Compression of Deep Learning Models for Text: A Survey

Manish Gupta and Puneet Agarwal

Abstract—In recent years, the fields of natural language processing (NLP) and information retrieval (IR) have made tremendous progress thanks to deep learning models like Recurrent Neural Networks (RNNs), Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTMs) networks, and Transformer [1] based models like Bidirectional Encoder Representations from Transformers (BERT) [2]. But these models are humongous in size. On the other hand, real world applications demand small model size, low response times and low computational power wattage. In this survey, we discuss six different types of methods (pruning, quantization, knowledge distillation, parameter sharing, tensor decomposition, and Linear Transformer based methods) for compression of such models to enable their deployment in real industry NLP projects. Given the critical need of building applications with efficient and small models, and the large amount of recently published work in this area, we believe that this survey organizes the plethora of work done by the 'deep learning for NLP' community in the past few years and presents it as a coherent story.

Index Terms—Model compression, Deep Learning, Pruning, Quantization, Knowledge Distillation, Parameter Sharing, Tensor Factorization, Linear Transformers.

1 Introduction

Deep learning models have revolutionized multiple fields of information systems including natural language processing (NLP), computer vision, and speech analysis. While they have enabled multiple tasks to attain very high accuracy values, model sizes and prediction latencies have increased significantly as well. Specific to text, Recurrent neural networks (RNNs), Gated Recurrent Units (GRUs) and long short term memory (LSTM) networks have been used for quite some time for various natural language processing (NLP) tasks. These models are large especially because of the input and output embedding parameters.

In the past three years, the field of NLP has made significant progress as is evident from the GLUE [3] and SuperGLUE [4] leaderboards[1,2]. Transformer [1] based models like Bidirectional Encoder Representations from Transformers (BERT) [2], Generative Pre-training Transformer (GPT-2) [5], Multi-task Deep Neural Network (MT-DNN) [6], Extra-Long Network (XLNet) [7], MegatronLM [8], Text-to-text transfer transformer (T5) [9], T-NLG [10] and GShard [11] have been major contributors to this success. But these models are humongous in size: BERT (340M parameters), GPT-2 (1.5B parameters), MegatronLM (8.3B parameters), T5 (11B parameters), T-NLG (17B parameters) and GShard (600B parameters). Bianco et al. [12] and Sanh et al. [13] provide a great overview of the sizes of recent deep learning models in computer vision and NLP respectively.

Deployment of such gigantic models is difficult even on high-end servers. Indeed a large number of real world applications run on machines with resource constrained environments, for example, edge devices, sensors and mobile phones. Edge devices could include offline medical equipment, and modules on drones, robots, glasses, etc. Often times, besides desiring a small model size, low response times are critical. For example, applications like driverless cars or apps to aid the blind require processing speeds of around 30 frames per second. Similarly, search engines need to be able to process billions of queries per day. Overall, the following factors motivate us to study compression of deep learning models.

- Memory (RAM) usage
- Prediction latency
- Power dissipation
- Inference on resource constrained devices
- Ease of training/finetuning
- Ease of deployment and update
- Ease of distributed training

Large networks do not fit in on-chip storage and hence require the more costly external DRAM accesses. Running a 1 billion connection neural network, for example, at 30 frames per second would require (30Hz)(1G)(640pJ) = 19.2W just for DRAM access – well beyond the power envelope of a typical mobile device. This implies that a mobile phone running such an app could suffer from fast draining of the battery, leading to overheating of the phone. Han et al. [14] discuss details of power dissipation for deep learning models. Another option to avoid large RAM, high prediction times and high power dissipation, is to run such massive deep learning models on cloud servers. But for many real world applications, it is desirable to run them on local client devices to avoid network delay, to guard user privacy and to avoid power dissipation in terms of input/output data communication.

- *Manish Gupta (gmanish@microsoft.com) works as a Principal Applied Scientist at Microsoft, India.*
- *Puneet Agarwal (punagr@microsoft.com) works as a Principal Engineering Manager at Microsoft, India.*

Manuscript received Aug, 2020; revised XXXX.

1. https://gluebenchmark.com/leaderboard
2. https://super.gluebenchmark.com/leaderboard

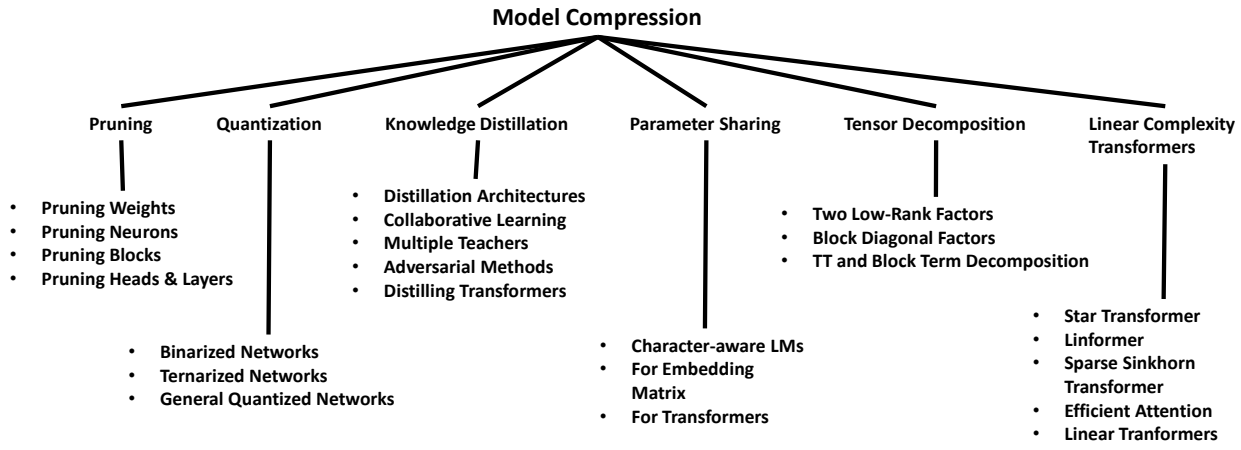

Fig. 1. Overview of Model Compression Methods for Text

Small models can indeed also lead to low prediction latencies. For example, Diamos et al. [15] showed that for small models, one can cache the RNN weights in on-chip memory such as caches, block RAM, or register files across multiple timesteps. This could lead to as much as 146x speedup if the entire RNN layer can be stored in registers rather than the GPU DRAM of an NVIDIA TitanX GPU.

Finally, it is easier to perform software development, deployment and updates with smaller models. Large models are difficult to handle. For example, it is impossible to fine-tune pretrained BERT-large model on a GPU with 12-16 GB RAM. This poses a large barrier of entry for communities without the resources to purchase several large Graphic Processing Units (GPUs). For large models, tuning various configuration parameters needs lots of resources. Smaller models lead to improved speed of learning and allow for more hyper-parameter configurations to be evaluated. Mobile-first companies dislike large apps. App stores are very sensitive to the size of the binary files. For example, iPhone App Store has the restriction "apps above 150 MB will not download until you connect to Wi-Fi". Smaller models are easier to use and deploy in real world systems. Large models need multiple server nodes. On the other hand, multiple instances of smaller models can be run simultaneously on the same machine leading to higher QPS (queries per second). Lastly, smaller models also decrease the communication overhead of distributed training of the models.

Fortunately, there is a large amount of redundancy among the weights of these large neural networks. A small subset of the weights are sufficient to reconstruct the entire network. Denil et al. [16] showed that by simply using $\sim$5% of the weights, it is possible to predict the remaining weights without any drop in accuracy. This observation led to a large number of research efforts across multiple communities on compression of large deep learning models. In this survey, we aim to systematically explore this large body of literature in the NLP community by organizing it into various categories. Figure 1 shows a broad organization of model compression methods for text. In this survey we do not focus on specific methods proposed in other communities like vision and speech only and which make use of image/audio specific architectures and hence cannot be applied to text. Also, we do not discuss methods on hardware optimizations to reduce latency. While there are other surveys in the broad area of model compression [17], [18], they are either old or focus on computer vision related problems.

2 Model Compression Methods: Overview

In this survey, we discuss compression methods using pruning, quantization, knowledge distillation, parameter sharing, tensor decomposition and linear Transformers.

The most obvious way to reduce model size is to sparsify weight matrices. Pruning methods differ based on what is pruned and the actual logic used to prune. Given a matrix, one can prune (1) some weight entries, (2) rows or columns (i.e., neurons), (3) weight blocks, (4) attention heads (in case of Transformer based methods), (5) layers or a combination of the structures. How to decide which weights/neurons/blocks/heads to prune? Should you prune large networks or build small networks? Should you do one-shot pruning versus iterative/gradual pruning? How does regularization help when pruning? We discuss these aspects of pruning based methods in Section 3.

Besides removing the weights themselves, another way to reduce the size of weight matrices is to reduce the number of bits needed to to represent each weight. Typically weights are stored as 32-bit double values. In an extreme case, weights can be quantized to two values (binary 1 bit). But other popular ways include quantization to three values (ternary) or multiple bits. Quantization can be uniform vs non-uniform. Quantization methods can be deterministic or stochastic. Quantization can be performed in a loss-aware or unaware manner. Quantization bin ranges can be trained versus tuned. Finally, the level of quantization needs to be different across layers of RNNs, LSTMs or Transformers to attain a favorable model size versus accuracy tradeoff. We discuss these aspects of quantization based methods in Section 4.

A third way of doing model compression is using knowledge distillation (also broadly known as student teacher networks). In such methods, the idea is to first train a deep teacher model using labeled data, and then transfer "dark knowledge" from teacher to train a shallow student network. Thus, the student model is trained to mimic a pre-trained, larger teacher. After the student is trained, it is

deployed while the teacher network can be thrown. Distillation methods vary based on (1) different types of teacher model, (2) different types of loss function like squared error between the logits of the models, KL divergence between the predictive distributions, or some other measure of agreement between the model predictions, (3) different choices for what dataset the student model trains on (a large unlabeled dataset, a held-out data set, or the original training set), (4) Mimic what – teachers class probabilities, teachers feature representation, and (5) learn from whom – teacher, teacher assistant, or other fellow students. We discuss these aspects of knowledge distillation based methods in Section 5.

Another way that reduces overall weights is to find weights which are similar and use a single number to represent them. Broadly, the method is called parameter sharing. Methods differ depending on (1) which parameters are shared, (2) technique used to share parameters, and (3) the level at which sharing is performed. We discuss these aspects of parameter sharing based methods in Section 6.

Yet another way to avoid large matrices is to approximate them using a combination of smaller matrices. Such tensor decomposition methods for model compression factorize large tensors into multiple smaller components. Methods differ (1) in the type of factorization technique, (2) matrices being factorized, and (3) the property of weight matrix being exploited. We discuss these aspects of tensor decomposition methods in Section 7.

In Transformer based models, besides the model size, latency is a concern because of quadratic complexity in terms of the input sequence size. Hence, very recently, there have been several efforts on designing Transformers with linear complexity. Such methods use various techniques for enforcing linearity – the broad idea is to compute a transformed representation for every token using attention over a fixed small number of other tokens. Methods differ in terms of defining the set of other tokens to be used to compute a transformed representation for the current token. We discuss such methods in Section 8.

3 Pruning

The first proposed methods for model compression were based on pruning. One can prune away weights from a weight matrix depending on various criteria (e.g., prune away low magnitude weights). Such unstructured pruning methods lead to sparse matrices and need special sparse matrix manipulation libraries at inference time. Hence, various structured pruning methods have also been proposed which aim to prune away structures like neurons, weight matrix blocks, attention heads or layers. In this section, we provide an organized overview of such methods. Fig. 2 provides a broad overview of various pruning styles.

In pruning, the main idea is to grow a large model and then prune away weights to end up with a much smaller but effective model. This is inspired by the following biological observation. Trillions of synapses are generated in the human brain during the first few months of birth. At one year old, synapse count peaks at 1000 trillion. And then natural pruning begins to occur. A ten year old child has nearly 500 trillion synapses. This 'pruning' mechanism removes redundant connections in the brain [19].

