SplitNet: Learning to Semantically Split Deep Networks for Parameter Reduction and Model Parallelization

JUYONG KIM, YOOKOON PARK, GUNHEE KIM, SUNG JU HWANG

Chosen Paper

- Discussing *SplitNet: Learning to Semantically Split Deep Networks for Parameter Reduction and Model Parallelization* by Juyong Kim, Yookoon Park, Gunhee Kim and Sung Ju Hwang
- Published in ICML 2017
- Research is affiliated with Samsung

Embarrassingly Model-Parallelizable

We propose a novel deep neural network that is both lightweight and effectively structured for model parallelization. Our network, which we name as *SplitNet*, automatically learns to split the network weights into either a set or a hierarchy of multiple groups that use disjoint sets of features, by learning both the class-to-group and feature-to-group assignment matrices along with the network weights. This produces a tree-structured network that involves no connection between branched subtrees of semantically disparate class groups. SplitNet thus greatly reduces the number of parameters and required computations, and is also embarrassingly model-parallelizable at test time, since the evaluation for each subnetwork is completely independent except for the shared lower layer weights that can be duplicated over multiple processors, or assigned to a separate processor. We validate

Introduction & The Problem

Recently, deep neural networks have shown impressive performances on a multitude of tasks, including visual recognition (Krizhevsky et al., 2012; Szegedy et al., 2015; He et al., 2016), speech recognition (Hinton et al., 2012), and natural language processing (Bengio et al., 2003; Sutskever et al., 2014). However, such remarkable performances are achieved at the cost of increased computational complexity at both training and test time compared to traditional machine learning models including shallow neural networks. This increased complexity of deep networks can be problematic if the model and the task size becomes very large (*e.g.* classifying tens of thousands of object classes), or the application is time-critical (*e.g.* real-time object detection).

Related Work: Parameter Reduction

Parameter reduction for deep neural networks. Achieving test-time efficiency is an active research topic in deep learning. One straightforward approach is to remove weak connections during the training, usually implemented using the ℓ_1-norm (Collins & Kohli, 2014). However, the ℓ_1-norm often results in a model that trades-off the accuracy with the efficiency. Han et al. (2015) presented an iterative weight pruning technique that repeatedly retrains the network while removing of weak connections, which achieves a superior performance over ℓ_1-regularization. Recently,

Regularization: L2-Norm

By far the most common choice for regularizing the network, thus avoiding overfitting, is to impose a squared ℓ_2 norm constraint on the weights:

$$R_{\ell_2}(\mathbf{w}) \triangleq \|\mathbf{w}\|_2^2 \,. \tag{3}$$

In the neural networks' literature, this is commonly denoted as 'weight decay' [33], since in a steepest descent approach, its net effect is to reduce the weights by a factor proportional to their magnitude at every iteration. Sometimes it is also denoted as Tikhonov regularization. However, the only way to enforce sparsity with weight decay is to artificially force to zero all weights that are lower, in absolute terms, than a certain threshold. Even in this way, its sparsity effect might be negligible.

Scardapane, S. & Comminiello, D. & Hussain, A. & Uncini, A. (2016). "*Group Sparse Regularization for Deep Neural Networks*", arXiv preprint arXiv 1607.00485

Regularization: L1-Norm

As we stated in the introduction, the second most common approach to regularize the network, inspired by the Lasso algorithm, is to penalize the absolute magnitude of the weights:

$$R_{\ell_1}(\mathbf{w}) \triangleq \|\mathbf{w}\|_1 = \sum_{k=1}^{Q} |w_k| \,. \tag{4}$$

The ℓ_1 formulation is not differentiable at 0, where it is necessary to resort to a subgradient formulation. Everywhere else, its gradient is constant, and in a standard minimization procedure it moves each weight by a constant factor towards zero (in the next section, we also provide a simple geometrical intuition on its behavior). While there exists customized algorithms to solve non-convex ℓ_1 regularized problems [34], it is common in the neural networks' literature to apply directly the same first-order procedures (e.g., stochastic descent with momentum) as for the weight decay formulation. As an exam-

Scardapane, S. & Comminiello, D. & Hussain, A. & Uncini, A. (2016). "*Group Sparse Regularization for Deep Neural Networks*", arXiv preprint arXiv 1607.00485

Related Work: Parameter Reduction by Grouping

the group sparsity using $\ell_{2,1}$-norm has been explored for learning a compact model. Alvarez & Salzmann (2016) applied (2,1)-norm regularization at each layer to eliminate the hidden units that are not shared across upper-level units, thus automatically deciding how many neurons to use at each layer. Wen et al. (2016) used the same group sparsity to select unimportant channels and spatial features in a CNN, and let the network to automatically decide how many layers to use. However, they assume that all classes share the same set of features, which is restrictive when the number of classes is very large. On the contrary, our proposed SplitNet enforces feature sharing only within a group of related classes, and thus semantically reduce the number of parameters and computations for large-scale problems. Recently, Shankar et al. (2016) also addressed the use of symmetrical split at mid-level convolutional layers in CNNs for architecture refinement. However, they did not learn the splits but predefine them, as opposed to SplitNet which learns semantic splits along with network weights.

Groups

The basic idea of this paper is to consider *group-level* sparsity, in order to force all outgoing connections from a single neuron (corresponding to a group) to be either simultaneously zero, or not. More specifically, we consider three different groups of variables, corresponding to three different effects of the group-level sparsity:

1) **Input groups** $\mathcal{G}_{\text{in}}$: a single element $\mathbf{g}_i \in \mathcal{G}_{\text{in}}, i = 1, \ldots, d$ is the vector of all outgoing connections from the ith input neuron to the network, i.e. it corresponds to the first row transposed of the matrix $\mathbf{W}_1$.
2) **Hidden groups** $\mathcal{G}_{\text{h}}$: in this case, a single element $\mathbf{g} \in \mathcal{G}_{\text{h}}$ corresponds to the vector of all outgoing connections from one of the neurons in the hidden layers of the network, i.e. one row (transposed) of a matrix $\mathbf{W}_k, k > 1$. There are $\sum_{k=2}^{H+1} N_k$ such groups, corresponding to neurons in the internal layers up to the final output one.
3) **Bias groups** $\mathcal{G}_{\text{b}}$: these are one-dimensional groups (scalars) corresponding to the biases on the network, of which there are $\sum_{k=1}^{H+1} N_k$. They correspond to a single element of the vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_{H+1}\}$.

