Improving Generalization of Feed-Forward Classifiers with Malicious Dropout

Jack Maris
Stanford University
jmaris@stanford.edu

Iskandar Pashayev
Stanford University
pashayev@stanford.edu

Abstract

Dropout is a technique proposed by Hinton et al. to improve the generalization capability of neural network models by randomly dropping out neurons at some hidden layer during training. We propose an alteration to the dropout technique – which we call Malicious Dropout – to improve generalization without adding too much computational overhead. Instead of randomly dropping out neurons, we drop out the neurons that may be useful to correct classification. The MalDrop computation increases training time approximately by a factor of two. Initial results are inconclusive.

1. Introduction

A network which successfully classifies many images of a particular category may have poorly learned notable features characteristic of that category. For example, consider a network that classifies most dogs by the curvature and location of their tails. The network would possess scant information about canine snouts, paws, body postures, etc. This then leads to generalization issues. In our example, front-facing images where the tail is occluded may be misclassified. The network failed to learn a fundamental feature of the category.

Simple measures like Dropout and L2 normalization can promote ”backup” associations. Dropout is especially appealing because it encourages the network to be robust to prominent missing features. When humans see a picture of a dog whose tail has been digitally edited out, they still recognize that the picture is of a dog; dropout encourages networks to do the same. We propose a modification to the Dropout algorithm that specifically drops neurons which are valuable to correct classifications. That is, if on some trial a dog-tail neuron $H^{Mal}_{dogtail}$ being 0 would most significantly increase L_i (say, $\mathbf{W}_{dogtail,\,dog}$ is high and all $H^{Mal}_{\neq\,dogtail}$ are all near 0), we set it to 0. The motivation is that by specifically setting the most prominent features to 0, we can encourage the network to learn other features as well. This may improve generalization and better capture important characteristic features of each class.

2. Problem Statement

We implement a measure we call *Malicious Dropout* towards the task of classifying images, *e.g.* in ImageNet. In our conception it serves as a drop-in replacement for a standard dropout layer in the penultimate hidden layer of a convolutional neural network. The goal is to formulate a layer which leads to better generalization than standard Dropout, perhaps in fewer epochs, without incurring dramatic computational overhead.

3. Literature Review

Dropout [5][1] is a technique which prevents overfitting in deep neural networks by randomly (uniformly) dropping neurons, thereby ”sampling” thinned networks from an exponential number of networks at training time [8]. While the uniformity of this drop probability makes sense from a regularizing perspective, it is not quite clear if it results in the *optimal* possible performance at test time. We found no papers studying the idea of modifying Dropout to target useful features. But significant work has been done on improving Dropout generally.

Nested dropout [6] is a modified dropout technique for data with ordered dimensions. The technique assigns a prior distribution – usually a geometric distribution – over the indexes of representation units, samples a random index b, and drops all of the units that have an index greater than b. However this technique is more effective with unsupervised learning methods and is not effective compared to the current supervised deep learning models.

Dropconnect [11] randomly drops weights rather than neurons. It is simple, performant, and has results comparable to Vanilla Dropout's. An aside: a Malicious version of Dropconnect would drop weights which are seen as important (by some metric) on a given trial. It would be an interesting topic for further research.

[1]Hereafter we will refer to it as vanilla Dropout to avoid confusion with Malicious Dropout.

4. Method and Implementation

4.1. Malicious Dropout layer

The MalDrop layer H^{Mal} takes in D neurons and outputs D neurons. ρ neurons are set to 0. We would like to determine exactly the set of neurons which maximize loss, but with high ρ this would be intractable, requiring $\binom{D}{\rho}$ forward passes from the MalDrop layer to the end. So we need an efficient approximate heuristic. It would be useful to have a heuristic value we could quickly assign each neuron H_j^{Mal} because once all are computed, we could select the top ρ neurons in $O(D\ log(\rho))$ time [1]. And we would compute the network's loss D times: once given that each neuron independently has been set to 0 (call it $L_{j\rightarrow 0}$). The neurons with the highest $L_{j\rightarrow 0}$ scores would be the ones to set to 0. But this assumes that the impact that setting H_j^{Mal} to 0 has on the loss depends little on how many other neurons had been set to 0 (and it also requires us to compute L exactly D times, which is intractable if the layer is early in the network). This is a problem.

