Pattern Recognition Letters
journal homepage: www.elsevier.com

Block-wise Dynamic Sparseness

Amir Hadifar[a,], Johannes Deleu[a], Chris Develder[a], Thomas Demeester[a]

[a]Ghent University – imec, Department of Information Technology, Technologiepark 126, 9052 Zwijnaarde, Belgium

ABSTRACT

Neural networks have achieved state of the art performance across a wide variety of machine learning tasks, often with large and computation-heavy models. Inducing sparseness as a way to reduce the memory and computation footprint of these models has seen significant research attention in recent years. In this paper, we present a new method for *dynamic sparseness*, whereby part of the computations are omitted dynamically, based on the input. For efficiency, we combined the idea of dynamic sparseness with block-wise matrix-vector multiplications. In contrast to static sparseness, which permanently zeroes out selected positions in weight matrices, our method preserves the full network capabilities by potentially accessing any trained weights. Yet, matrix vector multiplications are accelerated by omitting a pre-defined fraction of weight blocks from the matrix, based on the input. Experimental results on the task of language modeling, using recurrent and quasi-recurrent models, show that the proposed method can outperform a magnitude-based static sparseness baseline. In addition, our method achieves similar language modeling perplexities as the dense baseline, at half the computational cost at inference time.

1. Introduction

Deep Neural Networks (DNNs) have been a success story in recent years, due to their impressive performance on various domains. Theoretically and empirically, it has been shown that DNNs, trained by first-order-methods such as stochastic gradient decent (SGD), are able to represent a wide variety of complex functions (Hornik et al., 1989). Although in general it seems that utilizing large and fully parameterized networks is a reasonable way to increase prediction effectiveness, the computational complexity and memory demand of these models may become a bottleneck.

Many methods have been proposed to address that general issue of computational and memory complexity associated with large models. Researchers have focused on carefully redesigning *neural network architectures* to reduce the computational cost while maintaining effectiveness. Such approaches include: (i) *tensor decomposition* to express tensors in terms of a sequence of operations on simpler (e.g., smaller) tensors (Denton et al., 2014), (ii) *quantization* to reduce the precision of weights and activation functions with minimal impact on performance (Courbariaux et al., 2016), (iii) *knowledge distillation* to transfer knowledge of a larger model into another lightweight model (Hinton et al., 2015), and (iv) *network pruning* to remove redundant and uncritical connections in order to arrive at sparse models (Han et al., 2015a).

This paper can be situated in the area of network pruning. However, whereas most previous work focuses on what we call *static* pruning, i.e., permanently disabling specific network connections, we focus on *dynamic* pruning, whereby a well-chosen pruned version of the network is used, depending on the input.

Yet, to avoid inefficiency which would result from a fully flexible sparsity pattern for each input instance, we propose block-wise pruning: entire blocks of model parameters are jointly pruned, thus facilitating an efficient implementation with a limited number of additional parameters to compute the dynamic pruning mask.

We argue that our proposal of dynamic block-wise pruning can achieve a (reduced) computational cost similar to static pruning, while largely keeping the expressiveness of the non-sparse case, in the sense that overall (i.e., across all instances) the same number of network parameters can be tuned. The next section provides an extensive overview of related work in the area of pruning techniques for neural networks. The subse-

e-mail: `amir.hadifar@ugent.be` (Amir Hadifar) quent sections further detail our contributions, which can be summarized as follows: (i) We propose a dynamic block-wise pruning approach (Section 2), aimed at achieving low computational cost without affecting expressivity much, (ii) We experimentally validate our approach, illustrating its dynamic gating mechanism on MNIST (Section 4.1), as well as (iii) the effectiveness of dynamic sparseness in limiting computational complexity while maintaining performance for language modeling (Section 4.2). The final Section 5 summarizes our conclusions and indicates potential areas of follow-up research.

2. Related Work

This section provides an overview of related literature. We start by providing a high-level overview on various strategies for pruning, thereby attempting to bring clarity in the diverse terminology used in literature (Section 2.1). We then explore the following sub-domains in more detail: unstructured pruning (Section 2.2), structured pruning (Section 2.3), and conditional computation (Section 2.4).

2.1. Overview of Pruning Strategies

The first notion of sparsity in the area of artificial neural networks appeared as pruning techniques (Sietsma and Dow, 1988) designed to gradually switch off increasing numbers of network parameters during training, until either performance starts to drop, or a certain level of sparseness is obtained. It has been found to be a practical solution to reduce both network complexity and over-fitting (Han et al., 2015b). By discarding a fraction ς of the model parameters, the computational complexity may be reduced by a factor of $1 - \varsigma$. However, eliminating a substantial fraction of the parameters reduces the ability of the network to take advantage of detailed correlations in the data.

