Applying Partial Learning to Convolutional Neural Networks

Kyle Griswold
Stanford University
450 Serra Mall
Stanford, CA 94305
kggriswo@stanford.edu

Abstract

This paper will explore a method for training convolutional neural networks (CNNs) that allows for networks that are larger than the training system would normally be able to handle to be trained. It will analyze both the validation accuracy of each method and the time each method takes to train to determine the viability of this method of training.

1. Introduction

1.1. Motivations

One of the main bottlenecks in modern CNN design is the time and memory it takes to train a CNN. Even though increasing the depth of the network generally increases it's performance, researchers are not always able to take advantage of this fact due to system constraints. The new training methodology is designed to help mitigate these concerns.

1.2. Main Methodology

In order to help alleviate these problems, the new training methodology splits the desired CNN architecture into groups of layers (which we will refer to from this point forward as stages) and then trains each stage on the training set in sequence. From here on out we will refer to this training methodology as Partial Learning and the standard methodology of training every layer at once as Total Learning. The idea behind this is that training each stage seperatly will give comperable accuracy to training every stage together, but will require less memory and computational time, which will allow for larger networks.

1.3. Experimental Plan

In this paper, we will first evaluate CNNs with architectures that our system can train with Total Learning and compare the performance of these architectures between Total and Partial Learning. We will then implement architectures larger than our system can handle with Total Learning and see if the accuracy of these architectures with Partial Learning improves above the accuracy of the initial architecture with Total Learning.

2. Related Work

This paper does not deviate from the currently accepted methodologies of training CNNs (for example, those described by Karpathy [4]) except in using Partial Learning instead of Total Learning. The closest methodology to Partial Learning that we have found is in stacked auto-encoders (like those described in Gehring [2]), but there are several important differences between stacked auto-encoders and Partial Learning. The first is that while both methodologies split the training into stages, stacked auto-encoders use unsupervised training on the stages while Partial Learning trains each stage to classify the inputs into supervised classes. Another difference is that stacked auto-encoders are fine-tuned over the whole network after they are trained, while this fine-tuning is impossible for Partial Learning because it is used on networks too large to train all at once.

3. Approach

We will now go into the specifics of our experimental strategy.

3.1. Concrete Description of Training Methodology

We will start the training process by spliting the CNN architecture we want into stages, where the output of one stage is exactly the input to the next stage (that is, the next stage doesn't have any more inputs and the output is not sent to any other stages) (splitting the CNN into stages that form a directed acyclic graph instead of a linear chain is an experiment for another paper). We then start with the first stage and attach a fully connected linear classifier onto the end of it and train this CNN on the original training data inputs and outputs. When the training for stage 1 is done we remove the linear classifier and run each training example through stage 1 to get our new feature vector and use this