Scardapane, S. & Comminiello, D. & Hussain, A. & Uncini, A. (2016). "*Group Sparse Regularization for Deep Neural Networks*", arXiv preprint arXiv 1607.00485

(2,1)-Norm

Group sparse regularization can be written as [23]:

$$R_{\ell_{2,1}}(\mathbf{w}) \triangleq \sum_{\mathbf{g}\in\mathcal{G}} \sqrt{|\mathbf{g}|}\,\|\mathbf{g}\|_2 \ , \qquad (5)$$

where $|\mathbf{g}|$ denotes the dimensionality of the vector $\mathbf{g}$, and it ensures that each group gets weighted uniformly. Note that, for one-dimensional groups, the expression in (5) simplifies to the standard Lasso. Similarly to the ℓ_1 norm, the term in (5) is convex but non-smooth, since its gradient is not defined if $\|\mathbf{g}\|_2 = 0$. The sub-gradient of a single term in (5) is given

Scardapane, S. & Comminiello, D. & Hussain, A. & Uncini, A. (2016). "*Group Sparse Regularization for Deep Neural Networks*", arXiv preprint arXiv 1607.00485

Related Work: Parallel and Distributed Deep Learning

Parallel and distributed deep learning. As deep networks and training data become increasingly larger, researchers are exploring parallelization and distribution techniques to speed up the training process. Most parallelization techniques exploit either 1) data parallelism, where the training data is distributed across multiple computational nodes, or 2) model parallelism, where the model parameters are distributed. Dean et al. (2012) used both data and model parallelism to train a large-scale deep neural network on a computing cluster with thousands of machines. For CNNs, Krizhevsky et al. (2012) used both data

Model Parallelization: Problems

with greatly reduced the communication overheads. However, all these are systems-based approaches that work under the assumption that the model structure is given and fixed. Our approach, on the other hand, leverages semantic knowledge of class relatedness to learn a network structure that is well-fitted to a distributed machine learning setting.

Related Work: Tree-structured DN

Tree-structured deep networks. There have been some efforts to exploit hierarchical class structures for improving the performance of deep networks. To list a few recent work, Warde-Farley et al. (2014) proposed to group classes based on their weight similarity, and augmented the original deep network with the softmax loss for fine-grained classification for classifying classes within each group. Yan et al. (2015) proposed a convolutional network that combines predictions from separate sub-networks for coarse- and fine-grained category predictions, which share common lower-layers. Goo et al. (2016) exploited the hierarchical structure among the classes to learn common and discriminative features across classes, by adding in simple feature pooling layers. Murdock et al. (2016) generalized dropout to stochastically assign nodes to clusters which results in obtaining a hierarchical structure. However, all of these methods focus on improving the model accuracy, at the expense of increased computational complexity. On the contrary, SplitNet is focused on improving memory/time efficiency and parallelization performance of the model.

Tree-structured DNN

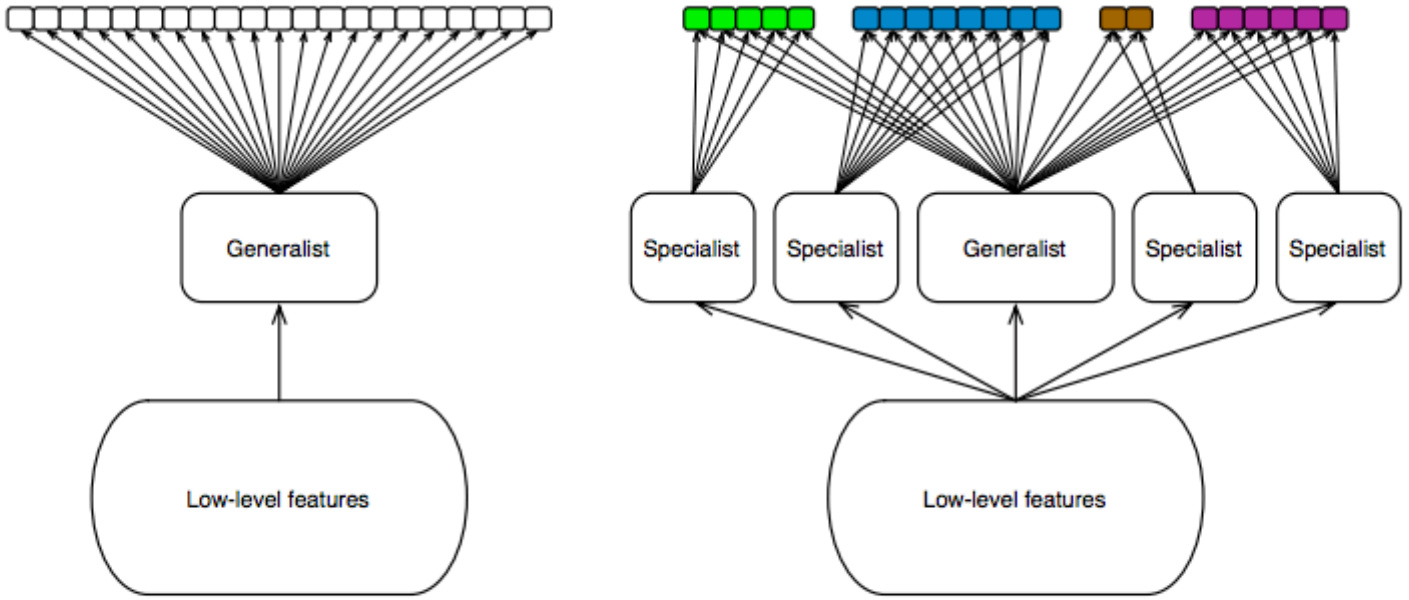


Figure 1: A schematic of the augmentation process. Left: the original network. Right: the network after augmentation.