Say we have a layer H^i, where two neurons H_a^i and H_b^i both have high (positively) weighted connections with one neuron in the next layer H_z^{i+1} (and scantly weighted connections with the others). Also, $H_a^i >> 0, H_b^i << 0$. And, the layer after that is a ReLU. If we set only H_a^i to 0 then $H_z^{i+1}$is negative and ReLU'd to 0. If we set only H_b^i to 0 then $H_z^{i+1}$is a high positive value (and remains that way after the ReLU). But if we set both to 0, the result is close to 0. The gist is that neural networks encode complex non-linear relationships, so in most situations taking the top ρ neurons by $L_{j\rightarrow 0}$ is a poor heuristic.

But some layers can represent relationships we can work with. An obvious case is the last layer, which in our case is then softmax'ed out to determine final scores. To maximize loss, we can set $H_{y_i}^{Final}$to 0 if its score is greater than 0, and set those neurons $H_{\neq y_i}^{Final}$ which have scores less than 0 to 0. But this case is not useful because then the network would not learn much: we could set $H_{y_i}^{Final}$ as high as we would like, and the best we would be able to do would be to have it at parity with $\rho - 1$ other categories.

Instead we can implement MalDrop in a penultimate layer which is followed by an Fully-Connected layer, which is softmax'ed. And our heuristic (which I reiterate is approximate) is thus (with the hyperparameter ω): for each neuron H_j^{Mal}, compute

$$V_j = H_j^{Mal} W_{j,y_i} - \omega H_j^{Mal} \sum_k (W_{j,k\neq y_i})$$

This is what H_j^{Mal} "contributes" to the last layer's right category minus the sum of what H_j^{Mal} "contributes" to the last layer's wrong categories (weighted by ω, usually less than 1[2].). Because $\frac{e^{x_j}}{\sum_{j'} e^{x_{j'}}}$ is monotonically increasing with respect to x_j, we at least know exactly whether any neuron in H^{Mal} will increase or decrease some category c's presoftmax value. There is no guarantee that the order of "contribution" corresponds to the best-possible combination of ρ neurons to set to 0. That is, a category $c \neq y_i$ with a low score should be unimportant, while one with a score near y_i *should* be important. It is an empirical question whether this ends up mattering. But the simple weighted sum is efficient enough, and the heuristic does not have to be perfect. We run tests on 2 networks later to determine how well this heuristic works.

4.2. Test Scaling

During training we track how frequently a neuron is let through. For neuron H_j^{Mal}, call it p_j^{Pass}. In normal Dropout by the Law of Large Numbers $\lim_{i\to\infty} p_j^{Pass} = p$. In MalDrop we have no such guarantee. The computation of each p_j^{Pass} is not computationally expensive, however.

At validation and test time we do dropping, only multiplying each neuron by its p_j^{Pass}, the expected probability of passing during training. And this is similar to Vanilla Dropout's scaling by p.

4.3. Network Architectures

We begin with a simple network architecture as a proof-of-concept, where we can compare no-Dropout, Vanilla Dropout, and Malicious Dropout for efficiency and performance with reasonable ρ. Our first network has this structure:

- A convolutional layer with depth=10, a kernel that is 3x3, padding=1, and stride=1,
- A ReLU,
- A spatial batchnorm layer,
- Another convolutional layer, with depth=1, a kernel that is 3x3, padding=1, and stride=1,
- Another ReLU,
- One of **No Dropout**, **Vanilla Dropout**, or **MalDrop**.
- An affine layer.

This network is easy to work with, but powerful enough to give us $\approx$ 0.55% Top-1 accuracy on CIFAR-10 with Vanilla Dropout and some ρ-schedules of MalDrop.

We wanted to compare performance with a conventional network too. We searched for a well-studied network whose last two layers were Dropout and then Fully-Connected.

[2]To balance the positive and negative components, consider setting $\omega = \frac{1}{D-1}$

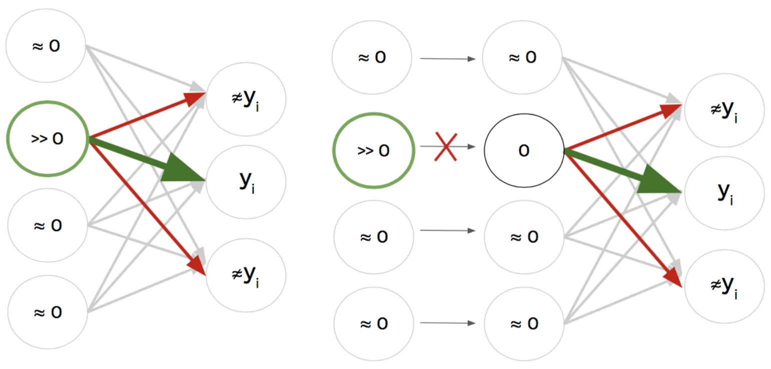


Figure 1. *left:* A normal affine layer. *right:* An affine layer after MalDrop has been applied, with $\rho = 1$. Grey weights correspond to weights near 0, red to significantly negative, green to significantly positive.

