CS 229 Final Project: Automated Curriculum Learning

Rahul Palamuttam rpalamut@stanford.edu
Nimit Sohoni nims@stanford.edu
Halwest Mohammad halwestm@stanford.edu

Abstract

(Motivation) A long-standing challenge in deep learning is accelerating training time, i.e. the time required for neural networks to learn near-optimal layer weights on complex datasets. It is well known that the human brain learns best by perceiving progressively more complex examples of a concept. Similarly, a curriculum of progressively more complex tasks could accelerate a neural network's training (Elman, 1993). Curriculum Learning attempts to train deep networks in phases of progressively more difficult objectives. We propose a methodology for automating curriculum learning using simple classifiers to estimate the difficulty of training examples and thereby automatically design a curriculum. We also present the results of applying this approach to two image classification tasks.

1. Motivation and Related Work

1.1. Deep learning

Deep learning has taken machine learning and computer vision by storm ever since AlexNet[8] greatly outperformed any other model at the time on the ImageNet[10] dataset using a deep convolutional network. Deep learning has since become a very active area of research and many questions in the field have not found definitive answers.

Compared to conventional machine learning algorithms, algorithms based on deep learning often require orders of magnitude more data (and computations) to train. In fact, one of the main reasons for the recent resurgence of neural networks is that the required calculations can be performed efficiently in parallel on GPUs, which have rapidly grown in power in recent years.

1.2. Curriculum learning

A natural question to ask is if we can somehow accelerate convergence during training. There are several approaches to this, like good weight initialization[4] and modified optimization methods[6, 1, 7]. *Curriculum learning*[2] takes a different approach: we attempt to feed the training data to the model in an order which encourages fast convergence.

Humans tend to learn more easily by building from simple examples. Similarly, when it comes to machine learning, and deep learning in particular, we might suspect that it is beneficial for an algorithm to be trained on simpler examples first before moving on to more difficult ones. Using curriculum learning, one hopes that we can accelerate training by allowing the network to learn basic characteristics of the task before moving on to progressively more difficult details. Indeed, it was observed in [2] that curriculum learning for neural networks improved the rate of convergence on a shape recognition task.

1.3. Choosing a curriculum and our approach

The concept of curriculum learning begs the question: what does it mean for a certain image to be "easy" vs "difficult"? [2] used prior human knowledge to separate the data into easier and more difficult examples. However, some recent results [5, 9] show promising outcomes by automatically learning the curriculum. In both of these cases, however, there was an auxiliary model that explicitly learned the curriculum for the main model.

Our approach is a bit different. We hypothesize that it is possible to automatically distinguish between easy and difficult examples by training simpler models on the data and then observing the performance of the simple model on the examples. Intuitively, we can expect that if a certain example is "easy", both the simple model and more complex main model will easily be able to classify this example. We then sort the training examples depending on how well our simple model performed on them and thus get a curriculum for the main model. The assumption is that the simple model will train very quickly in comparison to the main model, and thus that we will still decrease computational time overall since the number of training iterations to reach a desired accuracy in the main model should be reduced.

2. Methodology

2.1. Description of framework

We want to train a classifier f_W for some task. To help us train this model, we will have a simple classifier g_θ that takes an input and calculates a confidence score in $[0, 1]$ for each of the classes. We will define the *difficulty* of a training example (X, Y) [where X is the input and Y is the correct class label] with respect to g_θ as $-g_\theta(X)_Y$. So, an example is *difficult* with respect to g_θ if g_θ gives a low score to the correct class, and conversely, *easy* if it gets a high score for the correct class.

The first step of the framework is then to train the model g_θ on the training data. After we have done this, we sort the data into k bins by their difficulty with respect to g_θ, let's

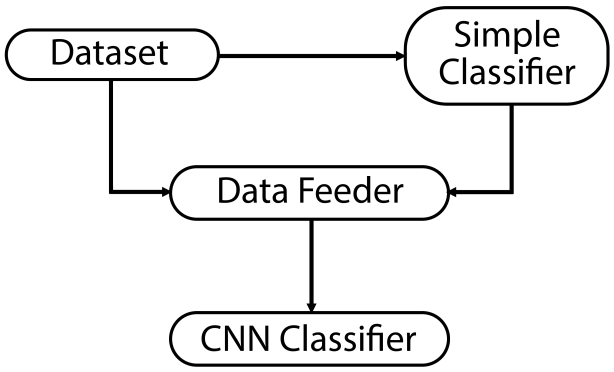


Figure 1: High-level diagram of our framework

say $B_1, \ldots, B_k$, where all B_i have (roughly) the same number of training examples, and the examples in B_j are more difficult with respect to g_θ than those in B_i iff $j > i$. We then split the training into k stages where at the j'th stage, we train on all examples in $\bigcup_{i \leq j} B_i$ using batch stochastic gradient descent.

