PER-WEIGHT CLASS-BASED LEARNING RATES VIA ANALYTICAL CONTINUATION

Anonymous authors
Paper under double-blind review

ABSTRACT

We study the problem of training deep fully connected neural networks. Despite much progress in the design of activation functions, novel normalization techniques, and various skip-connection techniques, such networks remain challenging to train due to vanishing or exploding gradients. Our method is based on employing a different class-dependent learning rate to each network weight. Since the learning rates are hyperparameters and not part of the network, we perform an analytical continuation of the network, and create a generalized network. Following this reparameterization, the set of per-class per-weight learning rates are being manipulated during the training iterations. Our results show that the new algorithm leads to improved classification accuracy for both classical and modern activation functions.

1 INTRODUCTION

The success of deep neural networks in computer vision, voice recognition and synthesis, and NLP is undeniable. What is common in all of these application domains is the use of weight sharing, either spatially, in CNNs, or through time, in recurrent architectures such as RNN or LSTM. It is far less clear whether vanilla fully connected networks demonstrate the same advantage over alternative machine learning methods, such as the gradient boosted trees algorithm (Friedman, 2000), which seems to be the method of choice in many machine learning competitions.

Indeed, learning deep, fully-connected, neural networks is prone to converge to poor local minima. This can be mitigated by advanced optimization methods, by the usage of normalization, and by carefully designing the activation functions. In this work, we propose a new alternative, which is to manipulate the learning rate of each neuron during training in a manner that is class dependent.

The learning rate of each individual weight is based on the label of the learned sample. In other words, the label is used not only for the loss, but also to control the weight updates during training. What makes our work different than an ineffective brute force attempt to learn a predefined subnetwork per class, is the ability to share activations based on patterns that emerge during training. The class-based differentiation occurs due to local modifications of the learning rates, and does not follow a predetermined or a global program.

The influence of each sample on the individual weights is controlled by a set of hyperparameters. The optimization of these hyperparameters is usually not clear. We introduce a method that is based on the idea of network lifting, i.e., we construct a generalized network that contains both the original network's parameters and hyperparameters as its own parameters. The learned hyperparameters are not used for inference. Therefore, the resulting network is as fast in its inference time as a regular feed-forward network.

2 RELATED WORK

Fully Connected Networks are typically shallow, as a result of vanishing or exploding gradients. This challenge is often tackled by normalization schemes, such as Batch Normalization (Ioffe & Szegedy, 2015), Layer Normalization (Ba et al., 2016) and Weight Normalization (Salimans & Kingma, 2016). These methods aim to have the output or input of each layer follow a zero-mean and a standard deviation of 1 distribution.

Very recently, a new type of activation function named SELU Klambauer et al. (2017) was constructed with a specific goal of creating networks that are self-normalized (do not require added normalization). This is done by using exponential linear units (Clevert et al., 2015) with pre-calculated coefficients such that the normalization property stays intact during the network's forward propagation.

Our method also works well with all other activation functions we have experimented with, including the time tested $tanh$ non-linearity, and the common ReLU activation. However, acknowledging the experimental success of SELU, we focus on demonstrating that our method is able to improve the results of fully connected networks with SELU activations, and this is evaluated where SELU does best.

Somewhat related is the task of learning the network's structure during training, e.g., (Saxena & Verbeek, 2016; Wen et al., 2016; Liu et al., 2015; Feng & Darrell, 2015; Lebedev & Lempitsky, 2016). This differs from our work, in which the network structure remains fixed, except that during the weight update step of the training phase, different parts of the network change at a difference pace, depending on the class of the training sample.

3 METHOD

In order to add class dependency to our weights, we modify the weight update procedure of the gradient descent algorithm. Specifically, we learn a set of per-class learning rates for each of the network's weights.

The parametric form of the modified weight update, for every single network weight w, is given by

$$w \to w - \alpha(\mu_w^T y)\frac{\partial L(w,x,y)}{\partial w} \tag{1}$$

where α is a global learning rate, $\mu_w \in \mathbb{R}^k$ is a vector of learned parameters, k being the number of classes, $y = \left[y^1, y^2, \ldots, y^k\right]^\top \in \mathcal{Y} = (0,1)^k$ is a one-hot encoding vector that has the value of 1 for the class of sample $x \in \mathcal{X}$, and $L(w,x,y)$ is the network loss for the training sample (x,y) as a function of w.