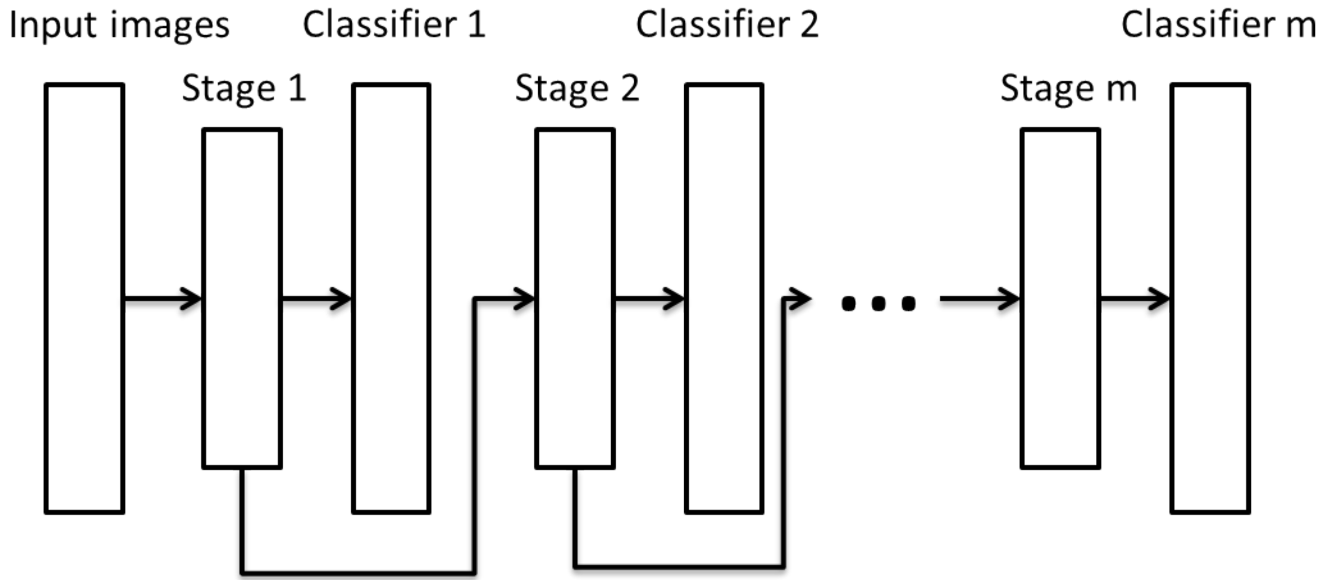

Figure 1. Diagram of the information flow in a CNN trained with Partial Learning

new feature vector (with the corresponding labels) to train stage 2 with the same process. We repeat this process until all of the stages are trained, at which point our final CNN is fully trained (Note that we keep the last stage's linear classifier to use as the final score generator). Figure 1 shows a diagram of this process.

4. Experiment

4.1. Data Set

The data set we will be using is CIFAR-10. Since this is an experimental methodology, it is more efficient to work with a smaller dataset to quickly run experiments than larger datasets that will take a long time to train on and may not give any good results.

4.2. Implementation Details

We used numpy [1] as the matrix algebra system to implement our architectures, along with initial code from the CS231n course at Stanford distributed from [3]. To train each architecture, we trained each stage with one epoch and with a batch size of 50. To find the hyper-parameters, we first found a learning rate and regularization that gave reasonable results on the baseline CNN - CNN 1. The learning rate and regularization we chose was a learning rate of 10^-4 and a regularization of 1. We used these as the initial hyper parameters for every stage (we didn't hand optimize each stage because then the results would depend more on how much time we spent hand-optimizing each stage than the actual merits of each one) and then used 10 iterations of random exponential search from a normal distribution with mean 0 and standard deviation 1 to find the best hyper parameters for each stage. We did this hyper-parameter tuning on a per-stage basis, so each stage could be trained with a different learning rate and regularization. Since this means that the training on identical prefixes for different architectures should give the same results for each prefix, we only trained each prefix once and used it for both architectures to save on time (for example the first stage for CNNs 1,6 and 8). After this was finished, we repeated the process, but this time with two epochs and only 5 additional hyper parameter iterations.

We then ran T-SNE (using an implementation we got from [5]) to analyze the features produced by our CNNs.

4.3. Experiments

We will be experimenting with the 8 CNN architectures outlined in Table 1. Note that in the table, Conv-n-m means that the layer is a convolutional layer with a filter size of nxn and has m filters. Pool-n means that this layer is a max-pool layer with an nxn viewing range and a stride of n. Also note that the linear classifier for each CNN is a fully connected linear classifier at the end of each stage to convert it into the scores for each label - this is just omited from the table for the sake of brevity.

