DEEP NEURAL NETWORKS AND PROCEDURE FOR TRAINING DROPOUT NEURAL NETWORKS

[1]Punith Kumar, [2]Nabi M, [3]Devaraj Biradar
[1]Guest Faculty, [2]PG Student, [3]PG Student
[1]Department of Computer Science,
[1]Bangalore University, Bangalore, India

Abstract: Deep neural nets with a large number of parameters are very powerful machine learning systems. However, over fitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with over fitting by combining the predictions of many different large neural nets at test time. Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much.

***IndexTerms* -** neural networks, regularization, deep learning

I INTRODUCTION

Deep learning is the name we use for “stacked neural networks”; that is, networks composed of several layers. The layers are made of *nodes*. A node is just a place where computation happens, loosely patterned on a neuron in the human brain, which fires when it encounters sufficient stimuli. A node combines input from the data with a set of coefficients, or weights that either amplify or dampen that input, thereby assigning significance to inputs for the task the algorithm is trying to learn. (For example, which input is most helpful is classifying data without error?) These input-weight products are summed and the sum is passed through a node’s so-called activation function, to determine whether and to what extent that signal progresses further through the network to affect the ultimate outcome, say, an act of classification.

Dropout is one of the recent advancement in Deep Learning that enables us to train deeper and deeper network. Essentially, Dropout act as a regularization, and what it does is to make the network less prone to over fitting. As we already know, the deeper the network is, the more parameter it has. For example, VGGNet from ImageNet competition 2014, has some 148 million parameters. That’s a lot. With that many parameters, the network could easily overfit, especially with small dataset.

Enter Dropout. In training phase, with Dropout, at each hidden layer, with probability `p`, we kill the neuron. What it means by ‘kill’ is to set the neuron to 0. As neural net is a collection multiplicative operations, then those 0 neuron won’t propagate anything to the rest of the network.

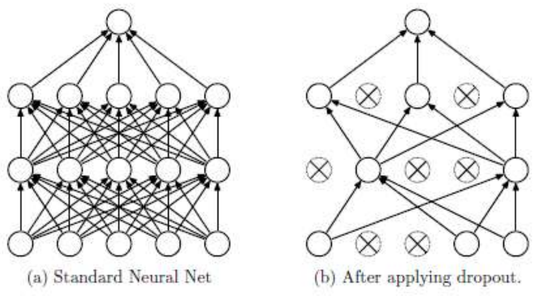

Figure 1: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Let n be the number of neuron in a hidden layer, then the expectation of the number of neuron to be active at each Dropout is p*n, as we sample the neurons uniformly with probability p. Concretely, if we have 1024 neurons in hidden layer, if we set p = 0.5, then we can expect that only half of the neurons (512) would be active at each given time.

Because we force the network to train with only random p*n of neurons, then intuitively, we force it to learn the data with different kind of neurons subset. The only way the network could perform the best is to adapt to that constraint, and learn the more general representation of the data.It's easy to remember things when the network has a lot of parameters (overfit), but it's hard to remember things when effectively the network only has so many parameters to work with. Hence, the network must learn to generalize more to get the same performance as remembering things.

So, that's why Dropout will increase the test time performance: it improves generalization and reduce the risk of over fitting. Dropout is a technique that addresses both these issues. It prevents over fitting and provides a way of approximately combining exponentially many different neural network architectures efficiently. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Figure 1. The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability p independent of other units, where p can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks. For the input units, however, the optimal probability of retention is usually closer to 1 than to 0.5.