Given μ_w for all weights w in the network, the update rule is well-defined, and clearly class dependent. Since $\{\mu_w\}_w$ are external to the network, and do not take part in its optimization, one cannot update these weights by using a direct gradient method.

In order to derive the update step for these weights, we consider an analytical continuation of our network, where a "handle" z_w is placed. In other words, we define an extended, sample-dependent, network with parametrized weights $z_w(\alpha,x,y)$, such that at $z_w(0,x,y) = w$ for all pairs $(x,y) \in \mathcal{X} \times \mathcal{Y}$. Specifically, we define z to be (omitting for brevity w, x and y)

$$z(\alpha) = w - \alpha(\mu_w^T y)\frac{\partial L}{\partial w}. \tag{2}$$

Note that for $\alpha \ll 1$, $L(z(\alpha),x,y) \leqslant L(w,x,y)$, since $L(z(\alpha),x,y)$ represents the original network after one step of SGD.

When the class label is not j, i.e., $y^j = 0$, the jth weight of μ_w, denoted by μ_w^j does not play any role, since in that case $z(\alpha)$ is independent of it. When $y^j = 1$, the gradient descent update role of μ_w^j is proportional to the square of the derivative of the loss by the specific weight w. Both these cases stem from the chain rule:

$$\frac{\partial L(z(\alpha),x,y)}{\partial \mu_w^j} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial \mu_w^j} = \frac{\partial L}{\partial z}\left(-\alpha y^j \frac{\partial L}{\partial w}\right) \tag{3}$$

A Taylor expansion w.r.t α can be performed on Eq. 3. We assume $\alpha \ll 1$ and consider the elements up to order $O\left(\alpha^2\right)$. Then, we can write it as

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial w}\frac{\partial w}{\partial z} = \frac{\partial L}{\partial w}\left(1 + \alpha\left(\mu_w^T\right) y \frac{\partial L}{\partial z \partial w}\right) = \frac{\partial L}{\partial w} + O(\alpha). \tag{4}$$

The gradient of the loss w.r.t to μ_w^j then follows immediately:

$$\frac{\partial L}{\partial \mu_w^j} = -\alpha y^j \left(\frac{\partial L}{\partial w}\right)^2 + O\left(\alpha^2\right) \approx -\alpha y^j \left(\frac{\partial L}{\partial w}\right)^2 \tag{5}$$

Given the gradient of the loss w.r.t μ_w^j, we obtain the update rule

$$\mu_w^j \rightarrow \mu_w^j + \alpha_\mu \alpha y^j \left(\frac{\partial L}{\partial w}\right)^2, \tag{6}$$

where in our experiments we set the learning rate of μ_w using cross validation.

The initialization of each of μ_w^j is set to 1, which results in the conventional gradient descent. However, at each optimization step, μ_w^j can only increase, since a positive element is added to it. Noting that unlike the network weights, μ are specific for the specific training landscape, one can reset these after each update. However, since we learn them by a gradient descent update rule, we opt to accumulate the updates for exactly half an epoch, after which we reset all μ_w^j to be 1.

4 EXPERIMENTS

We evaluate our method on the UCI repository on datasets that contain more than 1000 samples. These are exactly the datasets where SELU was shown to allow fully connected neural networks to outperform other learning methods[1]. Moreover, instead of performing cross validation over the architectures, we select the architecture pointed to by Klambauer et al. (2017), after extensive cross validation experiments. These hyperparameters, which are listed in Tab. 1, were extracted from the authors' github repository[2]. Following that work, all experiments were also run for 100 epochs, however, we differ in that no early stopping is employed. In some experiments, Klambauer et al. (2017) report using a high learning rate of 0.1 for some of the models. However, in our runs these models diverged. We, therefore, decided to use a learning rate of 0.01 for all experiments.

The choice of using the hyperparameters that provided the best SELU outcomes clearly challenges us, since we compete against SELU specifically where it excels. Therefore, due to the diminishing returns principle, our ability to greatly outperform is reduced. It is also likely that the results of our algorithm would improve, given the chance to select different hyperparameters.