Alternatively, we can also run batch SGD and sample examples to form our batch with probabilities according to their "easiness". Specifically, we used the following approach: (1): Rank all training examples from easiest to hardest. (2): To form batch k, we first construct a probability distribution over all training examples: $P(x_i) = \max(0, 1 - \frac{1}{\alpha|X|}(rk(x_i) - \beta kB)$, where $|X|$ is the total number of training examples, B is batch size, $rk(x_i)$ is the 0-indexed rank of x_i from easiest to hardest, and α, β are previously chosen hyperparameters. We then normalize these probabilities so that they sum to 1, and finally sample k examples according to these probabilities to get our next batch [sampling with replacement, for implementation simplicity]. Intuitively speaking, this equation assigns higher probability to easier examples (and truncates "probabilities" that go below zero), but as the network trains, the sampling probabilities even out due to the βkB term. The reason that we do not simply sample according to the difficulty score directly is that our basic classifiers are not guaranteed to be well-calibrated, so this might lead us to highly oversample or undersample certain examples due to skewed predictions. By contrast, a ranking-based sampling function offers reliable behavior agnostic of the choice of simple classifier.

2.2. Simple classifiers

For the simple classifier, we have tried the following: (1) a multinomial logistic regression (or softmax regression) [which could also be seen as a fully connected network with no hidden layers and a cross-entropy loss]; (2) A multi-class one-vs-all linear SVM; (3) a shallow convolutional neural network with one convolutional layer and one fully-connected hidden layer. The former two were trained using `scikit-learn`, and the latter with TensorFlow (as with our main model). The choice of these classifiers was motivated by their lower runtimes relative to training our full model, thus offering the potential for overall speedup. We also considered k-Nearest Neighbors and multi-class kernelized SVMs, but due to their high (superquadratic) computational cost, these "simple" models actually take far longer to train than the full neural network [f_W] itself, and thus would not be useful without implementing highly parallel versions of these algorithms ourselves. So, we focused on the three classifiers mentioned above.

2.3. Main model (Multilayer CNN)

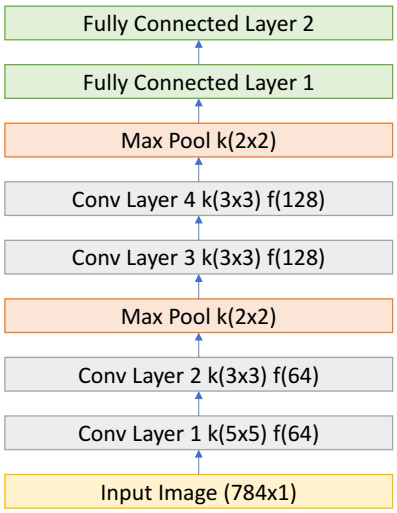


Figure 2: Main Convolutional Architecture

We first pass the image through two convolutional layers, with 64 output channels each, and a kernel size of 5x5 and 3x3 respectively. The output of the 2nd convolution layer is fed into a max pool layer with a 2x2 kernel. From here the output is passed through two more convolutional layers each with 128 channels and 3x3 kernels.

We then pass the signal through a final max pooling layer with a 2x2 kernel and feed it to two fully-connected (FC) layers. The first fully connected layer maps the embedding to a 1024 vector, and the second maps the 1024 size vector to the class mappings, i.e. a size 10 vector. We used 50 percent dropout while training as well as L2 weight regularization. We initialized our weights using Xavier initialization. We used the leaky ReLU (LReLU) activation function with parameter 0.01: LReLU(x) = max(x, 0.01x). The final activation function is a softmax (as is typical for multiclass prediction tasks).

2.4. Adapted Inception Module

We wanted to evaluate the effectiveness of an adapted Inception module. The first branch was a 1x1 convolution of the input image with 8 filters. The second branch was a 1x1 convolution followed by a 3x3 convolution. The third branch was a 1x1 convolution followed by a 5x5 convolution. Finally the fourth branch consisted of a 3x3 convolution followed by a 1x1 convolution. The output of the inception module was passed to a 5x5 average pooling layer and a fully connected layer mapping the input to a vector the size of the class dimensions. The adapted Inception module was not as effective as the multilayer CNN, which could be due to the fact that we did not incorporate the "stem" of the Inception module. Furthermore, stacking Inception modules has also shown to yield better performance, as well as adding additional filters in the convolutional layers; however, the computational requirements severely limited the number of modules we could stack, so we could not get better performance than our original multilayer CNN (topping out at around 30% on a noise-free validation set when training on CIFAR with noise parameter 5, a far cry from the 49% we get on this task with our previous model).