Early studies (Reed, 1993) consider two main types of pruning, namely sensitivity and penalty-term methods. Sensitivity methods rely on the estimation of influence of a specific node or weight. Penalty-term methods modify the objective function to force neural networks to remove redundant weights during training. Although some methods combine both approaches (Finnoff et al., 1993), others cannot be easily added to the other family of methods (Whitley, 1990).

Pruning algorithms are sometimes divided into the following broad categories: weight pruning vs. neuron pruning (Li et al., 2018). The distinction is in the fact that neuron pruning removes entire neurons (columns/rows in weight matrix) whereas weight pruning is applied on individual entries of the weight matrices. This terminology is less suited to describe some recent methods, where entire subnetworks may be pruned; e.g., heads in transformers or filters in convolution layers (Voita et al., 2019; He et al., 2018).

Recent studies therefore divide pruning techniques into *structured* and *unstructured* methods. Unstructured pruning is usually applied on individual parameters and does not follow a specific pattern or constraint (e.g., LeCun et al. (1990); Han et al. (2015b)), while structured methods keep the network architecture intact and pruning is applied at the level of filters, channels, or layers (Wen et al., 2016; Lin et al., 2017). An argument against unstructured pruning is that obtaining real performance gains seems hard to achieve without dedicated hardware/libraries (Han et al., 2016). In turn, some authors have argued that models obtained by structured pruning seem to be less accurate (Vooturi et al., 2018).

Finally, there is another strategy on how to use network pruning, quite different from the ideas mentioned above. The goal is to train large models which at inference time require the same computational cost as smaller ones, by pruning certain computation paths depending on the input. This idea is referred to as *dynamic execution* (Gao et al., 2018), *runtime pruning* (Lin et al., 2017), or more broadly *conditional computing* (Bengio, 2013), and it is this paradigm that our work follows.

2.2. Unstructured Network Pruning

An early contribution of unstructured pruning was the use of hyperbolic and exponential biases to decay network weights (Hanson and Pratt, 1989). The Optimal Brain Damage and Surgeon methods (LeCun et al., 1990; Hassibi and Stork, 1993) allowed reducing the number of connections based on the Hessian of the loss function. Han et al. (2015b) proposed a magnitude-based pruning method, whereby weights below a user-defined threshold were truncated. Narang et al. (2017a) applied magnitude pruning on weight matrices of Recurrent Neural Networks (RNNs). They pruned weights below a threshold, monotonically increased during training. As opposed to previous techniques, where pruned connections were lost permanently, Guo et al. (2016) proposed a magnitude-based pruning schema in which eliminated connections could be recovered with some probability. Some studies employed regularization as a proxy to induce sparseness. Louizos et al. (2017) utilized the L_0-norm to learn sparse networks. Molchanov et al. (2017) devised variational dropout to truncate redundant weights. However, these regularization methods usually suffer from the lack of control over the level of sparseness.

2.3. Structured Network Pruning

Recent research on network pruning has seen several interesting contributions on structured pruning methods.

Regularization-based pruning can be devised in a structured way. Wen et al. (2016) proposed a framework based on a group-lasso penalty to remove network components in filter-wise, channel-wise, shape-wise, and depth-wise formats. Similarly, Wen et al. (2017) showed how rows and columns in weight matrices of the Long Short Term Memory (LSTM) can be pruned via a group lasso regularization. Liu et al. (2017) induced sparsity on the batch normalization scaling factors to prune channels. They impose L_1 regularization on the scaling factors in batch norm to push them toward zero in order to identify insignificant channels. He et al. (2018) brought ideas from Guo et al. (2016) into structured pruning by introducing soft filter pruning, whereby during training the filters could recover after first being pruned. More recently, Voita et al. (2019) proposed a method which eliminates the attention heads in transformers using a gating mechanism and an L_0 regularization.

Neuron pruning, as introduced in the previous Section 2.1, can be considered a form of structured pruning. Srinivas and Babu (2015) presented a variation of the ReLU activation function that has learnable parameters which multiply with neurons output. The parameters are encouraged to take binary values with the help of regularizers that reward values close to 0 or 1. Those neurons with a zero value can simply be removed. Hu et al. (2016) proposed to prune individual neurons based on statistics of the network output.

Structured pruning is sometimes referred to as group pruning, whereby entire groups of elements are pruned at once (Zmora et al., 2019). Various approaches exist, but the most popular idea is block pruning in which entire blocks of weights are removed. Notable contributions are from Narang et al. (2017b), who applied block-wise pruning on RNNs, and Varma et al. (2019), who introduced Dynamic Block Sparse Reparameterizations (DBSR) where sparse patterns for convolutional layers are learned with an L_1 regularizer. However, contrary to our approach, their generated patterns are static at inference time. Demeester et al. (2018) proposed ideas to enforce blockwise sparseness *up front* in word embedding layers and recurrent networks, to also benefit from sparseness during training. Their block-sparse RNN layers were shown to be equivalent to multiple smaller dense RNNs in parallel, each focusing on subregions of the input, with concatenated outputs.