One natural question is should you prune large networks or build small dense networks? Pruning involves extra processing plus sparse matrices need special handling. Can we avoid it by training smaller models? Zhu et al. [20] experimented with models of various sizes with/ without pruning of stacked LSTMs models for language modeling, and seq2seq models for NMT. They found that large-sparse models consistently outperform small-dense models and achieve up to 10x reduction in number of non-zero parameters with minimal loss in accuracy.

3.1 Pruning Weights

3.1.1 Hessian based Methods

In their seminal work (Optimal Brain Damage or OBD) on proposing weight pruning as a method for model compression, LeCun et al. [21] defined saliency of a weight as change in the objective function E caused by deleting that parameter. Using Taylor series and making multiple assumptions, they propose that $\frac{1}{2}\sum_i h_{ii}\delta u_i^2$ can be used as a measure of saliency of weight u_i where $h_i = \frac{\partial^2 E}{\partial u_i \partial u_j}$. Weights with low saliency can be pruned and the pruned network can be retrained to adjust the remaining weights. The procedure for computation of the diagonal of the Hessian is very similar to the back-propagation algorithm used for computing the first derivatives. Hence, computing the diagonal of the Hessian is of the same order of complexity as computing the gradient.

OBD ignores cross terms in the Hessian matrix. But on most real datasets, Hessian is strongly non-diagonal. Hence, to avoid pruning of important weights, Hassibi et al. [22] proposed a method called Optimal Brain Surgeon (OBS) which considers cross terms as well. Using a similar derivative of E wrt weight w_i, saliency of the weight is $L_i = \frac{1}{2}\frac{w_i^2}{[H^{-1}]_{ii}}$. Computing H^{-1} is difficult. Hence, they provide a faster recursion relation for calculating H^{-1} from training data and structural information of the network. Also, unlike other methods (like OBD or magnitude pruning), OBS does not demand (typically slow) retraining after the pruning of a weight. In every step, we delete w_i with min L_i and update all weights ($\delta w = -\frac{w_i}{[H^{-1}]_{ii}} H^{-1} e_i$) where e_i is the unit vector in weight space corresponding to (scalar) weight w_i. Unfortunately, these methods (OBD and OBS) are computationally prohibitive as second derivative computations are expensive.

3.1.2 Magnitude Pruning Methods

A more computationally feasible method for pruning connections and relearning weights based solely on the magnitude of the original weights is to simply prune away weights with small magnitudes. The idea was first proposed by Han et al. [23]. Pruning away low magnitude weights makes the matrix sparse. Sparse matrices can be stored in Compressed Sparse Row/Column (CSR/CSC) formats. Further space can be saved by storing the index difference instead of the absolute position, and encode this difference using small fixed number of bits. See et al. [24] experimented with these pruning methods on encoder-decoder LSTM NMT (neural machine translation) models. They perform magnitude

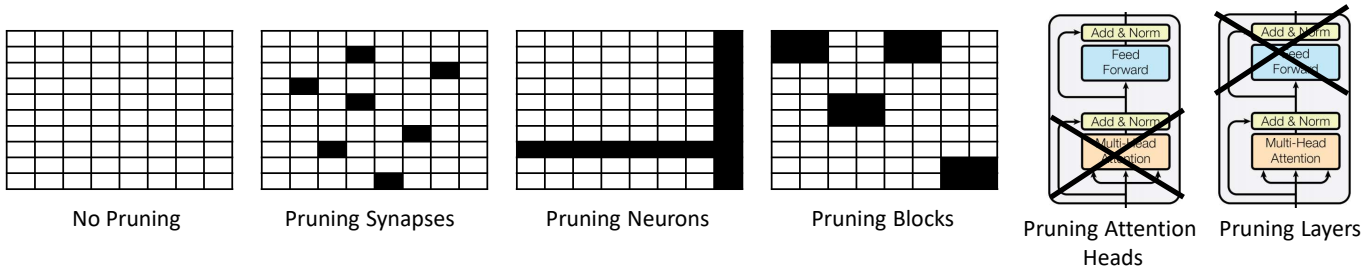

Fig. 2. Different Types of Pruning

pruning on all weight matrices of a 4-layer LSTM. They find that higher layers, attention and softmax weights are the most important, while lower layers and the embedding weights hold a lot of redundancy. At the lower layers the parameters for the input are most crucial, but at higher layers the parameters for the gates also become important. These methods typically have a target pruning percent as a hyper-parameter and pruning is either performed statically (after training the full model) or dynamically (while training itself). Retraining the sparse pruned network helps in improving accuracy.

In a typical encoder-decoder LSTM model, there are these weight classes: source embedding weights, target embedding weights, source layer weights, target layer weights, attention weights and softmax weights. An important consideration related to magnitude pruning is how do we distribute the pruning over these different weight classes of a model, given a target x% pruning? Three ways suggested by See et al. [24] include class-blind, class-uniform and class-distribution. In the class-blind way, we take all parameters, sort them by magnitude and prune the x% with smallest magnitude, regardless of the weight class. So some classes are pruned proportionally more than others. In the class-uniform way, Within each class, we sort the weights by magnitude and prune the x% with smallest magnitude. So all classes have exactly x% of their parameters pruned. In the class-distribution scheme, for each class c, weights with magnitude less than $\lambda\sigma_c$ are pruned. Here, σ_c is the standard deviation of that class and λ is a universal parameter chosen such that in total, x% of all parameters are pruned. Class-blind pruning is the simplest and adheres to the principle that pruning weights (or equivalently, setting them to zero) is least damaging when those weights are small, regardless of their locations in the architecture. Class-uniform pruning and class-distribution pruning both seek to prune proportionally within each weight class, either absolutely, or relative to the standard deviation of that class. They observe that class-blind pruning outperforms both other schemes.

3.1.3 *Iterative Magnitude Pruning Methods*

Typically, it has been seen that rather than pruning in one-shot, it is a good idea to prune gradually over epochs. This way of pruning is called iterative or gradual pruning. For starting proportion x% and ending proportion y%, iterative magnitude pruning procedure prunes x% of each of the weights, does re-training, and then prunes $(y-x)/T$% of the weights every K iterations. T is the number of times, pruning is done. Sometimes, pruning is started after few warmup iterations have already been performed. Magnitude pruning has been seen to be very effective with regularization (L_1/L_2) while training. Dropouts also help in effective pruning. In some pruning methods, a weight once pruned cannot be a part of the network in future iterations. On the other hand, other methods do not modify the gradients of a pruned weight in the back-propagation step. In that case, it is possible for the updates of a pruned weight to be larger than the threshold of that layer, and then the weight will be involved in the forward pass again. Also, in every pruning iteration, we could either use a fixed threshold [25] or monotonically increase it [26].

In case of gradual pruning [26], where pruning threshold ϵ is monotonically increased, ϵ is determined as follows in every iteration i. Let f be the number of iterations after which ϵ is updated. Also, after a few warmup iterations, weights are sorted using absolute values and we pick the weight corresponding to the 90^{th} percentile as q. Pruning threshold ϵ is increased in two stages. In the first stage (which starts at start iteration s and continues until ramp iteration r, we use θ as the initial slope to prune weights. In the second stage (which starts at ramp iteration r and continues until end iteration e), we use ϕ as the ramp slope to change the rate of pruning. Typically, ϕ is set to 1.5θ where θ is calculated as $\theta = \frac{2qf}{2(r-s)+3(e-r)}$. Thus, from iteration s to r, we set $\epsilon = \theta(i-s+1)/f$; while from iterations $r+1$ to e, we set $\epsilon = (\theta(r-s+1)+\phi(i-r+1))/f$. Typically when pruning, biases are not pruned since they are much fewer in number. Overall, RNN/LSTM model size can be reduced by 90% and speed-up is around 2x to 7x using gradual pruning with no deterioration in accuracy. Also, layers closer to input are pruned more aggressively compared to the final layers.

Another way of performing iterative pruning is to set a pruning target per iteration [20]. In this scheme, we start with an initial sparsity value s_0. To achieve a final sparsity value of s_f after n pruning steps with pruning frequency f, pruning target in iteration i can be computed as $s_i = s_f + (s_0 - s_f)(1 - \frac{i}{nf})^3$. Thus, the sparsity of the network is gradually increased while allowing the network training steps to recover from any pruning-induced loss in accuracy. We prune the network rapidly in the initial phase when the redundant connections are abundant and gradually reduce the number of weights being pruned each time as there are fewer and fewer weights remaining in the network.

Cheong et al. [27] found that iterative pruning leads to poor results when pruning Transformer models like BERT. Guo et al. [28] found that there are two problems with pruning especially when done with regularization. (1) The larger weights w_j are penalized more heavily than smaller weights w_i in L_1 regularization, which violates the original

intention of weight pruning, "removing the unimportant connections". (2) Direct optimization of a regularization penalty term causes divergence from the original loss function and has negative effect on the effectiveness of gradient-based update. They propose to perform reweighted L_1 minimization where $\alpha_i > 0$ are inversely proportional to magnitude of corresponding weights $|w_i|$. Thus, they solve $\min_w f(w) + \gamma \sum_i \alpha_i |w_i|$ where $f(w)$ is the original loss function for the network. This optimization is solved using a reweighted proximal pruning (RPP) method (which depends on proximal operators). RPP decouples the goals of high sparsity from minimizing loss, and hence leads to improved accuracy even with high levels of pruning for BERT.

3.1.4 Iterative Magnitude Pruning and Densification

Further, the effectiveness of pruning can be improved by performing pruning and densification [29], [30] alternately across multiple iterations. There are two ways of doing this. In the first method [29], in each iteration, either pruning is performed or densification. The sparse training regularizes the model, and the dense training restores the pruned weights, increasing the model capacity without overfitting. Sparsification helps the optimizer escape saddle points, and leads to regularizd training which converges to a significantly better minima. In the second method [30], in every iteration some dormant weights can reappear in the network while other active ones can get pruned out. A dormant $w \in W$ is activated iff $|w.grad|$ is larger than the $(100\alpha)^{th}$ percentile of all elements in $|W.grad|$. A $w \in W$ is removed iff $|w|$ is smaller than the $(100\beta)^{th}$ percentile of all elements in $|W|$. α and β refer to growth ratio, and pruning ratio, respectively.

3.2 Pruning Neurons

It is difficult to implement unstructured pruning practically since, at inference time, special support is needed for matrix multiplication in the sparse space. Pruning away neurons leads to removal of a row or a column from a weight matrix, thereby avoiding sparse matrix handling. However, compared to pruning weights, pruning neurons is less flexible since we need to find entire rows/columns for deletion. In this section, we discuss ways of determining neurons that can be pruned.

3.2.1 Removing Low Importance Neurons

He et al. [31] proposed three node importance functions to determine importance score for neurons.

- Entropy: Let a_i (d_i) be the #instances with node output $>$ (or $\leq$) 0.5 for binary classification with a sigmoid activation. Then $\text{Entropy}(i) = \frac{d_i}{a_i+d_i} \log_2 \frac{d_i}{a_i+d_i} + \frac{a_i}{a_i+d_i} \log_2 \frac{a_i}{a_i+d_i}$. The intuition is that if one node's outputs are almost identical on all training data, these outputs do not generate variations to later layers and consequently the node may not be useful.
- Output-weights Norm (onorm): average L_1-norm of the weights of its outgoing links.
- Input-weights norm (inorm): average L_1-norm of the weights of its incoming links.

All the neurons are sorted by their scores and nodes with less importance values are removed. In most cases, onorm has been found to be the best among these importance functions.

Special regularizers have also been proposed to force neurons to push either all incoming or outgoing connection weights towards zero [32], [33]. Specifically, for handling incoming connections, the following two regularizers are popular: (1) L_2 norm on weight matrix W defined as $\sum_i ||W_{i:}||_2 = \sum_i (\sum_j W_{ij}^2)^{1/2}$. This puts equal pressure on each row, but within each row, the larger values contribute more, and therefore there is more pressure on larger values towards zero. (2) L_∞ norm on weight matrix W defined as $\sum_i ||W_{i:}||_\infty = \sum_i max_j |W_{ij}|$. This puts equal pressure on each row, but within each row, only the maximum value (or values) matter, and therefore the pressure towards zero is entirely on the maximum value(s). Similar regularizers can easily be defined for outgoing connections as well.