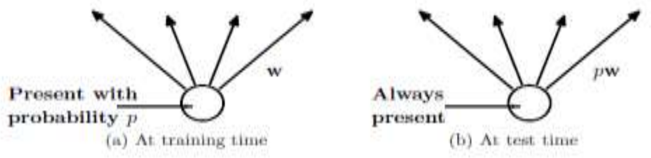

Figure 2: Left: A unit at training time that is present with probability p and is connected to units in the next layer with weights w. Right: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Applying dropout to a neural network amounts to sampling a "thinned" network from it. The thinned network consists of all the units that survived dropout (Figure 1b). A neural net with n units, can be seen as a collection of 2n possible thinned neural networks. These networks all share weights so that the total number of parameters is still O(n2), or less. For each presentation of each training case, a new thinned network is sampled and trained. So training a neural network with dropout can be seen as training a collection of 2n thinned networks with extensive weight sharing, where each thinned network gets trained very rarely, if at all. At test time, it is not feasible to explicitly average the predictions from exponentially many thinned models. However, a very simple approximate averaging method works well in practice. The idea is to use a single neural net at test time without dropout. The weights of this network are scaled-down versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by past test time as shown in Figure 2. This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time. By doing this scaling, 2n networks with shared weights can be combined into a single neural network to be used at test time. We found that training a network with dropout and using this approximate averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods.

II RELATED WORK

Dropout can be interpreted as a way of regularizing a neural network by adding noise to its hidden units. The idea of adding noise to the states of units has previously been used in the context of Denoising Autoencoders (DAEs) by Vincent et al. (2008, 2010) where noise is added to the input units of an autoencoder and the network is trained to reconstruct the noise-free input. Our work extends this idea by showing that dropout can be effectively applied in the hidden layers as well and that it can be interpreted as a form of model averaging. We also show that adding noise is not only useful for unsupervised feature learning but can also be extended to supervised learning problems. In fact, our method can be applied to other neuron-based architectures, for example, Boltzmann Machines. While 5% noise typically works best for DAEs, we found that our weight scaling procedure applied at test time enables us to use much higher noise levels. Dropping out 20% of the input units and 50% of the hidden units was often found to be optimal. Since dropout can be seen as a stochastic regularization technique, it is natural to consider its deterministic counterpart which is obtained by marginalizing out the noise.

Recently, van der Maaten et al. (2013) also explored deterministic regularizers corresponding to different exponential-family noise distributions, including dropout (which they refer to as “blankout noise"). However, they apply noise to the inputs and only explore models with no hidden layers. Wang and Manning (2013) proposed a method for speeding up dropout by marginalizing dropout noise. Chenet al. (2012) explored marginalization in the context of denoising autoencoders. In dropout, we minimize the loss function stochastically under a noise distribution. This can be seen as minimizing an expected loss function. Previous work of Globerson and Roweis (2006); Dekel et al. (2010) explored an alternate setting where the loss is minimized when an adversary gets to pick which units to drop. However, this work also does not explore models with hidden units.

III PROCEDURE FOR TRAINING DROPOUT NEURAL NETS

Backpropagation:

Dropout neural networks can be trained using stochastic gradient descent in a manner similar to standard neural nets. The only difference is that for each training case in a mini-batch, we sample a thinned network by dropping out units. Forward and back propagation for that training case are done only on this thinned network. The gradients for each parameter are averaged over the training cases in each mini-batch. Any training case which does not use a parameter contributes a gradient of zero for that parameter. Many methods have been used to improve stochastic gradient descent such as momentum, annealed learning rates and L2 weight decay. Those were found to be useful for dropout neural networks as well. One particular form of regularization was found to be especially useful for dropout constraining the norm of the incoming weight vector at each hidden unit to be upper bounded by a fixed constant c. In other words, if w represents the vector of weights incident on any hidden unit, the neural network was optimized under the constraint $||w||2 \leq c$. This constraint was imposed during optimization by projecting w onto the surface of a ball of radius c, whenever w went out of it. This is also called max-norm regularization since it implies that the maximum value that the norm of any weight can take is c. The constant c is a tunable hyperparameter, which is determined using a validation set. Max-norm regularization has been previously used in the context of collaborative filtering (Srebro and Shraibman, 2005). It typically improves the performance of stochastic gradient descent training of deep neural nets, even when no dropout is used. Although dropout alone gives significant improvements, using dropout along with maxnorm regularization, large decaying learning rates and high momentum provides a significant boost over just using dropout. A possible justification is that constraining weight vectors to lie inside a ball of fixed radius makes it possible to use a huge learning rate without the possibility of weights blowing up. The noise provided by dropout then allows the optimization process to explore different regions of the weight space that would have otherwise been difficult to reach. As the learning rate decays, the optimization takes shorter steps, thereby doing less exploration and eventually settles into a minimum.

