Activation, Dropout, Pruning: Versatile and Interrelated Methods for Improving Generalization

Matthew Resnick
Matthew.Resnick@colorado.edu
SID: 109570948
CSCI 5922

***Abstract*—In the endeavor to build an artificial neural model for inference or prediction, it may often seem like two worlds must be reconciled into one: a world in which the model must capture something fundamental about known information and a world in which it must only capture something ethereal for the sake of the unknown. In this project, I seek not for such reconciliation of worlds, but rather to ask questions about bridges between them. I explore methods which both pose effective solutions to problems of generalization and lead to questions about how/why normalization, regularization, and sparsity may affect a model's ability to generalize. First, I will discuss and implement interesting activation functions which use some or all of these ideas to shrink the gap between representations of seen and unseen data. I will also discuss methods and research directions behind model pruning, and then I will present new methods of pruning which I have devised to further explore the role of sparsity in a model's generalization capacity.**

I. Introduction

Goodfellow, Bengio, and Courville said it best in *Deep Learning*: "What separates machine learning from optimization is that we want generalization error [to be low]," as opposed to simply minimizing training error [1]. However, when constructing a new model to learn some unseen distribution, it seems that the overfitting boogeyman ever lurking. Some unrepresentative training data here or a bloated model design there, all cause a vastly divergent training and test error. There are ways to improve generalization, of course, but they too are numerous beyond measure, and the quest to wrangle innumerable contributors to overfitting then becomes a quest to find their equally numerous counterpart solutions.

For some, there is a simple conclusion: different tasks require a specifically engineered solution, every time. Each component of a model must be carefully tailored to a given task, and there is no quick substitute for such meticulous tinkering. While I agree that each problem may have a unique solution, I disagree that widely applicable solutions can't be found for certain types of issues. And if some method can solve an entire class of problems blindly, it is all the more elegant to me, and a tool which warrants further study.

In this project, I will investigate such tools from three different areas of deep learning which I find to be the most interesting, and more importantly, the most powerful in improvement of generalization: activation functions, dropout, and neuronal pruning.

In the realm of activation functions, I explore those which affect structural aspects of a model and can be applied to a wide range of tasks without regard for underlying data distributions, model hyper-parameters, and other such pesky details. I will discuss their function, implement some from scratch, and compare results on real datasets from a variety of viewpoints. For each of these, a particular form of dropout regularization has been specifically tailored, and thus I will implement and discuss these as well.

Finally, I will discuss neuronal pruning —itself a form of regularization—which will culminate in a presentation of a new forms of pruning. I hope to combine the concepts learned from special activation functions, regularization, and existing pruning methods into new methods in order to explore questions of model sparsity and salience of parameters.

II. Activation Functions

When one thinks of activation functions in a deep learning setting, the first thing that comes to mind may not be generalization improvement. In fact, for most, activation functions probably constitute a single thought during the development of a model. There's good reason for this, too: the Rectified Linear Unit (ReLU) [2] usually works perfectly well, in spite of its simplicity. In many cases, even a lack of an activation function may be suitable for obtaining sufficient results. In *Deep Learning*, Bengio et. Al. suggest that new activation functions which perform similarly to existing functions "are so common as to be uninteresting," but that there are certainly more useful types which have not yet been discovered.

And right they were, as we have seen in the years since they published this sentiment. The progress, however, has not just come from improving slight issues with how the design of the function affects optimization, as with many ReLU variants, but rather with altering what the activation function does in the network entirely. Originally, the attitude towards activation functions was that they served the purpose of introducing a necessary non-linearity to the model. And while this is usually still an important purpose, other applications have only recently begun to be realized.

A. Functions Which Self-Normalize

The benefits of normalizing the input to a neural network model are widely known, including more efficient optimization of the loss function and better generalization [3]. Sergey Ioffe and Christian Szegedy of Google have also shown a variety of benefits from performing normalization in the inputs to each layer of the network, not the least of which being a reduction of internal covariate shift [4]. Thus, batch normalization has become a common practice in the construction of stable neural network learners, but as usual it has become a source of overfitting in many situations. As Hochreiter et. Al. point out, the stochastic nature of some training techniques cause great instability in the ability of a model to learn a distribution if batch normalization is in play, and further the training time can be dramatically increased [5].