Warde-Farley, D. & Rabinovich, A. & Anguelov, D. (2014). "*Self-informed Neural Network Structure Learning*", arXiv:1412.6563

HD-CNN

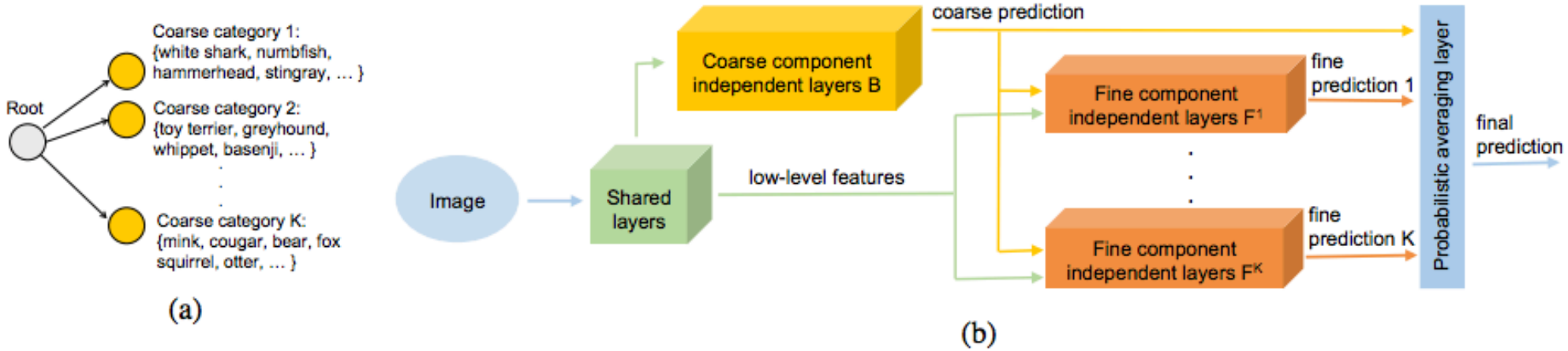


Figure 1: (a) A two-level category hierarchy where the classes are taken from ImageNet 1000-class dataset. (b) Hierarchical Deep Convolutional Neural Network (HD-CNN) architecture.

Yan, Z. & Zhang, H. & Jagadeesh, V. & Decoste, D. & Di, W. & Yu, Y. (2015). "*HD-CNN: Hierarchical Deep Convolutional Neural Network for Image Classification*", ICCV

Taxonomy-Regularized CNN

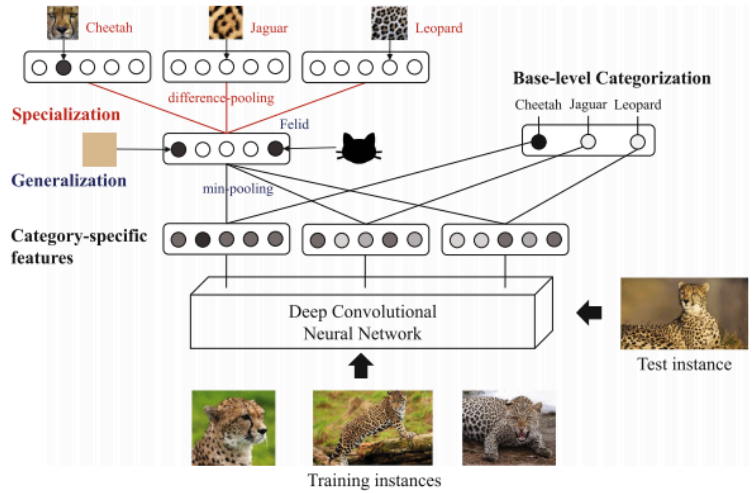


Fig. 1. Concept: Our taxonomy-regularized deep CNN learns grouped and discriminative features at multiple semantic levels, by introducing additional regularization layers that abstract and differentiate object categories based on a given class hierarchy. (1) At the generalization step, our network finds the commonalities between similar object categories that help recognize the supercategory, by finding the common components between per-category features. (2) At the specialization step, our network learns subcategory features as different as possible from the supercategory features, to discover unique features that help discriminate between sibling subcategories. These generalization and specialization layers work as regularizers that help the original network learn the features focusing on those commonalities and differences.

Goo, W. & Kim, J. & Kim, G. & Hwang, S. (2016). "*Taxonomy-Regularized Semantic Deep Convolutional Neural Networks*", ECCV

SplitNet

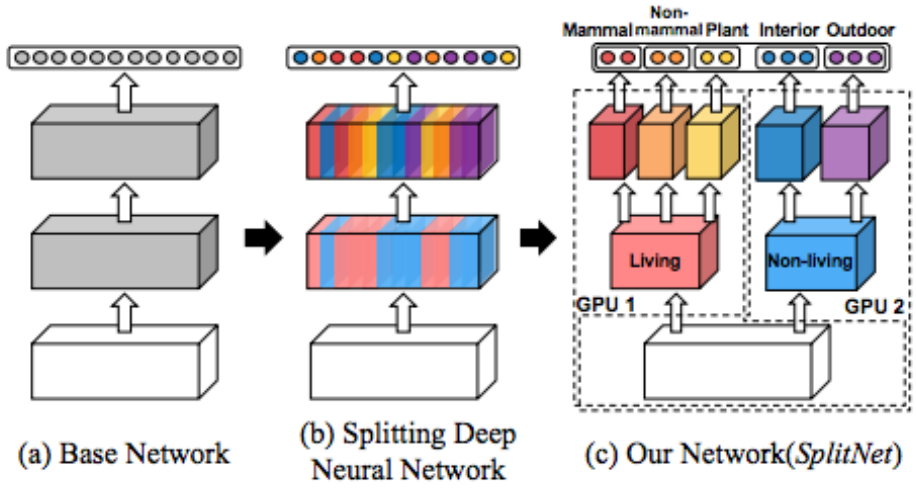


Figure 1. **Concept**. Our network automatically learns to split the classes and associated features into multiple groups at multiple network layers, obtaining a tree-structured network. Given a base network in (a), (b) our algorithm optimizes network weights as well as class-to-group and feature-to-group assignments. Colors indicate the group assignments of classes and features. (c) After learning to split the network, the model can be distributed to multiple GPUs to accelerate training and inference.

