Toward Genetically Generated Ensembles of Neural Networks

Robert Reiss
Department of Computer Science
University of British Columbia
rreiss@gmail.com

Saifuddin Syed
Department of Mathematics
University of British Columbia
ssyed@math.ubc.ca

Abstract

Neural networks are powerful yet notoriously fickle beasts. They require tuning by hand over many parameters. Meanwhile the best tuning methods are only marginally instructive, generally consisting of anecdotal evidence and values that are known to work. Furthermore, even with the same parameters, neural networks often exhibit a range of efficacy, as training is stochastic in nature. We present a method which utilizes a genetic algorithm to train capable neural networks, while simultaneously producing an ensemble to harness their combined power. We demonstrate that this can lead to excellent candidates for further hand tuning, and powerful ensembles which are a match for finely hand-tuned neural networks. These ensembles are surprisingly robust. Multiple training runs produced very similar results, which were remarkably stable. This approach allows for an automated process for training capable neural networks, reducing the drudgery of hand-tuning.

1 Introduction

Over the last two decades, neural networks have become a force to be reckoned with in machine learning literature. Their effectiveness at solving both modern and classical problems is undeniable. In practice however, it is incredibly difficult to train neural networks. There are two main issues which arise when training neural networks: the loss function is highly non-convex, and neural networks are very sensitive to architecture, i.e. the number of layers and hidden units. Stochastic gradient descent and back propagation have been quite effective at tackling the first issue, but addressing the architecture problem does not have a standard, elegant solution. Neural networks are often fine-tuned by hand based on heuristics, anecdotal evidence, and best practice. Since the connections in each layer are highly dependent on one another, it takes a vast amount of trial and error to get a well-trained model. Overall the process can be very labour intensive and frustratingly inefficient. Moreover, it is generally not known what the upper bound on the performance is for a given problem, nor if neural networks are even able to treat the problem with great precision. Luckily, there is often a "good-enough" error rate; a rate sufficiently low as to be acceptable given real world constraints. Thus any solution which provides a "good-enough" result will be viable for a given application. This provides the motivation for constructing an ensemble of networks, which may surpass the "good-enough" threshold without further intervention.

We use a genetic algorithm to automate the process of finding and training suitable architectures for neural networks with L layers. We simultaneously produce an ensemble from the trained networks. In short, we explore the state space of model parameters by randomly initializing a population of neural networks. We then assign a fitness score based on how well the networks do on a validation set after training. The fitter networks have a better chance of passing on their traits to successive generations, while the weaker ones die out. As the generations progress, the average fitness tends to increase. We then create an ensemble from the best models found during the search.

Submitted to Mark Schmidt as a final project for the 2016 offering of *CPSC 540, Machine learning*.

Our method produces well trained neural networks, trading off additional computational time for manual labour. We simultaneously produce better networks while constructing an ensemble in an unsupervised manner (in the colloquial sense). Thus, if the ensemble produces results which are "good-enough", we are done. On the other hand, if additional precision is required, we observe successful models as the generations progress. These serve as a good foundation for further fine-tuning. Finally the method can be extended beyond neural network architecture to also optimize for other hyperparameters: the regularization rate, learning rate, dropout rate, etc.

2 Genetic Algorithm's for NN Architecture

A genetic algorithm is an optimization strategy that mimics the principles of evolution to search the parameter space Θ, where Θ is a d dimensional space of parameters. This occurs in four stages: initialization, selection, crossover, and mutation [**?**].

We begin with a population $P \subset \Theta$, called *phenotypes*. The phenotypes are encoded into a d-dimensional vectors; with each component representing a characteristic of the population called a *gene*. Each phenotype's fitness is computed via a fitness function $f : \Theta \to \mathbb{R}$. Fitter phenotypes are more likely to be selected to proceed into the next generation. Some of the selected phenotypes breed by exchanging genes (known as *crossover*) to produce offspring - the next generation. To allow exploration of other genes not part of the initial population, we *mutate* some of the genes. That is we allow some genes to reinitialize with given probability. The process is repeated until some stopping criterion is met. In principal this should produce a population that converges to a higher fitness level as the generations progress.

2.1 Initialization

Given a neural network with L layers, the architecture is specified by the number of hidden units in each layer. Thus, the parameter space searched is the set of possible architectures $\Theta = \mathbb{N}^L$. Each $x \in \Theta$ is a vector where x_i represents the number of hidden units in layer i. Instead of $\mathbb{N}$, we limit our search to the cube $\Theta = [1, N]^L$, for some N. We initialize our population size M by setting $P_0 = \{x_1, \ldots, x_p\}$, where x_j is a random vector chosen uniformly from Θ. A population in generation t is denoted by P_t.