Van Keirsbilck et al. (2019) presented a more general exploration of ways to induce sparseness up front on recurrent architectures. As a result, they observed that some types of sparse RNNs (e.g., the DiagonalRNN) offer better parallellization and acceleration possibilities in comparison to standard architectures.

2.4. Conditional Computation

Conditional computation has been proposed previously to improve model performance without a proportional increase in computational costs. Bolukbasi et al. (2017) created a pipeline by stacking multiple DNN models and designed a decision function for what they called 'early-exit'. Its purpose was to decide for each input, which of the intermediate models would already allow for a correct prediction. In effect, the system avoided the computational time associated with full evaluation of the pipeline. Almahairi et al. (2016) introduced the so-called dynamic capacity network, whereby it adaptively focused on task-specific regions of the input data. The proposed model consisted of two modules: a low capacity network that is activated on the whole input to find task-specific regions, and a high capacity network that is directed by an attention mechanism to focus on the selected regions.

Imposing sparseness in a Mixture of Experts (MoE) can also be seen as a form of conditional computation. Shazeer et al. (2017) proposed the gated MoE, consisting of up to thousands of experts, where a trainable mechanism determines a sparse combination of the experts to use for each example. The gating mechanism cannot always be represented by differentiable functions and thereby some studies resorted to reinforcement learning techniques. For instance, Lin et al. (2017); Wu et al. (2018) trained a policy network with reinforcement learning to choose which filters or residual-blocks to enable for a given input. However, these methods usually require significant computing resources for training.

The studies that are most closely related to our approach are Gao et al. (2018) and Chen et al. (2019), which applied conditional gating on channels and filters in Convolution Neural Networks (CNNs). However, in our approach, we devise a gating mechanism on a block level and it is applicable for any matrix-vector multiplication (rather than being CNN-specific).

3. Dynamic Sparse Linear Layer

In this section, we motivate and explain our proposed blockwise linear layer with dynamic sparseness. It can be applied in any neural network component that contains a matrix-vector product, to speed up computation at inference time.

Consider a neural network layer that requires a matrix-vector product $W\boldsymbol{h}$ between a weight matrix $W \in \mathbb{R}^{m \times n}$ and a vector $\boldsymbol{h} \in \mathbb{R}^n$, which in general would be the output from a previous layer. Our goal is to strongly reduce the computational load of this operation ($m\,n$ multiplications, $m\,(n-1)$ summations) at inference time. This could be achieved by a static sparse counterpart, mentioned in Section 2. Indeed, setting a number of well-chosen entries in W to zero would lead to reduced requirements both in computation and memory. However, the model with the remaining entries in W has to learn a suitable transformation for the entire space of possible inputs $\boldsymbol{h}$, which may become harder when fewer parameters in W remain available to the model. Indeed, limiting the number of non-zero entries in W may correspondingly reduce the expressiveness of the model. The underlying idea behind dynamic sparseness is to decide at run-time which entries in W can be ignored, based on $\boldsymbol{h}$. Dynamic sparseness therefore stems from the pursuit of achieving a similar gain in computations at inference time, while still

having the flexibility of training all individual weight matrix entries.

3.1. Sparseness through Gating

A sparse matrix can be seen as the result of applying a binary masking matrix G to a dense matrix W in an element-wise product. This mechanism is also called gating: entry W_{ij} is zeroed out when the corresponding gate is closed ($G_{ij} = 0$), and remains unchanged when the gate is open ($G_{ij} = 1$).

More generally, gating can be categorized into *continuous* and *discrete* gating. On the one hand, continuous forms of gating such as soft attention (Bahdanau et al., 2014) and gated linear units (Dauphin et al., 2017) have desirable properties which allow *learning* the gating coefficients, but they do not offer actual sparseness. For example, when calculating gating coefficients with a sigmoid function, closed gates correspond to very small values, which are never exactly zero, though. Discrete gating, on the other hand, where closed gates correspond to $G_{ij} = 0$, effectively achieve sparse operations. For instance, the static sparseness where only selected entries in W are non-zero, can be seen as a predetermined form of discrete gating.

In the following paragraphs, we propose a gating mechanism which combines the advantages of both: it leads to sparse operations, but is trainable with gradient-based methods as in the continuous gating case. Note that other techniques than the one proposed here are possible, and we discuss some of these in Section 3.3.