Stage Number	CNN 1	CNN 2	CNN 3	CNN 4	CNN 5	CNN 6	CNN 7	CNN 8
1	Conv-3-32 Conv-3-32 Pool-2	Conv-3-32	Conv-3-32 Conv-1-32	Conv-3-32 Conv-3-32	Conv-3-32 Conv-1-32 Pool-2	Conv-3-32 Conv-3-32 Pool-2	Conv-3-32 Conv-1-32 Pool-2	Conv-3-32 Conv-3-32 Pool-2
2		Conv-3-32 Pool-2	Conv-3-32 Conv-1-32 Pool-2	Conv-3-32 Conv-3-32 Pool-2	Conv-3-32 Conv-1-32 Pool-2	Conv-3-32 Conv-3-32 Pool-2	Conv-3-32 Conv-1-32 Pool-2	Conv-3-32 Conv-3-32 Pool-2
3							Conv-3-32 Conv-1-32 Pool-2	Conv-3-32 Conv-3-32 Conv-3-32

Table 1. CNN archetectures we will experiment with.

4.4. Explaination of Architectures

CNN 1 is the standard CNN with 2 conv layers and trained all in one stage, which is analogous to Total Learning. CNN 2 is the same architecture as CNN 1, but trained with Partial learning instead. CNN 3 takes each conv layer in CNN 2 and converts it to a 2-layer Network-in Network layer instead of a linear layer. CNN 5 adds an additional pooling layer at the end of the first stage of CNN 3 to seperate each stage of Conv layers from each other. CNN 7 adds an additional third stage to CNN 5 with the same architecture as the other two stages. CNNs 4,6, and 8 are all extensions of CNNs 3,5, and 7 respectively where we convert the network-in-network layer into a standard 3x3 convolutional layer.

4.5. Evaluation Methods

We will mainly be evaluating our CNNs on their validation accuracy and training time. Specifically, we will be testing if CNNs trained with Partial Learning have faster training times than when trained with Total Learning, and whether architectures too large for Total Learning have higher accuracy than baseline architectures trained with Total Learning.

4.6. Expected Results

We anticipate that CNNs trained with Partial Learning will have slightly lower accuracy than CNNs with the same architecture but trained with Total Learning. The Partial Learning trained CNNs will have lower training time requirements than the Total Learning trained CNNs though, which should allow for the larger CNNs trained with Partial Learning to have similar memory and time requirements to the inital CNN trained with Total Learning, but with greater accuracy.

4.7. Actual Results

The results we got for best hyper-parameters for each stage, as well as the training time, training accuracy, and validation accuracy for those hyper parameters are detailed in Tables 2 and 3. We also ran T-SNE on the features produced by CNNs 7 and 8 from the 1 epoch and 10 hyper parameter iterations set of experiments. We didn't run it on any other architectures because CNNs 1-6 gave too many output dimemsions for our T-SNE implementation to handle, and the CNNs 7 and 8 in the 2 epoch case didn't have good validation accuracies, and thus their output features won't be linearly seperable like we are hoping to see. The results of those T-SNEs are in Figures 2 and 3.

5. Conclusion

5.1. Primary Analysis of Results

We first compare the time CNNs 1 and 2 take to see if Partial Learning makes the CNNs train faster. Looking at the results, we see that this is not the case though since the time for CNN 2 is either approximatly equal (in the 1 epoch case), or CNN 2 is slower (in the 2 epoch case). This is not to say that a more heavily optimized implementation would not give better results (eg. we did not have enough memory to convert every input image into the features of each stage all at once, so we had to do the conversion repeatedly for each batch), just that our implementation does not give a speed up.

Next, we compare the validation accuracies of CNN 1 and CNN 2. Since these two have the same architecture, the only accuracy differences between them will be due to the Partial vs. Total Learning. In each case, CNN 2 has lower accuracy, which is to be expected. In the 1 epoch case, the validation accuracy is only 3% lower, which is promising - this means that the accuracy cost of splitting the architecture into stages may be low enough to make Partial Learning viable. The 2 epoch case gives us a 6.5% difference in accuracy. This is not as good as the 3% accuracy from the 1 epoch case, but it is still low enough to make Partial Learning potentially viable.

