Image Classification with Pyramid Representation and Rotated Data Augmentation on Torch 7

Keven (Kedao) Wang
Stanford University
kvw@stanford.edu

Abstract

This project classifies images in Tiny ImageNet Challenge, a dataset with 200 classes and 500 training examples for each class. Three network architectures are experimented: a traditional architecture with 4 convolutional layers + 2 fully-connected layers; a Tiny GoogleNet with 3 inception layers; and a pyramid representation-based network. Tiny GoogleNet achieved the highest top-1 validation accuracy of 47%. Work is done to reduce overfitting. Dropout improves validation accuracy by 10%. Data-augmentation of random crop and horizontal flip increased validation accuracy by 10%. Rotation does not appear to improve validation accuracy. Pyramid representation shows significant computational efficiency, achieving similar top result 240% faster computation time per batch. Training accuracy converges at 65 - 70% for all three networks. Future work is to increase expressive power of network. Training was done on Torch 7 with Facebook's Deep Learning Extension.

1. Introduction

The task with image classification is to assign an image with a label from a set of classes, and strive for the highest accuracy possible. This project uses the Tiny ImageNet classification challenge in CS231N class. This dataset is a reduced version of the ImageNet Challenge dataset. Tiny ImageNet dataset has 200 classes. Each class has 500 training samples, 50 validation samples, and 50 test samples. Each image is 64×64, with bounding box of object specified. Each image has exactly one class of interest. The metrics of interest is top-1 accuracy.

This project uses Convolutional Neural Networks, which assumes local image features can be extracted the same way regardless of location - i.e. the same 3×3 filter can be used to extract features from anywhere on the image. Training is done on one Nvidia K520 GPU with 4Gb memory.

This project uses Torch7 library with Deep Learning CUDA Extension (fbcunn) [2], the open-source library by Facebook to construct, train, and test the network.

2. Background

2.1. History

Convolutional Neural Networks was first put into practical use for MNIST dataset (60k handwritten digits) classification in 1995 [5]. LeNet 5 achieved near-human-level test error rate of 0.9% on this 10 class classification task.

CNN was first used for large-scale image classification in 2012 for ImageNet challenge [4]. The test error rate of 17% is almost half of the error rate of previous year's models, which was based on non-CNN approaches.

2.2. Convolutional Neural Networks

2.2.1 Convolutional Layer

Convolutional layer consists of small filters (usually of size $3 \times 3, 5 \times 5$) to convolve in 2D space on an image. Each filter is 3D in shape, with depth equal to the depth of input data (3 in the first layer, for 3 RGB channels). Each filter outputs a $1 \times 1 \times depth$ column. Each filter, depending on the weights trained, is responsible for a particular task (horizontal / vertical edge detection, etc.) The reasons to use filters are:

1. It greatly saves parameters. Each convolutional filter has a small number of weights (e.g. a 3×3 filter on the first stage only has $3 \times 3 \times 3 + 3 = 27$ weights), regardless of the size of input.

2. It assumes local features can be extracted the same way regardless of location, which is true for object position that is randomly distributed.

The early convolutional layers are responsible for extracting low level features, with the late stages responsible for higher level features [10]. This is because each subsequent layer can 'see' a larger patch of the original image. For example, taking a filter size of 3×3 the first layer 'sees'

3×3 neighboring pixels, and 'squeezes' it into a single pixel column. The next layer can effectively see 5×5 neighboring pixels of original image. The later stages 'see' a growing window aggregated by previous layers.

2.2.2 Pooling Layer

A pooling layer effectively downsamples an input. Max pooling, which takes max value of a patch, is used in this project. The reasons for pooling layer are:

1. It saves computation for later stages by reducing the data size.
2. It enables later convolution stages to "see" a larger image patch.

2.3. Optimization

Batch Gradient Descent with momentum update is used. This technique adds inertia and friction to the update function, which is shown to converge more quickly than vanilla gradient descent.

$$v = momentum \cdot v - lr \cdot dx$$
$$x = x + v$$

2.4. Hardware progress

Today, computation is the biggest bottleneck for training large scale CNNs. Techniques of pooling is used largely to speed up computations. The number of weights used in LeNet in 1995 was 600k. The number of weights used by AlexNet in 2012 has grown to 100M. The training time still takes weeks on state-of-art GPU, which is optimized for highly parallel matrix multiplication tasks.