1) SELU: Hochreiter and his team thus developed an activation function which is self-normalizing, and as they show, is resistant to the issues from stochastic training mechanisms, more stable in training loss optimization, and more computationally efficient. As an added benefit for generalization, this type of activation function is more compatible with many regularization techniques and is more resistant to the issues of exploding and vanishing gradients, as a result of more stable variance of activation. These scaled exponential linear units (SELU) are described by the following piece-wise function:

$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$$

Fig. 1. *From [5]*

The authors of [5] also introduce a new method of regularization specifically suited for SELU activation called alphaDropout. Standard dropout zeros neurons with given probability p, and then scales output either by $\frac{1}{1-p}$ in training or by $1-p$ in testing to preserve the mean of the input [6]. The idea behind this new method of dropout is focused on the concept that standard dropout works well for a function like ReLU because as the input becomes arbitrarily large in the negative, the value is zero, a value often reached because of dropout zeroing neurons. However, the value of SELU does not approach zero as the input grows negative, and thus alphaDropout sets neurons to the value it approaches $(-\lambda\alpha)$ instead of zero. In order to preserve the new mean and standard deviation created by this alteration, they also describe a new transformation instead of the aforementioned scale factor:

$$z = a(xd + \alpha'(1-d)) + b$$

Where:

- $\alpha' = -\lambda\alpha$
- $a = (1 - p + \alpha'^2(1-p) * p)^{-1/2}$
- $b = -a(p\alpha')$
- and $d \sim Binomial(1, 1-p)$

2) SERLU: Influenced by the self-normalizing property of SELU and the overall form of Swish [7] (that is, it is asymptotically similar to ReLU, with a "bump" local to small negative inputs), the authors of "Effectiveness of Scaled Exponentially-Regularized Linear Units (SERLUs)" produced a more viable and widely applicable activation function than its predecessors [8]. It can be described by:

$$\text{SERLU}(x) = \lambda_{serlu} \begin{cases} x & x \geq 0 \\ \alpha_{serlu} x e^x & \text{otherwise} \end{cases}$$

Fig. 2. *From [8]*

For essentially the same reasoning as with alphaDropout, the authors of SERLU also developed a new dropout method, called shift-dropout. In the case of shift-dropout, the minimum value is described by $f_{min} = -\lambda_{serlu}\alpha_{serlu}e^{-1}$ and the mean-preserving transformation is described by:

$$\hat{z} = \frac{1}{1-p}[\tilde{z} - pf_{min}]$$

Where $\tilde{z}$ represents the altered dropout activation with Bernoulli distribution.

3) Shape Comparison: The shapes of each activation function is made immediately obvious by the following plot (along with a few other functions):

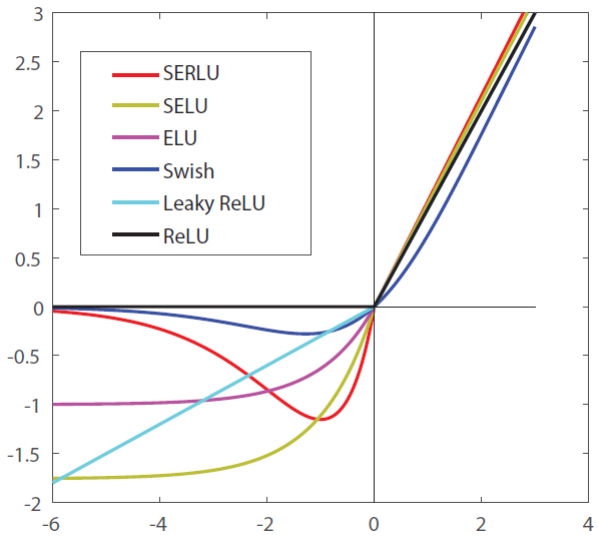

Fig. 3. *From [8]*

It can be seen that all of these functions follow ReLU asymptotically for positive input, but only Swish and SERLU do for negative input. All others increase monotonically to zero. This gives SERLU a few benefits: 1) being asymptotically similar to ReLU for positive inputs keeps the aspect of simpler calculation of gradient direction during loss surface optimization —due to the derivative essentially being 1. 2) For small negative input, SERLU is more expressive, in that the activation will not be zero. 3) Large negative inputs have activations of essentially zero, which seems to make both learning and generalizing easier.

B. Functions Which Induce Sparsity

From previous and simultaneous work on activation functions which can allow layer activations to self-normalize, it was apparent to some researchers that identifying operations which were more dynamic in function could be a rich endeavor. In the paper in which the Swish function was first described, the researchers set out to search the space of activation function types themselves in order to improve upon ReLU's established performance [7]. However, a more dynamic approach had been underway in years prior: allow a function to simply approximate *any* piecewise continuous function, and let its form be determined during training.

1) Maxout: Goodfellow et. Al. developed the maxout activation in 2013 which should do exactly the aforementioned operation, and provably so [9]. For layer output z and pool size k, maxout can be described as:

$$h_i(x) = \max_{j \in [1,k]} z_{ij}$$

Fig. 4. *From [9]*

Thus, ReLU is a specific form a maxout. Simply allow the pool size to be 2 (that is, to approximate a two-part piecewise function), and let the parameters be such that the first input be zero and the second be the usual affine transformation. Then the activation would be the max of zero and this transformation, i.e. ReLU itself.