In order to make the gating mask applied to W dependent on the vector $\boldsymbol{h}$, we need a decision function $\mathcal{G}(\boldsymbol{h}, \theta, \varsigma)\colon \mathbb{R}^n \mapsto \mathbb{R}^{m \times n}$ which maps $\boldsymbol{h}$ to an $m \times n$ gating mask. The function $\mathcal{G}$ should be able to meaningfully differentiate between individual $\boldsymbol{h}$, and it should be trainable in terms of its parameters θ. Furthermore, for a user-specified sparseness level ς, a fraction ς of its entries should be zero. It is defined as follows:

$$\mathcal{G}(\boldsymbol{h}, \theta, \varsigma) = \frac{S_\varsigma\Big(\mathrm{FF}(\boldsymbol{h}, \theta)\Big)}{\Big\langle S_\varsigma\Big(\mathrm{FF}(\boldsymbol{h}, \theta)\Big)\Big\rangle}. \tag{1}$$

The function $\mathrm{FF}(\boldsymbol{h}, \theta)\colon \mathbb{R}^n \mapsto \mathbb{R}^{m \times n}$ represents a feed-forward neural network with trainable parameters θ. In our experiments we use a single layer, with ReLU activation. The $m\,n$ output activations of $\mathrm{FF}(\boldsymbol{h}, \theta)$ are organized in an m by n masking matrix. The function S_ς induces sparseness by retaining the top k activations and setting all others to zero, where k corresponds to the required number of non-zero entries, i.e., $k = (1 - \varsigma)\, m\, n$ (rounded to the nearest integer). Each entry in the resulting matrix is subsequently divided by its mean value, denoted by the denominator in Eq. (1). With this normalization, the mean value of the gates is always 1. As such, for any distribution of scores in the gating mask, or when changing ς during training (see Section 4.2), the net 'mass' of applied gating always remains the same. Our experimental results are in line with this intuition: the normalization is needed for robustness during training and leads to better results.

Finally, the dynamic sparse matrix-vector product is obtained as $(\mathcal{G}(\boldsymbol{h}, \theta, \varsigma) \odot W)\, \boldsymbol{h}$. The general formulation outlined above is however computationally heavier than the original product $W\, \boldsymbol{h}$. The next paragraphs outline our suggested simplifications of Eq. (1) to arrive at an overall more efficient model.

3.2. Modifications for Efficiency

The gating mechanism outlined above introduces extra parameters and computational complexity. We can reduce that overhead in practice by using block-wise sparse operations, which limits the output size of the feed-forward layer $\mathrm{FF}(\boldsymbol{h}, \theta)$, and can be made considerably more efficient than other sparse representation methods such as the Compressed Sparse Row (CSR) format (Aktulga et al., 2014). Note that the block-wise implementation limits the granularity on the sparseness level that can be achieved, but compared to unstructured sparseness it allows avoiding issues related to indexing storage overhead and irregular memory accesses (Narang et al., 2017b).

In particular, we partition the row-dimension of W into r segments, and its column-dimension into c segments. We can then write W as a block matrix

$$W = \begin{pmatrix} W_{11} & \cdots & W_{1c} \\ \vdots & \ddots & \vdots \\ W_{r1} & \cdots & W_{rc} \end{pmatrix}$$

and assign a single gate value to each submatrix W_{ij}, rather than to each individual scalar entry. We assume that while training, the neural network is capable of organizing its hidden state dimensions in a way that permits the simultaneous switching on or off of these entire submatrices through the gating. The output dimension of the feed-forward layer FF hence reduces from $n\,m$ to $r\,c$.

We can go even further, and assume that the network should be able to organize itself if only part of $\boldsymbol{h}$ is actually used as the key to calculate the gating coefficients. As a result, the input dimension of $\mathrm{FF}(\boldsymbol{h}, \theta)$ can be reduced as well.

The gating mechanism becomes potentially much cheaper thanks to the ideas of (i) using a coarser grid of gate coefficients in combination with W as a block matrix, and (ii) using a specific subregion from the $\boldsymbol{h}$ space to calculate these gates. In our experimental results in Section 4.2 we will quantify the resulting computational gain.

3.3. Alternatives for Dynamic Gating

As mentioned above, the top k selection of scores through S_ς as non-zero gate values is our method of choice, because it is fast, and yields good results with gradient based optimization techniques. Alternatively, dynamic gating could be achieved by sigmoid functions, as in recurrent networks such as an LSTM. This would however leave little control over the sparseness level. Using a softmax function might constitute another alternative. However, the softmax would focus on a single gating coefficient. This can be circumvented by using the sum of as many softmaxes as one wants non-zero gating values:

$$G(\boldsymbol{h}, \theta, \varsigma) = \frac{1}{1 - \varsigma} \sum_{i=1}^{k} \mathrm{softmax}(\mathrm{FF}_i(\boldsymbol{h}, \theta_i)/\tau), \tag{2}$$

in which the FF_i ($i = 1 \ldots k$) are feed-forward layers tuned to steer the gating of each softmax component. The parameter τ is a temperature value. By gradually decreasing it, each softmax becomes more peaky and learns to selects a single entry. At inference time, at most k gates would open, corresponding to a sparseness level of at least $(1 - \varsigma)\, r\, c$ (for the block-wise implementation similar to Section 3.2). Note the normalization in Eq. (2) leads to mean gating values of 1, similar to Eq. (1). While this approach appeared to actually work for small cases, it is not computationally feasible for large-scale problems.