2.1.1 Fitness

Given a trained neural neural network with architecture x, the fitness $f(x)$ is the performance of the network on a validation set.

2.2 Selection

Recall the population at generation t is P_t. The fitter phenotypes in P_t are more likely to be selected than the weaker ones by letting $p(x)$, the probability that $x \in P_t$ is chosen, be proportional to its fitness. Namely:

$$p(x) = \frac{f(x)}{\sum_{y \in P_t} f(y)}.$$

Let $\tilde{P}_t$ denote the new population of size M generated and sampled from P_t via $p(x)$.

2.3 Crossover

During the crossover phase, each phenotype in $\tilde{P}_t$ is chosen to breed with some probability κ - the *crossover rate* . In preliminary testing, we found $\kappa = 0.3$ to be a good value. Generally, the algorithm was not very sensitive to the crossover rate.

Let $\tilde{x}_1, \ldots, \tilde{x}_m \subset \tilde{P}_t$ denote the phenotypes that are chosen for breeding. Since they are already random, we decided to breed the genes of $\tilde{x}_i$ with $\tilde{x}_{i+1}$, where the subscripts are mod m. For each $1 \leq i \leq m$, we pick a random integer $1 \leq k_i < L$ and define the child $\hat{x}_i$ by the vector

$$(\hat{x}_i)_j = \begin{cases} (\tilde{x}_i)_j & \text{if } 1 \leq j \leq k_i \\ (\tilde{x}_{i+1})_j & \text{if } k_i < j \leq L \end{cases}$$

Let $\hat{P}_t$ denote $\tilde{P}_t$, where $\tilde{x}_i$ is replaced with $\hat{x}_i$.

2.4 Mutation

Finally, to bring some genetic diversity into the population, we allow some of the genes to mutate. For each $\hat{x} \in \hat{P}_t$, and for each $1 \leq j \leq L$, we let $(\hat{x})_j$ reinitialize to an integer between 1 and N with probability μ, called the *mutation rate*. We found that $\mu = 0.3$ worked well. As with the crossover rate, the algorithm was not particularly sensitive to the mutation rate. Where a greater mutation rate proved useful was when an evolution got stuck in a rut - when every member of the population has the same low fitness. In these cases, a higher mutation rate allowed the population to exit the rut faster. After mutation has occurred the generation t is over. The new post-mutation population is denoted P_{t+1}.

The procedure now repeats starting with the fitness phase for a specified number of generations.

2.5 Ensemble Classifier

To get to generation t, we must train each model in P_s for $s < t$. Although the latter generations generally produced fitter individuals than previous generations, many already trained networks in previous generations exhibited strong fitness. Pooling all the networks above a fitness threshold F_0 allows us to create a robust ensemble. Let Q_t denote all the trained networks $x \in P_s$, such that $f(x) \geq F_0$ for $s \leq t$. We then create a new model M_t, where we take the mode of the predictions of networks in Q_t. M_t is a diverse ensemble. The new networks added to the ensemble are the ones that survive the death-match described above, and thus are getting progressively fitter. This process leads the ensemble to become fitter as the evolution progresses.

2.6 Related work

There have been many approaches to the problem of finding a suitable model for the structure of a neural network. Adams, Wallach, and Ghahramani in [AWG10] used a non-parametric Bayesian approach to find an optimal network. They used a cascading Indian buffet process to create a prior over the number of layers and hidden units, with a Gaussian prior on the weights. To make inferences from the model, the (non-analytic) posterior is sampled using MCMC. The benefit of their approach is that they do not need to assume the number of layers. In addition to the complexity of the model, this approach is very dependent on the prior imposed on all the parameters. Moreover, MCMC can take a long time to converge. Therefore, this approach is highly sensitive to assumptions, and may be ill suited for certain problems.

There have been many different neuro-evolution approaches to construct and train neural nets. Stanley & Miikkulainen in [SM02], and Stepniewski & Keane in [SK96], have used genetic algorithms to encode both the structure and the weights; training and evolving the model without the need for a backpropagation implementation. Their approach allows for the possibility of certain connections between nodes to not exist, instead of having each layer fully connected. In contrast to our model, running each generations is computationally inexpensive, although the number of generations required to achieve a reasonably well trained network is very high, and convergence can be slow.