Unsupervised Pretraining:

Neural networks can be pretrained using stacks of RBMs (Hinton and Salakhutdinov, 2006), autoencoders (Vincent et al., 2010) or Deep Boltzmann Machines (Salakhutdinov and Hinton, 2009). Pretraining is an effective way of making use of unlabeled data. Pretraining followed by finetuning with backpropagation has been shown to give significant performance boosts over finetuning from random initializations in certain cases. Dropout can be applied to finetune nets that have been pretrained using these techniques. The pretraining procedure stays the same. The weights obtained from pretraining should be scaled up by a factor of 1/p. This makes sure that for each unit, the expected output from it under random dropout will be the same as the output during pretraining. We were initially concerned that the stochastic nature of dropout might wipe out the information in the pretrained weights. This did happen when the learning rates used during finetuning were comparable to the best learning rates for randomly initialized nets. However, when the learning rates were chosen to be smaller, the information in the pretrained weights seemed to be retained and we were able to get improvements in terms of the final generalization error compared to not using dropout when fine tuning.

Let’s see the concrete code for Dropout:

```
# Dropout training
u1 = np.random.binomial (1, p, size=h1.shape)
h1 *= u1
```

First, we sample an array of independent Bernoulli distribution, which is just a collection of zero or one to indicate whether we kill the neuron or not. For example, the value of u1 would be np.array ([1, 0, 0, 1, 1, 0, 1, 0]). Then, if we multiply our hidden layer with this array, what we get is the original value of the neuron if the array element is 1, and 0 if the array element is also 0.

For example, after Dropout, we need to do h2 = np.dot (h1, W2), which is a multiplication operation. What is zero times x? It's zero. Then the subsequent multiplications would be also zero. That's why those 0 neurons won't contribute anything to the rest of the propagation.

Now, because we're only using p*n of the neurons, the output then has the expectation of p*x, if x is the expected output if we use all the neurons (without Dropout). As we don't use Dropout in test time, then the expected output of the layer is x. That doesn't match with the training phase. What we need to do is to make it matches the training phase expectation, so we scale the layer output with p.

```
# Test time forward pass
h1 = X_train @ W1 + b1
h1[h1 < 0] = 0

# Scale the hidden layer with p
h1 *= p
```

In practice, it's better to simplify things. It's cumbersome to maintain codes in two places. So, we move that scaling into the Dropout training itself.

```
# Dropout training, notice the scaling of 1/p
u1 = np.random.binomial (1, p, size=h1.shape) / p
h1 *= u1
```

With that code, we essentially make the expectation of layer output to be x instead of px, because we scale it back with 1/p. Hence in the test time, we don't need to do anything as the expected output of the layer is the same.

Dropout backprop

During the backprop, what we need to do is just to consider the Dropout. The killed neurons don't contribute anything to the network, so we won't flow the gradient through them.

```
dh1 *= u1
```

Test and Comparison

Test time! But first, let's declare what kind of network we will use for testing.

```
def make_network(D, C, H=100):
    model = dict(
        W1=np.random.randn(D, H) / np.sqrt(D / 2.),
        W2=np.random.randn(H, H) / np.sqrt(H / 2.),
        W3=np.random.randn(H, C) / np.sqrt(H / 2.),
        b1=np.zeros((1, H)),
        b2=np.zeros((1, H)),
        b3=np.zeros((1, C))
    )

    return model


model = nn.make_network(D, C, H=256)
```