During our experiments we set $\alpha_\mu = 10^5$ for SELU and tanh, and $\alpha_\mu = 10^4$ for ReLU. If the network did not converge, we reduced it to by a factor of 10. For computational reasons, we limited the number of layers with our per-weight/per-class learning rates to be the bottom 16 layers, whereas any additional layer is a regular fully connected one. In addition, the topmost (representation) layer is not manipulated, since softmax is applied instead of the activation function. Therefore, for example, 8 or 16 layers architectures employ specific learning rates to all layers, except the top one, whereas in our experiments, 32 layer architectures would have such layers only for the bottom 16 layers. The reason that the bottom layers were selected is that the lower layers, which are further away from the loss, would benefit more from the additional class-based information.

The loss function is generally non-convex, however around the minima, we can treat it locally as a convex function. Given a convex function, following the regular gradients would bring us to the minima, and adaptive learning rates are no longer needed. Moreover, the μ_w^j parameters change in a cyclic manner every half an epoch, which might hinder the last steps of convergence. We, therefore, introduce a cutoff of 10, 20 or 30 epochs, after which we stop using our local learning rates, and return to using the normal gradients. We use cross-validation in order to decide on the position of this cutoff for each of the datasets. Except for the clear rule for selecting α_μ above, this binary choice is the only added hyperparameter parameter introduced by our method that is not set to a single fixed value throughout the experiments.

The right way to use the UCI datasets for comparing algorithms is debated in the literature (Fernández-Delgado et al., 2014; Klambauer et al., 2017; Wainberg et al., 2016). We follow

[1]The SELU work had an exceptionally high number of validating experiments, beyond what we could clone in reasonable time using our resources.

[2] https://github.com/gklambauer/SelfNormalizingNetworks/tree/master/Hyperparameters/UCI

Table 1: The hyperparameters used during our experiments for each of the UCI datasets. Shown are the number of layers, the number of hidden neurons per layer, the architecture, and the dropout constant. Conic layers start with the given number of hidden units in the first layer and then decrease the number of hidden units to the size of the output layer, according to the geometric progression. Rectangular ones have a fixed number of hidden neurons per layer.

dataset	**# layers**	**#hidden**	**Architecture**	**α-Dropout**
Abalone	8	512	Conic	0
Adult	8	512	Conic	0.05
Bank	3	512	Conic	0.05
Car	4	512	Rectangular	0
Cardiotocography_10clases	16	512	Conic	0
Cardiotocography_3clases	4	256	Rectangular	0.05
Chess_krvk	32	1024	Rectangular	0
Chess_krvkp	8	512	Rectangular	0.05
Connect_4	8	1024	Rectangular	0
Contrac	16	512	Rectangular	0.05
Hill_Valley	32	256	Conic	0.05
Image_Segmentation	3	512	Conic	0.05
Led_Display	3	512	Conic	0.05
Letter	4	512	Rectangular	0.05
Magic	3	512	Conic	0
Miniboone	4	1024	Rectangular	0
Molec_biol_splice	8	256	Rectangular	0.05
Mushroom	16	256	Rectangular	0.05
Nursery	3	1024	Conic	0
Oocytes_merluccius_nucleus_4d	8	1024	Conic	0
Oocytes_merluccius_states_2f	8	256	Rectangular	0
Optical	8	256	Rectangular	0.05
Ozone	3	512	Conic	0.05
Page_blocks	16	256	Conic	0
Pendigits	4	256	Rectangular	0.05
Plants_margin	4	256	Rectangular	0.05
Plants_shape	8	256	Rectangular	0
Plants_texture	2	512	Rectangular	0.05
Ringnorm	4	256	Conic	0
Semeion	16	1024	Rectangular	0.05
Spambase	3	512	Rectangular	0
Statlog_german_credit	2	256	Rectangular	0.05
Statlog_image	3	1024	Conic	0
Statlog_landsat	16	1024	Rectangular	0
Statlog_shuttle	3	512	Rectangular	0
Steel_plates	16	1024	Rectangular	0
Thyroid	3	1024	Conic	0.05
Titanic	8	256	Conic	0
Twonorm	8	256	Conic	0
Wall_following	4	256	Conic	0
Waveform	4	256	Rectangular	0.05
Waveform_Noise	2	1024	Conic	0.05
Wine_quality_red	32	1024	Rectangular	0
Wine_quality_white	32	1024	Rectangular	0
Yeast	3	256	Conic	0.05