How to split?

Thus, to maximize the utility of this splitting process, we need to cluster classes together into groups so that each group uses a subset of features that are completely disjoint from the ones used by other groups. One straightforward way to obtain such mutually exclusive groupings of classes is to leverage a semantic taxonomy, since semantically similar classes are likely to share features, whereas dissimilar classes are unlikely to do so. However, in practice, such semantic taxonomy may not be available, or may not agree with actual hierarchical grouping based on what features each class uses. Another simple approach is to perform (hierarchical) clustering on the weights learned in the original network, which is also based on actual feature uses. Yet, this grouping may be still suboptimal since the groups are highly likely to overlap, and is inefficient since it requires training the network twice.

The Goal

Given a dataset $\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$, where $\boldsymbol{x}_i \in \mathbb{R}^d$ is an input data instance and $y_i \in \{1, \ldots, K\}$ is a class label for K classes, our goal is to learn a network whose weight at each layer l, $\boldsymbol{W}^{(l)}$ is a block-diagonal matrix, where each block $\boldsymbol{W}_g^{(l)}$ is associated with a class group $g \in \mathcal{G}$, where $\mathcal{G}$ is the set of all groups. Such a block-diagonal $\boldsymbol{W}^{(l)}$ ensures that each disjoint group of classes has exclusive features associated with it, such that no other groups use those features; this allows the network to be split across multiple class groups, for faster computation and parallelization.

Group Assignment

We assume that the number of groups G, is given. Let p_{gi} be a binary variable indicating whether feature i is assigned to group g $(1 \leq g \leq G)$, and q_{gj} be a binary variable indicating whether class j is assigned to group g. We define $\boldsymbol{p}_g \in \mathbb{Z}_2^D$ as a feature group assignment vector for group g, where $\mathbb{Z}_2 = \{0, 1\}$ and D is the dimension of features. Similarly, $\boldsymbol{q}_g \in \mathbb{Z}_2^K$ denotes a class group assignment vector for group g. That is, $\boldsymbol{p}_g$ and $\boldsymbol{q}_g$ define a group g together, where $\boldsymbol{p}_g$ represents features associated with the group and $\boldsymbol{q}_g$ indicates a set of classes assigned to the group.

Constraint

We assume that there is no overlap between groups, either in features or classes, i.e. $\sum_g \boldsymbol{p}_g = \mathbb{1}_D$ and $\sum_g \boldsymbol{q}_g = \mathbb{1}_K$, where $\mathbb{1}_D$ and $\mathbb{1}_K$ are the vectors with all-one elements. While this assumption imposes hard regularizations on group assignments, it enables the weight matrix $\boldsymbol{W} \in \mathbb{R}^{D \times K}$ to be sorted into a block-diagonal matrix, since each class is assigned to a group and each group depends on a disjoint subset of features. This greatly reduces the number of parameters, and at the same time, the multiplication $\boldsymbol{W}^\top \mathbf{x}$ can be decomposed into smaller and faster block matrix multiplications.

Objective function

The objective for training our SplitNet is then defined as:

$$\min_{\omega, \boldsymbol{P}, \boldsymbol{Q}} \mathcal{L}(\omega, \boldsymbol{X}, \boldsymbol{y}) + \sum_{l=1}^{L} \lambda ||\boldsymbol{W}^{(l)}||_2^2 + \sum_{l=S}^{L} \Omega(\boldsymbol{W}^{(l)}, \boldsymbol{P}^{(l)}, \boldsymbol{Q}^{(l)}), \tag{1}$$

where $\mathcal{L}(\boldsymbol{W}, \boldsymbol{X}, \boldsymbol{y})$ is the cross entropy loss on the training data, $\omega = \{\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(L)}\}$ is the set of network weights at all layers, $\|\boldsymbol{W}\|_2^2$ is the weight decay regularizer with a hyperparameter λ, S is the layer where splitting starts, $\Omega(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q})$ is the regularizer for splitting the network, and $\boldsymbol{P}^{(l)}$ and $\boldsymbol{Q}^{(l)}$ are the set of feature-to-group and class-to-group assignment vectors respectively, for each layer l.

Splitting regularization

Our regularization assigns features and classes into disjoint groups; it consists of three objectives as follows:

$$\begin{aligned}\Omega(\boldsymbol{W},\boldsymbol{P},\boldsymbol{Q}) = \gamma_1 R_W(\boldsymbol{W},\boldsymbol{P},\boldsymbol{Q}) \\ + \gamma_2 R_D(\boldsymbol{P},\boldsymbol{Q}) + \gamma_3 R_E(\boldsymbol{P},\boldsymbol{Q})\end{aligned} \tag{2}$$

where $\gamma_1, \gamma_2, \gamma_3$ controls the strength of each regularization, which will be discussed in following subsections.

$$\Omega(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) = \gamma_1 R_W(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) + \gamma_2 R_D(\boldsymbol{P}, \boldsymbol{Q}) + \gamma_3 R_E(\boldsymbol{P}, \boldsymbol{Q})$$