The interesting properties of maxout go beyond just function approximation, though, as it was in part developed to accompany the properties of the usual dropout regularization technique. Maxout itself does not produce a sparse representation, but dropout —which will be covered in more detail in a later section—will sparsify the representation as sub-networks are ignored randomly during training. Further, it is pointed out in [10] that the neuron pooling property of maxout makes the maxout unit partially invariant to input changes.

2) Channel-out: If maxout is thought of as a generalized form of ReLU, then the subsequent channel-out activation can be thought of as a generalization of maxout [11]. The researchers behind channel-out sought two goals: explain maxout's performance on a fundamental level, and then use such an understanding to generalize it.

In the first regard, they came to a deep conclusion. The central idea behind this conclusion they refer to as "sparse pathway encoding," which describes maxout's operation as one which can both train parameters to make sufficient predictions given an example, but can also encode information in the network structure itself during training. Thus, when a test example is provided, the network has information on which pathways in the network are best suited for mapping such an example nearest to a value in the prediction space with least error. So this conclusion can explain the exceptional performance, but why is it so conceptually deep? Without explicit intention, this emulates a biological neural network according to Hebbian theory of neural plasticity [12].

Qi Wang and Joseph JaJa expanded this idea into an activation function, "channel-out," which is specifically engineered to better encode information into the network's pathways. Channel-out is very much like maxout, except every channel out node links to multiple output paths (and thus the size of each layer remains the same) and the decision function on what information to keep is variable, rather than being the max of inputs. For input a and indicator function $\mathbf{I}$, channel-out can be described by:

$$h_i = \mathbf{I}_{\{i \in f(a_1, a_2, \ldots, a_k)\}} a_i$$

Fig. 5. *From [11]*

Where Wang and JaJa place the following restrictions on $\mathbf{I}$: 1) it must be piecewise constant, 2) the pull-back of each output must be of essentially the same size and 3) the computational cost of evaluation must be low.

The operation and difference between maxout and channel out is best revealed in the following representation from [11]:

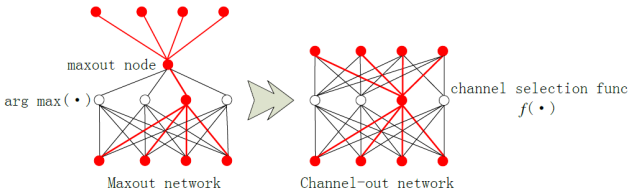

Fig. 6. *From [11]*

C. Comparative Performance

With the mechanisms, properties, and conceptual foundations of each of the preceding functions illuminated, I now turn to demonstrating their comparative experimental performance. The performance of each method here are not at state of the art levels, as I try to keep aspect of each model as similar as possible besides the method in question for the sake of careful comparison. State of the art performance benchmarks can be seen directly in each function's respective paper.

I selected three data sets for comparison, all with varying amounts of noise and predictive tasks, with two having binary targets and one having multi-class targets. The data sets are all publicly available, with MNIST being accessible from a variety of sources, and the Higgs and Gisette data sets all being available from the University of California Irvine. In this section, I have selected results only from MNIST, which I believe to be fairly representative, for analysis and discussion, but the remaining results can all be computed from the accompanying code. Further, most model components were constructed using the PyTorch framework, though I implemented SERLU, shift-dropout, maxout, and channel-out from scratch as they do not yet have public distributions for PyTorch.

First I explore the results of each activation with and without its corresponding regularization technique:

Model	Accuracy	Train Time (s)	Best Test Error
SELU_NoDropout	0.9984	151.306	1.5209
SELU_Standard_Dropout	0.9982	162.093	1.518
SELU_Alpha_Dropout	0.9981	181.523	1.5631
SERLU_NoDropout	0.9986	272.702	1.493
SERLU_Standard_Dropout	0.9991	274.572	1.5844
SERLU_Shift_Dropout	0.9989	387.597	1.5318
Maxout_NoDropout	0.9989	165.138	1.49
Maxout_Standard_Dropout	0.999	175.577	1.4926
Channelout_NoDropout	0.9981	263.274	1.4992
Channelout_Standard_Dropout	0.9988	276.23	1.4917

While most activations don't reveal too much of a difference in accuracy between a model without the corresponding regularization and with, the difference in generalization is readily apparent:

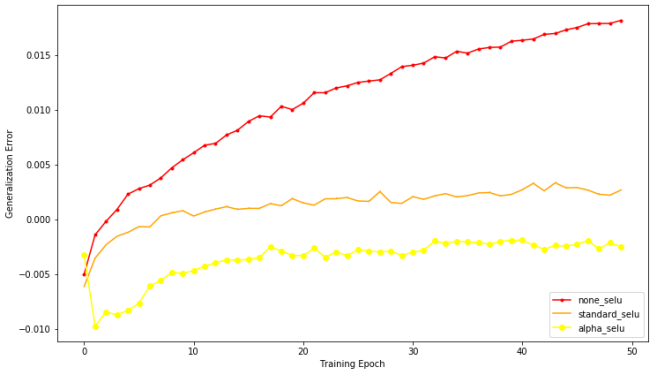

Fig. 7. SELU

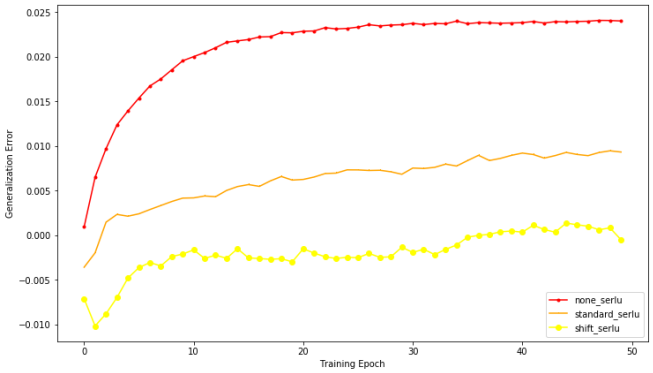

Fig. 8. SERLU

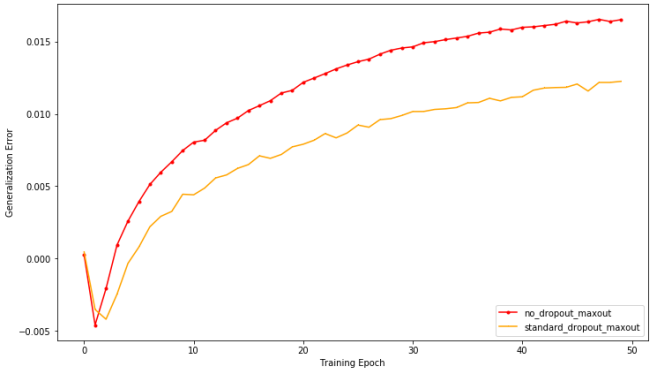

Fig. 9. Maxout

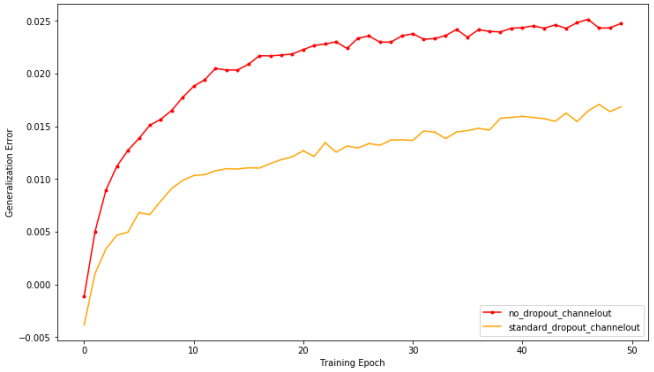

Fig. 10. Channel-out

Then I looked at different hyper parameters for each model, specifically the dropout ratio and, where appropriate, the pooling size:

Model (p=dropout %)	Accuracy	Train Time (s)	Best Test Error
SELU (p=0.2)	0.9983	178.219	1.5367
SELU (p=0.3)	0.9981	181.028	1.5494
SELU (p=0.4)	0.9983	175.149	1.5625
SELU (p=0.5)	0.9984	176.948	1.5775
SERLU (p=0.2)	0.9991	378.467	1.6139
SERLU (p=0.3)	0.9989	380.54	1.5302
SERLU (p=0.4)	0.9988	379.102	1.6204
SERLU (p=0.5)	0.9987	376.755	1.5432
Maxout (p=0.2, poolsize=2)	0.9989	169.882	1.4904
Maxout (p=0.2, poolsize=4)	0.9989	171.217	1.4893
Maxout (p=0.3, poolsize=2)	0.9989	168.78	1.4911
Maxout (p=0.3, poolsize=4)	0.9987	169.875	1.4893
Maxout (p=0.4, poolsize=2)	0.9988	172.5	1.4915
Maxout (p=0.4, poolsize=4)	0.9989	171.508	1.4895
Maxout (p=0.5, poolsize=2)	0.999	174.886	1.4928
Maxout (p=0.5, poolsize=4)	0.9991	169.686	1.4915
Channelout (p=0.2, poolsize=2)	0.9987	284.788	1.4897
Channelout (p=0.2, poolsize=4)	0.9986	269.657	1.4918
Channelout (p=0.3, poolsize=2)	0.9989	273.162	1.489
Channelout (p=0.3, poolsize=4)	0.9988	275.655	1.4916
Channelout (p=0.4, poolsize=2)	0.999	273.87	1.4892
Channelout (p=0.4, poolsize=4)	0.999	275.015	1.491
Channelout (p=0.5, poolsize=2)	0.9988	281.235	1.4911
Channelout (p=0.5, poolsize=4)	0.9989	285.525	1.4928