Finally, we compare the accuracies of CNN 1 with the extended architectures of CNNs 3-8. Looking at the validation accuracies for each case tells us that only one CNN was able to beat CNN 1, specifically CNN5 in the 2 epoch

Result	CNN 1	CNN 2	CNN 3	CNN 4	CNN 5	CNN 6	CNN 7	CNN 8
Learning Rate	0.000069	0.000100/ 0.000721	0.003244/ 0.008321	0.000269/ 0.005026	0.000547/ 0.000111	0.000069/ 0.000957	0.000547/ 0.000111/ 0.000000	0.000069 0.000957/ 0.009278
Regularization	0.282346	1.000000/ 0.481662	0.153250/ 0.000687	0.074457/ 0.067618	0.131378/ 0.018597	0.282346/ 0.023609	0.131378/ 0.018597/ 0.875551	0.282346/ 0.023609/ 0.001889
Time (in seconds)	2767.89	2762.73	4750.57	6753.39	2881.07	5106.26	4265.47	7104.93
Training Accuracy	0.535	0.474	0.463	0.537	0.135	0.465	0.122	0.501
Validation Accuracy	0.540	0.510	0.460	0.531	0.144	0.512	0.127	0.531

Table 2. Results for the CNNs with 1 epoch and 10 hyper parameter iterations

Result	CNN 1	CNN 2	CNN 3	CNN 4	CNN 5	CNN 6	CNN 7	CNN 8
Learning Rate	0.000705	0.000100/ 0.000128	0.000146/ 0.000216	0.000340/ 0.012252	0.000761 0.017744	0.000705/ 0.000602	0.000761/ 0.017744/ 0.000021	0.000705/ 0.000602/ 0.000180
Regularization	0.018425	1.000000/ 0.302135	0.994257/ 0.226451	0.067439/ 0.105700	0.078767/ 0.004098	0.018425/ 0.006584	0.078767/ 0.004098/ 8.292548	0.018425/ 0.006584/ 1.096897
Time (in seconds)	5499.41	6436.17	8748.34	14565.57	5709.78	9950.23	8036.87	13911.29
Training Accuracy	0.654	0.529	0.201	0.508	0.677	0.549	0.113	0.161
Validation Accuracy	0.618	0.553	0.223	0.542	0.630	0.573	0.134	0.212

Table 3. Results for the CNNs with 2 epochs and 5 hyper parameter iterations

case, but since the difference is only 1.2% this could easily just be due to random chance. At first glance this seems to pretty clearly indicate that Partial Learning doesn't help increase the size of the architecture because the validation accuracy cost is just too high.

Looking closer at the value of the accuracies instead of just comparing them tells a different story though. Specifically, if you look at CNNs 5 and 7 in the one epoch case and CNNs 3,7,8 in the two epoch case, we see that their accuracies are abnormally low - too low to mearly be caused by it simply being a poor choice of architecture - these accuracies indicate that the networks were not trained properly. A first guess at a cause would be a bug in the code somewhere, but all of the architectures with low accuracy share all of their code with other architectures that did fine (eg. the code for the stages in CNN 5 and 7 is used by stage 2 in CNN 3, even though CNNs 5 and 7 have terrible accuracies with one epoch and CNN 3's accuracy is fine). This, coupled with the fact that the optimal learning rate for the last stage of CNN 7 with one epoch is so low that we can't tell it apart from 0, seems to indicate that these abnormal results are simply the product of poor hyper-parameters.

Looking at the way we chose our hyper parameters supports this further - we started by hand optimizing for CNN 1, so the initial hyper parameters already started with CNN 1 having the advantage. We then only used a random search with a relatively short number of iterations, which means that if the hyper parameters didn't start off at reasonable values for a given architecture, then they might never find a reasonable value through random search before we stopped trying hyper parameters, which would result in the abysmal accuracies that we see in our tables.

