Review for Variational Dropout Sparsifies Deep Neural Networks

Bao Jinge(A0214306U)[1] and Zhang Runtian(A0212492L)[2]

[1]Centre for Quantum Technologies, National University of Singapore
e0522065@u.nus.edu
[2]School of Computing, National University of Singapore
e0504862@u.nus.edu

1 Problem

As we knows, people need a good method to keep deep neural networks from overfitting, and it's even better if it can compress the model effectively.
This problem is important for theory and application, because DNNs require training of a large number of parameters and are prone to overfitting, which not only hurt the capacity of model, but also waste computing resources. Some regularization methods was tried before dropout such as l_0 and l_1 regularization, but the co-adaption of weights makes it not the perfect solution [1]. When neural networks go deeper, some specific weights get control of majority weights, and more iterations only strengthens these weights in fact, which induce the network into sparsity. Thus, we use dropout to stop the co-adaptation between weights of different layers.
The paper we mainly refer to and review is *Variational Dropout Sparsifies Deep Neural Networks* on ICML17 [2].

2 Contribution

2.1 From Binary Dropout to Variational Dropout

The paper reviewed the whole theory contribution process of dropout technique. Consider a fully-connected layer in a DNN, and denote the output matrix as O_{m*n}, input matrix as A_{m*i}, weight matrix as W_{i*n}. n is the number of output neurons and i is the number of input neurons. We have $O = A * W$ and after using dropout it is $O = [A \oplus P]W$, which means product in each element of matrix A. Binary Dropout set each element of P to be 0 with probability p, the drop rate.
In Gaussian Dropout the $p \sim \mathcal{N}(1, \alpha)$, and multiplying Gaussian noise on inputs is equivalent to multiplying it on weights, which obtains a posterior distribution over weights.

$$w_{ij} = \theta_{ij}\xi_{ij} = \theta_{ij}(1 + \sqrt{\alpha}\epsilon_{ij}) \sim \mathcal{N}(w_{ij}|\theta_{ij}, \alpha\theta_{ij}^2) \tag{1}$$

with $\epsilon_{ij} \sim \mathcal{N}(0, 1)$. The matrix w per minibatch can be seen as a sample from $N(\theta_{ij}, \alpha\theta_{ij}^2)$. They also introduced Variational Drop, which is equivalent to Gaussian Dropout if we chose the prior

distribution $p(W)$ to be improper logscale uniform.

$$p(log|w_{ij}|) = const \Leftrightarrow p(|w_{ij}|) \propto \frac{1}{|w_{ij}|} \tag{2}$$

Then because α is fixed, the variational lower bound $\mathcal{L}(\Phi) = L_{\mathcal{D}}(\Phi) - D_{KL}(q_{\Phi(w)}||p(w))$ depends on α only, where $q_{\Phi}(w)$ is a parametric distribution which posterior distribution $p(w|\mathcal{D})$ is approximated by. They noticed that the drop rate α is a variational parameter which can be trained for each layer, each neuron, or each weight.

2.2 Sparse Variational Dropout

The biggest innovation of this paper, is that they liberated the bound ≤ 1 of the original variational dropout, and solved the difficulties training the model with large drop rates. The core contributions are additive noise reparameterization and an approximation of KL divergence.

2.2.1 Additive Noise Reparameterization

The original variational dropout faces the problem of huge variance of stochastic gradients, which is caused by multiplicative noise. To get the gradient

$$\frac{\partial L^{SGVB}}{\partial \theta_{ij}} = \frac{\partial L^{SGVB}}{\partial w_{ij}} * \frac{\partial w_{ij}}{\partial \theta_{ij}} \tag{3}$$

we need compute Equation 1. Thus we have

$$\frac{\partial w_{ij}}{\partial \theta_{ij}} = 1 + \sqrt{\alpha} * \epsilon_{ij} \tag{4}$$

Here it's obvious that when α is large, then variance will be large too. The reparameterization replaces $1 + \sqrt{\alpha} * \epsilon_{ij}$ by $\sigma_{ij}\epsilon_{ij}$, and $\sigma_{ij} = \sqrt{\alpha\theta_{ij}}$, thus $\frac{\partial w_{ij}}{\partial \theta_{ij}} = 1$ and there is no injected noise. Variance are reduced and now $\alpha \in (0, \infty)$.