Group Weight Regularization

Let $\boldsymbol{P}_g = diag(\boldsymbol{p}_g)$ and $\boldsymbol{Q}_g = diag(\boldsymbol{q}_g)$ be the feature and class group assignment matrix for group g respectively. Then $\boldsymbol{P}_g \boldsymbol{W} \boldsymbol{Q}_g$ represents the weight parameters associated with group g, *i.e.* intra-group connections between features and classes. Since our goal is to prune out inter-group connections to obtain block-diagonal weight matrices, we minimize off block-diagonal entries as follows:

$$R_W(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) = \sum_g \sum_i ||((\boldsymbol{I} - \boldsymbol{P}_g)\, \boldsymbol{W}\, \boldsymbol{Q}_g))_{i*}||_2 + \sum_g \sum_j ||(\boldsymbol{P}_g\, \boldsymbol{W}\, (\boldsymbol{I} - \boldsymbol{Q}_g))_{*j}||_2 \quad (4)$$

where $(\boldsymbol{M})_{i*}$ and $(\boldsymbol{M})_{*j}$ denote i-th row and j-th column of $\boldsymbol{M}$. Eq.(4) imposes row/column-wise $\ell_{2,1}$-norm on the inter-group connections. Figure 2 illustrates this regularization, where the portions of the weights to which the regularization is applied are colored differently. [1]

$$\Omega(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) = \gamma_1 R_W(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) + \gamma_2 R_D(\boldsymbol{P}, \boldsymbol{Q}) + \gamma_3 R_E(\boldsymbol{P}, \boldsymbol{Q})$$

Visualization

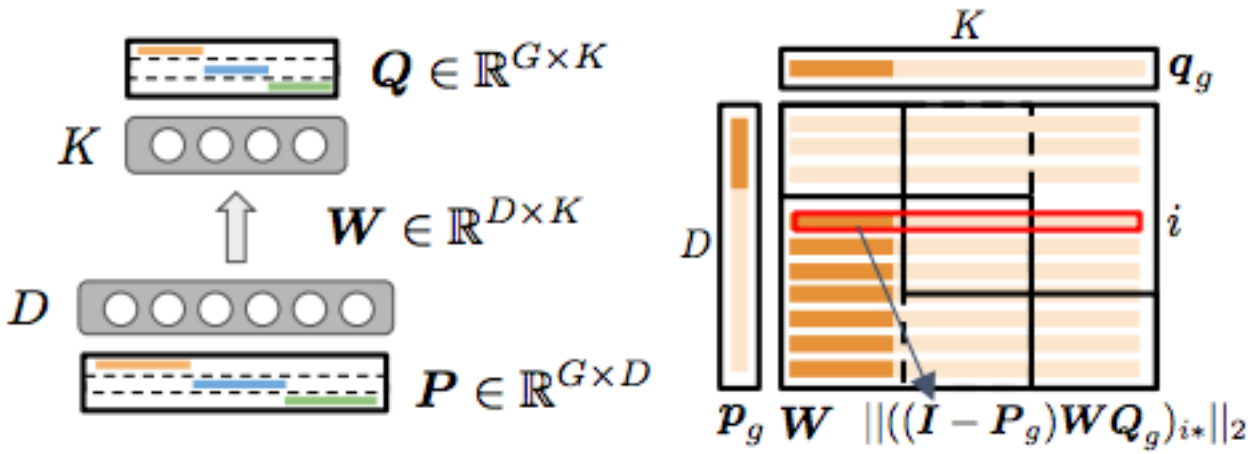


Figure 2. **Group Assignment and Group Weight Regularization.** (**Left**) An example of group assignment with $G = 3$. Colors indicate groups. Each row of matrix $\boldsymbol{P}, \boldsymbol{Q}$ is group assignment vectors for group g: $\boldsymbol{p}_g$, $\boldsymbol{q}_g$. (**Right**) Visualization of matrix $(\boldsymbol{I} - \boldsymbol{P}_g)\boldsymbol{W}\boldsymbol{Q}_g$. The group assignment vectors work as soft indicators for inter-group connections. As the groupings converge, $\ell_{2,1}$-norm is concentrated on inter-group connections.

$$\Omega(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) = \gamma_1 R_W(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) + \gamma_2 R_D(\boldsymbol{P}, \boldsymbol{Q}) + \gamma_3 R_E(\boldsymbol{P}, \boldsymbol{Q})$$

Disjoint Group Assignment

For the group assignment vectors to be completely mutually exclusive, they should be orthogonal; *i.e.* they should satisfy the condition $\boldsymbol{p}_i \cdot \boldsymbol{p}_j = 0$ and $\boldsymbol{q}_i \cdot \boldsymbol{q}_j = 0, \forall\, i \neq j$. We introduce an additional orthogonal regularization term:

$$R_D(\boldsymbol{P}, \boldsymbol{Q}) = \sum_{i<j} \boldsymbol{p}_i \cdot \boldsymbol{p}_j + \sum_{i<j} \boldsymbol{q}_i \cdot \boldsymbol{q}_j. \qquad (5)$$

where the inequalities avoid the duplicative dot products.

$$\Omega(\boldsymbol{W},\boldsymbol{P},\boldsymbol{Q}) = \gamma_1 R_W(\boldsymbol{W},\boldsymbol{P},\boldsymbol{Q}) + \gamma_2 R_D(\boldsymbol{P},\boldsymbol{Q}) + \gamma_3 R_E(\boldsymbol{P},\boldsymbol{Q})$$

Balanced Group Assignment

The disjoint group assignment objective in Eq.(5) alone may drive one group to dominate over all other groups; that is, one group includes all features and classes, while other groups do not. Therefore, we also constrain the group assignments to be balanced, by regularizing the squared sum of elements in each group assignment vector.

$$R_E(\boldsymbol{P},\boldsymbol{Q}) = \sum_g \Big((\sum_i p_{gi})^2 + (\sum_j q_{gj})^2 \Big). \quad (6)$$

Splitting DNNs

Our weight-splitting method in section 3.2 can be applied to deep neural networks (DNN), which has two types of layers: 1) the input and hidden layers that produce a feature vector for a given input, and 2) the output fully-connected (FC) layer on which the softmax classifier produces class probabilities. The output FC layer can be split by directly applying our method in section 3.2 on the output FC weight matrix $\boldsymbol{W}^{(L)}$. Our splitting framework can be further extended into deep splits, involving either multiple consecutive layers or recursive hierarchical group assignments. Algorithm 1 describes the deep splitting process.