2.5. Open-source frameworks

There has been multiple open source frameworks to train CNNs on GPU. The most popular one is caffe, developed at Berkeley [3]. The advantage of Caffe is that it is plug-and-play, but it is not very configurable. On the other hand, the Torch 7 deep learning extension developed by Facebook [2] is supposed to provide much more flexibility with the layers, optimization process, data augmentation, etc. It also supports data- and model-parallelism across GPUs. Although I was not utilize that in this project (due to Terminal.com having only a single GPU device per instance [9]), it is the trend going forward.

3. Approach

3.1. Framework

Torch 7 with Deep Learning CUDA Extension is used for this project. The extension has the following advantages:

- It offers great flexibility in defining the optimization criteria, data augmentation, network architectures, since all of these are defined in Lua.
- Lua has smaller memory footprint than python. The interpreter is faster than python's.
- It supports model and data-parallelism. As CNNs get bigger, multi-GPU training will be the trend to scale. A typical model in this project takes 8 hours to train till convergence. Leveraging 2 GPUs means doubling the iteration speed for choosing network architectures and hyper-parameters.
- It supports reading from file system directly via multiple threads. No intermediate data store is necessary.
- It uses CUDNN, the experimental deep learning framework released by NVidia. In the long term this will be faster than naive GPU implementation, since the optimization is done at a lower level [6]. Currently it only offers 1.2x performance improvement.

It has also these disadvantages:

- Lua has a bit of a learning curve. Doing the same task in python would easily be 2x to 4x faster, given my familiarity with Python and better IDE / debugging tools available.
- Lua has a much smaller developer community, and therefore a smaller library ecosystem / documentation. I did not find this to be an issue in this particular project. Since for all the tasks, such as image manipulation, there are readily available Lua packages.

3.2. Architectures

Traditional, Tiny GoogleNet, and Pyramid representation architectures are experimented with.

3.2.1 Traditional

The following traditional architecture produces the best result among traditional architectures. It has 4 convolutional layers + 2 fully connected layers. It is a scaled-down version of AlexNet. [4]

3.2.2 Tiny GoogleNet

A downsized 3-layer GoogleNet is used. GoogleNet argues that features generated at different scales are equally interesting [8]. It therefore concatenates output from $1 \times 1, 3 \times 3, 5 \times 5$ into the output feature vector. An additional pooling output is used to allow for higher level features to be extracted by the next layer. In this project, each 'inception' layer uses filter sizes of $3 \times 3, 5 \times 5, 7 \times 7$ to further increase the view window.

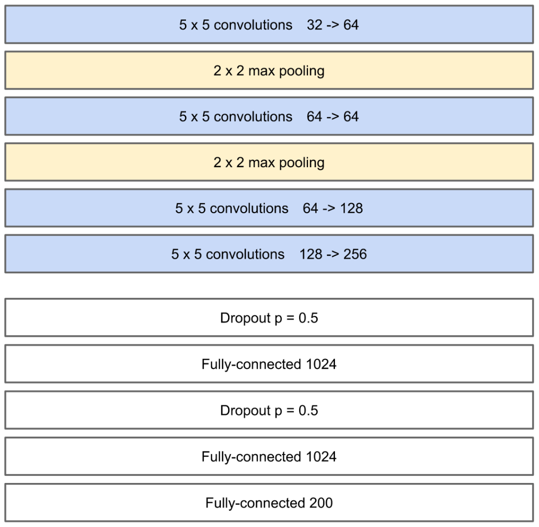

Figure 1. Traditional architecture: 4 convolutional layers + 2 Fully-Connected Layers.

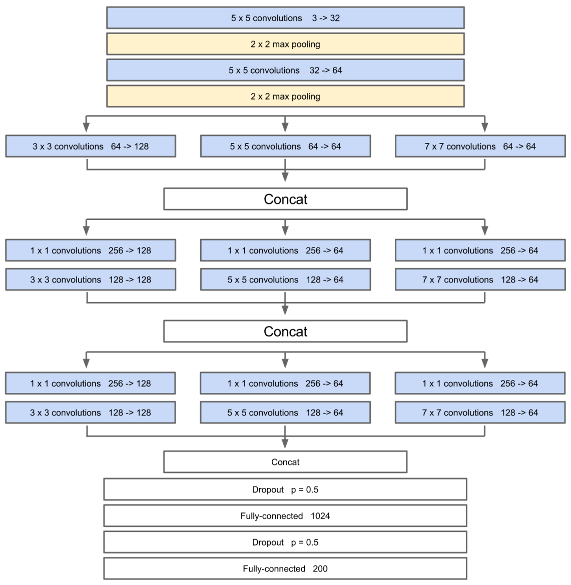
5 x 5 convolutions 3 -> 32
2 x 2 max pooling
5 x 5 convolutions 32 -> 64
2 x 2 max pooling
3 x 3 convolutions 64 -> 128
5 x 5 convolutions 64 -> 64
7 x 7 convolutions 64 -> 64
Concat
1 x 1 convolutions 256 -> 128
3 x 3 convolutions 128 -> 128
1 x 1 convolutions 256 -> 64
5 x 5 convolutions 128 -> 64
1 x 1 convolutions 256 -> 64
7 x 7 convolutions 128 -> 64
Concat
1 x 1 convolutions 256 -> 128
3 x 3 convolutions 128 -> 128
1 x 1 convolutions 256 -> 64
5 x 5 convolutions 128 -> 64
1 x 1 convolutions 256 -> 64
7 x 7 convolutions 128 -> 64
Concat
Dropout p = 0.5
Fully-connected 1024
Dropout p = 0.5
Fully-connected 200