the splitting of train set and test set suggested by Fernández-Delgado et al. (2014). For validation we leave out a random subset of 15% of the training set. The same train, validation, and test splits are used across the various methods. This protocol follows the one of Klambauer et al. (2017). However, different validation splits are used, and our SELU results, which employ the public PyTorch implementation release with PyTorch 0.20, do not fully match the numbers reported by Klambauer et al. (2017).

Nothing in the proposed method is tailored toward SELU or any other activation function. Our method is general and could work well with any other activation function. Therefore, in addition to the SELU experiments, we also experiment with other activation functions. These experiments employ exactly the same architectures used per dataset for SELU. The only architectural difference between these experiments and the SELU ones is in the use of a regular Dropout instead of the AlphaDropout prescribed by Klambauer et al. (2017).

The results are reported in Tab. 2 As can be seen, using the SELU activation together with our purposed method benefits 21 experiments out of 45, where only 14 experiments suffer a loss in performance. In 10 datasets the results are identical. Since, in many cases, the results are similar, we applied McNemar's test in order to compare the algorithms. The test is applied for correct/incorrect classification of test samples. In 13 cases, our method outperforms the baseline SELU method with a p-value smaller than 0.05. In 6 of the datasets the baseline method outperforms at that significance level.

Two common activation functions are ReLU and the hyperbolic tangent (tanh). When using the ReLU activation function, 23 datasets benefit from using our method, whereas 11 datasets have suffered a performance reduction. In 11 datasets, the results are identical. When using tanh as an activation function, in 15 datasets an improvement is seen and only 9 datasets suffer a loss in performance. In 21 datasets the results are identical.

An interesting question to ask is whether the different learning rates per class lead to per-class specialization of the neurons. In order to answer this, we mark a neuron as class-specific if its activations across the test samples have a Wilcoxon ranksum p-value lower than 10^{-7} (the low threshold is taken to mitigate the multiple hypothesis situation). In other words, for each class, we separate the activations of each neuron to those obtained for the class and to those obtained for all other classes, and compute the Wilcoxon ranksum p-value for the difference between the two distributions. In Tab. 3 we report the mean over all classes of the ratio of neurons which are found to be "class-specialized". As can be seen, there is little difference for the baseline networks, in comparison to the situation when applying our method.

5 Conclusions

We employ the technique of analytical continuation in order to control the learning rate hyperparameter during training. We chose to manipulate this parameter in a per-class per-neuron fashion. Other options include per sample manipulation, per-layer, grouping based on side-information and many combinations of these. In addition, the technique can be used in order to dynamically control other hyperparameters, e.g., introducing per-weight regularization terms or determining class-based trade-off parameters.

Our experimental results show that compared with the SELU technique, exactly where it was shown to excel, our method leads to an improvement in accuracy nearly two out of three times. The situation is similar for other classical and modern activation functions. Being able to win convincingly in a diverse set of experiments, with variations in both datasets and activation functions, indicates that our method is a useful tool to add to one's toolbox. The underlying technique of analytical continuation can also be used to create many other such tools.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

Table 2: All datasets from the UCI Repository with over 1000 samples. For each activation function, we run the baseline network and the network with our per-class learning rate method. The bold fonts mark the best performance per each pair for matching experiments.