2.2.2 Approximation of the KL Divergence

They decomposed KL-divergence into a sum

$$D_{KL}(q(\theta, \alpha)||p(w)) = \sum_{i,j} D_{KL}(q(\theta_{ij}, \alpha_{ij})||p(w_{ij})) \tag{5}$$

and proposed a tighe approximation of KL-divergence, where $D_{KL}(q(\theta, \alpha)||p(w))$ is approximated by

$$-D_{KL}(q(\theta, \alpha)||p(w)) \approx k_1\sigma(k_2 + k_3 \log a_{ij}) - 0,5log(1 + a_{ij}^{-1}) + C \tag{6}$$

where $k_1 = 0.63576$, $k_2 = 1.87329$, $k_3 = 1.48695$. When α approaches to infinity, if we let $C = -0.63575$ the KL-divergence will be zero.

2.3 Sparsity Generation

Now the drop rates are determined by each weight, with the above approximation, when α grows the regularization term works fine, so they proved that it generates the sparsity that set the specific weights to be zero.
They also proposed another angle that because of the unbounded α means the value of this weight might be completely random and the magnitude maybe extremely high, so to maintain the prediction ability of the model the corresponding weight θ_{ij} is better to be 0, and then $q(\theta_{ij}, \alpha_{ij})$ is effectively a delta function which centered at 0.
They proved that if α is fixed, the optimal value of θ can be obtained by

$$(X^T X + diag(X^T X)diag(\alpha))^{-1} X^T y \tag{7}$$

When α_{ij} tends to infinity both θ_{ij} and $\alpha\theta_{ij}$ tends to 0. Sparsity is proved in linear situation.

2.4 Network Compression

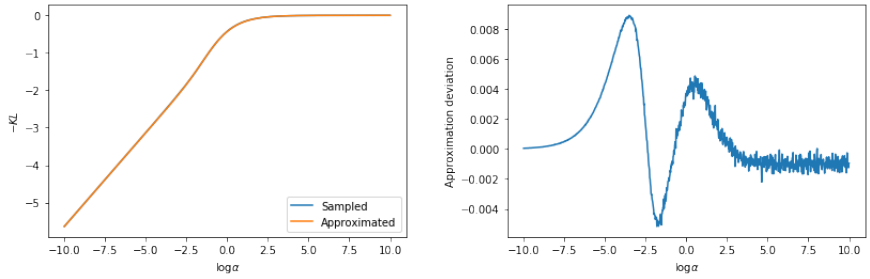
They concluded that Sparse Variational Dropout can reduce number of weights obviously, and can be used with quantization and Huffman coding techniques to compress network, because sparsity step is the intermediate process of these techniques.

3 Interpretation

3.1 Experiment and Analysis

By referring to the reference code given by the original author, we re-run the relevant experiments given in the article. However, since the reference code was written by Theano 1.0 and Python2.7, and Theano team has stopped updating in 2018. Therefore, we reimplemented the algorithm in this article through Pytorch framework, and implemented a richer experiment on models and datasets that are different from the models and data sets given by the paper [2].