Here are the top performing models, in terms of accuracy and generalization error, side-by-side (where ReLU and Maxout have dropout of 50%, and the rest have 40%):

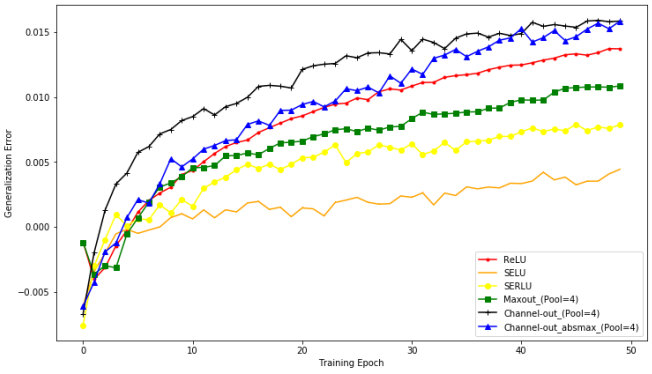

Fig. 11.

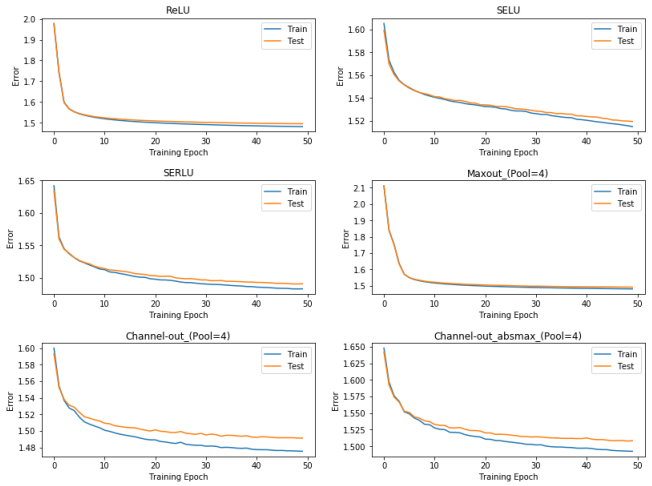

Fig. 12.

So, at least for MNIST, SELU and SERLU perform the best in terms of generalization error, with SELU being the clear winner in terms of computation time. This makes a great deal of sense —and I believe many situations would see a similar result—because the addition of dropout to a self-normalizing activation would allow for the additional sparse pathway encoding, similar to that of maxout and channel-out. As seen in [13], it may be the maxout and channel-out see their best performance when there is an additional, explicit ensemble mechanism such as bagging or boosting. Boosting works by subset of data being focused on during training by a unique model, and thus the sparse pathway encoding of both activations would be fit much more snugly to the underlying data. Whereas fitting parameters snugly to the training data might cause more overfitting, I believe this might cause smaller generalization error because the sparse pathway encoding can only capture vague information about a given input sample, as opposed to making an exact calculation for every feature.

III. Pruning

Research in the field of pruning is a troublesome area, to say the least. A recent survey of pruning research has revealed that there is very little consistency in the development and evaluation of methods, and some methods are being entirely rediscovered because older literature has been largely ignored [14]. The same survey, from Blalock et. Al. of MIT, suggests that despite such a disorganized effort to conduct research, pruning is an often successful method of model compression and, in many cases, of regularization. They maintain, "One of the clearest findings about pruning is that it works."

Due to the lack of consistency in the field, it may be that pruning methods will change drastically in the future. However, as of now, the two main types are structured —where architectural components of the model are trimmed away—and unstructured —where individual parameters are zeroed. The primary difference is that structured methods have a far superior compression effect, as there is less computation to be done (though, it is not inconceivable to imagine methods created in the near future which are built for specific frameworks that can replicate the compression effects in unstructured methods). Further, each type of pruning has two sub-types, related to the way in which the final structure of the model is determined: predefined and automatic [15].

In both modern and legacy literature on pruning, the most widely used and arguably most widely successful method is that of magnitude-threshold pruning. Papers such as [16] and [17], which study the intricacies allowing for the success of pruning, employ this method, either by including a penalty term or by manually removing connections with a magnitude below a certain threshold. There appears to be evidence for the promise of pruning in nature as well, because, as pointed out in [18], this type of pruning can be thought of as evaluating the energy processing of the neurons to induce sparsity. While not known to be the model for sparsity in biological neural networks, it still is convincing that methods rooted in the biological version of a process are the most successful.

A. Predefined Neuron Pruning on a Schedule

Since pruning research is still essentially in infancy, I will incorporate the information I have discussed in this report so far, as well as some external methods of model evaluation, to create new pruning methods. The current paradigm for the process of unstructured pruning, as described in [16], is to create a large, over-parameterized network and then iteratively prune and continue training to "fine-tune" the resulting network.