Algorithm for splitting DNNs

Algorithm 1 Splitting Deep Neural Networks

Input: Number of groups G, layers to split $S \leq L$ and hyper-paramaters $\gamma_1, \gamma_2, \gamma_3$
Initialize weights and group assignments
while groupings have not converged **do**
  Optimize the objective using SGD with a learning rate η

$$\mathcal{L}(\omega, \boldsymbol{X}, \boldsymbol{Y}) + \lambda \sum_{l=1}^{L} ||\mathbf{W}^{(l)}||_2^2 + \gamma_1 \sum_{l=S}^{L} R_W(\boldsymbol{W}^{(l)}, \boldsymbol{P}^{(l)}, \boldsymbol{Q}^{(l)})$$

$$+\gamma_2 \sum_{l=S}^{L} R_D(\boldsymbol{P}^{(l)}, \boldsymbol{Q}^{(l)}) + \gamma_3 \sum_{l=S}^{L} R_E(\boldsymbol{P}^{(l)}, \boldsymbol{Q}^{(l)})$$

end while
Split the network using the obtained group assignments and weight matrices
while validation accuracy improves **do**
  Optimize $\mathcal{L}(\omega, \boldsymbol{X}, \boldsymbol{Y}) + \lambda \sum_{l=1}^{L} ||\mathbf{W}^{(l)}||_2^2$ using SGD
end while

Layers to split

The lower layers of a DNN learn low-level, generic representations, which are likely to be shared across all classes. The higher level representations, on the contrary, are more likely to be specific to the classes in a particular group. Therefore we do not split all layers but split layers down to S-th layer ($S \leq L$), while maintaining lower layers ($l < S$) to be shared across class groups.

Deep Split

Each layer consists of input and output nodes with the weights $\boldsymbol{W}^{(l)}$ that connects between them, and $\boldsymbol{P}_g^{(l)}$, $\boldsymbol{Q}_g^{(l)}$ for input-to-group and output-to-group assignments. Since the output nodes of each layer correspond to the input nodes of the next layer, the grouping assignment are shared as $\boldsymbol{q}_g^{(l)} = \boldsymbol{p}_g^{(l+1)}$. This enforces that no signal is passed across different groups of layers, so that forward and backward propagation in each group is independent from the processes in other groups. This allows the computations for each group to be parallelized, except for the softmax layer. The softmax layer includes a normalizing operation over all classes which requires aggregating logits over groups; however, during inference it suffices to identify the class with the maximum logit, which can be simply obtained by first identifying the class with maximum logit in each group, and then selecting the class with maximum logit among the identified group-specific maximums. This requires minimal communication overhead.

Application to CNNs

When applied to CNNs, the proposed group splitting process can be performed in the same manner on the convolutional filters. Suppose that a weight of a convolutional layer is a 4-D tensor $\boldsymbol{W}_c \in \mathbb{R}^{M \times N \times D \times K}$, where M, N denote height and width of receptive fields and D, K denote the number of input and output convolutional filters. We reduce the 4-D weight tensor $\boldsymbol{W}_c$ into a 2-D matrix $\boldsymbol{W}'_c \in \mathbb{R}^{D \times K}$ by taking the root sum squared of elements over height and width dimensions of $\boldsymbol{W}_c$, i.e. $\boldsymbol{W}'_c = \{w'_{dk}\} = \{\sqrt{\sum_{m,n} w^2_{mndk}}\}$. Then the weight regularization objective for the convolutional weight is obtained by Eq.(4), using $\boldsymbol{W}'_c$ instead.

$$R_W(\boldsymbol{W}, \boldsymbol{P}, \boldsymbol{Q}) = \sum_g \sum_i ||((\boldsymbol{I} - \boldsymbol{P}_g)\, \boldsymbol{W}\, \boldsymbol{Q}_g))_{i*}||_2 + \sum_g \sum_j ||(\boldsymbol{P}_g\, \boldsymbol{W}\, (\boldsymbol{I} - \boldsymbol{Q}_g))_{*j}||_2 \quad (4)$$

Hierarchical Grouping

Assume that the grouping branches at the l-th layer and the output nodes of the l-th layer are grouped by G supergroup assignment vectors $\boldsymbol{q}_g^{(l)}$ with $\sum_g \boldsymbol{q}_g^{(l)} = \mathbb{1}_D$. Suppose that in the next layer, for each supergroup $g \in \{1, ..., G\}$ there are corresponding S subgroup assignment vectors $\boldsymbol{p}_{gs}^{(l+1)}$ with $\sum_{s,g} \boldsymbol{p}_{gs}^{(l+1)} = \mathbb{1}_D$. As aforementioned, the input nodes to the l+1-th layer corresponds to the output nodes of l-th layer. By defining $\boldsymbol{p}_g^{(l+1)} = \sum_s \boldsymbol{p}_{gs}^{(l+1)}$, we can map subgroup assignments into corresponding supergroup assignments. This allows us to impose the constraint $\boldsymbol{q}_g^{(l)} = \boldsymbol{p}_g^{(l+1)}$ as in Deep Split.

Parallelization of SplitNet

Our learning algorithm produces a tree-structured network whose subnetworks have no inter-group connections. This results in an embarrassingly model-parallel network where we can simply assign the obtained subnetworks to each processor, or a machine. In our implementation, we consider two approaches for model parallelization: 1) Assigning both the lower-layers and group-specific upper layers to each node. At the test time the lower layers are not changed; thus this approach is acceptable, although it causes unnecessary redundant computations across the processors. 2) Assigning the lower layer to a separate processor. This eliminates redundancies in the lower layer but incurs communication overhead between the lower layer and the upper layers.