dataset	**SELU**	**SELU+ class lr**	**ReLU**	**ReLU+ class lr**	**tanh**	**tanh+ class lr**
Abalone	65.18	**65.33**	64.51	**65.33**	64.99	64.99
Adult	**81.50**	81.36	81.81	**81.82**	81.81	**81.86**
Bank	89.25	**89.38**	88.89	**89.25**	**89.16**	88.63
Car	96.18	96.18	95.95	**96.06**	93.40	93.40
Cardiotocography_10clases	**82.41**	79.77	**75.82**	74.13	80.34	**80.53**
Cardiotocography_3clases	88.62	**88.71**	89.28	89.28	**91.72**	90.40
Chess_krvk	79.51	**81.69**	76.58	**77.02**	78.49	**80.27**
Chess_krvkp	98.00	98.00	97.93	**98.19**	98.00	**98.50**
Connect_4	84.00	**85.00**	**84.61**	84.48	85.80	**85.85**
Contrac	**53.40**	51.36	50.68	**50.95**	50.27	50.27
Hill_Valley	51.49	51.49	50.83	50.83	**55.45**	50.83
Image_Segmentation	75.24	75.24	74.29	74.29	73.33	73.33
Led_Display	70.00	**70.60**	69.00	**69.20**	70.00	70.00
Letter	96.40	**96.47**	96.20	**96.25**	96.40	**96.45**
Magic	86.77	**87.36**	**86.94**	86.35	**86.76**	86.21
Miniboone	92.77	**93.03**	92.15	**92.32**	92.55	92.55
Molec_biol_splice	80.75	80.75	**77.99**	77.68	81.07	**81.38**
Mushroom	100.00	100.00	100.00	100.00	100.00	100.00
Nursery	**99.68**	99.65	99.31	99.31	99.34	**99.35**
Oocytes_merluccius_nucleus_4d	75.34	**75.73**	**78.86**	76.32	**81.02**	80.63
Oocytes_merluccius_states_2f	91.39	**91.98**	**91.98**	91.78	92.56	92.56
Optical	96.86	**96.91**	97.12	**97.17**	96.91	**96.96**
Ozone	**96.77**	96.69	96.77	**97.16**	96.69	96.69
Page_blocks	**96.75**	96.56	**96.86**	96.56	**96.86**	96.49
Pendigits	98.64	**98.67**	99.23	99.23	99.17	99.17
Plants_margin	79.12	**79.38**	76.00	**76.63**	78.87	**79.00**
Plants_shape	**62.62**	61.88	45.87	**46.00**	58.00	58.00
Plants_texture	77.35	77.35	77.85	77.85	78.72	78.72
Ringnorm	**96.81**	96.68	96.46	**96.57**	**91.86**	91.24
Semeion	**89.20**	88.57	**84.30**	78.77	89.70	89.70
Spambase	92.57	92.57	93.26	93.26	**93.09**	92.87
Statlog_german_credit	74.00	**74.60**	75.00	**75.80**	74.00	**74.60**
Statlog_image	95.50	**95.58**	95.67	95.67	95.50	95.50
Statlog_landsat	**89.35**	87.96	88.41	**88.59**	87.64	87.64
Statlog_shuttle	99.93	99.93	99.84	**99.87**	**99.89**	99.88
Steel_plates	**68.97**	68.04	62.78	**65.15**	71.75	71.75
Thyroid	97.88	**98.09**	**97.61**	97.56	98.04	98.04
Titanic	78.45	78.45	78.45	78.45	78.45	78.45
Twonorm	96.84	**96.86**	96.65	96.65	97.22	**97.24**
Wall_following	**88.75**	88.67	86.73	**86.95**	87.98	**88.01**
Waveform	**86.10**	85.80	86.20	**86.24**	86.00	**86.72**
Waveform_Noise	84.50	**84.72**	85.20	**85.40**	85.00	**85.12**
Wine_quality_red	54.44	**54.82**	**56.32**	56.20	57.07	57.07
Wine_quality_white	53.61	**54.19**	54.84	**55.37**	53.61	53.61
Yeast	**60.65**	60.51	**60.38**	60.24	62.80	62.80

J. Feng and T. Darrell. Learning the structure of deep convolutional networks. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2749–2757, Dec 2015. doi: 10.1109/ICCV.2015.315.