2.5. Overall Algorithm Description

We sorted the training data in ascending order of difficulty and bucketized it into n different buckets [where n is an input parameter to the algorithm]. The main model (the multilayer CNN) was trained in n stages, initially starting at 1; during stage k, all training examples only come from the easiest k buckets. After a predetermined number of steps [which is a tunable hyperparameter that we set to $\frac{3000}{n}$, the stage is advanced to $k+1$, unless we are already in the final stage (stage n), in which case we are already drawing training examples from the entire training set. The CNN was trained using minibatch gradient descent, with a batch size of 50. Note that the algorithm with 1 bucket simply corresponds to standard minibatch gradient descent, since the first (and only) bucket contains the entire training set. This is our baseline for evaluating the performance of our method. It is important to note that the basic classifier is only run once, before the training of the CNN is begun.

Within each stage, the order of training examples was randomized, as is standard practice (and the examples were re-shuffled each time the stage was advanced). This was done in order to avoid explicitly training on the examples in strict order of easiness, which would be a deterministic training ordering that could lead to undesirable artifacts.

We also tested the version of our model that samples examples according to their difficulty, rather than explicitly bucketizing the examples. However, in no case did it perform better than all bucket-based models. A more extensive hyperparameter search would need to be done to determine whether it can offer a benefit, for which we unfortunately do not have the computational resources. For the sake of brevity, we thus avoid reporting these results.

3. Evaluation and Discussion

3.1. Description of Data

We have used the MNIST [3] and CIFAR datasets for our experiments. We found that these tasks were quite easy for our final convolutional architecture, in that it achieved good validation set accuracy after a low number of minibatch gradient descent steps. To more easily assess the potential benefits of curriculum learning, we thus made the task harder by adding noise to our data as follows:

For each training image x_i [represented as a floating-point vector with values in $[0,1]$ and length equal to the number of pixels in each direction times the number of channels], we selected a value σ_i^2 uniformly at random in $[0,S]$, roughly corresponding to the desired amount by which we are increasing the difficulty of that example, where S is a previously chosen parameter (we used $S=5$ for CIFAR and $S=20$ for MNIST). We then generated a vector of random noise of the same length as x_i by independently sampling a Gaussian with mean 0 and standard deviation σ_i^2 for each of the entries, adding the resulting vector to the original x_i, and finally clipping the result to lie in [0,1]. (See Appendix for examples of such 'noised' images.)

3.2. Results

In Figures 3 and 4, we show the learning curves for training our main model, with logistic regression and CNN basic classifiers, on noised CIFAR data (with noise parameter equal to 5), and validating on noised and un-noised data, respectively. We plotted only the validation set accuracy (every 10 training steps), as the curve is smoother and easier to interpret visually, and also represents the quantity that we truly want to optimize. We see that the logistic regression basic classifier offers a clear performance boost in the initial stages (more pronounced and long-lasting [over 5000 batches] when validating on un-noised data - see Fig. 4), while the shallow CNN performs little better than default training. Using SVM as a basic classifier also performs only as well as default training, so we have not included the plots. Clearly, the accuracy is much lower when validating on data that also contains noise, which is to be expected; nevertheless, the logistic regression classifier does show a benefit here as well, if only for about 1500 batches.

It is perhaps more informative to compare the time taken for a model to reach a specified accuracy threshold than simply looking at the final validation accuracy when evaluating the ability of our approach to accelerate learning, since it is likely that the models will eventually saturate around the same accuracy. Thus, in Table 1 we report the number of

Table 1: Main model performance on noised MNIST task, compared to simple classifier training accuracy. The first column shows the training accuracy of the *basic* classifier. Columns 2 and 3 show the final test accuracy, number of batches processed until the main model reaches 80% validation accuracy and same for 90%, when using each of the models as the basic classifier. The final main model test accuracy was 0.11 when not trained with any basic classifier - barely better than random.