3.2.2 Removing Redundant Neurons

Consider a simple network with one hidden layer with n neurons. Thus, the output can be computed as $z = a_1 h(W_1^T X) + a_2 h(W_2^T X) + ... + a_n h(W_n^T X)$ where a_i and W_i indicate weights. In case $W_1 == W_2$, $h(w_1^T X) = h(w_2^T X)$. Thus, we can compute output as $z = (a_1 + a_2) h(W_1^T X) + 0.h(W_2^T X) + ... + a_n h(W_n^T X)$. In general, whenever two weight sets (W_1, W_2) are equal, one of them can effectively be removed. This should be done with a surgery step, i.e., we need to alter the co-efficient a_1 to $a_1 + a_2$. Of course, for many pairs of weight sets (i.e., neurons), W_1 and W_2 are not exactly the same. Hence, Srinivas et al. [34] proposed this 3 step method for redundant neuron identification and removal. (1) Compute saliency s_{ij} for all possible neuron pairs (i, j) as $s_{ij} = \langle a_j^2 \rangle ||\epsilon_{ij}||_2^2$ where $\langle a_j^2 \rangle$ denotes the average of the quantity over all output neurons. Let S be the matrix with all s_{ij} values. (2) Pick the indices (i', j') corresponding to the minimum s_{ij}. Delete the j' neuron, and update $a_i' \leftarrow a_i' + a_j'$. (3) Update S by removing the j'^{th} column and row, and updating the i'^{th} column (to account for the updated a_i').

3.3 Pruning Blocks

In weight pruning, irregularity of sparse matrices limits the maximum performance and energy efficiency achievable on hardware accelerators. Pruning neurons avoids sparse matrix issues but is limited in term of overall pruning possible. Block-sparse formats store blocks contiguously in memory reducing irregular memory accesses. If the maximum magnitude weight of a block is below the current threshold, we set all the weights in that block to zeros. Similar to iterative weight pruning, block pruning [35] can also be done iteratively using a monotonically growing threshold. Any blocks that had been zeroed out are held at zero even after pruning has ended resulting in a sparse model at the end of training. Just like weight pruning (as discussed in Section 3.1), the start slope θ and ramp slope ϕ determine the rate at which the threshold increases. For block pruning, we need to modify the start slope to account for the number of elements in a block (N_b). Thus, the start slope for block pruning is typically set to $\theta_b = \theta \times \sqrt[4]{N_b}$. Further, to enable

effective removal of blocks, Narang et al. [35] propose group Lasso regularization method. Group lasso is a type of weight regularization that works on groups of weights and can zero out all the weights in a group. For each block, we add a loss term proportional to the L_2 norm of the block. Thus, we optimize for $\min_w f(w) + \lambda_g \sum_{g=1}^{G} ||w^{(g)}||_2$. When we combine group lasso with block pruning, group lasso guides the selection of blocks to prune. Group lasso regularization is applied to coincide with the pruning schedule, i.e., we turn off regularization when the pruning schedule ends. Typically, inducing block sparsity with 4x4 blocks in vanilla RNNs and GRUs works well, compared to larger block sizes. Larger blocks require lower sparsity to maintain similar accuracy.

Unfortunately, it becomes challenging to maintain the same model accuracy when block sparsity is applied. Also, block sizes (i.e., pruning granularity) are application-sensitive, making it another hyper-parameter to tune. To avoid these problems, Cao et al. [36] proposed a new method called Bank-Balanced Sparsity (BBS). BBS splits each weight matrix row into multiple equal-sized banks, and adopts fine-grained pruning to each bank independently to obtain identical sparsity among banks. Each bank has the same number of non-zero values. For example, retaining top two weights in each bank of size 4 implies a sparsity of 50%. We apply the BBS pruning method iteratively to a pre-trained network, and fine-tune the network after each pruning iteration to restore the model accuracy. BBS achieves almost the same model accuracy as unstructured sparsity and significantly outperforms block sparsity when pruning weights at the same sparsity level. BBS is also amenable to FPGA (Field Programmable Gate Arrays) acceleration because it inherently provides a balanced matrix partitioning for parallel computing.

3.4 Pruning Heads and Layers

Besides neurons and blocks, for Transformer based models, structured pruning can also be applied to attention heads and entire layers.

3.4.1 *Pruning Attention Heads*

BERT BASE model consists of 12 layers each with 12 attention heads. Similarly, a typical NMT encoder-decoder Transformer with 6 layers each for encoder as well as decoder contains 16 attention heads per layer. Michel et al. [37] found that majority of attention heads can be removed without deviating too much from the original score. Surprisingly, in some cases removing an attention head results in an increase in accuracy. When these heads are removed individually, only 8 (out of 96) heads in 6-layer WMT NMT Transformer (16 heads/layer) cause a statistically significant change in performance when they are removed from the model, half of which actually result in a higher BLEU score. For most layers, one head is indeed sufficient at test time, even though the network was trained with 12 (BERT) or 16 (WMT Transformer) attention heads. One can also do iterative pruning of multiple heads (rather than just one at a time) across layers. For iterative pruning, head importance score is defined using the expected sensitivity of the model to the mask variables ξ_h as follows.

$$I_h = E_{x \sim X} \left| \frac{\partial L(x)}{\partial \xi_h} \right| = E_{x \sim X} \left| Att_h(x)^T \frac{\partial L(x)}{\partial Att_h(x)} \right| \quad (1)$$

where X is the data distribution, $L(x)$ is the loss on sample x, and $Att_h(x)$ is the output of the attention head h for instance x, . Intuitively, if I_h has a high value then changing ξ_h is liable to have a large effect on the model. Hence, in every iteration heads with low I_h values are pruned out. Michel et al. [37] observed that pruning up to 20% and 40% of heads from NMT and BERT models respectively, did not lead to any noticeable negative impact on accuracy.

Voita et al. [38] used two other head importance scores to prune attention heads from the NMT model. The two scoring methods were: (1) Layer-wise relevance propagation (LRP) [39]. LRP is a method for computing the relative contribution of neurons at one point in a network to neurons at another. (2) "confidence" of a head which is computed as the average of its maximum attention weight excluding the end of sentence symbol, where the average is taken over tokens in a set of sentences used for evaluation. For pruning the heads, they propose a method based on stochastic gates and a differentiable relaxation of the L_0 penalty. L_0 norm equals the number of non-zero components and pushes the model to switch off less important heads. They find that only a small subset of heads are important for translation. On the English-Russian WMT dataset, pruning 38 out of 48 encoder heads results in a drop of only 0.15 BLEU.

3.4.2 *Pruning Layers*

Note that dropping attention heads does not reduce runtime as they are usually computed in parallel. While one can prune weights, neurons or attention heads, how can we design a scheme to prune away layers? The LayerDrop idea proposed in [40] is inspired by DropConnect. DropConnect randomly drops weights while training on a batch. LayerDrop does structured dropout: it drops groups of weights, heads, feed forward network (FFN) matrices, or layers. The layers to be pruned can be decided using one of these ways: (1) Every Other: Prune every other layer (with rate p), e.g., every 3^{rd} layer in a 12-layer BERT model. (2) Search on Validation: Search for a set of layers to be pruned by checking their impact on a validation set. This entails trying various combinations. (3) Data Driven Pruning: Learn the drop rate p_d of each layer in a data driven manner. Given a target drop rate p, we learn an individual drop rate p_d for the layer at depth d such that the average rate over layers is equal to p. At inference time, we forward only the fixed top-k highest scoring layers based on the softmax output. Across the three methods, "Every Other" strategy works surprisingly well across many tasks and configurations. "Search on Validation" and "Data Driven Pruning" only offer marginal gains.

3.4.3 *Pruning General Structures*

Lastly, Prasanna et al. [41] experiment with pruning both the FFN layers as well as attention heads in a BERT network. Just like [37], they assign a mask variable to each of these structures. To decide which structures to prune, we look at the expected sensitivity of the model to the mask variables. High sensitivity implies large impact on the model output

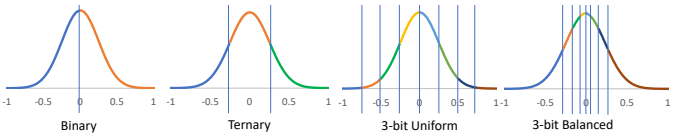

Fig. 3. Different Types of Quantization

and hence corresponding structures should be retained. They find that it is possible to find a subnetwork of elements that achieves performance comparable with that of the full model, and similarly-sized subnetworks sampled from the rest of the model perform worse.

3.5 Summary

To summarize, pruning has been the most popular method for model compression. Pruning methods can be unstructured (prune weights) or structured (prune neurons, blocks, attention heads, layers). While weight pruning theoretically leads to pruning to a large extent, practical implementation of sparse data structures is difficult. Pruning and regularization need to be done together carefully. Also, it is critical to define the importance functions for various structures carefully. Among weight pruning methods, while iterative magnitude pruning with regularization works well for RNNs and LSTMs, RPP performs better for Transformer based models. Pruning blocks using BBS is better than pruning blocks or pruning neurons. For Transformer models, pruning just the heads may not provide latency improvements, but dropping a combination of attention heads and layers is better.

4 Quantization

While pruning saves on the model size by removing weights, quantization aims to reduce the number of bits needed to store weights. Most computer architectures use 32 bits to represent weights. However, estimated precision of the brain (hippocampal spine) synapses is around 4.6 bits [42]. Empirical evidence suggests that most quantities in the nervous system (for instance, firing of the neurons) have variability of a few percent due to biological noise, or a precision of 1 in 100 at best [43]. Thus, each decision could depend on $\log_2(100)$=6.64 bits. Thus, we should be able to store weights in our artificial neural networks on average in a space of 4–7 bits. Given this motivation, various methods have been proposed which perform 1-bit (or binary quantization), ternary quantization, and general quantization exploring the spectrum between 3 and 32 bits. We discuss such methods in this section. Fig. 3 provides a broad overview of various quantization styles.

4.1 Binarized Networks

Quantizing weights to 1 bit provides a compression of 32x but leads to a significant drop in accuracy across many tasks. However, in a hybrid quantization scheme, such binary quantization can be very helpful for some layers in a network. Binarization can be done using deterministic methods or could be stochastic in nature. Also, while naïve binarization has a very simple way of fixing the binary boundary threshold, one could perform a complex loss aware binarization as well. We discuss these variants of binarization in this section.

4.1.1 Deterministic Binarization

Simplest way of binary quantization is to set the weight as 1 for non-negative weights, and to -1 for negative weights. This leads to 32x compression. Also, the matrix multiplication for binary matrices is ∼7x faster [44] leading to faster model inference. In the forward pass, binary networks drastically reduce memory size and accesses, and replace most arithmetic operations with bit-wise operations, which leads to great increases of power efficiency. Also, in the simplest version, binarization can be performed in a static manner, i.e., after the training is done. However, this method leads to large loss in accuracy.

A variant of this simple method is to set the weight to a constant c_1 for non-negative weights, and to another constant c_2 for negative weights. Binary Scheme (BS)-Fixed method [45] stores the original weights and during the forward pass replaces the values with a masked value of c_1 or c_2, where c_1 and c_2 are fixed and chosen with hyper-parameter tuning. Full precision weights are used during training. At the end of training, the weights are replaced with the index of its masked value. Choosing the values of c_1 and c_2 can be difficult and time-consuming in BS-Fixed. Thus, in the BS-flexible method [27], we initialize c_1 and c_2 using KMeans with two centroids over the weights, and then update c_1 and c_2 using back-propagation. Also, in the BS-Flexible method, weights are quantized as follows.

$$w_b = \begin{cases} c_1 & \text{if } w \geq (c_1 + c_2)/2 \\ c_2 & \text{if } w < (c_1 + c_2)/2 \end{cases} \tag{2}$$

Note that w is the original weight value while w_b is the binarized weight value. These changes eliminate the need for hyper-parameter tuning.