4. Experimental Results

In this section, we present our experimental results. We will start by visualizing gate coefficients for a simple feed-forward network (Section 4.1). This serves as a sanity check that our method is trainable, and does not lead to the collapsing of the open gates to the same positions for any input. After that, we will evaluate the presence of block-wise dynamic layers in neural sequence models for language modeling (Section 4.2), and provide further insights on dynamic gate distributions.

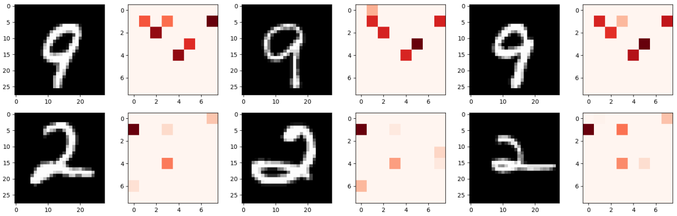

Fig. 1: Block specialization in the last hidden layer of a feed-forward network. The generated heat-map demonstrates that the model selects similar patterns for given instances of the same class.

4.1. Illustration on MNIST

We first demonstrate the dynamical sparseness model on a simple model for the task of digit recognition on the MNIST dataset (LeCun, 1998). Although CNN-based architectures such as AlexNet (Krizhevsky et al., 2012) would reach higher accuracies, our goal is mainly to apply our ideas to the most basic model, and visualize the gating mechanism.

We therefore chose for a simple feed-forward neural network, with 5 dynamic sparse hidden layers (each with 1024 units, and block size of 128×128), with sparseness level $\varsigma = 90\%$. The sparseness level was kept constant during the whole training process.

We trained the model for 45 epochs using momentum SGD, with a batch size of 128, and a learning rate of 5×10^{-3}. It is worth noting that using an overly high learning rate causes drastic updates on the gating mechanism, which leads the model to only focus on a few blocks.

We observe that after around 21 epochs the test error stabilizes to 2%, which is comparable to the dense case. Moreover, to validate our initial hypothesis that a dynamic sparseness model can benefit from addressing different blocks for distinct inputs, we looked into the last hidden layer of the network for two different classes. As shown in Fig. 1, for each class, a particular pattern specialization can be perceived. The dynamic gating not only employs an almost identical pattern of non-zero blocks for instances of the same class, but also the values are correlated among those same-class inputs (cf. similar color intensities). These results suggest that the dynamic sparsity model we applied has sufficient capacity to act in place of a dense one.

4.2. Language Modeling Experiments

In this section we consider the task of language modeling with recurrent sequence models, on two standard datasets, i.e., the Penn TreeBank (PTB) from Marcus et al. (1993) and Wikitext2 from Merity et al. (2016). The PTB has a vocabulary size of 10,000 words, and contains 1,036,580 tokens, whereas Wikitext2 has a vocabulary size of 33,278, for a total of 2,551,843 tokens. We apply our proposed model on two well-known architectures, the Quasi-RNN (Bradbury et al., 2017) and the regularized LSTM (Zaremba et al., 2014).

In the following experiments, we maintain the original dense word embedding layer for these models, since the vocabulary size is relatively small and selecting a vector from a matrix is already computationally efficient.

For the language modeling experiments, the dynamic sparseness was introduced by increasing ς linearly from zero to the reported final level, over a limited number of epochs during training. We noticed that this yields slightly better results compared to training with a fixed ς from the start, as was done for the MNIST experiments.

4.2.1. Quasi-RNN Language Model

A Quasi-RNN layer alternates convolutional layers, parallel across timesteps, and a recurrent pooling function, parallel across channels. As a result it is more time-efficient than fully recurrent models. We use a 4-layer Quasi-RNN, with embedding size of 512, hidden size $|\boldsymbol{h}| = 1536$, and tied weights for encoder/decoder. Note that the dimensions are slightly different from the original paper (embedding size 400, hidden size 1550), because our implementation assumes powers of 2 for the dimensions of the blocks in the block-wise dynamic sparseness model. We use the Adam optimizer (Kingma and Ba, 2014) with learning rate of 0.001, and linearly raise the target sparseness level ς up to the intended value between training epochs 350 and 450. Otherwise we keep the training setup from Bradbury et al. (2017). For reasons of fair comparison, we do not perform any additional hyper-parameter tuning for our models.[1]