	Model Train Acc (MNIST)	(Final CNN Val Acc, # to 80%, # to 90%): 2 Buckets	(Final CNN Val Acc, # to 80%, # to 90%): 3 Buckets
LR	0.2534	(0.93, 2000, 2500)	(0.94, 500, 800)
SVM	0.1121	(0.11, —, —)	(0.82, 2900, —)
Shallow CNN	0.4068	(0.93, 900, 1700)	(0.94, 500, 900)

batches to 80% and 90% validation accuracy, respectively, when training our main model on noised MNIST data [with noise scale of 20] and validating on un-noised data. We observe that the SVM, which barely performs better than random on this task, does not help accelerate learning at all, while the other two classifiers do. The Shallow CNN, while being a better model by itself than logistic regression, does not further accelerate training of the main model when more three or more buckets are used.

When we performed similar tests on un-noised CIFAR data (or un-noised MNIST data), the curriculum learning approach did not show any benefit, as the validation curves looked roughly the same with or without curriculum learning. We hypothesize that this is actually due to our core model being "very good" at these tasks (it saturates at over 98% validation accuracy on MNIST and over 80% on CIFAR). To support this assertion, we experimented with using a shallow CNN as our *main* model, with logistic regression as its basic classifier for curriculum learning, for the un-noised CIFAR task. This CNN does not do well on this (relatively easy) task overall, saturating at only around 46% validation accuracy. Nevertheless, when we used curriculum learning and varied the number of buckets, we saw consistently that the curriculum learning approach sped up training in the very early stages (although we still saw the curves catch up later on; see Table 2). This supports the idea that the addition of noise is not always necessary for our approach to be beneficial, at least early in training.

Table 2: Results for *basic* CNN, trained with Logistic Regression as basic classifier on *un-noised* CIFAR data. VA stands for Validation Accuracy. Mean accuracies were taken over five runs each. Batch size was 50.

# Buckets	VA after 200 steps	VA after 300 steps
1	0.325	0.353
2	0.347	0.378
3	0.392	0.397
4	0.381	0.414

Figure 3: Learning curves for main model with various numbers of buckets trained on noised CIFAR data but validated on un-noised data. Left: Logistic regression as basic classifier. Right: A simple CNN as basic classifier.

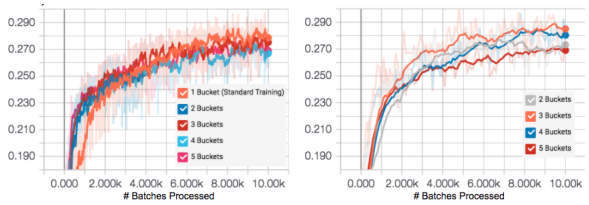
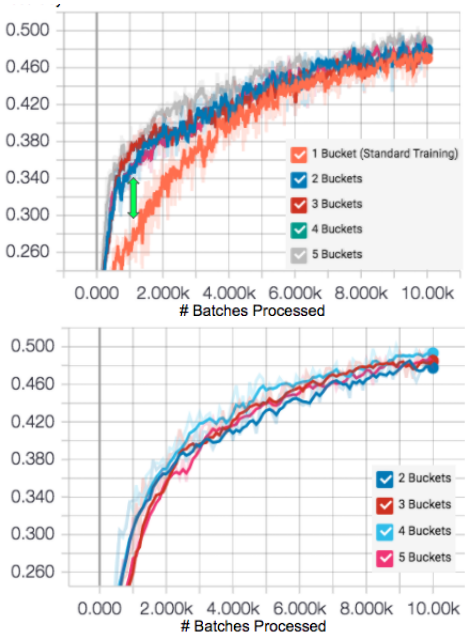


Figure 4: Learning curves with various numbers of buckets validated on unnoised CIFAR. Upper: Logistic regression as basic classifier. Lower: A simple CNN as basic classifier.



4. Discussion and Future Work

It is notable that learning is almost always significantly accelerated for the first few hundred to thousand iterations when using curriculum learning with a good basic classifier, but that the difference generally goes away after a while, and the standard network catches up in terms of validation accuracy. It would be an interesting future experiment to test whether similar models are learned with and without the curriculum approach, or whether the two training methods arrive at very different local minima. This could, for instance, be done by looking at which training examples are misclassified by each of the models and comparing these sets.

Another note is that our approach seems to work better when noise is added compared to when training on the data without noise. One explanation for this is that classification on CIFAR and MNIST is an inherently easy task, in which case the models already learn fairly quickly and the usefulness of curriculum learning diminishes. An example of this happening is when we train on MNIST without noise, in which case the convergence happens so quickly that its difficult to see a difference. Thus, with more time and computing power, it would be interesting to see how the automated curriculum learning approach would perform on a more difficult task, like classification on ImageNet.