4.1.2 Stochastic Binarization

Stochastic [46] binarization is performed as follows.

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{3}$$

Here, $\sigma(w) = clip(\frac{w+1}{2}, 0, 1) = max(0, min(1, \frac{w+1}{2}))$, also called as hard sigmoid. We only binarize the weights during the forward and backward propagations but not during the parameter update. Keeping good precision weights during the updates is necessary for Stochastic Gradient Descent (SGD). This is possible using something called as "Straight Through Estimator (STE) trick" [47]. As per STE, as the quantized value is an approximation of the original value, we can substitute the gradient with respect to the quantized value for the gradient of original value. The trick allows the inclusion of quantization into the computation graph of back-propagation and allows QNNs to represent parameters, activations and gradients with low bitwidth numbers. For test-time inference, there are 3 options using such a quantization method: (1) Use the resulting binary weights w_b (this makes most sense with the deterministic binarization). (2) In the stochastic case, many different networks can be sampled by sampling a w_b for each weight. The ensemble output of these networks can then be obtained by averaging the outputs from individual networks. (3) Use original weights. But this does not reduce model size.

Besides this, there have been further efforts that make train/test faster but do not reduce model size. For example,

Lin et al. [48] convert multiplications in the backward pass into bit-shifts by restricting the activations to be power-of-two integers. Hubara et al. [49] binarize weights and activations, at the inference phase and the entire training phase of a deep network.

4.1.3 Loss Aware Binarization

The naïve binary quantization methods divide the real number line into two parts and each part was mapped to a quantized weight value. Can we decide per weight value which of the two weights it should be quantized to? Thus the idea behind Binary Weight Networks (BWN) [50] is to approximate the weight vector $W \in R^n$ using a binary vector $B \in \{+1, -1\}^n$ and a scaling factor $\alpha \in R^+$ such that $W \approx \alpha B$. Thus, we wish to find $\alpha^*, B^* = \text{argmin}_{\alpha,B} ||W - \alpha B||^2$. We can expand and write $||W - \alpha B||^2 = \alpha^2 B^T B - 2\alpha W^T B + W^T W$. Since $B \in +1, -1^n$, $B^T B = n$. Also $W^T W$ is a constant. Thus $B^* = \text{argmax}_B W^B$ such that $B \in +1, -1^n$. This optimization can be solved by simply assigning $B_i = +1$ when $W_i \geq 0$, and $B_i = -1$ otherwise. To compute α^*, we set the derivative of $||W - \alpha B||^2$ wrt α to 0 and get $\alpha^* = \frac{\sum |W_i|}{n}$. Thus, besides the binarized weight matrix, a scaling parameter is also learned in BWN.

To take this idea further, can we learn α and B to minimize the overall network's loss function? Thus, now, the Weight binarization can be formulated as the following optimization problem: $\min_{\hat{w}} loss(\hat{w})$ such that $\hat{w}_l = \alpha_l b_l$, $\alpha_l > 0$, $b_l \in +1, -1^{n_l}$, $l = 1, ..., L$ where L is the number of layers, n_l is the number of weights in layer l. This loss aware binarization [51] problem can be solved using proximal Newton algorithm [52] to find the best α_l and B_l.

4.2 Ternarized Networks

Unfortunately, binary quantization of the recurrent weights in RNNs/LSTMs never worked [53]. When the true value of a weight is near zero, its quantized value is either set to -1 or 1. This results into an artificial increase in the magnitude of the weights and the vanishing/exploding gradients problem becomes more severe. Hence, another popular form of quantization is ternary quantization. Ternary quantization can help achieve a min of 16x compression (up to 32x compression if hardware allows to avoid storing zeros). In this section, we discus different variants of ternary quantization from the simplest ternary connect networks to hybrid ternary networks like HitNets.

4.2.1 Ternary Weight Networks

The simplest method for ternary quantization is ternary connect [48] whose deterministic form is as follows.

$$w_t = \begin{cases} +1 & \text{if } w > 0.5 \\ 0 & \text{if } -0.5 < w \leq 0.5 \\ -1 & \text{if } w \leq -0.5 \end{cases} \quad (4)$$

Note that w is the original weight value while w_t is the ternarized weight value. Like binary connect, ternary connect also eliminates all multiplications in the forward pass. In the stochastic form, assuming original weights have been normalized to be in the range [-1,1], ternary quantization is done as follows.

$$w_t = \begin{cases} +1 & \text{with prob } w \text{ if } w \in (0, 1] \\ 0 & \text{with prob } 1 - w \text{ if } w \in (0, 1] \\ 0 & \text{with prob } 1 + w \text{ if } w \in [-1, 0] \\ -1 & \text{with prob } -w \text{ if } w \in [-1, 0] \end{cases} \quad (5)$$

A slightly related way called as Bernoulli Ternary Quantization where w_t is set to +1 (or -1) with prob p if $w > 0$ (or) < 0, and set to 0 with prob 1-p where $p \sim$Bernoulli($|x|$). Yet another way to set the boundaries for the three ranges is to use Gaussian based ternary weights [54] as follows.

$$w_t = \begin{cases} +1 & \text{if } w > -(\mu + \sigma/2) \\ 0 & \text{if } -(\mu + \sigma/2) < w \leq (\mu + \sigma/2) \\ -1 & \text{if } w \leq -(\mu + \sigma/2) \end{cases} \quad (6)$$

where μ and σ are the mean and standard deviation of the weight matrix being quantized.

4.2.2 Trained Ternary Quantization

Rather than using the rules for ternary quantization as mentioned above, one can learn the boundary ranges or the quantized values for individual weights. One way of learning the right ternary representation per weight value is to minimize the Euclidean distance between full precision weights W and the ternary weights T along with a scaling factor [55]. This can be expressed as the following optimization problem: $\alpha^*, T^* = \text{argmin}_{\alpha,T} ||W - \alpha T||_2^2$ such that $\alpha \geq 0$, $T_i \in -1, 0, 1$, $i = 1, 2, ..., n$. Note that this is equivalent to the BWN method [50]. This does not lead to a closed form solution. Hence, we approximate the solution with threshold-based ternary function.

$$w_t = \begin{cases} +1 & \text{if } w > \Delta \\ 0 & \text{if } -\Delta < w \leq \Delta \\ -1 & \text{if } w \leq -\Delta \end{cases} \quad (7)$$

The approximation works when we set $\Delta^* = \text{argmax}_{\Delta > 0} \frac{1}{|I_\Delta|} (\sum_{i \in I_\Delta} |W_i|)^2$ where I_Δ is the number of weights with magnitude$> \Delta$. Again, this has no straightforward solution, unless we assume that original weights W_i's are generated from uniform or normal distribution. When W_i's are uniformly distributed in $[-a, a]$ and Δ lies in $(0, a]$, the approximated Δ^* is $a/3$, which equals to $\frac{2}{3} E(W)$. When W_i's are generated from normal distributions $N(0, \sigma^2)$, the approximated Δ^* is 0.6σ which equals to $0.75E(|W|)$. Thus, we can use a rule of thumb that $\Delta^* \approx 0.7E(W) = \frac{0.7}{n} \sum_{i=1}^{n} |W_i|$ for fast and easy computation.

Another way to learn the quantization step size Δ in Eq. 7 is to learn in a loss-aware manner [56], i.e., tuning it to minimize the overall network loss. Given a multi-layered network, we need to perform such quantization layer by layer in a greedy manner. We first train the network with full precision weights. We quantize all input data and signals of hidden layers. Next, we start with the weight quantizer between the input layer and the first hidden layer, try several step sizes around the initial step size and measure the output error of the network with the training set. The initial step size is determined using Lloyd-Max algorithm [57]. Choose the step size that minimizes the output error and

quantize the weights. Further, we perform these steps for the next few layers until the output layer. Finally, the quantized neural network is retrained.

Yet another way of training ternary quantization [58] is to quantize weights to one of $-W_l^n$, 0, W_l^p for each layer l, where W_l^n and W_l^p are trainable parameters, learned using back-propagation. First, we normalize the full-precision weights to the range [-1, +1] by dividing each weight by the maximum weight. During SGD, we back propagate the gradient to both W_l^n and W_l^p and to the latent full-precision weights. This makes it possible to adjust the ternary assignment (i.e. which of the three values a weight is assigned). To decide the quantization step size Δ_l for a layer l, two heuristics can be used: (1) set $\Delta_l = t \times max(|w_l|)$ where t is a constant and w_l are the full precision weights in layer l. (2) maintain a constant sparsity r for all layers throughout training. By adjusting the hyper-parameter r we can obtain ternary weight networks with various sparsities.

4.2.3 Hybrid Ternary Quantization

Given various ternary quantization methods proposed so far, one can combine them and use different methods for different layers. Wang et al. [59] found that threshold ternary quantization (TTQ) (Eq. 7) is preferable for weights in an RNN while Bernoulli Ternary Quantization (BTQ) is preferable for activations. This is based on the observation that in an RNN, the distribution of weights follows normal distribution (with different ranges across different weight matrices), while for activations, the range is [0,1] and most of the values are located near to the two poles instead of the middle of the range. In the training phase (where we need to store the full precision weights), ternary quantization of weights only saves 1.4x memory consumption but quantizing both weights and activations can achieve up to 16x memory savings.

The HitNet architecture [59] with this hybrid ternary quantization can be defined using these equations, where i_t, f_t, o_t are the input, forget and output gates; x_t is input at time t; c_t is the cell output; and h_t is the hidden layer output; W_x, W_h, b_x, b_h are weights.

$$\begin{aligned} i_t, f_t, g_t, o_t &= \sigma(TTQ(W_x)x_t + TTQ(b_x) \\ &+ TTQ(W_h)h_{t-1} + TTQ(b_h)) \\ c_t &= f_t \times c_{t-1} + i_t \times g_t \\ h_t &= BTQ(o_t \times \sigma(c_t)) \end{aligned} \tag{8}$$

4.3 General Quantized Networks

So far we discussed methods designed specifically for binary and ternary quantization. Now, we discuss general k-bit quantization methods. We will discuss (1) uniform quantization methods which perform equal width binning, (2) non-uniform methods which are closer to equal frequency binning, (3) loss-aware quantization methods, and (4) methods specifically designed for Transformer models.

4.3.1 Uniform Quantization

Uniform k-bit Quantization simply splits the range of original weights into $2^k - 1$ equal size intervals [44], [50], [60]. If original weights are in range [-1,1], they can be quantized as follows.

$$q_k(x) = 2\left(\frac{round[(2^k-1)(\frac{x+1}{2})]}{2^k-1} - \frac{1}{2}\right) \tag{9}$$

Similarly, if entries are in range [0,1], we could use $q_k(x) = \frac{1}{2^k-1}\lfloor(2^k-1)x + \frac{1}{2}\rfloor$. When the weights in matrix X are not in the range [0,1], we can first scale weights as $\tilde{X} = \frac{X-\beta}{\alpha}$ where $\alpha = max(X) - min(X)$ and $\beta = min(X)$. After quantization, we can apply a reverse transform to approximate the original values. Overall, the quantized result is: $Q(X) = \alpha q_k(\tilde{X}) + \beta$.

Given any quantization function $q_k(x)$, one can use it for quantizing weight matrices of various recurrent models like RNNs, GRUs and LSTMs [53]. Typical inference equations for a GRU can be written as follows.

$$z_t = \sigma(W_z.[h_{t-1}, x_t]); r_t = \sigma(W_r.[h_{t-1}, x_t]) \tag{10}$$

$$\tilde{h_t} = tanh(W.[r_t \times h_{t-1}, x_t]); h_t = (1 - z_t)h_{t-1} + z_t\tilde{h_t} \tag{11}$$

Besides the matrix multiplications needed to compute z_t, r_t and $\tilde{h_t}$, the gate structure of $\tilde{h_t}$ and h_t brings in the need for element-wise multiplication. As $\tilde{h_t}$ and h_t are also the inputs to computations at the next timestamp, and noting that a quantized value multiplied by a quantized value will have a larger bit-width, we need to insert additional quantization steps after element-wise multiplications. Another problem with quantization of GRU structure lies in the different value range of gates. The range of tanh is [-1, 1], which is different from the value range [0, 1] of z_t and r_t. Keeping in mind these observations, the equations for a quantized GRU can be written as follows, after the weights W_z, W_r and W and input x_t have already been quantized to [-1,1].

$$z_t = \sigma(W_z.[h_{t-1}, x_t]) \tag{12}$$

$$r_t = \sigma(W_r.[h_{t-1}, x_t]) \tag{13}$$

$$\tilde{h_t} = tanh\left(W.\left[2q_k\left(\frac{1}{2}(r_t h_{t-1}) + \frac{1}{2}\right) - 1, x_t\right]\right) \tag{14}$$

$$h_t = 2q_k\left(\frac{1}{2}((1 - z_t)h_{t-1} + z_t\tilde{h_t}) + \frac{1}{2}\right) - 1 \tag{15}$$

Following a similar method, we can also quantize LSTM networks.

