dropout hyperparameter, and a 0.4 dropout. All others tested (0.3, 0.7, 0.8) were significantly worse than both of these. Though the data is noisy, a best-fit log trend line shows that it is likely that 0.5 will succeed in the long run, which intuitively follows from the research papers on dropout (see Section 3.1).

3.4. Data Augmentation

MiniPlaces is has a training data set of 100,000 images, which is quite small compared to state of the art data sets, such as Places2. When the training data is small compared to network, the limited data set will prevent the network from generalizing as well as it could. Therefore, more data can be artifically made by augmenting the current data to provide slight and random modifications. This process is known simply as data augmentation [14]. Simple data augmentation such as random crops and flips are standard practice [9]. These prevent the network from learning classification based on specific locations of features or patterns with an image and prevents a specific ordering of the pixels to recognize a classification. Further data augmentation methods can be created to provide even more variation in images. Rotation, colour jittering, and Fancy PCA [13] can also aide in creating the appearance of much more training data than actually exists. Rotation, color changes, and PCA don't change a human's ability to recognize the scene however. Two other data augmentation changes are blurring and adding random noise (often known as salt and pepper) to the images. The "search space" is incredibly broad for data augmentation techniques. therefore, we only trial a small sampling of the possible augmentations. By the end of our testing, we augment the data with random crops, flips, scaling, rotation, and salt-and-pepper noise.

3.5. Ensembling

It was shown by Hansen and Salamon [5] that performance can be increased by training a number of different neural networks on the same dataset and then ensembling

them. Furthermore, it was shown that it may be better to use many models rather than all those available [16]. Better results can furthermore be achieved than that gained through just averaging models by performing a weighted average according to the performance of each individual network on the validation set [4]. Our final accuracy scores thus rests on the weighted means of the best networks found according to their top5 accuracy on the test data.

4. Experiments

At the beginning of the project, early experiments involved trying to run AlexNet on AWS, as well change the batch size to be able to train data for long enough without getting a "Resources Exhausted" error (out of memory on the GPU) or a "MemoryError" (running out of RAM after a certain number of iterations). We experimented with these methods in a variety of ways. Our general workflow was to first understand the batch size relation to iteration speed, training, and RAM. Next, we experimented with various dropout changes. We then used data augmentation techniques. Finally, major network architecture changes were implemented. While there were only minor effects of changing the dropout hyperparameter, p, than we were expecting, 0.5 did appear to produce the best results after 30,000 iterations (compared to 0.3, 0.4, 0.7, and 0.8). See Figure 2 for a graph of the experiment. Decreasing the learning rate did show noticeable improvements in Training and Validation Top5 results for any given model. As you can see in Figure 3, there is a spike in success after the learning rate was lowered by a factor of 10.

Moreover, data augmentation was attempted in four different styles:

- Flipping and Cropping
- Flipping, Cropping, and Rotating
- Flipping, Cropping, Rotating, and Gaussian Noise
- Flipping, Cropping, Rotating, and Salt-and-Pepper Noise

The rotation augmentation did not work as well as we had expected, and the Gaussian noise and Salt-and-Pepper noise made minimal difference. It did take much longer to reach a high success rate in training, but by the time training top5 accuracy was high, overfitting took place (noticeable by a distinct gap between a significantly lower validation accuracy and a higher training accuracy). We implemented rotation by randomly rotating the image between $-90°$ and $90°$ and then scaled and cropped the image a second time to remove the black bars in the corner of the rotated image. We then tried a reduced rotation range of $-30°$ to $30°$. This appeared to work slightly better, and was used in a result that was included in the final ensemble result.

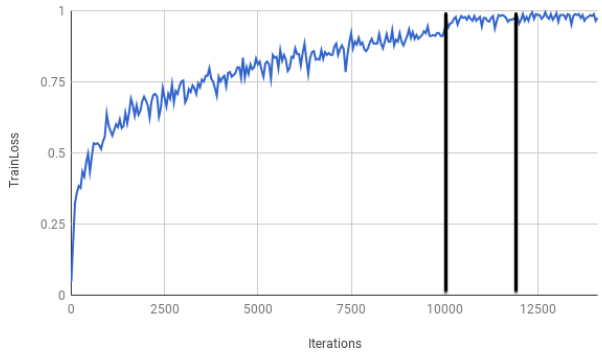

Figure 3. In this test, as the training stopped improving, we decreased the learning rate twice, each time by one order of magnitude. We decreased the learning rate manually: once at 10,000 iterations and once at 12,000 iterations, noted by the black bar lines. You can visibly see the distinct jump in performance after these changes were made.