We're using three layers network with 256 neurons in each hidden layer. The weights are initialized using Xavier divided by 2, as proposed by He, et al, 2015. The data used are MNIST data with 55000 training data and 10000 test data. The optimization algorithm used is RMSprop with 1000 iterations, repeated 5 times and the test accuracy is averaged.

```
# Without Dropout
rmsprop => mean accuracy: 0.9640, std: 0.0071

# With Dropout
rmsprop => mean accuracy: 0.9692, std: 0.0006
```

Looking at the result, model which use Dropout yield a better accuracy across the test set. The difference of 0.005 might be negligible, but considering we have 10000 test data, that's quite a bit. The standard deviation of the test tells different story though. It seems that the network that uses Dropout for training perform consistently better during test time. Compare it to the non-Dropout network, it's an order of magnitude worse in term of consistency. We can see this when comparing the standard deviation: 0.0006 vs 0.0071.However, when we look at the convergence of the network during training, it seems that non-Dropout network converge better and faster. Here, we could see at the loss at every 100 iterations.

```
# Without Dropout
Iter-100 loss: 0.7141623845363005
Iter-200 loss: 0.5242217596766273
Iter-300 loss: 0.37112553379849605
Iter-400 loss: 0.38909851968506987
Iter-500 loss: 0.25597567938296006
Iter-600 loss: 0.30120887447912315
Iter-700 loss: 0.24511170871806906
Iter-800 loss: 0.23164132479234184
Iter-900 loss: 0.18410249409092522
Iter-1000 loss: 0.21936521877677104

# With Dropout
Iter-100 loss: 0.8993988029028332
Iter-200 loss: 0.761899148472519
Iter-300 loss: 0.6472785867227253
Iter-400 loss: 0.4277826704557144
Iter-500 loss: 0.48772494633262575
Iter-600 loss: 0.35737694600178316
Iter-700 loss: 0.3650990861796465
Iter-800 loss: 0.30701377662168766
Iter-900 loss: 0.2754936912501326
Iter-1000 loss: 0.3182353552441539
```

This indicates that network without Dropout performs better at the training phase while Dropout network perform worse. The table is turned at the test time, Dropout network is not just perform better, but *consistently better*. One could interpret this as the sign of over fitting. So, really, we could see that Dropout regularize our network and make it more robust to over fitting.

IV CONCLUSION

We look at one of the driving force of the recent advancement of Deep Learning: Dropout. It's a relatively new technique but already made a very big impact in the field. Dropout act as regularizer by stochastically kill neurons in hidden layers. This in turn force the network to generalize more.

We also implement Dropout in our model. Implementing Dropout in our neural net model is just a matter of several lines of code. We found that it’s a very simple method to implement. We then compare the Dropout network with non-Dropout network. The result is nice: Dropout network performs consistently better in test time compared to the non-Dropout Network.

REFERENCES

[1] O. Dekel, O. Shamir, and L. Xiao. Learning to classify with missing and corrupted features. Machine Learning, 81(2):149{178, 2010.

[2] A. Globerson and S. Roweis. Nightmare at test time: robust learning by feature deletion. In Proceedings of the 23rd International Conference on Machine Learning, pages 353{360. ACM, 2006.

[3] I. J. Goodfellow, D.Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. In Proceedings of the 30th International Conference on Machine Learning, pages 1319{ 1327. ACM, 2013.

[4] G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. Neural Computation, 18:1527{1554, 2006.

[5] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541{551, 1989.

[6] S. Wager, S. Wang, and P. Liang. Dropout training as adaptive regularization. In Advances in Neural Information Processing Systems 26, pages 351{359, 2013.

[7] S.Wang and C. D. Manning. Fast dropout training. In Proceedings of the 30th International Conference on Machine Learning, pages 118{126. ACM, 2013.

[8] N. Srebro and A. Shraibman. Rank, trace-norm and max-norm. In Proceedings of the 18th annual conference on Learning Theory, COLT'05, pages 545{560. Springer-Verlag, 2005.

[9] N. Srivastava. Improving Neural Networks with Dropout. Master's thesis, University of Toronto, January 2013.