VGG[7] fit the bill (we specifically used PyTorch's VGG-16 from `torchvision.models`).

Both networks use cross-entropy loss, and the optimization function is Stochastic Gradient Descent. We did little hyperparameter tuning: our task is theoretical so we focus on comparing relative performance, not squeezing out optimal performance. And while we did explore different settings of ρ, we did not do so with ω.

4.4. ρ Scheduling

We describe five different ways of setting the hyperparameter ρ for a MalDrop layer.

Standard. The user passes a constant ρ value, and for each iteration we drop the top ρ neurons contributing the lowest loss in the penultimate layer.

Randomized. For each batch, we sample a random ρ. Then as in the standard scheduling, we drop the top ρ neurons contributing the lowest loss in the penultimate layer. The distribution from which ρ is sampled is the hyperparameter in this case. However, we expect the Uniform distribution to be the best performer in this case. As such, we sample $\rho \sim Uniform(0, L)$, where L is the number of neurons in the MalDrop layers. Other choices of distribution are left as a future exercise.

Forward Annealing. The number of neurons dropped for each batch increases with each successive epoch. In this case, the user passes in an initial value of ρ from which we compute the number ρ_i of neurons to drop, where i is the current epoch. We use a logistic function bounded by 0 and L to have an increasing sequence of ρ_i, where as before L is the number of neurons in the MalDrop layer. More specifically, ρ_i is determined by the logistic function

$$\rho_i = \frac{L}{1 + e^{-(\rho+i)}}. \tag{1}$$

Reverse Annealing. This form of scheduling is similar to forward annealing, but instead of monotonically increasing the number of neurons dropped, we monotonically decrease it. To achieve this, given a user-supplied ρ, we determine the number ρ_i of neurons to drop during epoch i as

$$\rho_i = L \cdot \left(1 - \frac{1}{1 + e^{-(\rho+i)}}\right). \tag{2}$$

Note, that we also could have retained the formula for Forward Annealing and simply subtracted i from ρ instead of adding it. However, we decided against this because when $i = 0$, both forward and reverse annealing then begin with the same number of neurons dropped. In such a case, unless $\rho_0 = \frac{L}{2}$, the rate of change of ρ_i for forward annealing would be different from reverse annealing, and we wished to preserve the symmetry of these rates of change.

Periodic. The MalDrop layer is only active every t_{period} batches (t_{period} being a hyperparameter). The idea behind this is that MalDrop may be most useful as a counterweight against a normal network. Our early recommendation is $t_{period} = 5$: rare enough that the network learns, but frequent enough that less dominant features are allowed to be learned.

4.4.1 Datasets and Features

We intend to focus on classifying the ImageNet dataset using convolutional neural networks. Here, vanilla dropout has proven effective [8]. Other architectures seem poorly suited for dropout, e.g. recurrent neural networks [12]. For our preliminary results we use the MNIST and CIFAR10 datasets to compare the effectiveness of networks with vanilla dropout to network with malicious dropout. For a more detailed evaluation of the effectiveness of the MalDrop technique, we use the Tiny ImageNet Dataset.

5. Results

The code that the following results are based on is hosted at `git.io/maldrop`. These results are based on an implementation within the PyTorch[4] and NumPy[3] frameworks. The code references example projects on Justin Johnson's Github profile[2] and PyTorch's Github profile. These experiments were performed on a Google Cloud Compute Engine VM instance of Ubuntu 16.04 LTS, run with one GPU accelerated by an NVIDIA Tesla K80.

5.1. MNIST

We test our model on the MNIST handwritten digits dataset as a preliminary evaluation of our model. The following table provides a comparison of the loss after 10 epochs and the validation accuracies at the specified epochs for the aforementioned basic CNN, with different choices of dropout applied to the penultimate layer.