4.3.2 Balanced Quantization

Uniform quantization is easy to implement but far from optimum when quantizing non-uniform data, which is believed to be the trained weights and activations of deep neural network. One way of performing non-uniform quantization is exponential quantization [53]. It quantizes the weight values to an integer power of 2. If we let $p = \frac{|W|}{2^{\lfloor \log_2 |W| \rfloor}} - 1$, deterministic exponential quantization can be written as follows.

$$\log_2 W_q = \begin{cases} \lceil \log_2 |W| \rceil & \text{if } p > 0.5 \\ \lfloor \log_2 |W| \rfloor & \text{otherwise} \end{cases} \tag{16}$$

Similarly, stochastic exponential quantization can be written as follows.

$$\log_2 W_q = \begin{cases} \lceil \log_2 |W| \rceil & \text{with prob } p \\ \lfloor \log_2 |W| \rfloor & \text{with prob } 1 - p \end{cases} \tag{17}$$

Exponential quantization enables storing weights in low precision and eliminating multiplications. However, it still does not perform quantization in a way which is sensitive to

the distribution of the weights. Distributions of parameters in neural networks are often imbalanced, such that the uniform quantization determined from extremal values may under utilize available bitwidth. When we quantize values, it may be desirable to make the quantized values have balanced distributions, to take full advantage of the available parameter space. Balanced quantization method [61] starts by partitioning numbers into 2^k bins containing roughly the same number of entries (percentiles). Each partition is then mapped to an evenly-divided interval in the closed interval [0, 1]. Finally, the quantization step maps intervals into discrete values using Eq. 9 and transforms the value range to be approximately the same as input.

A naïve implementation using percentiles as thresholds would require sorting of weight values during each forward operation in back-propagation, which may slow down the training process. The 2^k evenly spaced percentiles required in histogram equalization can be computed from the recursive application of partitioning of numbers by medians. Further, the mean μ can be used to approximate the median m. Thus, we can perform approximate histogram equalization without doing sorting.

4.3.3 KMeans based Quantization Schemes

Yet another way of performing non-uniform quantization is to decide bin boundaries using clustering in a static manner. In this static-KMeans method [62], We first train the neural network with full-precision parameters. Then apply KMeans to the weights. After clustering, the value of each pixel is set to the value of the center of the cluster it belongs to. We also need to store mapping from integers to cluster centers. Given k clusters, we only need $\log(k)$ bits to code the clusters.

A better approach is to perform KMeans clustering during training. In this method [23], multiple connections (belonging to the same cluster) share the same weight, and we fine-tune those shared weights. For the forward pass, the cluster index stored for each connection is mapped to a centroid which is then used as the weight. For back-propagation, during update, all the gradients are grouped by the cluster index and summed together, multiplied by the learning rate and subtracted from the shared centroids from last iteration. We use KMeans clustering to identify the shared weights for each layer of a trained network, so that all the weights that fall into the same cluster will share the same weight. Weights are not shared across layers. To calculate the compression rate, given k clusters, we only need $\log_2 k$ bits to encode the index. In general, for a network with n connections and each connection is represented with b bits, constraining the connections to have only k shared weights will result in a compression rate of: $r = \frac{nb}{n \log_2 k + kb}$.

There are two other ways of using KMeans for non-uniform quantization: Product Quantization (PQ) and Residual Quantization (RQ) [63]. In product quantization (PQ), we partition the vector space into many disjoint subspaces, and perform quantization (KMeans) in each subspace. Weight matrix W is partitioned columnwise: $W = [W^1, W^2, ..., W^s]$ where $W^i \in R^{m \times n/s}$ assuming n is divisible by s. Then we perform KMeans on each submatrix W^i to obtain clusters $c_1^i, ..., c_k^i$. Thus, we get s codebooks. The reconstructed matrix is $\hat{W} = [\hat{W}^1, \hat{W}^2, ..., \hat{W}^s]$ where $\hat{W}_j^i$ is the closest centroid c_j^i. PQ can be applied to either the x-axis or the y-axis of the matrix. We need to store the cluster indexes and codebooks for each subvector. The compression rate for this method is $(32mn)/(32kn + log_2(k)ms)$. Residual quantization (RQ) is similar. In RQ, we first quantize the vectors into k-centers. Next we find out the residuals for each data point $(w - c)$ and perform KMeans on the residuals. Do it recursively t times. Then the resultant weight vectors are calculated as $\hat{W}_z = c_j^1 + c_j^2 + ... + c_j^t$ given we have recursively performed t iterations. We need to store all the codebooks for each iteration, which potentially needs large amount of memory. The compression rate is $m/(tk + log_2(k)tn)$.

4.3.4 Loss Aware Quantization

Generalizing the loss aware binarization approach (Sec. 4.1.3) [50], we can perform k-bit quantization [64] by attempting to solve the following problem. $\min_{\{\alpha_i, b_i\}_{i=1}^k} \left|\left| w - \sum_{i=1}^k \alpha_i b_i \right|\right|^2$ where $w \in R^n$ is the original weight vector, $\alpha_i \in R$ and $b_i \in \{-1, +1\}^n$ are variables to be learned. This NP-hard problem can be solved using an iterative greedy approximation which sequentially minimizes the residue. In each iteration, first the residue is computed as $r_{i-1} = w - \sum_{j=1}^{i-1} \alpha_j b_j$, and then α_i and b_i are computed as $\alpha_i = \frac{1}{n}||r_{i-1}||_1$ and $b_i = \text{sign}(r_{i-1})$. Further, refined greedy approximation [64] extends this to further decrease the quantization error. In the j^{th} iteration after α_j and b_j have been updated, the method adds one extra step to refine all computed $\{\alpha_i\}_{i=1}^j$ with the least squares solution: $[\alpha_1, ..., \alpha_j] = ((B_j^T B_j)^{-1} B_j^T w)^T$ where $B_j = [b_1, ..., b_j]$. Typically refined greedy is more accurate than the greedy approach. In refined greedy approximation, after modification on the computed α's, b's are no longer optimal while the method keeps all of them fixed. To improve the refined greedy approximation, alternating minimizing α's and b's becomes a natural choice. Xu et al. [65] find that only two alternating cycles is good enough to find high precision quantization. Further, similar to [50], for an LSTM, we can combine overall network loss minimization with the multi-bit quantization loss minimization using this bi-level optimization. $\min_{w, \{\alpha_i, b_i\}_{i=1}^k} \text{LSTM}\left(\sum_{i=1}^k \alpha_i b_i\right)$ such that $\{\alpha_i, b_i\}_{i=1}^k = \text{argmin}_{\{\alpha_i', b_i'\}_{i=1}^k} \left|\left| w - \sum_{i=1}^k \alpha_i' b_i' \right|\right|^2$.

4.3.5 Quantization for Word Embeddings and Transformers

Each word vector is typically represented as a 300–500 dimensional vector, with each parameter being 32 bits. As there are millions of words, word vectors may take up to 3–6 GB of memory/storage. Can we quantize word vectors? We can clearly quantize them after training. But, we could also quantize when learning word embeddings. For example, Lam et al. [45] perform 1-bit and 2-bit quantization while performing word2vec [66] training using the Continuous Bag of Words (CBOW) method. They observe that quantization while training leads to better results compared to quantization after training.

Cheong et al. [27] applied BS-Fixed and BS-Flexible binary quantization to Transformer models. They observed

that the Transformer architecture is highly resistant to quantization, and is able to match the original model up to a 4-bit representation. Simple iterative pruning is much worse compared to quantization. Lastly, Shen et al. [67] propose mixed-precision quantization for BERT based on the observation that different encoder layers should use different number of bits for quantization. Layers that exhibit flatter curvature of the loss gradient surface can be quantized to lower bit precision. Thus, they use different number of bits at different levels of granularity: layers, attention heads and groups of neurons. They observe that quantizing embedding layers with 8 bits and other weight matrices with 2–4 bits leads to results comparable with full-precision BERT.

4.4 Summary

Quantization performs model compression by reducing the number of bits per weight value. Binary quantization does not work well by itself for text based neural models. But ternary and higher-bit quantization lead to significant model size reduction without loss in accuracy across tasks. Non-uniform quantization methods like balanced quantization or KMeans based quantization methods are better than uniform quantization methods. Loss aware quantization done while training is better than static loss-unaware quantization. Mixed-precision quantization combined with pruning is highly effective for Transformer based models.

5 Knowledge Distillation (KD)

KD methods are the most popular model compression methods for Transformer networks. Also called student-teacher networks, the main idea is to first train a deep teacher network, and then learn a shallow student network that mimics the teacher. After training, the student model is deployed. What information ("dark knowledge") from the teacher can be used to train the student? What loss functions can be used to ensure right flow of information from teacher to student? Can we have an ensemble of teachers, or teacher assistants or rather fellow students who can train the student? How can we optimize student training using adversarial training examples? We discuss these aspects in this section.

5.1 Various Distillation Architectures

Ba and Caruna [68] proposed Student Teacher networks (or mimic models) where the student uses the logits before softmax from the teacher network for training. The student model is not trained on the original labels; it is trained to learn the function that was learned by the teacher model. Thus, the student model is optimized to minimize the L2 loss between the teacher logits and the student logits across all training instances. Such distilled student models are more accurate than the same shallow student trained directly on the original labeled training data mainly because: (1) Teacher removes noisy labels, if any. (2) The uncertainty from the teacher is more informative to the student than the original $0/1$ labels. (3) The original targets may depend in part on features not available as inputs for learning, but the student sees targets that depend only on the input features. The dependence on unavailable features has been eliminated by filtering targets through the teacher.

Yet another way of utilizing logits is to have the student learn from noisy teacher logits [69]. After obtaining logits from the teacher, Gaussian noise with mean 0 and standard deviation σ is added to teachers logits. This perturbation can be applied to samples selected with probability α. The perturbed outputs produce the effect of a regularizer.

While Ba and Caruna [68] suggested using only the logits, Hinton et al. [70] suggested training the student by minimizing the cross entropy loss between the teacher softmax output and the student softmax output, besides minimizing the cross entropy between student prediction and actual label. The first part is called the soft loss and the second one is called the hard loss. Typically hard loss is given much lower weight compared to the soft loss term. To make the softmax output non-peaked and thereby transfer more useful information from teacher to student, softmax with temperature> 1 should be used. The same temperature should be used for training both the teacher and the student, but after the student has been trained the temperature can be set to 1 at test time. Besides logits and softmax output, Sobolev training for neural networks is a method for incorporating target derivatives in addition to the target values while training student network. Czarnecki et al. [71] experiment with first two derivatives of the targets.

KD has also been used along with quantization for better model compression [47], [72], [73]. We start with a trained full-precision large teacher network and an apprentice (student) network that has been initialised with full-precision weights. The apprentice network's precision is lowered and is fine-tuned using KD.

Why just use the output from the last layer of the teacher for training the student? In FitNets [74], the student performs hint-based training, i.e., the student is trained using not only the outputs but also the intermediate representations learned by the teacher as hints to improve the training process and final performance of the student. we choose a hidden layer of the FitNet, the guided layer, to learn from the teacher's hint layer. Because the student intermediate hidden layer will generally be smaller than the teacher's intermediate hidden layer, additional parameters are introduced to map the student hidden layer to the prediction of the teacher hidden layer.