Language modeling effectiveness. We now investigate the impact of our dynamic sparseness on the language modeling effectiveness for PTB. In particular, we apply our model on the filters of the convolutional layers. The results are shown in Table 1, for varying sparseness levels. We compare our method ('dyn. sparse') with the original model ('orig. dense') as well as a dense model with reduced $|\boldsymbol{h}|$ ('small dense') which requires the same number of multiplication-addition operations at inference time as our method. Note that we do not tune any hyper-parameters for that baseline either. For reference, Table 1 also lists the results from Bradbury et al. (2017) as 'orig. ref'. The number of parameters ('params') in their model is 3.6M lower than the 'orig. dense' model, because of the lower dimension for embedding layers (see above). Also, their perplexity is slightly higher than our dense baseline, which may be related to our different implementation with another optimizer. Besides $|\boldsymbol{h}|$ and the perplexity ('ppl'), the table lists the sparseness level ς, and the fraction of multiplication-addition computations in the matrix-vector product at inference time (denoted as 'comput.'), relative to the dense model.

Introducing sparsity, i.e., reducing the number of matrix-vector products, leads to increased perplexities, both for the dynamic sparse model and for the small dense baseline. However, the *dynamic* sparse models consistently outperform the *small dense* baselines with corresponding computational cost. For example, our technique allows halving the number of multiplication-addition operations (comput. = 0.5) at the cost of only +1.2 perplexity points, compared to +2.3 for the smaller dense model at the same computational cost.

[1] We use an existing implementation of the Quasi-RNN from fast.ai, adapted for pytorch 1.0.0

Table 1: Language modeling perplexity (ppl) of the reference implementation (orig. ref) from Bradbury et al. (2017), original Quasi-RNN (orig. dense), the small dense, and the dynamic sparse model, on the PTB test set. Column 'comput.' denotes the fraction of matrix-vector product computations compared to the dense baseline.

Model	$\|\boldsymbol{h}\|$	ς	ppl	comput.	params
orig. ref	1550	0%	56.70	-	43.8M
orig. dense	1536	0%	54.93	1.0	47.4M
small dense	810	0%	57.23	0.5	20.3M
	660	0%	58.76	0.4	16.3M
	500	0%	60.82	0.2	12.6M
	350	0%	64.08	0.1	9.7M
dyn. sparse (ours)	1536	50%	56.17	0.5	47.9M
	1536	60%	57.93	0.4	47.9M
	1536	80%	60.70	0.2	47.9M
	1536	90%	63.59	0.1	47.9M

Note that the mentioned increase in perplexity in going from dense to 50% sparse is an upper bound, and may be related to the additional regularization effect of the dynamic sparseness. This effect is not compensated for, as the dropout probabilities are not tuned. Indeed, we noticed that switching off weight drop regularization in both models, leads to the same (yet, higher) perplexity in the dense and the 50% sparse model.

We observe a small increase in the number of parameters, from 47.4M to 47.9M, for the calculation of the gate coefficients. Yet, in our experiments, the amount of corresponding additional multiplication-addition operations is an order of magnitude lower than the total computational gain that can be achieved by dynamic sparseness.

Analysis of the gating mechanism. We expect three different ways the gating mechanism may function in a network with dynamic computations: (i) Some gates will be always active: we expect certain blocks in a network to be of key importance for all types of inputs; (ii) Other gates would become active conditioned on the input: the blocks in the weight matrices that are more specialized for certain features are dynamically selected based on the input; (iii) Finally, some gates may always remain closed, corresponding to static sparsity. We categorized gates as 'always' on/off, if they are on/off for more than 95% of the instances on the test set. Figure 2 shows how the gates of the Quasi-RNN model with $\varsigma = 0.50$ are distributed among these categories, for each of the 4 Quasi-RNN layers. The total height of each bar indicates the total number of gates for the corresponding layer (with smaller numbers of gates for the first and last layer, as the embedding size is smaller than the hidden state size). We observe that at least 60% of the gates in all layers are actually input-dependent. This suggests that the model prefers dynamic sparsity over static sparsity.

4.2.2. LSTM Language Model

We now train an LSTM language model on the PTB and Wikitext2 datasets, using a model and training procedure similar to the one described in Zaremba et al. (2014). The model is composed of an embedding layer, 2 LSTM layers, and a softmax layer. Each LSTM layer has 1536 hidden units, and a 65% dropout rate is applied on the non-recurrent connections. The dimension of word embeddings in the input layer is 1536. We train the model for 55 epochs and start to increase the sparseness level from epoch 25 to 40 to reach the predefined level. Dynamic sparseness was applied to the input-to-hidden and hidden-to-hidden weight matrices in the LSTMs. Besides the original dense baseline, we also apply the static sparsity model called Automated Gradual Pruning (AGP) (Zhu and Gupta, 2017), which gradually prunes parameters based on their weight magnitude.[1]