Dropout	Loss	Epoch 1	Epoch 5	Epoch 10
None	0.099	91.63%	97.31%	97.91%
Vanilla	0.072	91.57%	96.95%	97.81%
Malicious	0.082	91.04%	94.83%	97.46%

As we see, the architecture with MalDrop ($\rho = 256$, out of a layer with 4,096 neurons) learns slower than the other architectures. This is as expected since the network actively drops the features that would help it classify correctly with the most ease. However, by the end of the full 10 epochs, MalDrop's performance is on par with that of the other two networks. Given the simplicity of the dataset, we cannot draw any conclusions about the generalization capability of the MalDrop architecture. However, the architecture is still able to learn more specialized and simpler datasets like the handwritten digits of MNIST.

5.1.1 Time-Space Tradeoff.

We do notice a significant slowdown in the time it takes for the network to learn with Malicious Dropout as opposed to the other two choices of dropout. However, the extra time it takes for the computation can be mitigated by using extra space. More specifically, we can use more space and perform parallelized computations when calculating the contribution of each neuron to the final loss. However, this is a huge problem when training more complex networks on larger datasets, since the space required to parallelize these computations outscales memory on modern computing hardware.

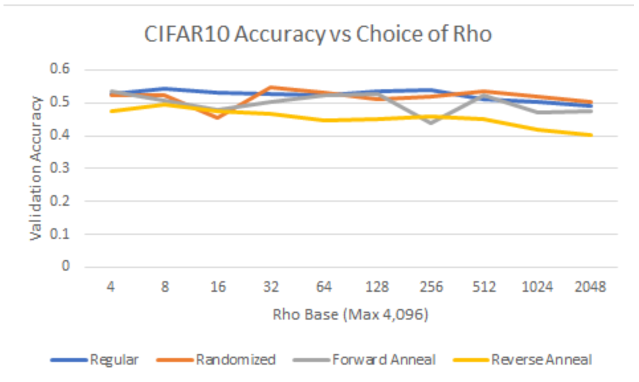
As a baseline, the architecture with Vanilla Dropout runs in 79 seconds and the architecture with no dropout runs in 77 seconds. By using more memory, the MalDrop layer runs in 184 seconds. Even though the time taken to train doubles, the computational overhead is still manageable. Without using space to improve the computation time, we expect the network with the MalDrop layer to take approximately 28,125 seconds, or almost eight hours for 10 epochs. This computational overhead is significantly more inefficient and impractical compared to the architectures with vanilla dropout and without dropout.

5.2. CIFAR10

We use the same architecture to evaluate MalDrop on CIFAR10 as we did for MNIST. After ten epochs, the No Dropout architecture yielded a validation accuracy of 52.5%, taking approximately 96 seconds to train, and the Vanilla Dropout architecture yielded an accuracy of 54.4%, taking approximately 97 seconds to train. The performance after ten epochs of the MalDrop architecture is similar to the performance of the Vanilla Dropout architecture; the MalDrop architecture fluctuates around 54.5%, when setting ρ carefully. Our MalDrop architecture utilizes the Time-Space tradeoff optimization that we also utilized for MNIST. As a result, training the MalDrop architecture for 10 epochs takes about 180-190 seconds. This increase is slightly relatively larger than the one for MNIST, but still a manageable computational overhead. Data for the MalDrop architecture is given in the figure in Section 5.2.1.

5.2.1 ρ Scheduling

We also used the CIFAR10 dataset to compare the performance of MalDrop under different schedules of ρ. The following figure illustrates the performance of the MalDrop architecture for different settings of rho under the Standard, Forward Annealing, and Reverse Annealing modes. We also show the performance under the Randomized mode, where ρ is sampled from a Uniform distribution bounded by 0 and 4,096, which is the number of neurons input into the final affine layer. We note that in this case, setting ρ for randomized mode does not have an impact. Rather, the choices of ρ under randomized shown in the figure serve as trials for which we sample a random ρ and measure the performance. This further helps us to visualize the variance that the MalDrop layer induces onto the network through its setting of ρ.



As we can see, the schedule of ρ does not induce much variation on the final loss. This is because as we see from the Standard mode, and indirectly from the Randomized mode, ρ itself does not affect the final accuracy by much. Even though the Reverse Annealing mode seems like it performs worse than the other three modes, this could be attributable to its definition. It is still unclear if equations (1) and (2) are optimal methods for annealing ρ. In particular, from superfluous checks it seems that the updating mechanism with which to ”trace” ρ_i along the logistic function (i.e. adding the epoch number to the base ρ) is inadequate for annealing ρ_i.