While the methods discussed so far use logits, softmax output or their derivatives to transfer knowledge, Yim et al. [75] proposed a "flow of solution procedure (FSP)" method where the distilled knowledge is transferred in terms of flow between layers, which is calculated by computing the inner product between features from two layers. What does this "flow" capture intuitively? If we view the input of a deep network as the question and the output as the answer, we can think of the generated features at the middle of the network as the intermediate result in the solution process. There are many ways to solve the problem of generating the output from the input. Hence, mimicking the generated features of the teacher can be a hard constraint for the student. Learning the solution process from teacher is important. More concretely, the student is trained to minimize the L2 difference between the teacher and student FSP matrices computed across various pairs of layers and across multiple training instances. A similar method called Representational distance learning (RDL) has

also been proposed in [76].

Lastly, multiple KD variants have been proposed for sequence-level predictions [77], [78], e.g., for neural machine translation (NMT). In word-level KD, cross-entropy is minimized between the student/teacher distributions for each word in the actual target sequence, as well as between the student distribution and the degenerate data distribution, which has all of its probability mass on one word. In sequence-level KD the student network is trained on the output from beam search of the teacher network that had the highest score. In sequence-level interpolation the student is trained on the output from beam search of the teacher network that had the highest similarity (say using BLEU score) with the target sequence.

5.2 Collaborative Learning

Can multiple students learn from each other? Is a powerful teacher really required? In the deep mutual learning (DML) method [79], different from the one-way transfer between a static pre-defined teacher and a student in model distillation, with DML, an ensemble of students learn collaboratively and teach each other throughout the training process. Surprisingly, no prior powerful teacher network is necessary – mutual learning of a collection of simple student networks works, and moreover outperforms distillation from a more powerful yet static teacher. Specifically, each student is trained with two losses: a conventional supervised learning loss, and a mimicry loss that aligns each student's class posterior with the class probabilities of other students.

Anil et al. [80] propose a similar method but suggest letting the students learn independently just using the conventional supervised learning (hard) loss at least for a few burn in iterations. After this, the mutual learning can be done as in DML. They also propose a variant of their Co-Distillation method to perform this training in a distributed scenario where communication efficiency is also important. To update the parameters of one network using co-distillation one only needs the predictions of the other networks, which can be computed locally from copies of the other networks' weights. Empirically, using stale predictions instead of up-to-date predictions for the other neural networks has little to no adverse effect on the quality of the final trained model produced by co-distillation.

5.3 Multiple Teachers

So far we have talked about a student mimicing a single teacher. However, it is interesting to explore if the student can learn better in presence of multiple teachers or from a teacher assistant.

Intuitively and also observed empirically, student network performance degrades when the gap between student and teacher is large. Given a fixed student network, one cannot employ an arbitrarily large teacher, or in other words, a teacher can effectively transfer its knowledge to students up to a certain size, not smaller. To alleviate this shortcoming, Mirzadeh et al. [81] introduced multi-step KD, which employs an intermediate-sized network (teacher assistant) to bridge the gap between the student and the teacher. The teacher assistant (TA) models are distilled from the teacher, and the student is then only distilled from the TAs. One could also perform multi-step TA distillation, for example, distillation path could be $10 \rightarrow 6 \rightarrow 4 \rightarrow 2$.

A simple way to do KD with multiple teachers is to train student with cross entropy loss between student predictions and average prediction from multiple teachers. A more effective method is to augment this with a relative dissimilarity (RD) loss [82] defined over intermediate layer outputs generated for a triplet of instances between the student and an ensemble of teachers. For the student, the middle layer is selected. For each teacher, we select the layer such that most teachers are consistent with the resulting order relationships under the voting strategy. We discuss the RD loss given a student and a teacher. Consider a triplet of instances (x_i, x_i^+, x_i^-) such that at an intermediate layer of the teacher network, distance between activations for x_i^+ and x_i is smaller than the distance between activations for x_i^- and x_i. Let p_i be the intermediate output from student for example x_i. Then the RD loss for the triplet (x_i, x_i^+, x_i^-) is given by $max(0, d(p_i, p_i^+) - d(p_i, p_i^-) + \delta)$ where d is the distance function, and δ is a small constant to prevent the trivial solution. To extend this loss function definition to multiple teachers, the order between the instances x_i^+ and x_i^- given x_i is decided based on majority voting between the teachers.

There are also specific settings when distilling from multiple teachers becomes natural, e.g., when the number of classes is large [70] or in multi-lingual settings [83]. When the number of classes is very large, the teacher model could be an ensemble that contains one generalist model trained on all the data and many "specialist" models, each of which is trained on data that is highly enriched in examples from a very confusable subset of the classes (like different types of mushroom). Softmax distribution vector of this type of specialist can be made much smaller by combining all of the classes it does not care about into a single dustbin class. Each specialist model is initialized with the weights of the generalist model. These weights are then slightly modified by training the specialist with half its examples coming from its special subset and half sampled at random from the remainder of the training set. To derive groupings of object categories for the specialists, we focus on categories that the full generalist network often confuses. When training the student, for each instance, we first find the $setkofn$ most probable classes according to the generalist model. Then, we take all the specialist models, m, whose special subset of confusable classes has a non-empty intersection with k and call this the active set of specialists A_k. Given student's full probability distribution q over all the classes, we minimize $KL(p^g, q) + \sum_{m \in A_k} KL(p^m, q)$ where p^g is output distribution from the generalist model, and p^m is the output distribution from the m^{th} specialist model.

An ensemble of teachers is also very useful in a multilingual NMT setting [83]. Individual models for each language pair are first trained and regarded as teachers, and then the multilingual model is trained to fit the training data and match the outputs of individual models simultaneously through KD. When the accuracy of multilingual model surpasses the individual model for the accuracy threshold τ on a certain language pair, we remove the distillation loss and just train the model with original negative log-likelihood loss for this pair. Lastly, when learning from a

teacher ensemble, it is burdensome to load all the teacher models in the GPU memory for distillation. Alternatively, we first generate the output probability distribution of each teacher model for each instance offline, and then just load the top-K probabilities of the distribution into memory and normalize them so that they sum to 1 for distillation. This reduces the memory cost from the scale of $|V|$ (the vocabulary size) to K.

5.4 Adversarial Methods

How should we select instances for KD such that the student training converges fast while being effective? The generalization performance of a classifier is closely related to the adequacy of its decision boundary, so a good classifier bears a good decision boundary. Therefore, transferring information closely related to the decision boundary can be a good attempt for KD [84]. To realize this goal, an adversarial attack is utilized to discover samples supporting a decision boundary, and then a student classifier is trained based on these samples. To obtain the informative samples close to the decision boundary, we utilize an adversarial attack. In general, an adversarial attack tries to find a small modification that can change the class of a sample, i.e., it tries to move the sample beyond a nearby decision boundary. A boundary supporting sample (BSS) is an adversarial sample that lies near the decision boundary of a teacher classifier. A BSS is obtained by a gradient descent method based on a loss function defined over classification scores, and it contains information about both the distance and the path direction from the base sample to the decision boundary.

Xu et al. [85] and Wang et al. [86] propose ideas based on Generative Adversarial Networks (GANs) for KD. In [85], the authors propose a method under which the knowledge is transferred from teacher to student through a discriminator in the GAN-based approach. Discriminator (a 2-layer MLP) is trained to distinguish whether the output logits is from teacher or student network, while the student (the generator) is adversarially trained to fool the discriminator, i.e., output logits similar to the teacher logits so that the discriminator can not distinguish. The deep teacher is pretrained offline. The student and discriminator are alternatively updated in the GAN-based approach. The number of nodes in each layer of the discriminator is the same as the dimension of logits, i.e., the number of categories C. Output of the discriminator D is a $C+2$ dimensional vector with C Label predictions and a Real/Fake prediction. A slight different variant is KDGAN [86]. In KDGAN, the student and the teacher learn from each other via distillation losses and are adversarially trained against the discriminator via adversarial losses. Within each epoch, we first train the discriminator, then the teacher and finally the student classifier. D aims to maximize the probability of correctly distinguishing the true and pseudo labels, whereas C and T aim to minimize the probability that D rejects their generated pseudo labels. C learns from T by mimicking the learned distribution of T. Also, T learns from C.

5.5 Distilling Transformers

Recently, there has been a lot of work around distilling Transformers to smaller Transformers with less number of layers or to Bidirectional LSTMs. Some of these methods aim at improving the accuracy versus model size tradeoff, while others focus on complex settings like mismatching student-teacher vocabulary [87] or mismatch number of attention heads.

Zhao et al. [87] learn a student with small vocabulary compared to the teacher using a dual training method. During distillation, for a given training sequence input to the teacher model, they mix the teacher and student vocabularies by randomly selecting tokens from the sequence to segment using the student vocabulary, with the other tokens segmented using the teacher vocabulary. As part of the masked language model (MLM) task, the model now needs to learn to predict words from the student vocabulary using context words segmented using the teacher vocabulary, and vice versa. The expectation is that the student embeddings can be learned effectively this way from the teacher embeddings as well as teacher model parameters. We perform dual training only for the teacher model inputs. The student model receives words segmented exclusively using the student vocabulary. Also, during MLM, the model uses different softmax layers for the teacher and the student vocabularies depending on which one was used to segment the word in question. Instead of distilling solely on the teacher model's final-layer outputs, layer-wise teacher model parameters can also be leveraged to directly optimize parameters of corresponding layers in the student model.

In Patient KD [88], the student learns from the teacher's output after every k layers or the output from the last few layers of the teacher. The student BERT is initialized using some layers of the pre-trained teacher BERT. TinyBERT [89] further extends this idea by using extensive knowledge from embedding layer, and attention and hidden sub-layers of multiple teacher layers, and also the overall teacher output. Each student layer is first mapped to a teacher layer before the student training. Liu et al. [90] distill a multi-task student from a multi-task teacher, given the soft targets of the training data across multiple tasks. If task t has a teacher, the task-specific loss is the average of two objective functions, one for the correct targets and the other for the soft targets assigned by the teacher. In MiniLM [91], the student is trained by deeply mimicking the self-attention behavior of the last Transformer layer of the teacher. Besides self-attention distributions, MiniLM introduces the self-attention value-relation transfer to help the student achieve a deeper mimicry. The value-relation is computed as pairwise correlation between different components of the value matrix across various attention heads of the final layer.

Lastly, Tang et al. [92] propose distillation of a BERT model to a single layer BiLSTM using KL divergence between student and teacher logits. Mukherjee et al. [93] also distill a multi-lingual BERT (mBERT) model to a BiLSTM. Representation transfer is done from Transformer-based teacher model to BiLSTM-based student model with different embedding dimensions and disparate output spaces. Distillation features include teacher logits and internal teacher representations for one teacher layer. To make all output spaces compatible, a non-linear projection of the parameters in student representation is done to have same shape as teacher representation for each token. The projection parameters are learned by minimizing the KL-

divergence (KLD) between the representations of the student and the chosen layer from the teacher. Overall there are multiple loss functions for the student training: supervised hard loss, soft loss wrt output logits, and soft loss wrt internal teacher layer. Rather than optimizing for all loss functions jointly, stage-wise training is performed where each loss function is sequentially used for optimization.

5.6 Summary

To summarize, KD is a popular method for text based models. Various methods have proposed information copying using logits, softmax output, attention sub-layer output, value relation, relative dissimilarity information from both the last layer as well as intermediate layers of the teacher. Many methods have been proposed to handle complex teacher-student configuration mismatches in terms of vocabulary, number of attention heads, and hidden layer sizes. Also, KD has been found to be very effective in complex problem settings like multi-lingual tasks and tasks with large number of classes. Learning from noisy teachers, teacher assistants, an ensemble of teachers has been found to be effective as well.