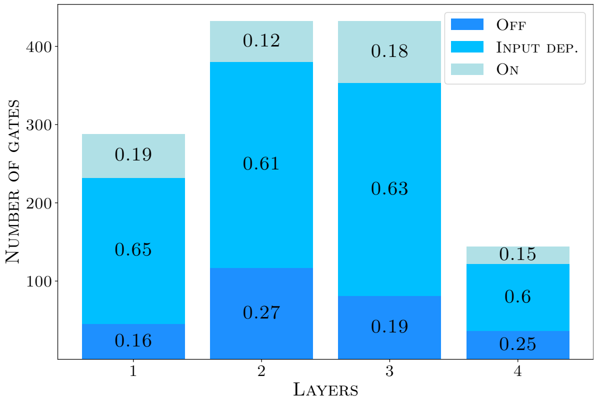

Fig. 2: Gate distribution on PTB test set: number of gates switched off or on over 95% of the time (Off and On, resp.) or in between (Input dep.), for the various Quasi-RNN layers. Corresponding fractions are shown on the bars.

Table 2 summarizes the results. There is a general degradation in the model quality with increasing sparseness levels, although with our 50% dynamic sparse model the baseline perplexity is still maintained. Our dynamic model consistently outperforms the static sparse model. This confirms our hypothesis that dynamic sparseness retains more expressiveness than static sparseness. If the computational cost at inference is more critical than memory, dynamic sparsity therefore presents a valid alternative. We further noticed that compensating for the regularization effect of the dynamic sparseness by slightly lowering the dropout rate compared to the dense case may lead to improved results for the block-wise dynamic sparseness model. Tuning the dropout rate over the values $\{0.4, 0.50, 0.55, 0.60, 0.65\}$ resulted in a 1.31 perplexity point decrement on PTB and 0.57 on wikitext-2 for the dynamic sparse model (with 0.60 as optimal dropout rate). However, similar tuning for the AGP method did not lead to further improvements.

5. Conclusion

We proposed the technique of block-wise dynamic sparseness, which can be used to reduce the computational cost at inference time for matrix vector products inside neural network building blocks. Our experimental results on Quasi-RNN and LSTM based language models provide a proof-of-concept of

[1] We used the existing implementation provided in the Distiller library (Zmora et al., 2019) for the AGP method

Table 2: LSTM language model test perplexity on PTB and Wikitext2. Comparing a standard dense LSTM model with a static sparse model and our dynamic sparse model, for different sparseness levels ς.

Model	ς	PTB	Wikitext2
Zaremba et al. (2014)	0%	78.40	-
orig. dense	0%	78.57	85.0
static sparse (AGP)	50%	80.92	86.91
	75%	81.06	88.45
	90%	81.75	89.48
dyn. sparse (ours)	50%	78.50	85.31
	75%	79.12	86.76
	90%	80.34	88.44

the proposed method, with significant reduction of the required computations at inference time at a very limited model effectiveness penalty. Under the same experimental settings, our method outperforms a baseline with static sparseness. Finally, the implementation of a cuda kernel to support the dynamic gating, and our code to reproduce the presented experimental results, are made publicly available .[1]

References

Aktulga, H.M., Buluç, A., Williams, S., Yang, C., 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations, in: IPDPS.

Almahairi, A., Ballas, N., Cooijmans, T., Zheng, Y., Larochelle, H., Courville, A., 2016. Dynamic capacity networks, in: ICML.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate, in: ICLR.

Bengio, Y., 2013. Deep learning of representations: Looking forward, in: SLSP.

Bolukbasi, T., Wang, J., Dekel, O., Saligrama, V., 2017. Adaptive neural networks for fast test-time prediction, in: ICML.

Bradbury, J., Merity, S., Xiong, C., Socher, R., 2017. Quasi-recurrent neural networks, in: ICLR.

Chen, Z., Li, Y., Bengio, S., Si, S., 2019. You look twice: GaterNet for dynamic filter selection in CNNs, in: CVPR.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Bengio, Y., 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or-1. arXiv preprint arXiv:1602.02830 .

Dauphin, Y.N., Fan, A., Auli, M., Grangier, D., 2017. Language modeling with gated convolutional networks, in: ICML.

Demeester, T., Deleu, J., Godin, F., Develder, C., 2018. Predefined sparseness in recurrent sequence models, in: CoNLL.

Denton, E.L., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R., 2014. Exploiting linear structure within convolutional networks for efficient evaluation, in: NeurIPS.

Finnoff, W., Hergert, F., Zimmermann, H.G., 1993. Improving model selection by nonconvergent methods. Neural Networks 6, 771–783.

Gao, X., Zhao, Y., Dudziak, L., Mullins, R., Xu, C.z., 2018. Dynamic channel pruning: Feature boosting and suppression, in: ICLR.