I propose, using an existing model for feature selection, that the neurons be pruned on the basis of L_1-norm as a measure of salience as usual, but the amount pruned per iteration should be based on an annealing schedule. This would keep the efficiency benefits of the usual magnitude pruning, while also adding to the effectiveness per pruning iteration.

1) Feature Selection with Annealing: First, it is necessary to describe the model which underlies the predictive task, called Feature Selection with Annealing (FSA) developed by Barbu et. Al. [19]. Given a single linear transformation of

input features and an appropriate loss function, the FSA algorithm would iteratively remove weights based on a schedule described by:

$$\mathrm{M}_i = k + (\mathrm{M} - k)\max\left(0, \frac{\mathrm{N}^{iter} - 2i}{2i\mu + \mathrm{N}^{iter}}\right)$$

Fig. 13. *From [19]*

At iteration i, where M is the input's feature dimensionality, μ is a tuneable parameter, and k is the max number of selectable features. This value is the number of parameters to be kept based on L_1-norm.

2) Neuron Pruning with Annealing: In a given training step of the original formulation of FSA, the schedule described above would be used to prune the input and the model parameters, along with an update to the parameters using gradient information from a loss calculation. Thus, FSA could be reconsidered as a neuron pruning step, where the input is the output of a layer in a neural network, and the gradient update happens in the usual way via backpropagation.

To this end, I modified the FSA algorithm to fit within the L_1 pruning practice in the following ways: After every epoch of training in a neural model, all parameters per layer are stacked into a single vector and ordered by L_1-norm. Then, the value M_i is calculated using a given compression factor for k, the current training epoch for i, and the total number of epochs for N^{iter}. Using the value for M_i, some number of parameters are selected, while the rest will be effectively pruned. A Boolean mask is then created using the selected parameters' locations, and ultimately applied to the corresponding layer during a forward pass of the network.

There are a few variations of this pruning method which I intend to explore. For instance, how do results compare if a network is pruned during training (the prune/fine-tune method) versus if a network which is trained from scratch without pruning, but given the masks from a previously trained and pruned network? Or even further, how will these compare to a completely new network with the same size? And how will these by-layer pruning methods compare to one in which all parameters from every layer are stacked and pruned at once?

3) Pruning with Annealing Results: The by-layer pruning schedule shows at least comparable performance with the base model down to a compression scale of 0.2 of the original parameters (on MNIST):

Compression %	Accuracy	Total_Gen_Error	Best_Test_Error
0.2	0.9461	-0.0261	1.5402
0.4	0.9559	0.059	1.5195
0.6	0.9583	0.0827	1.5145
0.8	0.9598	0.0995	1.5117
1	0.9602	0.0973	1.5119

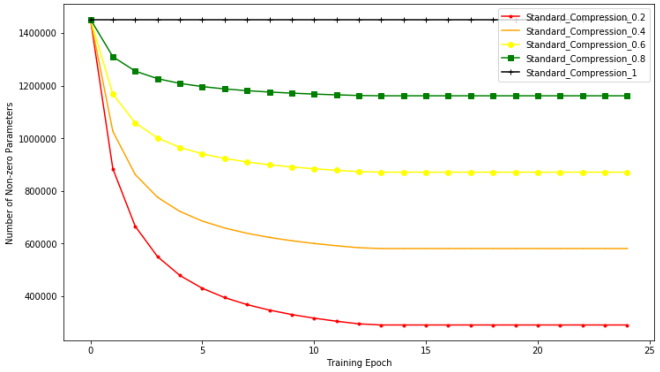

Fig. 14. Count of Nonzero Parameters per Training Epoch

Perhaps most intriguing though, is the results from other variations of this method (all pruned to 50% of original parameters:

Version	Accuracy	Train Time (s)	Total Gen. Error
No Pruning	0.9595	77.179	0.0967
By Layer During Training	0.9583	129.328	0.0787
With Dropout and Batchnorm	0.9708	151.738	0.2209
All Parameters at Once	0.9585	305.132	0.0716
Retrain from Scratch	0.943	206.913	-0.0185
Before Training	0.9574	145.225	0.0666
No Pruning Smaller Model	0.9582	48.078	0.1046

The results from essentially every version of this pruning method brought with it a wealth of useful information. It seems that while pruning all parameters at once brings comparable performance, it is over twice as computationally expensive to perform. Retraining from scratch also yielded substantial results (as [15] suggested it might), though any lost performance might be gained back with regularization techniques such as dropout and/or batch normalization.

However, the most interesting result comes from the temporal ordering of the pruning process. A natural inquiry could be made about what the point is for using a schedule for pruning if the same amount of parameters are pruned by magnitude, and doing so during or after training would yield similar results. At first, my answer would have been that the schedule is much more efficient, as the more pruned a model is, the faster it could train. What made this answer irrelevant was that pruning the model before training, even more efficient than during-training pruning, *also* yielded similar results.