Figure 2. Tiny GoogleNet: 3 inception layers to capture multi-scale image features.

3.2.3 Pyramid Representation

The pyramid representation of images is also explored. An alternative approach to capturing larger scale image features is to downsize the input first, and then use a smaller filter size on the downsampled image. For example, instead of applying a 5×5 filter on original image, a 3×3 filter is applied on a max-pooled image (size 2 stride 2). The viewing window proportion remains the same. The advantage is that the smaller filter computes much faster than a larger filter.

In this pyramid architecture, the 64×64 image is represented in a pyramid scheme as $64 \times 64, 32 \times 32, 16 \times 16$. Each representation undergoes three layers of convolution with 3×3 filters. This project's pyramid layer takes 64×64 input, and produces 16×16 output. The results are concatenated depth-wise into a single feature matrix. This brings two main challenges:

- Downsampling images makes a branch produce smaller size than the original image. This issue is addressed by adding pooling layers after convolutional layers in non-downsampling branches, essentially forcing higher-resolution branches to output the same size as the lowest-resolution branch. For example, the non-downsampling (64×64) branch uses two max-pooling layers, the half-downsampling branch (32×32) uses one max-pooling layer, while the quarter-downsampling branch (16×16) uses no max-pooling layer.

- The output size to a pyramid layer is smaller than its input size. This brings challenge to applying pyramid layers multiple times, as the output size will eventually be too small. In this project, only one pyramid layer is used.

In order to capture the image feature at even a larger scale, a fourth branch which down samples to 8×8 is used. In order to match the different sizes (8×8 vs 16×16), the output matrices are first reshaped to a one-dimensional vector, and then concatenated into a single feature vector.

3.3. Overfitting

Because of the small number of training examples (100k), overfitting is very likely on deep networks.

3.3.1 Reducing network size

Overfitting happens when a network's expressive power is too great for the amount of training data. The symptom is as training accuracy approaches 100%, the validation accuracy remains low. Reducing the depth of a network can enable a network to continue learning. Krizhevsky used cropping and flipping to increase training data size by a factor of 2048, on the 1.4 Million ImageNet training samples [7], yet still only used a network with 5 convolutional layers [4]. Therefore in this project, each network is limited to at most four convolutional layers. Without data augmentation, almost all three network architectures specified above overfit with training accuracy exceeding 99%.

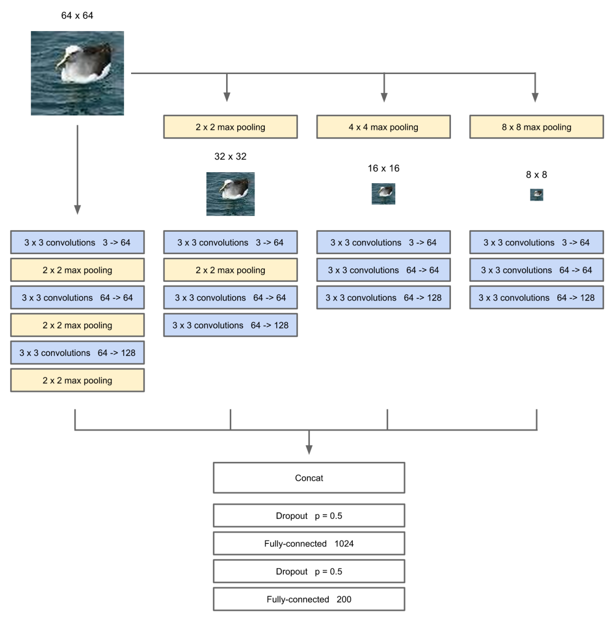
64 x 64
2 x 2 max pooling
4 x 4 max pooling
8 x 8 max pooling
32 x 32
16 x 16
8 x 8
3 x 3 convolutions 3 -> 64
2 x 2 max pooling
3 x 3 convolutions 64 -> 64
2 x 2 max pooling
3 x 3 convolutions 64 -> 128
2 x 2 max pooling
3 x 3 convolutions 3 -> 64
2 x 2 max pooling
3 x 3 convolutions 64 -> 64
3 x 3 convolutions 64 -> 128
3 x 3 convolutions 3 -> 64
3 x 3 convolutions 64 -> 64
3 x 3 convolutions 64 -> 128
3 x 3 convolutions 3 -> 64
3 x 3 convolutions 64 -> 64
3 x 3 convolutions 64 -> 128
Concat
Dropout p = 0.5
Fully-connected 1024
Dropout p = 0.5
Fully-connected 200