3.1.1 KL Approximattion



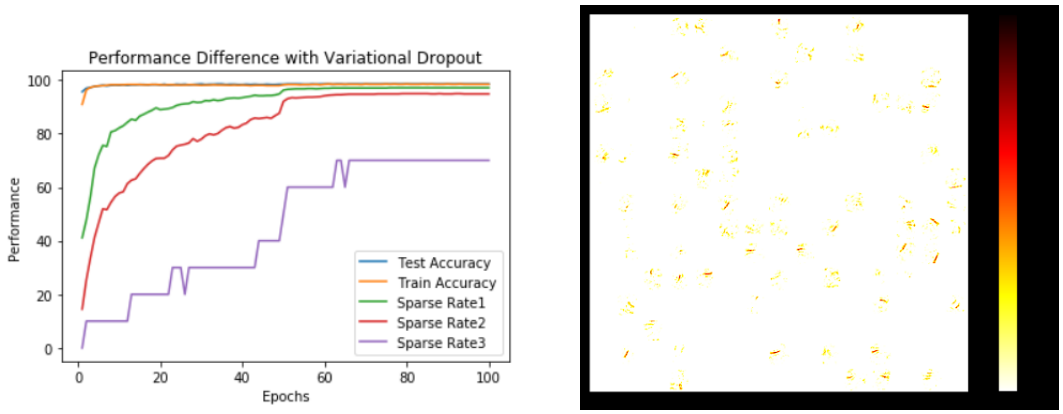
(a) Negatvie KL-divergence with $\log \alpha$ (b) Approximation Deviation with $\log \alpha$

Figure 1: KL Approximation Curve

At first, to verify the KL approximation mentioned in Equation 6. We draw the plots Figure 1. The first subfigure is relationship between log_α and $-KL$ and the second subfigure is relationship between $\log_\alpha$ and approximation deviation. As what we can see from figures, the approximation trick mentioned in paper [2] is valid.

3.1.2 Multi-Layer Perception With Sparse Variational on MNIST

On Multi-Layer Perception, we tested sparsity ability of Sparse Variational Dropout, the results show that it achieves a compression of 69.7%, and maintains good enough accuracy of 98.4%. The change of performance with sparsity under 100 epochs is as the Figure 2(a). After 63 epochs



(a) Training Curve of MLP with SparseVD on MNIST (b) Visualization Sparse Weights on First Layer

Figure 2: MLP with SparseVD on MNIST

training, the sparsity stop convergence and achieves 70%. We confirmed that the application of SparseVD doesn't hurt the model accuracy. Three sparse rate curve correspond to all three full connection layers.
In order to make the experiment result more visible, we draw a picture with the weights in first layer after training. The figure shows the obvious compression effect caused by Sparse VD. Red color means weight is large, and white color means weight approach to 0. Obviously, only a small part of weights remains after applying sparse variational dropout.

3.1.3 AlexNet With Variational Dropout on Cifar-10

We tested the performance of AlexNet with Sparse Variational Dropout on Cifar10 dataset.
Orginal Alexnet has accuracy of about 75% on Cifar10 dataset. Though the compression effect is not so obvious as on last MLP model, it also doesn't hurt the accuracy significantly. The Sparse-1 corresponds to the last full-connection layer, Sparse-2 to the second last full-connection layer, and so on.

We speculate that the SparseVD has a limitation that when model itself cannot achieves a good prediction, the Sparse VD is hard to generate a good sparsity. In our experiment, AlexNet cannot

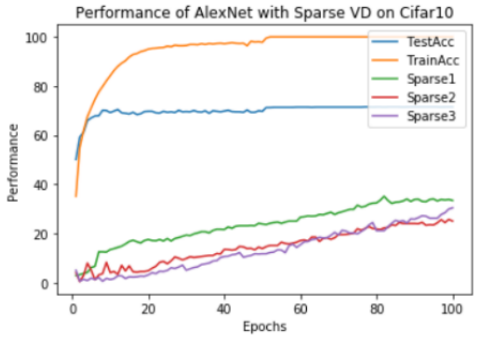
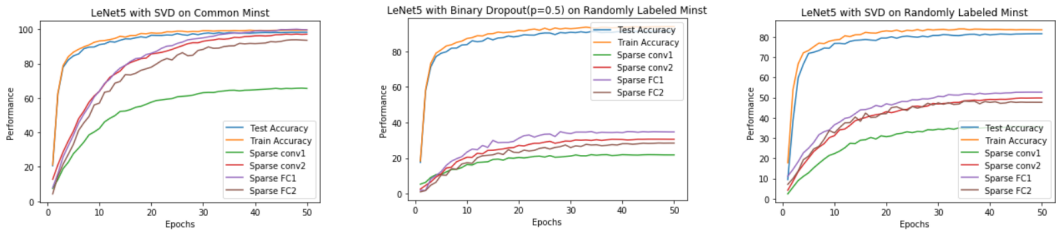