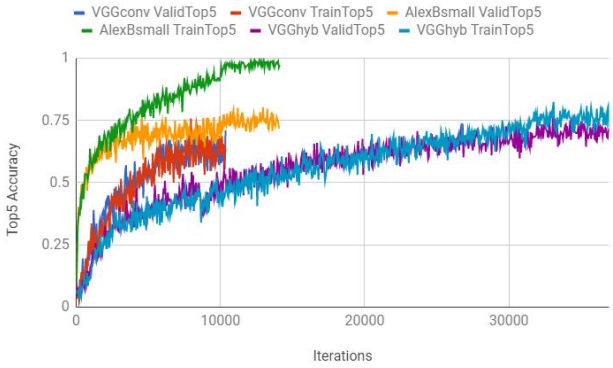

Figure 4. Top5 accuracies in training and validation for the top 3 performing architectures.

Model	Top5 Error
Ensemble	0.239
AlexBsmall	0.2458
VGGhyb	0.2539
VGGConv	0.2656

Table 1. Top5 error for the best 3 network architectures, in addition to a weighted ensemble of all three.

Ensembling proved to be a reliable method of improving overall accuracy. It can first be noted that ensembling of the predictions of architecturally similar networks produced only a marginal increase of 0.2% in the top5 classification accuracy on the test dataset. However, in a slight improvement, an ensembling of the best three architectures produced in improvement in the top5 accuracy of the best network by 0.6% as shown in Table [1].

5. Discussion

The base AlexNet implementation showed a very slow rate of learning. Replacing this model with the AlexNet implementation with batch normalization yielded far better results, ultimately resulting in a Top5 error of 0.239 on the test data. It was found that applying a batch normalization layer between convolutions and activation layers resulted in an increase in robustness to poorly initialized weights and allowed for higher learning rates to be used [7]. With AlexNet, we were eventually able to reach 100% Top5 accuracy in the training data, but the model was clearly overfitting as seen from the vertical offset between training and validation accuracies. We then spent a long time working to reduce overfitting by experimenting with dropout, data augmentation, and ensembling, ultimately increasing the accuracy by approximately 5% using these methods alone. Unfortunately, our data augmentation plans never worked as well as we had hoped. As described in Section 4, we attempted a variety of data augmentation systems, but none performed as well as simple flipping and cropping. A small (60° range) rotation did help one of our models, but it was not tested with an identical model without the rotation. Eventually, we made significant changes to the Alexnet network architecture, including a VGG-based model which showed significant improvement, giving our best score. I addition, we created a fully convolutional model. This convolutional model proved less accurate, likely as a result of sacrificing too many parameters in its implementation- it plateaus in its training very early, indicating a lack of complexity in the model and an insufficient number of parameters; further tailouring of the network could likely have helped. Conversely, the reduction in parameters helped prevent overfitting, which especially helped during the middle of this project, to achieve better-than-AlexNet performance for the first time.

5.1. Future Work

While increasing the depth of a network often leads to an increase in performance, at a certain depth an accuracy saturation is often reached that is not caused by overfitting; an alternative here is to incorporate a residual learning architecture as introduced in ResNet [6]. A simple additional hyperparameter to experiment would be Momentum; this is a term that acts as a leaky integrator, incorporating past information into the current gradient calculation which can help the stochastic gradient descent algorithm to jump out of local minima more quickly. Other possible improvements include increasing the complexity of the ensembling using bagging or boosting [3]. We would also hope to experiment further with data augmentation. We discovered many possible data augmentation ideas (discussed in Section 3.4), but only had time to test a few of these possibilities in the search space. In particular, we would like to try color jittering and Fancy PCA on top of those incorporated in this work.

5.2. Conclusion

In this project we researched and implemented a deep learning system to train scene recognition on the MiniPlaces dataset. We used a variety of techniques such as dropout, learning rate modifications, architectural changes, and data augmentation to improve our model. Our best result was achieved with an ensemble of various high-achieving models. Finally, we discussed future directions we could take to improve our model with more time and resources. Our final model performed with a 76.1% accuracy on the Top5 classifcation test data.

References

[1] P. Baldi and P. J. Sadowski. Understanding dropout. In *Advances in Neural Information Processing Systems*, pages 2814–2822, 2013.

[2] S. Bell, C. Lawrence Zitnick, K. Bala, and R. Girshick. Inside-outside net: Detecting objects in context with skip pooling and recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2874–2883, 2016.

[3] T. G. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 40(2):139–157, 2000.

[4] X. Frazao and L. A. Alexandre. Weighted convolutional neural network ensemble. In *Iberoamerican Congress on Pattern Recognition*, pages 674–681. Springer, 2014.

[5] L. K. Hansen and P. Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12(10):993–1001, 1990.

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[8] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[10] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[11] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[12] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[13] L. Taylor and G. Nitschke. Improving deep learning using generic data augmentation. *CoRR*, abs/1708.06020, 2017.

[14] D. A. Van Dyk and X.-L. Meng. The art of data augmentation. *Journal of Computational and Graphical Statistics*, 10(1):1–50, 2001.

[15] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva, and A. Torralba. Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[16] Z.-H. Zhou, J. Wu, and W. Tang. Ensembling neural networks: many could be better than all. *Artificial intelligence*, 137(1-2):239–263, 2002.