It is now apparent that while pruning may be effective, the method of determining salience may be less important than simply choosing an appropriate network size, the same problem we started with. While efficiency may change, the performance of the network did not change much whether it was pruned on a schedule, by layer, during training, after training, or even *before* training. We could prune weights right from their initialized values and it would have the exact same effect.

But without checking how every possible network structure affects the loss, how could we determine what the size of a network should be? I could see two routes. The first route

would be to still determine size during or after training by making assumptions about the distributions of the parameters. The second route would be to prune before training starts by making assumptions about the data itself, and how that might affect the network. Both routes will be discussed in the following section.

B. Automatic Neuron Selection with Pruning

The only remaining detail of the previous section that is a proverbial thorn in my side is the necessity of pre-defining some compression factor in order to even begin to think about pruning. The fact that this compression factor can be specified at many different levels of granularity and for different components of the model is a step in the right direction, but still at every step it must either be guessed —a completely unsatisfactory answer for such an otherwise rigorous field of inquiry—or it must be searched for empirically —a more rigorous solution, but far too computationally expensive in most situations.

In my eyes, a better solution would be to make a determination about the distribution of parameters and then prune based on a threshold constructed from it. Thus, I propose a few different ways of making this determination and compare the corresponding results.

1) Tetris Pruning: The first method of pruning I present can be thought of visually in a manner similar to the arcade game of Tetris. Every parameter in a given layer is first sorted into bins of equal width based on their exact value. Then, the lowest parameters (by magnitude) in each bin are zeroed such that the bin with the lowest number of parameters, P_{min}, has only one non-zero parameter remaining, while those original P_α parameters are left with $P_\alpha - P_{min} + 1$ non-zero parameters.

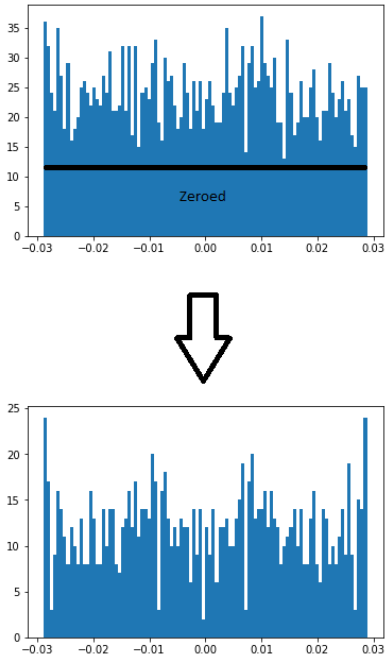

Fig. 15. *Visual representation of Tetris pruning. All but one zero-valued parameters are not shown in the second plot.*

The idea behind this method is that while a higher L_1-norm may indicate greater salience, there may be some nuanced contributions yet from parameters with smaller L_1-norms. Thus, salience is measured lexicographically, where the first measure of salience is abundance, and then after abundance is the usual L_1-norm. An additional nice property of Tetris pruning is that it need only be done in one step after training has concluded (or at a pre-determined point during training), and thus can be used in conjunction with other pruning methods. The only hyperparameter that would still require tuning is the the bin size, however the search space is much smaller since it may be checked by an order of magnitude at a time, depending on the total number of parameters.

This can be seen with performance on MNIST with various bin sizes:

Bin Size	Accuracy	Number of Parameters (% of Total)
10	0.1907	410 (3.42)
100	0.2892	2300 (19.17)
250	0.9441	4250 (35.42)
500	0.9521	7000 (58.33)
1000	0.9592	10000 (83.33)

2) Channel Pruning: Tetris pruning was developed largely from my exploration with self-normalizing activation functions, as it attempts to make use of the distribution of parameters to understand what to cut away. In a similar manner, I developed another pruning method which uses ideas from the activation functions which induce sparsity. This method, like the channel-out activation, groups parameters into pools of a pre-specified size and then makes a determination about what to keep with a given function. For channel-out this function could be a variety of mappings, but for channel pruning I decided specifically on the argmax of the absolute value of each parameter pool (i.e. magnitude pruning within pools). Again, there is a hyperparameter which must be tuned (pool size), however the search space is again significantly smaller, especially as the compression scale deteriorates rapidly with increasing pool size ($\frac{1}{pool\_size}$).