Figure 3: Training Curve of AlexNet with SparseVD on Cifar10

handle Cifar10 dataset as good as performance on Minst (Accuracy around 99%), it just has 70% accuracy on test data. In other way, AlexNet is not suitable for Cifar10 task, thus the Sparse VD are keen to remain more weights because it cannot distinguish whether it's important for specified task or not.
We also notice that the model is motivated to be more sparsified in deeper layers, and we deem the reason is that the features in deeper layer is more abstract, thus the fewer weight will contribute to the final classification.

3.1.4 LeNet5 With Sparse Varitional Dropout on Randomly Labeled MNIST

We tested LeNet-5 with Sparse VD on common Minst dataset, and on 50% randomly labeled Minst dataset. We also tested Lenet-5 with Binary Dropout with dropout rate 0.5 on randomly labeled Minst dataset as a comparison. After 50 epochs, we got the performance as Figure 4



(a) SVD+MNIST (b) BD+RL-MNIST (c) SVD+RL-MNIST

Figure 4: Training Curve of Lenet-5 On Original and Randomly Labeld MNIST

The results demonstrates that Sparse VD cannot fit randomly labeled data well, random labeling hurt the model's prediction ability as well as the sparsity. However, the Binary Dropout performed well on randomly labeled dataset. We give a explanation that the Sparse VD methods decides to drop every single weight and provide a constant prediction, which also means Sparse VD

Table 1: Result

Model	Train AR	Test AR	Conv1 SR	Conv2 SR	Fc1 SR	Fc2 SR
SVD+MNIST	99.40	98.22	65.56	97.03	99.66	93.53
SVD+RL-MNIST	83.51	81.57	35.58	49.83	52.70	47.70
BD+RL-MNIST	93.93	91.45	21.79	30.60	34.71	28.46

implicitly penalizes memorization and favors generalization.

3.2 Conclusion

Through theoretical analysis and experiments investigation, we confirmed Sparse Variational Dropout's ability of generating obvious sparsity without hurting the accuracy. We tested its performance on Multi-Layer Perceptron and AlexNet. We explained our understanding of the power and limitation of this technique, and implemented the whole computation codes.
As we can see from our experiments, with experiments given by the paper, we can some interesting conclusions as follows.
For different tasks, our method will lead to a very good sparsity on the model suitable enough for the task, but for the model that is not suitable for the task, the effect is not obvious or even worse. This might be the limitation of this algorithm.
To help calculate the gradient conveintly and save the training time, the paper introduce a new loss function with approximation KL divergence as Equation 6. However, this is just a numerical approximation function, without reasonable explanation. This is where the assumptions might be hard to interpretate.
In the future, we can focus on giving a better algorithm on interpretable loss function and try to modify the terrible performance on random labeling task.
Fortunately, with sparsifying methods like Sparse VD, we can get more sparse neural networks, which will motivate the research on interpretability of some Deep Neural Networks models like CNNs.
As what we can learn from lectures, Dropout is a cheap trick to save computer resourses and achieve good performance in training deep neural networks in practice. I believe Dropout research might bring great help to Network Architecture Search.