The ways to implement genetic algorithms to evolve neural networks are numerous, as evidenced by [MTH89],[MD89], and [tZM93]. These use different approaches than ours when it comes to interpreting the fitness functions, the genetic encodings, and crossover, allowing us to present a new methodology. The idea of using genetic algorithms to evolve neural networks is far from novel. Our contribution is that the described method is comparatively simple, and lends itself very well to constructing both strong neural nets and a robust ensemble of them, which none of the above attempted to do.

3 Experiments

The genetic algorithm with ensemble was implemented in MATLAB. We used a subset of the MNIST Database of handwritten digits to provide data for a classification task [LC]. To keep training tractable, we used only a portion of the available data set: 5000 training examples, 5000 validation examples, and 1000 test examples. Our selected training procedure ran an implementation of stochastic gradient

descent using a combination of Nesterov update with momentum parameter β, and learning rate α, as suggested by Sutskever et al. [SMDH13]. We annealed the learning rate with a further parameter δ. To regularize, we implemented L^2 weight decay with parameter λ. Finally we added dropout with probability p on the second hidden layer,which has been shown to help prevent over-fitting [SHK$^+$14]. We ran this for a constant I iterations to train each neural network. When the ensemble stopped improving on the validation result, we stopped producing new generations, as long the evolution was not in a rut. We define a rut as a state where all members of a generation are nearly equally fit, and produce a poor output (usually about random for classification).

3.1 Implementation Details

3.1.1 Stochastic Gradient Descent

We implemented stochastic gradient descent with the following Nesterov style updates:

$$w^{t+1} = w^t - \alpha_t \nabla f(w^t - \beta_t(w^t - w^{t-1})) + \beta_t(w^t - w^{t-1})$$

where α_t is the learning rate $\alpha\delta^\tau$ and $\tau = \lfloor \frac{10t}{I} \rfloor$. That is the learning rate decays by δ every $\lceil \frac{I}{10} \rceil$ iterations. The parameter β_t is the momentum parameter.

3.1.2 Regularizaton and Dropout

We implemented standard L^2 weight decay, as well as dropout. We performed the scaling of the weights during training rather than testing. This allows us to leave the weights untouched during testing and validation, making for a cleaner implementation.

3.1.3 Loss, Activation, and Fitness Functions

Our implementation used a $tanh$ activation function, with a standard least squares loss function. The fitness function used was the proportion of correctly classified items in a validation set.

3.2 Experimental Results

We trained networks with two, three, and four hidden layers respectively. The following parameters were used for each run:

- Max Population N = 250
- Population M = 8
- Crossover κ = 0.3
- Mutation μ = 0.3
- Fitness Cutoff F_0 = 0.8
- Iterations I = 250,000
- Learning Rate α = 0.01
- Learning Rate Annealing δ = 0.7
- Momentum β_t = 0.9 for all t
- Dropout Probability p = 0.5

Every ensemble of networks was trained five times. When the fitness of the ensemble stopped rising on the validation set, the training was ended and the learned weights were used on the test set. The fittest single neural net encountered during the evolution was also reported. The figure below summarizes the findings. The box plots represent the fitness of the trained ensembles on the test set, while the x marks represent the fittest and least fit individual neural network found over all training runs. The black x marks represent the fittest individuals, while the green x marks represent the least fit. The full data set is reported in Appendix A.

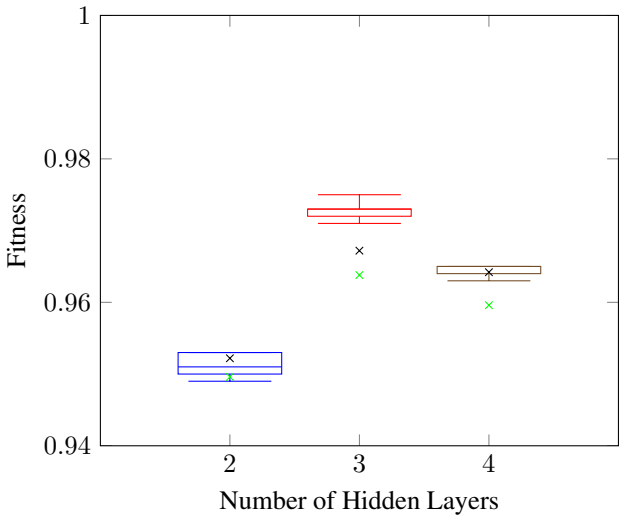

Figure 1: Fitness Evaluation

In figure 1, a few things are immediately noticeable. Firstly, except in a few cases, the ensemble outperformed the fittest individual. This is to be expected. This trend however was not observed with two hidden layers. This is likely due to the fact that two hidden layers are simply not complex enough to capture the full intricacies of the features. During the evolution of two hidden layer populations, we saw more frequent and longer ruts. In general very few two hidden layer neural networks performed better than a random classifier, and those that did perform better often did not perform particularly well. Thus the ensemble contained a number of relatively poor neural networks, even with a high fitness cutoff F_0 of 0.8. While a lower fitness cutoff did not perform more poorly in general, it also did not perform better. Training simply took longer, as a greater number of less fit individuals were admitted to the ensemble.