Unfortunately…

Training-time parallelization is currently done only at the finetuning step, after group assignments have been decided. We leave the parallelization from the initial network training stage as future work.

Comparison: Baseline

1) Base Network. Base networks with full network weights. For experiments on the CIFAR-100, we use Wide Residual Network (WRN) (Zagoruyko & Komodakis, 2016), which is one of the state-of-the-art networks of the dataset. We use AlexNet (Krizhevsky et al., 2012) and ResNet-18 (He et al., 2016) variants as the base network for the ILSVRC2012.

Comparison: SplitNet Variants

2) SplitNet-Semantic. A variant of our SplitNet that obtains class grouping from a provided semantic taxonomy. Before training, we split the networks according to the taxonomy, evenly splitting layers and assigning subnetworks to each group, and train it from scratch. We use the same approach for SplitNet-Clustering and SplitNet-Random.

3) SplitNet-Clustering. A variant of our SplitNet, where classes are split (hierarchical) performing spectral clustering of the pre-trained base network weights.

4) SplitNet-Random. SplitNet using random class splits.

Performance for WRN

Table 1. **Comparison of Test Errors According to Depths of Splitting (row) and Splitting Methods (column) on CIFAR-100.** Postfix S, C and R denote SplitNet variants – Semantic, Clustering and Random, respectively.

WRN-16-8 (BASELINE)						24.28
WRN-16-8 (DROPOUT)						24.52
METHOD	SPLIT DEPTH	G	SPLITNET-S	SPLITNET-C	SPLITNET-R	SPLITNET
FC SPLIT	1	4	23.80	23.72	24.30	24.26
SHALLOW SPLIT	6	2	24.46	24.54	25.46	**23.96**
DEEP SPLIT (DROPOUT)	11	2	25.04	26.04	27.12	24.62
HIER. SPLIT (DROPOUT)	11	2-4	24.92	25.98	26.78	24.80

Reduced Parameters/Computations

Table 2. **Comparison of Parameter/Computation Reduction and Test Errors on CIFAR-100.**

NETWORK	PARAMS(10^6)	% REDUCED	FLOPS(10^9)	% REDUCED	TEST ERROR(%)
WRN-16-8 (BASELINE)	11.0	0.0	3.10	0.0	24.28
FC SPLIT	11.0	0.35	3.10	0.0	24.26
SHALLOW SPLIT	7.42	32.54	2.64	14.63	**23.96**
DEEP SPLIT (DROPOUT)	5.90	46.39	2.11	31.97	24.66
HIER. SPLIT (DROPOUT)	4.12	62.58	1.88	39.29	24.80

Performance for AlexNet

Table 3. **Comparison of Parameter/Computation Reduction and Test Errors of AlexNet variants on ILSVRC2012.** The number of splits indicates the split in *fc6*, *fc7* and *fc8* layer, respectively. In 2×5 split, we split from *conv4* to *fc8* with $G = 2$.

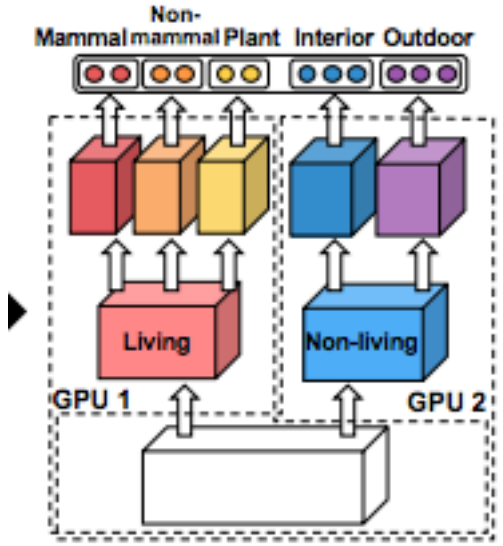
NETWORK	SPLITS	PARAMS(10^6)	% REDUCED	FLOPS(10^9)	% REDUCED	TEST ERROR(%)
ALEXNET(BASELINE)	0	62.37	0	2.278	0	41.72
SPLITNET	1-1-3	59.64	4.38	2.273	0.21	42.07
	1-2-5	50.69	18.72	2.256	1.00	42.21
	2-4-8	27.34	56.17	2.209	3.05	43.02
	2×5	31.96	48.76	1.847	18.95	44.60
SPLITNET-R	1-1-3	59.64	4.38	2.273	0.21	42.20
	1-2-5	50.70	18.70	2.256	0.99	43.20
	2-4-8	27.34	56.17	2.209	3.05	43.35
	2×5	31.94	48.79	1.846	18.93	44.99

Performance for ResNet

Table 4. **Comparison of Parameter/Computation Reduction and Test Errors of ResNet-18 variants(ResNet-18x2) on ILSVRC2012.** The number of splits indicates the split in *conv4-1&2*, *conv5-1&2* and the last fc layer, respectively.

NETWORK	SPLITS	SPLIT DEPTH	PARAMS(10^6)	% REDUCED	FLOPS(10^9)	% REDUCED	TEST ERROR(%)
RESNET-18X2	0	0	45.67	0	14.04	0	25.58
SPLITNET	1-1-3	1	44.99	1.49	14.04	0.01	**24.90**
	1-2-2	6	28.39	37.84	12.39	11.72	**25.48**
	2-2-2	11	24.21	47.00	10.75	23.42	26.45
SPLITNET-R	1-1-3	1	44.99	1.49	14.03	0.01	25.86
	1-2-2	6	28.38	37.86	12.39	11.72	26.41
	2-2-2	11	24.14	47.14	10.75	23.46	28.61

Model Parallelization



tion. We test two approaches: 1) Redundant assignment of lower layers (Deep and Hier. Split), and 2) assignment of lower layers to a separate GPU (Deep Split 3-way). With redundant assignment, the speedup becomes larger with

Test time performance

Table 5. **Model Parallelization Benchmark of SplitNet on Multiple GPUs.** We measure evaluation time performance of our SplitNet over 50,000 CIFAR-100 images with batch size 100 on TITAN X Pascal. Baseline implements layer-wise parallelization where sequential blocks of layers are distributed on GPUs.