Guo, Y., Yao, A., Chen, Y., 2016. Dynamic network surgery for efficient DNNs, in: NeurIPS.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J., 2016. Eie: efficient inference engine on compressed deep neural network, in: ISCA.

Han, S., Mao, H., Dally, W.J., 2015a. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, in: ICLR.

Han, S., Pool, J., Tran, J., Dally, W., 2015b. Learning both weights and connections for efficient neural network, in: NeurIPS.

Hanson, S.J., Pratt, L.Y., 1989. Comparing biases for minimal network construction with back-propagation, in: NeurIPS.

Hassibi, B., Stork, D.G., 1993. Second order derivatives for network pruning: Optimal brain surgeon, in: NeurIPS.

He, Y., Kang, G., Dong, X., Fu, Y., Yang, Y., 2018. Soft filter pruning for accelerating deep convolutional neural networks, in: IJCAI.

Hinton, G., Vinyals, O., Dean, J., 2015. Distilling the knowledge in a neural network. NeurIPS workshops .

Hornik, K., Stinchcombe, M., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural networks 2, 359–366.

Hu, H., Peng, R., Tai, Y.W., Tang, C.K., 2016. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. arXiv preprint arXiv:1607.03250 .

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization, in: ICLR.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks, in: NeurIPS.

LeCun, Y., 1998. The MNIST database of handwritten digits. http://yann. lecun. com/exdb/mnist/ .

LeCun, Y., Denker, J.S., Solla, S.A., 1990. Optimal brain damage, in: NeurIPS.

Li, G., Qian, C., Jiang, C., Lu, X., Tang, K., 2018. Optimization based layer-wise magnitude-based pruning for DNN compression., in: IJCAI.

Lin, J., Rao, Y., Lu, J., Zhou, J., 2017. Runtime neural pruning, in: NeurIPS.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., Zhang, C., 2017. Learning efficient convolutional networks through network slimming, in: ICCV.

Louizos, C., Welling, M., Kingma, D.P., 2017. Learning sparse neural networks through *l*_0 regularization, in: ICLR.

Marcus, M., Santorini, B., Marcinkiewicz, M.A., 1993. Building a large annotated corpus of English: The penn treebank .

Merity, S., Xiong, C., Bradbury, J., Socher, R., 2016. Pointer sentinel mixture models, in: ICLR.

Molchanov, D., Ashukha, A., Vetrov, D., 2017. Variational dropout sparsifies deep neural networks, in: ICML.

Narang, S., Elsen, E., Diamos, G., Sengupta, S., 2017a. Exploring sparsity in recurrent neural networks. arXiv preprint arXiv:1704.05119 .

Narang, S., Undersander, E., Diamos, G., 2017b. Block-sparse recurrent neural networks. arXiv preprint arXiv:1711.02782 .

Reed, R., 1993. Pruning algorithms – A survey. IEEE Transactions on Neural Networks 4, 740–747.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., Dean, J., 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, in: ICLR.

Sietsma, Dow, 1988. Neural net pruning-why and how, in: ICNN.

Srinivas, S., Babu, R.V., 2015. Learning neural network architectures using backpropagation, in: ICLR.

Van Keirsbilck, M., Keller, A., Yang, X., 2019. Rethinking full connectivity in recurrent neural networks. arXiv preprint arXiv:1905.12340 .

Varma, G., Kothapalli, K., et al., 2019. Dynamic block sparse reparameterization of convolutional neural networks, in: ICCV Workshops.

Voita, E., Talbot, D., Moiseev, F., Sennrich, R., Titov, I., 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned, in: ACL.

Vooturi, D.T., Mudigree, D., Avancha, S., 2018. Hierarchical block sparse neural networks. arXiv preprint arXiv:1808.03420 .

Wen, W., He, Y., Rajbhandari, S., Zhang, M., Wang, W., Liu, F., Hu, B., Chen, Y., Li, H., 2017. Learning intrinsic sparse structures within long short-term memory, in: ICLR.

Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H., 2016. Learning structured sparsity in deep neural networks, in: NeurIPS.

Whitley, D., 1990. The evolution of connectivity: Pruning neural networks using genetic algorithms, in: IJCNN.

Wu, Z., Nagarajan, T., Kumar, A., Rennie, S., Davis, L.S., Grauman, K., Feris, R., 2018. Blockdrop: Dynamic inference paths in residual networks, in: CVPR.

Zaremba, W., Sutskever, I., Vinyals, O., 2014. Recurrent neural network regularization. arXiv preprint arXiv:1409.2329 .

Zhu, M., Gupta, S., 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv preprint arXiv:1710.01878 .

Zmora, N., Jacob, G., Novik, G., 2019. Neural network distiller. doi:`10.5281/zenodo.3268730`.

[1] https://github.com/hadifar/dynamic-sparseness