4 Disccussion

From perspective from textbook [3]. Unnecessary complex structure is not ideal by Occam's Razor principle. The Sparse Variational Dropout in DNNs is really meaningful because it regularizes the model params effectively, keep the model from overfitting, and make it easier to store and computing.
Futhermore, the idea of constructing a neural network with more sparse weights coincides with the property of sparse signal transmission between neurons in computational neuroscience [4] in bio-interpretability view. Studying the more sparse neural networks will bring many beneficial parts to the interpretability of neural networks.

Besides the method proposed by this paper, we also have some popular methods to learn sparse neural networks. [5] proposed a way to use l_0 regularization to learn sparse neural networks. And [6] proposed to use pruning methods in neural networks to reduce 90% of parameter counts but not harm accuracy using "Lottery Hypothesis", where Sparse VD has similar effect. However, "Lottery" Pruning also has fatal shortcomings that if we train the model with the structure after pruning, the final convergence cannot be attached, which is against our common sense. Unfortunately, the paper didn't give a reason for the weird phenomenon.
Moreover, Google did work on comparing some popular sparsifying training methods recently in [7]. The work compare some popular sparse methods nowadays such as l_0 regularization, random pruning, magnitude pruning on resnet-50 and transfomers for some NLP task. They also achieve elegant performance, both accuracy and sparse rate.
We share some code patches in our appendix and submit source code in our supplementary materials, which is written in Pytorch 1.1.0 framework running on CUDA 9.0. However, the source code is not the release version, there might be some bugs. If you have some questions about code, you are welcome to email and discuss to authors of this report.

References

[1] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[2] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2498–2507. JMLR. org, 2017.

[3] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[4] Suraj Srinivas, Akshayvarun Subramanya, and R. Venkatesh Babu. Training sparse neural networks, 2016.

[5] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[6] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. 2018.

[7] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

Appendices

A Code Patches

```
# define LinearSVDO Layer
class LinearSVDO(nn.Module):
    def __init__(self, in_features, out_features, threshold, bias=True):
        super(LinearSVDO, self).__init__()
        self.in_features =in_features
        self.out_features =out_features
        self.threshold =threshold

        self.W =Parameter(torch.Tensor(out_features, in_features))
        self.log_sigma =Parameter(torch.Tensor(out_features, in_features))
        self.bias =Parameter(torch.Tensor(1, out_features))

        self.reset_parameters()

    def reset_parameters(self):
        self.bias.data.zero_()
        self.W.data.normal_(0, 0.02)
        self.log_sigma.data.fill_(-5)

    def forward(self, x):
        self.log_alpha =self.log_sigma *2.0 -2.0 *torch.log(1e-16 +torch.abs(self.W))
        self.log_alpha =torch.clamp(self.log_alpha, -10, 10)

        if self.training:
            lrt_mean = F.linear(x, self.W) +self.bias
            lrt_std =torch.sqrt(F.linear(x *x, torch.exp(self.log_sigma *2.0)) +1e-8)
            eps = lrt_std.data.new(lrt_std.size()).normal_()
            return lrt_mean +lrt_std *eps

        return F.linear(x, self.W *(self.log_alpha <3).float()) +self.bias

    def kl_reg(self):
        # Return KL here -- a scalar
        k1, k2, k3 =torch.Tensor([0.63576]), torch.Tensor([1.8732]), torch.Tensor([1.48695])
        kl = k1 * torch.sigmoid(k2 +k3 *self.log_alpha) -0.5 *torch.log1p(torch.exp(-self.log_alpha))
        a = - torch.sum(kl)
        return a
```

```
# Define New Loss Function -- SGVLB
class SGVLB(nn.Module):
    def __init__(self, net, train_size):
        super(SGVLB, self).__init__()
        self.train_size =train_size
        self.net =net

    def forward(self, input, target, kl_weight=1.0):
        assert not target.requires_grad
        kl = 0.0
        for module in self.net.children():
            if hasattr(module, 'kl_reg'):
                kl = kl + module.kl_reg()
        return F.cross_entropy(input, target) *self.train_size +kl_weight *kl
```