Figure 3. Pyramid Representation: represent an image at 4 different scales. 4 branches with 3 layers each.

3.3.2 Dropout

Dropout is done before each fully-connected layers. Dropout probability of 50% is used. This is extremely effective, and improved validation accuracy by 10%.

3.3.3 Data Augmentation

Work is done to generate more training examples.

Crop & Flip: 56×56 Random cropping is performed on 64×64 input. The ratio of $56/64 = 0.875$ follows the ratio of $224/256$, as used by AlexNet and GoogleNet. Horizontal flip is also performed. Cropping and horizontal flipping is done at random on training time. At test time, 10 images are fed into the model (4 corner crops + 1 center crop, each flipped). They are performed on CPU by multiple donkey threads, which does not add work onto GPU.

The 10-fold increase in training data significantly improved validation accuracy by 10%. At validation time, 10 images accuracy is only roughly 2% higher than 2 images accuracy (center-crop + flip).

Rotation: CNNs are highly sensitive to input rotations, as shown by [8]. Therefore training inputs are rotated between -8 to 8 degrees by random, with empty space padded with zeros, to provide more dynamic training samples. -8, 0, 8 discrete degrees of ration is used in this project, since it increases the amount of training examples relatively only slightly (by a factor of 3), and does not take too long to converge. Rotation is added in both training and validation time.

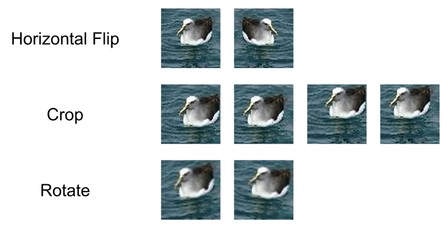

Figure 4. Data augmentation: 56×56 crop at 4 corners; horizontal flip; -8, 8 degree rotation.

Experiment is done on different rotation schemes, including different rotation ranges (e.g. -22.5 to 22.5 vs. -8 to 8 degrees), and granularity (continuous vs. discrete). Rotation does not improve validation accuracy in this project, while it takes longer to converge. It is possible that the network has potential to reach higher validation accuracy, but is not trained to convergence. It is also possible that the increase of training examples is not enough. Greater range of rotation and finer granularity can be used.

4. Experiment

Tiny ImageNet dataset is used. 200 classes, each with 500 training examples, 50 validation examples, and 50 test examples are classified on. A single output label is generated for each sample, and is used to compute accuracy. Top-1 validation accuracy is used as criteria.

4.1. Hyper-parameter tuning

Cross validation is done on learning rate and L2 regularization. Random grid-search is performed. Since it is suggested that a network is more sensitive to certain hyper-parameters than others, performing random grid search covers more ground per parameter as compared to strict grid search [1]. It was found that the learning rate of 0.015 works well for most models, and is therefore chosen as default learning rate. L2 regularization does not help with validation accuracy. Weight scale initialization is taken care of by Torch 7 fbcunn. Momentum decay rate of 0.95 is used.

4.2. Accuracy

The three architectures achieved roughly the same outcome. Tiny GoogleNet with 3 layers produces the best validation accuracy of 46%, at a training accuracy of 70%. All three networks have training accuracy converging at 65% - 70%. Rotation augmentation does not improve accuracy. The Pyramid network produces 43% validation accuracy, with only 3 convolutional layers per branch, higher than the 42% accuracy by traditional architecture, which has 4 convolutional layers.

More layers could be added to potentially improve the

Network Architectures	**Accuracy**	**Accuracy**: dropout	**Accuracy**: data augmentation: crop + flip	**Accuracy**: data augmentation: rotate
4 Conv + 2 Fully-connected	21%	30%	42%	39%
Pyramid		29%	43%	39%
Tiny GoogleNet			46%	45%

Table 1. Tiny GoogleNet performs the best. Dropout and data-augmentation improves accuracy greatly. Rotation does not help much.