6 Parameter Sharing

Rather than removing weights or reducing #bits to store them, parameter sharing methods reduce model size by finding weight blocks that can share the same weight. Character-based language models learn embeddings for characters and use them to compose word embeddings. In some senses, we can think of various words sharing these character embedding parameters. Further, various parameter sharing methods have been proposed to reduce the large word embedding matrix size. Finally, there are multiple Transformer architectures which benefit from the parameter sharing philosophy. We discuss these methods in this section.

6.1 Character-aware Language Models

Fig. 4 illustrates various charachter-aware language model architectures. Ling et al. [94] proposed their character to word (C2W) model which constructs vector representations of words by composing characters using BiLSTMs. Relative to traditional word representation models that have independent vectors for each word type, C2W requires only a single vector per character type and a fixed set of parameters for the compositional model. As input, we define an alphabet of characters C. For English, this vocabulary would contain an entry for each uppercase and lowercase letter as well as numbers and punctuation. Thus compared to the word embedding matrix, this model is much smaller. Despite the compactness of this model, this "composed" word representations yield comparable results across multiple text classification tasks. Jozefowicz et al. [95] propose two variants for composing word embeddings using character embeddings. In the first CNN-Softmax variant, they use character CNNs (Convolutional Neural Networks) to compose word embeddings from character embeddings both at the input side as well as at the output softmax layer. The character-CNN sub-networks at the input or the output

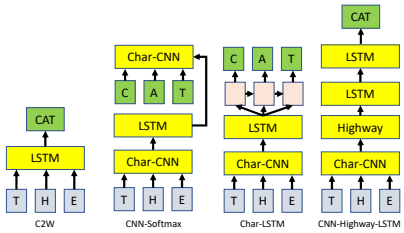

Fig. 4. Character-aware Language Models

do not share weights. The composed word embeddings are fed to an LSTM to generate the output. In the second Char-LSTM variant, character CNN is used to compose word embeddings on the input side. The composed word embeddings are fed to an LSTM to generate an output which is further fed to a small LSTM that predicts the target word one character at a time. Thus, the word and character-level models are combined, and predictions are made one character at a time, thus allowing to compute probabilities over a much smaller vocabulary. Kim et al. [96] propose another variant where at the output side they continue to use word embeddings, but at the input side they compose word embeddings using a highway network on top of a character CNN. The highway network's output is used as the input to a multi-layer LSTM, whose last hidden state output is fed to the output softmax layer.

6.2 Parameter Sharing in the Embedding Matrix

Given a weight matrix W and a budget K, we want to share weights within W to have a max of K unique values. A naïve implementation of random weight sharing can be trivially achieved by maintaining a secondary matrix consisting of each connection's group assignment. But this needs memory space itself. Hence, Chen et al. [97] propose to use hashing. HashedNets use a low-cost hash function (like xxhash[3]) to randomly group connection weights into hash buckets, and all connections within the same hash bucket share a single parameter value.

Unlike HashedNets where weights are randomly grouped, parameter sharing mechanisms in Toeplitz-like structured matrices [98] are highly specific and deterministic. Toeplitz matrices have parameters tied along diagonals. The displacement rank of all Toeplitz matrices is up to 2. Toeplitz-like matrices allow the displacement rank r to be higher. They include products and inverses of Toeplitz matrices, and their linear combinations. The displacement rank r serves as a knob on modeling capacity. High displacement rank matrices are increasingly unstructured. With displacement rank r, there are $2nr$ free parameters in the Toeplitz-like structured matrix. Toeplitz transforms can be applied not just to embedding matrix but to all weight matrices in an RNN model. Tay et al. [99] use a similar Toeplitz-like structured matrix method to propose Quaternion Transformers which lead to 75% parameter reduction in the Transformer architecture.

Another method for parameter sharing is to share low-rank factors across layers in a recurrent model. In this method, we first represent a weight matrix W using matrix factorization as $W = W_a W_b$. Thus, hidden layer output for

3. https://code.google.com/p/xxhash/

Embedding	x_1^c	x_2^c	x_3^c	x_4^c
x_1^r	january	february	...	...
x_2^r	one	two	...	...
x_3^r	...	...	...	...
x_4^r	...	...	...	...

TABLE 1
An Example of a Word Allocation Table

layer l at time t is $h_t^l = \sigma[W_a^l W_b^l h_t^{l-1} + U_a^l U_b^l h_{t-1}^l + b^l]$. But we can share some low-rank factors by setting $W_b^l = U_b^{l-1}$. The combination of matrix factorization and parameter sharing leads to large model compression.

Another way of compressing the embedding matrix is to divide the vocabulary V into frequent and infrequent word sets B and C respectively. Infrequent words' embeddings are represented with frequent words' by sparse linear combinations [100]. This is inspired by the observation that, in a dictionary, an unfamiliar word is typically defined by common words. A dense embedding is assigned to each common word; an infrequent word, on the other hand, computes its vector representation by a sparse combination of common words' embeddings. This compression is useful for both word embedding matrix as well as output layer of RNNs/LSTMs. Let $U \in R^{E \times |B|}$ be the learned embedding matrix of common words where E is the embedding dimension. For a word $w \in C$, we shall learn a sparse vector $x \in R^{|B|}$ as the sparse code of the word. Once we know x, embedding for a word $w \in C$ can be written as $\sum_{j=1}^{B} x_j U_j$ where U_j is the j^{th} column of U. To learn the sparse representation of word $w \in C$, the following problem needs to be solved: $\min_x ||Ux - A||_2^2 + \alpha||x||_1 + \beta|1^T x - 1| + \gamma 1^T max(0, -x)$ where A is embedding for the rare word w. The last two regularization terms favor a solution that sums to 1 and that is non-negative (for psychological interpretation concerns), respectively.

LightRNN [101] compresses word embedding matrix from $O(|V|)$ to $O(\sqrt{|V|})$. It uses a 2-Component shared embedding for word representations. We allocate every word in the vocabulary into a word-allocation table, each row of which is associated with a learned vector, and each column associated with another learned vector. Table 1 shows an example of a word allocation table. Depending on its position in the table, a word is jointly represented by two components: a row vector and a column vector. Thus, we only need $2\sqrt{|V|}$ vectors to represent a vocabulary of $|V|$ unique words, which are far less than the $|V|$ vectors. The input and output use different embedding row/column vectors but they share the same word-allocation table. Word Allocation table creation uses a bootstrap procedure to iteratively refine word allocation based on the learned word embedding. Embeddings (i.e. row and column vectors) are learned using language modeling loss using an RNN on top of the embedding layer.

Finally, Suzuki et al. [102] propose a Skipgram [66] training method with parameter sharing as follows. Split every embedding vector of size D into B equal sub-vectors of size C. Thus $D = B \times C$. We assign a limited number of reference vectors to each block of block-splitting vectors. E.g., the number of reference vectors becomes $K \times B$ if we assign K reference vectors to each block. Each reference vector is of size C. Skipgram training optimization remains the same except for these extra parameter sharing constraints (applied to both the input and output embedding vectors). Liu et al. [103] propose a very similar method where the embeddings are learned using a RNN language model rather than the Skipgram method.

6.3 Parameter Sharing in Transformers

A standard Transformer does not share parameters across layers and also has a fixed number of encoder layers. ALBERT [104] incorporates two parameter reduction techniques: (1) Factorized embedding parameterization. That is, it decomposes large vocabulary embedding matrix into two small matrices. Thus, it reduces the embedding parameters from $O(V \times H)$ to $O(V \times E + E \times H)$ where $H >> E$. (2) cross-layer parameter sharing: There are multiple ways to share parameters, e.g., only sharing feed-forward network (FFN) parameters across layers, or only sharing attention parameters. The default decision for ALBERT is to share all parameters across layers. An ALBERT configuration similar to BERT-large has 18x fewer parameters and can be trained about 1.7x faster. Dehghani et al. [105] propose Universal Transformers where the number of encoder layers are not pre-decided, and all the encoder layers share the parameters. Certain symbols (e.g. some words or phonemes) are usually more ambiguous than others. It is therefore reasonable to allocate more processing resources to these more ambiguous symbols. Thus, ambiguous symbols undergo more self-attention transformations compared to non-ambiguous ones. Thus, they provide a dynamic per-position halting mechanism for dynamically modulating the number of computational steps needed to process each input symbol (called the "ponder time") before the representation is passed on as input to the decoder. The idea of sharing weights across layers in Transformers has also been explored in [106].

6.4 Summary

Besides model compression, parameter sharing methods also act as a good regularizer. Parameter sharing in Transformers has been very successful. ALBERT was at the top of the GLUE leaderboard when it was proposed. Parameter sharing methods have also been widely used for compressing embedding matrix.

7 Tensor decomposition

Sparse Matrix decomposition has been traditionally used for applications like feature selection, collaborative filtering, topic mining from text, etc. In this section, we discuss how various popular tensor decomposition methods like Singular Value Decomposition (SVD), Tensor-Train [107], CP (CANDECOMP/PARAFAC) [108] and Tucker [109] can be used for model compression.

7.1 Two Low-Rank Factors

In this part, we will discuss methods where a matrix is factorized into two low-rank factors. Specifically, we replace a weight matrix W with $W_1 \times W_2$ such that the total number of parameters are significantly lesser.

A multi-layer RNN can be represented as follows.

$$h_t^l = \sigma(W_x^{l-1} h_t^{l-1} + W_h^l h_{t-1}^l + b^l) \quad (18)$$

$$h_t^{l+1} = \sigma(W_x^l h_t^l + W_h^{l+1} h_{t-1}^{l+1} + b^{l+1}) \quad (19)$$

Thus, there are two important weight matrices: the recurrent W_h^l and inter-layer matrices W_l^x. Prabhavalkar et al. [110] propose a method to jointly compress the recurrent and inter-layer matrices corresponding to a specific layer l by determining a suitable recurrent projection matrix, denoted by $P^l \in R^{r_l \times N_l}$ of rank $r^l < N^l$ such that $W_h^l = Z_h^l P^l$ and $W_l^x = Z_x^l P^l$. First, P^l is determined by computing a truncated SVD of the recurrent weight matrix, which we then truncate, retaining only the top r^l singular values. Thus, $W_h^l = (U_h^l \Sigma_h^l)(V_h^l)^T = Z_h^l P^l$. Thus, P^l is set to $(V_h^l)^T$. Further, we determine Z_x^l as the solution to the following least-squares problem: $Z_x^l = \text{argmin}_Y \, ||Y P^l - W_x^l||_2^2$. This solution can also be easily extended to LSTMs. Sak et al. [111] also proposed a similar solution based on a combination of parameter sharing and matrix decomposition but without SVD initialization. However, typically SVD initialization has been found to perform better.

Besides SVD, another way of matrix decomposition is sparse coding. Faruqui et al. [112] propose using sparse coding to decompose word embedding matrices. Thus, given vocabulary of size V, word embedding matrix $X \in R^{L \times V}$, sparse coding aims at representing each input vector x_i as a sparse linear combination of basis vectors a_i by solving the following problem. $\text{argmin}_{D,A} \sum_{i=1}^{V} ||x_i - D a_i||_2^2 + \lambda ||a_i||_1 + \tau ||D||_2^2$ where $D \in R^{L \times K}$ and $A \in R^{K \times V}$. Further, for interpretability, one can enforce all elements of A and D to be non-negative. For further compression, one can also enforce A to be binary or ensure that each column of A is a K sized one hot vector [113].

Lastly, Wang et al. [114] combine pruning with matrix factorization for model compression. Let W be a weight matrix. Structured pruning (removing a neuron, i.e., removing a column from weight matrix) can be achieved by replacing the computation Wx by WGx where diagonal sparsity-inducing matrix G is learned using L_0 regularization over WG along with the supervised loss. This effectively removes a subset of columns of W for column indices k with $z_k = 0$. One limitation is that this structured pruning method tends to produce lower performance than its unstructured counterpart. Hence, in the FL0P (Factorized L0 Pruning) model, we first factorize $W = PQ$. Let r be #columns of P (or equivalently #rows of Q), p_k and q_k be the k-th column of P and k-th row of Q respectively. We achieve structured pruning by introducing a pruning variable z_k for each component. $W = PGQ = \sum_{k=1}^{r} z_k \times (p_k q_k)$ where G is again the diagonal matrix of pruning variables. After training, only columns and rows corresponding to non-zero diagonal values need to be stored, resulting in much smaller (but still dense) matrices P and Q. The nonzero values of G can be absorbed into either P or Q. This structured pruning with factorization is much more effective compared to the vanilla structured pruning.