Secondly, there is a clear delineation between the different number of layers of hidden units. In particular, two hidden layers performed the worst, while three hidden layers performed the best. What is remarkable is how tightly the results were coupled to the network architecture. Not a single of the fittest two hidden layer networks in a run performed as well as any of the fittest in a four hidden layer network. Similarly, the three hidden layer networks summarily outperformed the two and four hidden layer networks. The only slight aberration was the fittest four hidden layer network in a single run marginally outperformed the least well performing of the individual fittest networks with three hidden layers; however the difference is so slight it can be ignored - 0.9638 vs 0.9642 which equates to two additional correct classifications per five thousand predictions. The same pattern holds for the ensemble classifiers. Every three hidden layer ensemble outperformed every four hidden layer ensemble, which in turn outperformed every two hidden layer ensemble.

A further delineation between the different number of layers of hidden units was that networks with more layers tended to have fewer ruts, and exited ruts faster. Both one hidden layer and five hidden layer networks were also tested, however one hidden layer networks performed abysmally. Due to the long computation times involved with training deeper networks, a full complement of five runs was not tested for five hidden layer networks. Five hidden layer networks performed in general no better than four hidden layer networks, based on the preliminary tests run.

Finally, the ensembles were remarkably consistent. The best and worst ensemble classifiers in a given architecture differed by a rate of only four correct predictions per thousand for the two and three hidden layer networks, and only three correct predictions per thousand for the four hidden layer network on the test set. The fittest individual networks in each run similarly showed a tight coupling. This is somewhat surprising, as the high non-convexity of the loss function, along with gradient descent can lead to considerable variability even when training two networks with the same parameters.

On the other hand, while the networks were clearly delineated by the number of hidden layers, within a family of networks with the same number of hidden layers, the results were much less clear. Well

over 50% of networks encountered during our training runs were no better than random. Those that were better than random usually displayed quite strong fitness (0.5000 or better, with most in the 0.7500 and up range). Furthermore amongst the fittest networks, there was quite a variety of phenotypes, though most of the fittest networks had a decreasing number of hidden units in the higher hidden layers. An increasing number of hidden units in the higher hidden layers was particularly bad for performance on this data set. When a given network layout worked well, perturbing the number of units per hidden layer by a slight amount generally had little effect. Thus while the search space is vast, as long as the perturbations were small, the exact structure was not overly sensitive.

The ensembles were compared to a hand tuned neural network which achieved an error rate (fitness) of 0.9710 on average, and a best fitness of 0.9750 over multiple training runs on the test set. This network however was trained with 500,000 iterations rather than 250,000 iterations. When comparing with the fittest reported three hidden layer networks, and then training those networks for 500,000 iterations, we saw results that were very close without further tuning, around 0.9680 on average. Thus the fittest networks produced automatically were within 0.3% or three correct predictions per thousand. Looking at the ensembles produced, we saw results that matched the hand tuned network.

4 Discussion and Future Work

4.1 Strengths/Contributions

The results witnessed where somewhat surprising. The initial motivation was to cut down the time needed to hand tune a neural network, by providing some insight into network configurations which produced acceptable results. The method however exceeded our expectations, providing results as good as those produced by many man-hours of laborious hand tuning. In particular, the ensembles produced were as capable as the best hand tuned networks, even over multiple training runs. The result is a robust method which provides both good candidates for further hand tuning and strong ensembles. The fittest networks can give good insight into an overall distribution over possible architectures, which can serve as a starting point for further Bayesian methods. The real strength of this approach lies in the automated nature of the procedure which is generally only limited by time and available computing power.