NETWORK	GPUS	TIME(S)	SPEEDUP(×)
BASELINE	1	24.27 ± 0.35	1.00
BASELINE-HORZ.	2	46.70 ± 0.48	0.52
BASELINE-VERT.	2	20.15 ± 0.67	1.20
BASELINE-VERT.	3	22.45 ± 0.77	1.08
SHALLOW SPLIT	2	17.78 ± 0.23	1.37
DEEP SPLIT	2	14.03 ± 0.13	**1.73**
HIER. SPLIT	2	14.22 ± 0.05	1.71
DEEP SPLIT 3-WAY	3	10.92 ± 0.44	**2.22**

Effect of regularization

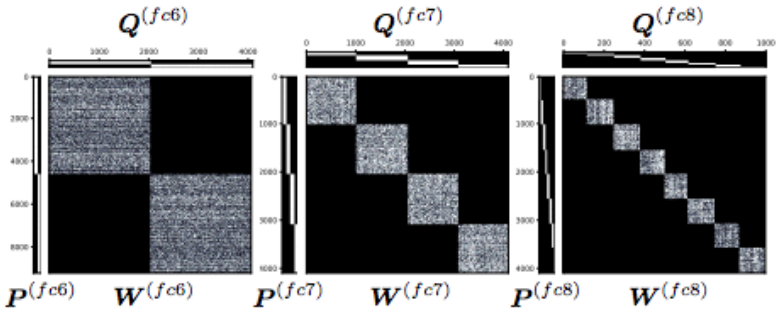


Figure 3. **Learned Groups and Block-diagonal Weight Matrices.** Visualization of the weight matrices along with corresponding group assignments learned in in AlexNet 2-4-8 split. Obtained block-diagonal weights are *split* for faster multiplication. Note the hierarchical grouping structure.

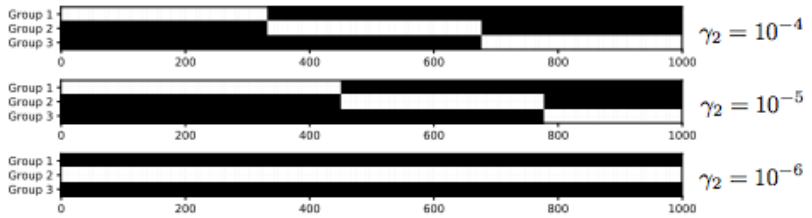


Figure 4. **Effect of Balanced Group Regularization** $R_E(\boldsymbol{P}, \boldsymbol{Q})$. The above figures show group-to-class assignment matrix $\boldsymbol{Q}^{(fc8)}$ for different values of γ_3 on ImageNet-1K with $G = 3$

Learned Groups

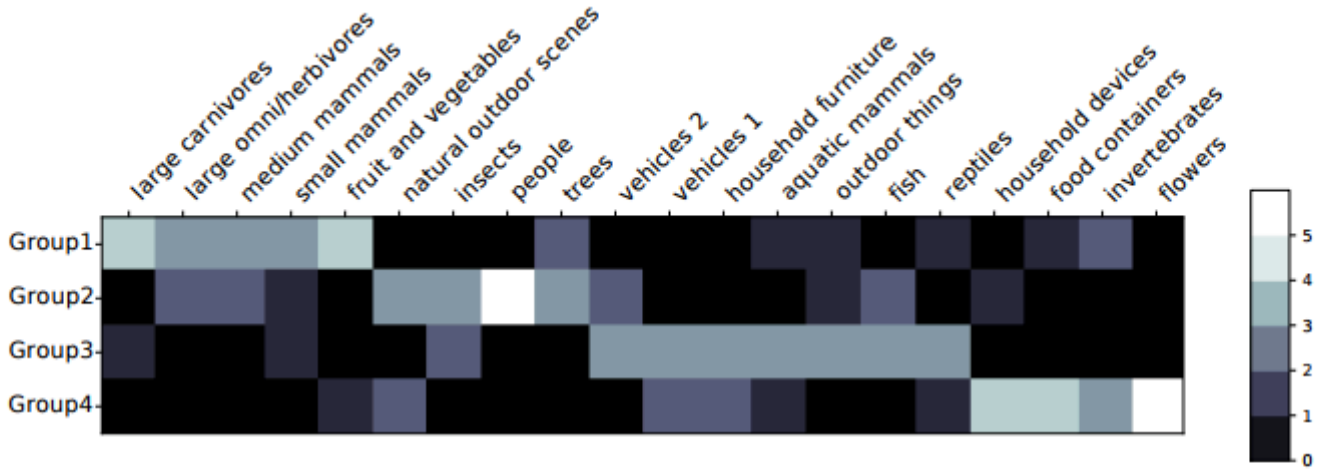


Figure 5. **Learned Groups.** Visualization of grouping learned in FC SplitNet on CIFAR-100. Rows denote learned groups, while columns denote semantic groups provided by CIFAR-100 dataset, each of which includes 5 classes. The brightness of a cell shows the agreement between learned groups and semantic groups.

Learned Groups

Figure 5 compares the learned group assignments in FC SplitNet ($G = 4$) with supercategories provided by the CIFAR-100. Each supercategory (column) includes five classes. For example, supercategory *people* includes *baby*, *boy*, *girl*, *man* and *woman*, which are grouped together by our algorithm into Group 2. Note that we have at least three classes from the same supercategory in each group. This shows that groupings learned by our method bear some resemblance to semantic categories even when no external semantic information is given.

Ending Notes

- Parallelization on the initial training stage
- Overall, well-written paper, convincing experiments
- Possible applications might be running DNNs on mobile devices, embedded computers, IoT?
- Splitting for RNNs?