5.3. Tiny ImageNet

We chose VGG16-Net to test MalDrop’s generalization on a more complex dataset, since it had a Dropout layer as its penultimate layer. Furthermore, when we ran our simplified network with two convolutional layers, the network was unable to learn any meaningful results, regardless of the No Dropout, Vanilla Dropout, and Malicious Dropout options. We tested the performance of VGG16-Net with its standard dropout layer, its dropout layer missing, and with a MalDrop layer. All three networks had a learning rate of 10^{-3}. The network with the MalDrop layer trained for 4 epochs, whereas the other two trained for 5 epochs. We discuss the number of epochs in Section 5.3.2. Figures 2, 3, and 4 summarize the performance of the models.

We observe that the performance of VGG16-Net with the MalDrop layer fails to learn at all. The loss is relatively constant, which suggests that perhaps the network parameters are not updated in a meaningful way. Even though the loss for the architectures with vanilla dropout and without dropout have increasing losses, their accuracies are nonetheless higher than that of the MalDrop architecture, the top-1 accuracy of which remains constant at 0. The increase in the top-5 accuracy of the MalDrop architecture after the fourth epoch is interesting to note, but could still be an aberration.

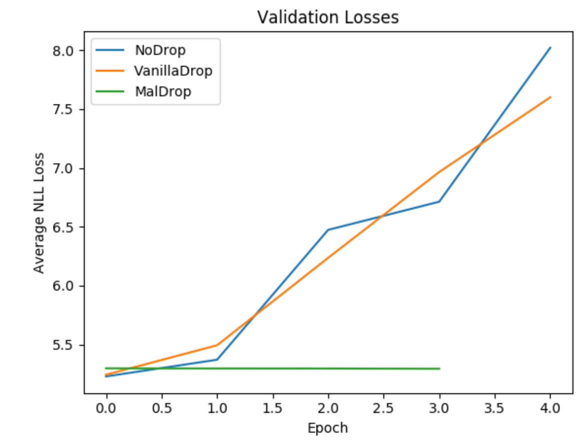


Figure 2. Validation losses after each epoch for VGG16-Nets with different settings of dropout.

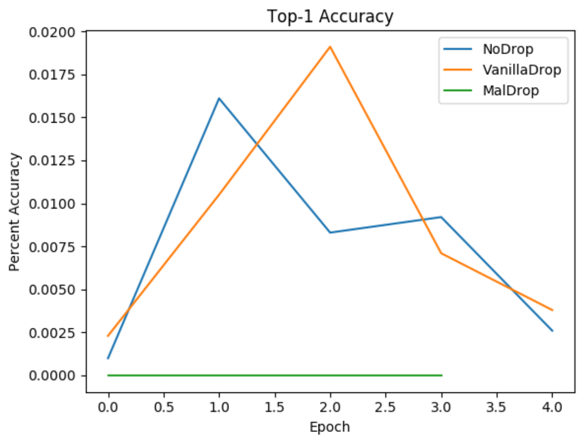


Figure 3. Top-1 accuracies after each epoch for VGG16-Nets with different settings of dropout.

5.3.1 Discussion

Overall, it appears that the VGG-16 MalDrop Network failed to begin learning. This could be a result of dropping the most important neurons during the start of training, preventing the network from learning. We have already seen in MNIST results, that even though MalDrop is as effective as regular dropout after 10 epochs, it converged slower than the VanillaDrop and NoDrop networks. Since MalDrop networks effectively ignore the most obvious features, they have to work more to ”catch up” during training. The complexity of the ImageNet dataset increases the difficulty with which a MalDrop architecture has to learn many different features. Furthermore, the low number of epochs for which the MalDrop network trained is most likely insufficient time for the MalDrop to begin learning. Training the network for more epochs could show a MalDrop network to begin learning.

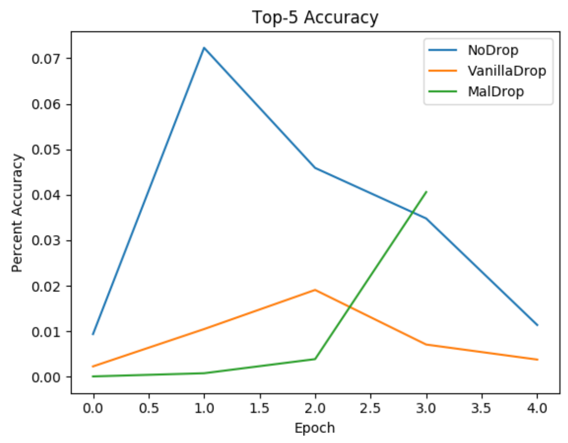


Figure 4. Top-5 accuracies after each epoch for VGG16-Nets with different settings of dropout.