4.2 Criticisms

One downfall to this method is that many neural networks need to be trained. For smaller, less complicated networks, they can be trained in reasonable time on standard computer hardware. The data set used is known to be somewhat simple which allows less complicated neural networks to perform well. Other data sets may require more complicated and deeper neural nets to show similar performance. For very deep nets with many units per layer, it could take weeks or even months to complete an evolution. For comparison, each evolution presented in this paper took about 10-20 hours to train on a standard laptop with an Intel i7 Processor. The algorithm however is eminently parallelizable. Training the networks in each generation can be done completely in parallel, and can be implemented in a straightforward manner to run on GPU architectures. Thus training can be sped up considerably.

We implemented and explored the architecture in a discrete way, setting the number of hidden layers manually. Networks with disparate numbers of hidden layers were never directly compared by our method. The method however, is perfectly general, and this can be easily accommodated. Additional parameters can be easily added to the phenotype vectors.

4.3 Next steps

The results warrant further work. Firstly, the additional parameters which were held fixed should be added to the phenotype vector, allowing more general mixing and evolution. This will likely lead to even better results, as the chosen values for the fixed parameters where generally just the typical values found in literature. Secondly, new data sets should be explored. While the method proved robust on the chosen dataset, this need not hold in general. Further parameters can be added as well. For example, the momentum parameter could be annealed.

The ensemble generation could also be improved. A soft-max style implementation of the classifier could be used. The results of the individual networks in the ensemble could also be scaled by their fitness. Furthermore, the fitness cutoff F_t could increase over time, allowing less fit networks into the ensemble initially, but only the very fittest as the gains from incremental networks decrease. One could also feed back the overall fitness of the ensemble to determine the best cutoff evolution as time progresses.

Finally in order to deal with larger and deeper nets, an implementation of this method using GPUs and parallel processing would be highly beneficial to further experimentation. MATLAB may not be the ideal platform, and other implementations may be better suited to environments such as THEANO or PYTHON.

5 Conclusion

We set out to produce capable neural networks with minimal user intervention and hand tuning. Turning to a genetic style algorithm, we were able to generate many fit neural networks, and combining them into an ensemble, found results as good as those of finely hand tuned networks. Our results exceeded our expectations, providing excellent candidates for further hand tuning, even with only a few parameters considered by the genetic algorithm. This technique shows great promise, and further work could allow for even fitter individuals and ensembles.

A
Appendix

Experimental Results Tables

Fitness Evaluation $L = 2$	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Max	Min	Mean
Individual Maximum	0.9502	0.9496	0.9512	0.9522	0.9512	0.9522	0.9496	0.9509
Ensemble Validation	0.9446	0.9454	0.9478	0.9508	0.9504	0.9508	0.9446	0.9478
Ensemble Test	0.9530	0.9500	0.9510	0.9530	0.9490	0.9530	0.9490	0.9512

Fitness Evaluation $L = 3$	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Max	Min	Mean
Individual Maximum	0.9668	0.9654	0.9672	0.9654	0.9638	0.9672	0.9638	0.9657
Ensemble Validation	0.9678	0.9706	0.9700	0.9710	0.9698	0.9710	0.9678	0.9698
Ensemble Test	0.9710	0.9730	0.9730	0.9750	0.9720	0.9750	0.9710	0.9728

Fitness Evaluation $L = 4$	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Max	Min	Mean
Individual Maximum	0.9642	0.9640	0.9626	0.9620	0.9596	0.9642	0.9596	0.9625
Ensemble Validation	0.9652	0.9650	0.9644	0.9640	0.9620	0.9652	0.9620	0.9641
Ensemble Test	0.9650	0.9630	0.9650	0.9640	0.9650	0.9650	0.9630	0.9644

References

[AWG10] Ryan P. Adams, Hanna M. Wallach, and Zoubin Ghahramani. Learning the structure of deep sparse graphical models. *Journal of Machine Learning Research: Workshop and Conference Proceedings (AISTATS)*, 9:1–8, 05/2010 2010.

[LC] Yann Lecun and Corinna Cortes. The MNIST database of handwritten digits.

[MD89] David J. Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, pages 762–767, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[MTH89] Geoffrey F. Miller, Peter M. Todd, and Shailesh U. Hegde. Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[SK96] Slawomir W. Stepniewski and Andy J. Keane. *Parallel Problem Solving from Nature — PPSN IV: International Conference on Evolutionary Computation — The 4th International Conference on Parallel Problem Solving from Nature Berlin, Germany, September 22–26, 1996 Proceedings*, chapter Topology design of feedforward neural networks by genetic algorithms, pages 771–780. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127, June 2002.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1139–1147. JMLR Workshop and Conference Proceedings, May 2013.

[tZM93] Byoung tak Zhang and Heinz Muhlenbein. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex Systems*, 7:199–220, 1993.