Additionally, if those architectures were affected by poor hyper parameters, there is no reason to think that the other architectures weren't also affected, albeit to a lesser degree. This means that the architectures we have might have been able to get better validation accuracies than CNN 1 if they had better hyper parameters, which would mean that Partial Learning might be viable.

This is still very speculative though - while the results do fit what we would expect if we had poor hyper parameter choices, there are still other explainations that could explain these results as well (eg. that the poor hyper parameters didn't affect the better architectures as much as we thought and even with better hyper parameters they would still not do significantly better than CNN 1). This means that while these results may indicate that Partial Learning may be viable, more study is needed to determine whether it is or not for sure.

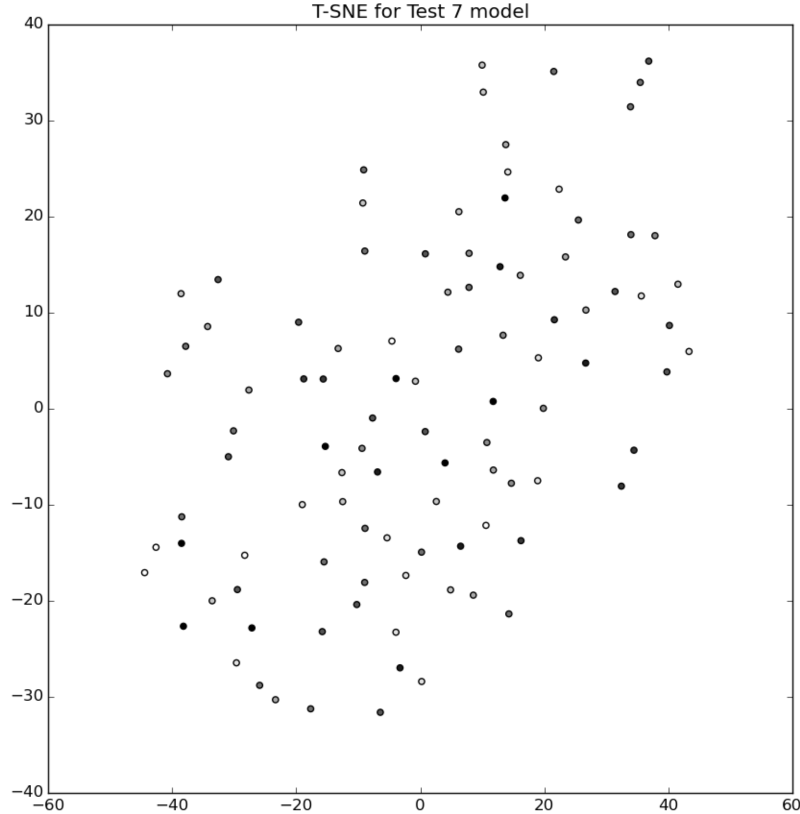

Figure 2. Results of T-SNE on CNN 7

5.2. Tangential Analysis of Results

First of all, if you look at the optimum learning rate/regularization for each stage in the multi-stage architectures, they seem to vary substantially from the optimums in other stages - sometimes by several orders of magnitude. This might simply be a fluke brought about by how we chose our hyper parameters, but considering the high variance of the optimum parameters between stages it would be worth experimenting with different learning rates and hyper parameters between different layers. It becomes even more interesting when you see that if you remove the architectures with abysmal validation accuracies (ie. those with poor hyper parameter choices), every architecture except one has the learning rate increasing and the regularization decreasing between stages (specifically CNN 6's learning rate and CNN 4's regularization in the 2 epoch case).

The construction of the T-SNE images was meant to show how much the CNNs had transformed the data to be linearly seperable, but they ended up still looking scattered with no decernable pattern.

5.3. Future Work

The first thing to do would be to use a better hyper parameter search algorithm. If the conclusions about poor hyper parameters being the cause of the poor performance is correct, this will give us results that show whether or not Partial Learning is viable or if the cost of splitting up the architecture is too high. We should be wary of attempting to hand optimize these though, because we could easily end up with results that correspond more to how much we hand optimized each architecture than whether each architecture was good or not.