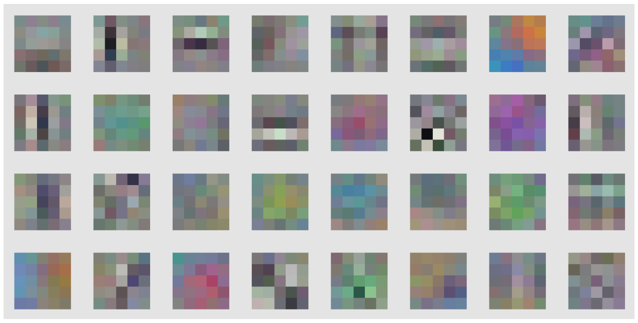
Figure 5. First layer weights of Tiny GoogleNet. Interesting features are learned.

Network Architecture	**Training time / batch (sec)**
4 Conv + 2 Fully-connected	0.838
Pyramid	0.443
Tiny GoogleNet	1.074

Table 2. Pyramid representations results in most efficient computation due to small filter size and downsampling.

expressive power of each network, and therefore improve accuracy. Tiny GoogleNet's training accuracy of 70% is relatively low, and therefore 3-layer GoogleNet is not expressive enough. As training accuracy converges toward 100%, validation accuracy could potentially improve as well.

4.3. Computational Efficiency

Training was done on all three network architectures. Tiny GoogleNet takes 8 hours to train till convergence. Pyramid network has the fastest training time of 0.443 seconds per mini-batch (256 images / batch), which is 240% faster than tiny GoogleNet. This efficiency is achieved by Pyramid network having small 3×3 filters, as compared to the $3 \times 3, 5 \times 5, 7 \times 7$ filter sizes used by GoogleNet. Introducing dropout and data-augmentation significantly increases time required to converge.

4.4. Features

The first layer weights of tiny GoogleNet show a clean representation of edges and dots. This means the network is learning the interesting information in an image.

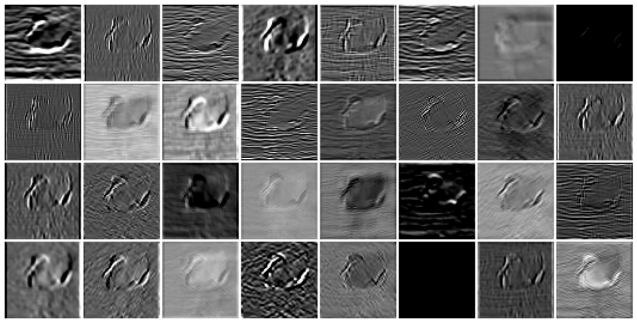
Figure 6. Convolved images after first 5×5 filter.

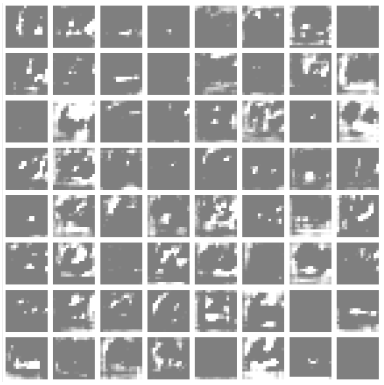
Figure 7. Convolved images after first inception layer 7×7 filter.

5. Conclusion

All three network types have training accuracy converging at roughly 65 - 70%. This seems low. More layers could be added to potentially improve the expressive power, and therefore increase accuracies of these models.

Rotation does not increase accuracy at all. It is not clear the reason why. It is possible that greater range and granularity in rotated examples is required, or that longer time is required to train network to convergence.

For the image classification task, the Pyramid representation of an image seems promising. The Pyramid network is very efficient, and produces almost identical results to tiny GoogleNet. Increasing the depth of Pyramid network can likely lead to better performance.

CNNs can be used to analyze the content of pixel-based data, and form surprisingly good understanding of the data as a whole. Armed with the tool of GPU, one can dive into understanding any other data format, as long as it can be decomposed into pixels. This includes audio, video, and beyond. There is no limit to where it can go from here.

5.1. References

References

[1] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[2] Facebook. Facebook's extensions to torch/cunn, 2015.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[4] 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.

[5] Y. LeCun, L. Jackel, L. Bottou, C. Cortes, J. S. Denker, H. Drucker, I. Guyon, U. Muller, E. Sackinger, P. Simard, et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective*, 261:276, 1995.

[6] Nvidia. Nvidia cudnn gpu accelerated machine learning, 2015.

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *arXiv preprint arXiv:1409.0575*, 2014.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[9] Terminal.com. The fastest linux cloud, 2015.

[10] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer, 2014.