There is an additional benefit to training on a larger dataset like ImageNet. In our runs, if we used too many buckets and did not advance between the buckets fast enough, we would overfit to the easy buckets (the validation loss would stop improving). This is probably in part due to the fact that the easy buckets are inherently easier to overfit to. However, it is also clear that a major issue can be the size of the buckets. On the other hand, if we have a larger dataset, we could train on the buckets for longer before advancing to the other buckets which might increase the effectiveness of curriculum learning. In the same vein, we could experiment with different criterion for switching between buckets. A sensible idea would be to switch when the validation accuracy stops increasing (which would suggest that we are starting to overfit on the current bucket). We experimented with this, but found that it was difficult to determine a good heuristic for switching, since the validation accuracy is not monotonic and the average rate of increase slows down as training proceeds, regardless of the training method used.

5. Conclusion

This automated approach to curriculum learning shows the potential to decrease network training time. However, the approach requires additional hyperparameters to consider when tuning models, especially the choice of basic classifier and number of buckets. Our experiments have shown that we can make a learning problem more challenging by adding random noise with differing levels of variance to the dataset, and correspondingly show a greater benefit to curriculum learning, though this is admittedly somewhat contrived and may not even work for all datasets (particularly those with discrete-valued features). Despite these drawbacks, this approach is promising because it may allow deep learning practitioners to expend less effort on data selection in the future, being able to simply apply the curriculum approach instead to filter bad examples. There are many potential applications of this; for instance, security footage might come from several different cameras, and each camera might have differing levels of image quality based on camera model, lens dust, and more. An image processing model might have difficulty learning from poorly captured images, but automated curriculum learning could help identify such images and set them aside until the end, so that the model can learn more basic features first from higher-quality images. We believe that this adaptive curriculum learning approach could be a promising avenue for several directions of future research, and hope to see it applied to real-world problems.

6. Appendix

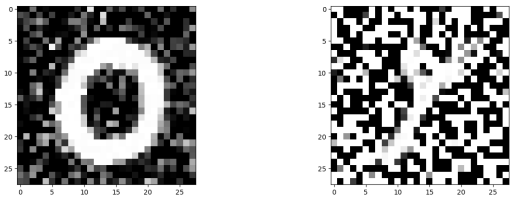


Figure 5: Examples of an “easy” [left] and “hard” [right] perturbed MNIST training example as identified by logistic regression classifier.

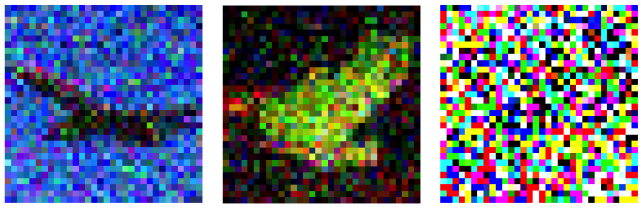


Figure 6: Examples of easy [airplane, left], medium [frog, middle], and hard [airplane, right] noised CIFAR examples as identified by the shallow CNN classifier.

References

[1] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in optimizing recurrent networks. In *ICASSP*, pages 8624–8628. IEEE, 2013.

[2] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning, 2009.

[3] L. Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, November 2012.

[4] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[5] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu. Automated curriculum learning for neural networks. *CoRR*, abs/1704.03003, 2017.

[6] G. Hinton. Neural networks for machine learning - lecture 6a - overview of mini-batch gradient descent. 2012.

[7] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[9] T. Matiisen, A. Oliver, T. Cohen, and J. Schulman. Teacher-student curriculum learning. *CoRR*, abs/1707.00183, 2017.

[10] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

Contributions

Halwest Mohammad

Halwest was responsible for implementing several of the models that were used (multinomial logistic regression, SVM, deep CNN), for running several of the experiments, for doing part of the literature review and for writing a significant part of the sections about motivations, relevant work, methodology and discussion. He created some of the architecture diagrams.

Nimit Sohoni

Nimit was responsible for using the basic model to generate difficulty scores for training examples, implementing the training logic that uses a given basic classifier to feed examples to the convolutional neural network, and running computational experiments (including writing automation scripts to do so). He also rearchitected portions of the code to make it extensible, and generated visualizations such as the ones in the appendix. He wrote the Evaluation and Discussion sections and portions of the Challenges and Next Steps sections in the report.

Rahul Palamuttam

Rahul was responsible for constructing and evaluating the main CNN model. He also plugged in TensorBoard to help compare the learning rates between the different experiments. Finally, he wrote the abstract and most of the Challenges and Next Steps sections, and portions of the methodology and motivation sections, and was also responsible for collating all references used in the project thus far. Rahul additionally created the Inception module from scratch and attempted to tune the model to perform well for the task. He created one of the architecture diagrams as well.