We could also experiment more with using unsupervised learning techniques. Our experiments trained each stage by trying to classify the input images directly, which one might think would give the best parameters for the main classification task, but it could also be that unsupervised learning would be better for the intermediate stages. This could be a

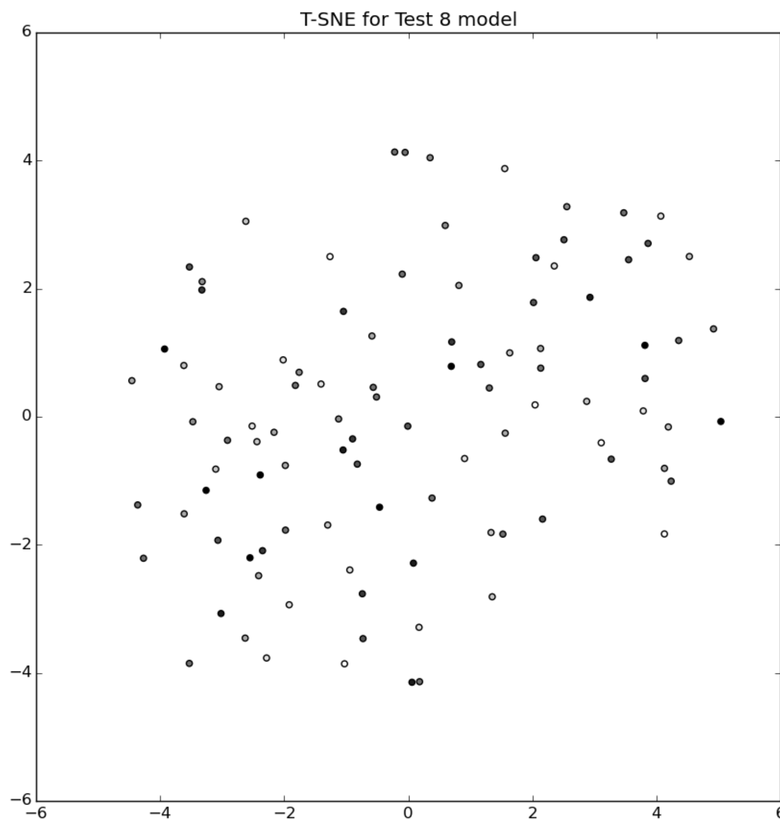

Figure 3. Results of T-SNE on CNN 8

relatively simple way to increase the performance of Partial Learning.

Also, our system was not able to hold both the original images and the derived features in memory at once, so we had to do the conversion for every batch. This is very inefficient, especially for multiple epochs, so if we are able to get a better system then we might be able to fix this. This would increase the training speed of Partial Learning, which could make it noticably faster than Total Learning.

It would also be helpful to be able to experiment with this on larger, more state-of-the-art architectures. This is because the information we find on small architectures might not generalize to the larger ones, and even if it did, using larger architectures could make any subtle differences between Partial and Total Learning more apparent.

It would also be worthwhile to experiment with different learning rates/regularizations for each layer. We could try independent learning rates for each layer, we could try having a base learning rate and then multiplying it by a constant for each layer we go up, or any number of other schemes. It might not work of course, but it would definitly be worth trying.

References

[1] N. developers. http://www.numpy.org/, 2013.

[2] Y. M. F. W. A. Gehring, J; Miao. Extracting deep bottleneck features using stacked auto-encoders. *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference*, pages 3377–3381, 2013.

[3] A. Karpathy. http://cs231n.github.io/assignment2/, 2015.

[4] A. Karpathy. http://cs231n.github.io/convolutional-networks/, 2015.

[5] L. van der Matten. http://lvdmaaten.github.io/tsne/, 2015.