Table 3: Mean % of neurons with class specialization based on Wilcoxon rank- sum test ($p < 10^{-7}$).

dataset	**SELU**	**SELU+ class lr**	**ReLU**	**ReLU+ class lr**	**tanh**	**tanh+ class lr**
Abalone	70.63	69.59	73.67	72.95	73.61	73.32
Adult	84.08	85.03	84.77	84.93	82.19	81.28
Bank	44.14	49.61	42.84	43.75	42.71	43.88
Car	22.29	22.36	34.61	34.63	27.23	27.21
Cardiotocography_10clases	43.15	43.04	56.77	55.45	47.10	47.89
Cardiotocography_3clases	58.33	58.27	61.82	61.69	58.04	58.11
Chess_krvk	56.42	56.37	73.64	73.36	61.74	61.82
Chess_krvkp	82.81	82.86	81.25	81.64	80.64	80.52
Connect_4	56.73	64.11	75.71	76.01	61.28	60.86
Contrac	23.95	23.60	36.92	36.57	18.28	18.14
Hill_Valley	0.07	0.05	0.00	0.00	0.37	0.44
Image_Segmentation	6.34	6.34	6.06	6.06	7.05	7.05
Led_Display	41.50	41.60	41.09	41.09	42.10	42.07
Letter	70.87	70.69	72.36	72.36	73.04	72.99
Magic	81.12	81.38	81.12	82.55	80.99	80.73
Miniboone	95.83	96.26	95.43	96.00	96.34	96.58
Molec_biol_splice	66.73	66.50	68.16	68.00	62.13	62.08
Mushroom	92.80	92.75	95.29	95.04	93.90	94.21
Nursery	62.15	61.98	62.13	62.25	64.24	64.29
Oocytes_merluccius_nucleus_4d	9.29	9.43	18.14	15.15	13.76	13.76
Oocytes_merluccius_states_2f	60.14	62.66	69.56	69.61	65.56	65.54
Optical	71.55	71.62	71.28	71.16	71.26	71.31
Ozone	11.59	11.72	10.68	10.68	12.76	13.41
Page_blocks	60.65	60.27	76.90	76.26	64.50	64.48
Pendigits	79.26	79.19	79.04	79.03	80.94	80.83
Plants_margin	0.00	0.00	0.00	0.00	0.00	0.00
Plants_shape	0.00	0.00	0.00	0.00	0.00	0.00
Plants_texture	0.00	0.00	0.00	0.00	0.00	0.00
Ringnorm	64.84	65.53	74.41	75.10	64.75	64.45
Semeion	51.01	51.02	55.63	56.09	42.84	42.85
Spambase	70.05	69.66	71.35	71.35	71.81	72.01
Statlog_german_credit	5.47	5.47	5.27	5.27	4.49	4.49
Statlog_image	64.80	64.65	64.82	64.69	67.39	67.38
Statlog_landsat	76.50	76.11	80.95	81.01	74.86	75.29
Statlog_shuttle	50.22	50.28	49.04	48.86	51.57	51.49
Steel_plates	36.94	36.97	49.45	49.46	42.66	42.31
Thyroid	45.65	45.52	45.72	45.55	45.72	45.54
Titanic	73.49	73.24	82.18	82.13	75.44	75.63
Twonorm	88.28	88.96	91.31	91.50	91.31	91.36
Wall_following	59.23	58.28	64.43	64.67	67.65	67.55
Waveform	83.04	82.68	80.11	79.75	80.92	80.96
Waveform_Noise	71.66	71.83	71.12	71.12	71.66	71.64
Wine_quality_red	11.80	11.98	25.45	23.97	13.70	13.51
Wine_quality_white	14.64	15.58	35.79	33.38	19.38	20.77
Yeast	26.56	26.77	24.80	24.69	25.33	25.27

Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.*, 15(1):3133–3181, January 2014. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2627435.2697065.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in neural information processing systems (NIPS)*, 2017.

Vadim Lebedev and Victor Lempitsky. Fast convnets using group-wise brain damage. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 806–814, 2015.

Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *arXiv preprint arXiv:1602.07868*, 2016.

Shreyas Saxena and Jakob Verbeek. Convolutional Neural Fabrics. In *Advances in Neural Information Processing Systems (NIPS)*, Barcelona, Spain, December 2016. URL https://hal.inria.fr/hal-01359150.

Michael Wainberg, Babak Alipanahi, and Brendan J. Frey. Are random forests truly the best classifiers? *Journal of Machine Learning Research*, 17(110):1–5, 2016. URL http://jmlr.org/papers/v17/15-374.html.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 2074–2082, 2016.