In order to see what might best suit such a pruning method, I obtained results on MNIST using three different activation functions and channel pruning post-training:

Activation (pool size)	Accuracy	Number of Nonzero Parameters (% of total)
ReLU (ps=2)	0.9711	726000 (50)
ReLU (ps=3)	0.9663	484000 (33.3)
ReLU (ps=4)	0.943	363000 (25)
ReLU (ps=8)	0.718	181500 (12.5)
Channel-out (ps=2)	0.9687	726000 (50)
Channel-out (ps=3)	0.9619	484000 (33.3)
Channel-out (ps=4)	0.9371	363000 (25)
Channel-out (ps=8)	0.6346	181500 (12.5)
Channel-out Absmax (ps=2)	0.9457	726000 (50)
Channel-out Absmax (ps=3)	0.9305	484000 (33.3)
Channel-out Absmax (ps=4)	0.8916	363000 (25)
Channel-out Absmax (ps=8)	0.6341	181500 (12.5)

So interestingly, despite a similar mechanism of determining relevant pathways within a network, channel-out does not

perform as well with channel pruning as ReLU does. It is likely that the sparse pathway encoding of channel-out is in some ways lost for certain samples of input when the network is pruned. With more time I would have liked to see the results of training with ReLU, using channel pruning to cut away some of the model parameters, and then retrain the pruned model with channel-out. I believe this might result in a trimmed model that still retains the necessary sparse pathway encoding.

3) Using Data to Determine Compression Factor: Returning to the first pruning model I discussed, if a determination could be made about the compression scale before training, then magnitude pruning could be used on the initialized weights (or, equivalently, the weights could be initialized sparsely) for perhaps the most efficient pruning method possible. Since the model has no information whatsoever before parameter initialization, the only way to make this determination is to gain some insight into the distribution of the input data itself. Thus, I conceived of a pipeline which first uses a lighter model to train and determine the compression factor, then passes this information to a neural model which uses it to prune the initialized weights and then train normally.

The lighter model I chose was none other than the original Feature Selection with Annealing model. It is fairly quick to train, the literature describes a nice, novel loss function which works well with magnitude as a measure of salience, and since it is a linear model we can assume that the compression will translate well into at least the first layer of a neural model. There is still the necessity of a pre-determined compression scale, however since the model is lighter than a neural model it is much faster to search for a sufficient value. Thus, I wrote FSA in Python as a SciKit-Learn compatible class for ease of use.

I again obtained results on MNIST for both the original FSA and a pruned neural model with the same set of compression scales:

Compression %	Accuracy (FSA)	Distance from Accuracy of Full Model (FSA)
0.2	0.8565	0.0255
0.4	0.8775	0.0045
0.6	0.8795	0.0025
1	0.882	0
Compression %	Accuracy (Pruned NN)	Distance from Accuracy of Full Model (Pruned NN)
0.2	0.9448	0.0160
0.4	0.9556	0.0052
0.6	0.96	0.0008
1	0.9608	0

While the results are not identical, the loss in accuracy at the same compression scales are in very similar ranges, indicating that this pruning pipeline might be a promising option, with some tweaks for efficiency. Further, I believe that, after exploring a variety of different methods of pruning, it may be more fruitful to understand how input to a model corresponds to the necessary structure (and especially an effective amount of sparsity) than it is to see how optimization of the model could inform the structure during or after training.

IV. Future Work

As mentioned throughout this report, there are a number of avenues that I would like to explore, given more time. In regards to the activation functions, I would be interested to implement maxout and channel-out in a variety of ensemble models to see how the sparse pathway encoding is affected. Promising results were certainly seen in [13], however, this was only one situation with one synthetic dataset, and further there was essentially no theoretical explanation for the performance.

With regards to pruning, I would like to further explore the relationship between the distribution/topological properties of input data and the most efficient structure of a model. This would include endeavors to test different mappings of input features and observe how the properties of the mappings affects performance under different model structures. I also came across a good deal of research on structural search methods, and I would be interested to see if I could construct a hierarchical pruning method that performs a similar task.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.

[2] R. Hahnloser, R. Sarpeshkar, M. Mahowald, R. Douglas, and H. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," *Nature*, vol. 405, pp. 947–51, 07 2000.

[3] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient BackProp*, pp. 9–48. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[4] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[5] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," 2017.

[6] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[7] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017.

[8] G. Zhang and H. Li, "Effectiveness of scaled exponentially-regularized linear units (serlus)," 2018.

[9] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," 2013.

[10] J. T. Springenberg and M. Riedmiller, "Improving deep neural networks with probabilistic maxout units," 2013.

[11] Q. Wang and J. JaJa, "From maxout to channel-out: Encoding information on sparse pathways," 2013.

[12] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, "19.1 models of hebbian learning," in *Neuronal Dynamics | From single neurons to networks and models of cognition*, Cambridge University Press.

[13] G. Melis, "Higgs boson machine learning challenge winning model documentation," 2014.

[14] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?," 2020.

[15] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," 2018.

[16] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," 2015.

[17] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," 2018.

[18] R. Reed, "Pruning algorithms-a survey," *IEEE Transactions on Neural Networks*, vol. 4, no. 5, pp. 740–747, 1993.

[19] A. Barbu, Y. She, L. Ding, and G. Gramajo, "Feature selection with annealing for computer vision and big data learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, p. 272–286, Feb 2017.