Another note to mention is that we found it surprising that the loss through 5 epochs was increasing, at least for the VGG16 networks with VanillaDrop and NoDrop. As a result, we reran the tests with the learning rate set to 10^{-1}. However, while testing with a higher learning rate resulted in improved performance (and decreasing loss) for the networks without MalDrop, the performance decreased for the MalDrop network. Indeed, both sets of top-1 and top-5 accuracies for the MalDrop network were 0 throughout the first four epochs.

5.3.2 Running Time

To compute the scores of one batch of size 320 and back-propagate gradients, the VGG16 NoDrop and VanillaDrop networks take approximately 15 seconds, whereas the MalDrop network takes 31 seconds. This is consistent with the compute time ratios seen for the MNIST and CIFAR10 datasets. Consequently, to train 5 epochs of 100,000 examples requires the MalDrop network to take approximately 12-13 hours, or double the 6-7 hours it takes to train the VGG16 networks with and without the penultimate vanilla dropout layer. Due to time constraints and the lengthy amount of time taken to train a MalDrop network, we trained the MalDrop network for four epochs. As such, while there is still a possibility that MalDrop may help networks generalize better, the added computational difficulty makes it impractical for networks to efficiently learn how to classify complex visual data.

6. Conclusion and Proposed Future Work

We have established the theoretical basis behind our MalDrop layer. Still the data is insufficient to conclude that MalDrop is useful in any practical domain. We contend that it may be useful with the right settings: ρ schedule, ρ_0 setting, ω setting[3], and network architecture. At the very least our proposed trade-off between efficiency and accuracy is not a computational bottleneck. And because our layer is efficient future work should focus on establishing useful settings. An immediate example here would be a smoother way of annealing ρ_i along the logistic function, as suggested in Section 5.2.1.

Other work should consider how useful it would be to calculate the *exact* best neurons to drop (*i.e.* those that conjunctively maximize L) exactly rather than using our heuristic. It would be comically expensive to compute *each* combination: VGG16-Mal has 4,096 neurons. If $\rho = 10$, this comes out to $\binom{4096}{10} = 3.62 \times 10^{29}$ loss computations. But on a small network with a small ρ, we would expect the computation to be tractable. To us it remains to be seen that *exactly* honing in on the right features will lead to much better performance on the training set or generalization. And even if the *exact* results are impressive, they are impossible to use on a normally-sized network. A MalDrop layer, for its flaws, manages to specifically target *some* important neurons while being only slightly slower than a vanilla-Dropout network.

Spatial dropout drops channels (vectors) rather than individual neurons [10]. It is worth studying Spatial Malicious Dropout. That is, dropping ρ channels instead of ρ neurons. It requires an architecture where the penultimate layer is not one-dimensional, but would otherwise be straightforward: compute $L_{j\to 0}$for each channel rather than each neuron.

Malicious Dropout could be applied at any level before one which take loss. Thus networks which propagate loss as multiple levels like GoogLeNet [9] would yield interesting results. However this would require the calculation to effectively ignore the loss coming from distant layers when determining which ρ neurons to drop.

MalDrop can be combined with Vanilla Dropout. This can be conceptualized as either one layer or a dual-part layer: a neuron is culled with some probability p, and then the remaining neurons are Maldropped[4]. Or, visa-versa. Future work could examine this.

References

[1] Finding the top k items in a list efficiently. `http://stevehanov.ca/blog/index.php?id=122`. Accessed: 2017-06-11.

[2] Justin johnson github profile. `https://github.com/jcjohnson/`. Accessed: 2017-06-12.

[3] Numpy github profile. `https://github.com/numpy/`. Accessed: 2017-06-12.

[3]Recall ω balances how much of a neuron's score comes from y_i versus $y_{\neq i}$. We recommend values orders of magnitude below 1, and experimenting.

[4]Though these conceptions are not strictly identical: it is possible for a neuron which equals 0 to be set to 0 in Maldrop.

[4] Pytorch github profile. `https://github.com/pytorch/`. Accessed: 2017-06-12.

[5] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[6] O. Rippel, M. Gelbart, and R. Adams. Learning ordered representations with nested dropout. In *International Conference on Machine Learning*, pages 1746–1754, 2014.

[7] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[8] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[10] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.

[11] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1058–1066, 2013.

[12] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.