7.2 Factorizing into Block Diagonal Matrices

The last layer of a language model is very large of the size HV where H is the size of the hidden layer and V is vocabulary size. Each word by an output embedding of the same size H. Chen et al. [115] propose a differentiated softmax method which varies the dimension of the output embeddings across words depending on how much model capacity is deemed suitable for a given word. In particular, it is meaningful to assign more parameters to frequent words than to rare words. By definition, frequent words occur more often in the training data than rare words and therefore allow to fit more parameters. They define partitions of the output vocabulary based on word frequency and the words in each partition share the same embedding size. Partitioning results in a sparse final weight matrix which arranges the embeddings of the output words in blocks, each one corresponding to a separate partition. The size of the final hidden layer H is the sum of the embedding sizes of the partitions. While this method does not involve creation of multiple factors, it factorizes the original matrix into multiple blocks while setting the remaining part of the matrix to 0.

Variani et al. [116] propose a method called Word Encoded Sequence Transducers (WEST) which factorizes a matrix $E = C \times D$ where D is constrained to be a block diagonal matrix. The block diagonal nature of the second factor leads to large compression rates.

7.3 Tensor Train and Block Term Decomposition

Tensor train decomposition (TTD) [107] is a standard tensor decomposition technique which decomposes a high dimensional tensor into multiple 2D and 3D tensors which can be multiplied together to reconstruct the original tensor. These factors are called TT-cores and theor dimensions are referred to as TT-ranks. TTD can be leveraged to compress various weight matrices in RNNs and LSTMs [117], [118]. The first step is to represent a matrix as a multi-dimensional tensor by simple reshaping transformation and then use TTD on it. The values of TTranks directly define the compression ratio, so choosing them to be too small or too large will result into either significant performance drop or little reduction of the number of parameters. Typically TT-ranks around 16 for small matrices and 64-192 for larger matrices result in a good trade-off between compression ratio and the accuracy metric of interest. Also, when we use TTD for weight matrices, we also need change the inputs appropriately to be compatible in terms of dimensions.

Compared with TT-RNN, Block-Term RNN (BTRNN) [119] is not only more concise (when using the same rank), but also able to attain a better approximation to the original RNNs with much fewer parameters. BTD decomposes a high order tensor into a sum of multiple Tucker decomposition models. The redundant dense connections between input and hidden state is first tensorized to a d-dimensional tensor and then decomposed using low-rank BTD into a sum of N different Tucker decompositions where N is the CP-rank. Each Tucker decomposition in turn consists of a core d-dimensional tensor and d 3-dimensional factor tensors. While Ye et al. [119] used BTD to compress RNNs, Ma et al. [120] used BTD to compress the self-attention matrix in Transformers. They first build a single-block attention based on the Tucker decomposition where the query, key and value are mapped

into three factor matrices and the core tensor is trainable and randomly initialized. It is then straightforward to represent the multi-head attention using BTD.

7.4 Summary

To summarize, matrix decomposition techniques are usually used in cmombination with parameter sharing. They have been very effective in dealing with large input/output embedding matricesin RNNs and LSTMs. SVD, Tensor Train, CP, Tucker, BTD have been the most popular decomposition techniques found to be useful for model compression.

8 Transformers with Linear Complexity

Time and memory in Transformers grows quadratically with the sequence length. This is because in every layer, every attention head attempts to come up with a transformed representation for every position by "paying attention" to tokens at every other position. Quadratic complexity implies that practically the maximum input size is rather limited. Thus, we cannot extract semantic representation for long documents by passing them as input to Transformers. Hence, there have been several efforts recently to reduce this quadratic complexity to linear. Most of these efforts choose a constant number of other positions to "pay attention" to so as to compute a transformed representation for any given position. They can model sequences tens of thousands of timesteps long using hundreds of layers. The methods differ in their approach towards selecting this constant number of other positions. We discuss a few of such recently proposed methods in this section.

Child et al. [121] propose sparse transformers where sparse factorizations of the attention matrix reduce the quadratic complexity to $O(n\sqrt{n})$. They propose two kinds of sparse factorizations: strided and fixed. Strided attention implies having one head attend to the previous l locations, and the other head attend to every l^{th} location, where l is the stride and chosen to be close to $\sqrt{n}$. More heads could be used with a different stride value. Fixed attention assumes that specific positions summarize previous locations and propagate that information to all future positions.

In Star-Transformers [122], to reduce model complexity from $O(n^2)$ to $O(2n)$, we replace the fully-connected attention matrix structure with a star-shaped topology, in which every two non-adjacent nodes are connected through a shared relay node. While ring connections connect a satellite node with two other satellite nodes, a radical connection connects a satellite node with the relay node. The idea is to update the star-center relay node based on satellite nodes and then update satellite nodes using information from the star node, and adjacent satellite nodes.

The Reformer architecture [123] replaces the dot-product attention in a typical Transformer by one that uses locality-sensitive hashing (LSH), changing its complexity from $O(n^2)$ to $O(n \log n)$, where n is the length of the sequence. In a standard Transformer, we compute scaled dot-product attention as $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d}})V$ where Q, K and V are the standard query, key and value components and d is a scaling factor. Reformer uses a Shared QK Transformer, i.e., $Q = K$ enabled by sharing the matrix that projects words/hidden layer to Q or K. Further, note that we are actually only interested in $\text{softmax}(QK^T)$. Since softmax is dominated by the largest elements, for each query q_i we only need to focus on the keys in K that are closest to q_i. How can we find the nearest neighbors among the keys? Reformer uses LSH. LSH is used to cluster (hash-bucket) the positions into various groups, and then every position needs to focus only on others within the same bucket.

Linformer architecture [124] exploits low-rank factorization of the self-attention matrix to reduce overall self-attention complexity from $O(n^2)$ to $O(n)$ in both time and space. The main idea is to add two linear projection matrices $E_i, F_i \in R^{n \times k}$ when computing key and value. We first project the original $(n \times d)$-dimensional key and value layers into $(k \times d)$-dimensional projected key and value layers. We then compute an $(n \times k)$-dimensional context mapping matrix using scaled dot-product attention. If we can choose a very small projected dimension k, such that $k << n$, then we can significantly reduce the memory and space consumption. Overall, it is $O(nk)$. Further, we can do two other forms of parameter sharing: (1) Headwise sharing: $E_i = E$ and $F_i = F$ across all heads i in a layer. (2) Key-value sharing: $E_i = F_i = E$ across all heads i in a layer. (3) Layerwise sharing: Single projection matrix E is used across all layers, all heads for both key and value.

Sparse Sinkhorn Attention based Transformer [125] is based on differentiable sorting of internal representations. First, they divide the input sequence into B equal sized blocks each of size n/B. A meta sorting network learns to generate latent permutations over these block sequences. Given sorted sequences, we are then able to compute quasi-global attention with only local windows, improving the memory efficiency of the attention module. They also propose Causal Sinkhorn Balancing and SortCut algorithms for causal scenarios for tailoring Sinkhorn Attention for encoding and/or decoding purposes. Their method reduces the memory complexity from $O(n^2)$ to $O(B^2 + (n/B)^2)$. The SortCut variant further reduces complexity to linear-time, i.e., $O(nk)$ where k is a user defined budget hyper-parameter much smaller than n.

Shen et al. [126] propose a very simple mathematical trick to reduce quadratic complexity to linear. A typical dot-product attention can be written as $\text{softmax}(QK^T)V$ ignoring the scale factor. This is quadratic because QK^T is n^2 in size. This can be rewritten as $\text{softmax}_r(Q)(\text{softmax}_c(K)^T V)$ where softmax_r and softmax_c are softmax applied to rows and columns respectively. This revised formulation has terms which are only linear in n. Finally, Katharopoulos et al. [127] express the self-attention as a linear dot-product of kernel feature maps and make use of the associativity property of matrix products to reduce the complexity from $O(n^2)$ to $O(n)$, where n is the sequence length.

To summarize, multiple methods have been proposed to reduce the quadratic complexity of the standard Transformer model. While Sparse Transformers reduce it to $O(n\sqrt{n})$, Reformers reduce it to $O(n \log n)$. Other methods like Star Transformer, Linformer, Sparse Sinkhorn Transformer, Efficient Attention and Linear Tranformers promise linear complexity.

9 Applications

The model compression mentioned in this survey have been used across a wide variety of text processing tasks. In Table 2 We list down the tasks, popular datasets and references where the readers can find more discussion around model size versus accuracy tradeoff.

10 Summary and Future Directions

We discussed various methods for compression of deep learning models for text. Broadly, we discussed pruning, quantization, knowledge distillation, parameter sharing, tensor decomposition, and Linear Transformer based methods. These methods not just help reduce the model size, but also lead to lower prediction latencies and low power consumption due to reduced computations.

However, there is a lot more work to be done. (1) With linear Transformer models, one can afford to have input with tens of thousands of tokens. Hence, many tasks need to be redesigned where large context can now be included as input to improve accuracy. (2) Combinations of several methods have not been tested well. Lot of experiments are needed to check how models respond to combination of model compression methods. (3) Latency results vary based on GPU architectures. With new GPU architectures (Nvidia RTX 3080, Nvidia T4), some methods like quantization may become more impactful. (4) Real world settings are often complex: multi-modal, multi-task, multi-label, small-data, noisy labels, multi-teachers, mismatching teacher-student architectures. Efficient ways of recommending the most promising method is necessary. (5) Different components/structures of a model may respond to different kinds of compression methods with specific hyper-parameters. A generic method to choose the right method for various structures is needed. (6) How does compression of models impact their interpretability? Can we design model compression mechanisms aimed at looking at a tradeoff between model accuracy, size, latency and interpretability. (7) None of the model compression methods performs any application specific compression. Can we obtain further compression by exploiting some task-specific patterns?

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017, pp. 5998–6008.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[3] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *ICLR*, 2019.

[4] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "SuperGLUE: A stickier benchmark for general-purpose language understanding systems," *1905.00537*, 2019.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, 2019.

[6] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," *arXiv:1901.11504*, 2019.

[7] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *arXiv:1906.08237*, 2019.

[8] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using gpu model parallelism," *arXiv:1909.08053*, 2019.

[9] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv:1910.10683*, 2019.

[10] C. Rosset, "Turing-nlg: A 17-billion-parameter language model by microsoft," *Microsoft Blog*, 2019.

[11] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *arXiv:2006.16668*, 2020.

[12] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018.

[13] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv:1910.01108*, 2019.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[15] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent rnns: Stashing recurrent weights on-chip," in *ICML*, 2016, pp. 2024–2033.

[16] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, "Predicting parameters in deep learning," in *NIPS*, 2013, pp. 2148–2156.

[17] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arXiv:1710.09282*, 2017.

[18] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[19] C. A. Walsh, "Peter huttenlocher (1931–2013)," *Nature*, vol. 502, no. 7470, pp. 172–172, 2013.

[20] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv:1710.01878*, 2017.

[21] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *NIPS*, 1990, pp. 598–605.

[22] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *NIPS*, 1993, pp. 164–171.

[23] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv:1510.00149*, 2015.

[24] A. See, M.-T. Luong, and C. D. Manning, "Compression of neural machine translation models via pruning," *arXiv:1606.09274*, 2016.

[25] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015, pp. 1135–1143.

[26] S. Narang, E. Elsen, G. Diamos, and S. Sengupta, "Exploring sparsity in recurrent neural networks," *arXiv:1704.05119*, 2017.

[27] R. Cheong and R. Daniel, "transformers. zip: Compressing transformers with pruning and quantization," Technical report, Stanford University, Stanford, California, 2019., Tech. Rep., 2019.

[28] F.-M. Guo, S. Liu, F. S. Mungall, X. Lin, and Y. Wang, "Reweighted proximal pruning for large-scale language representation," *arXiv:1909.12486*, 2019.

[29] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran *et al.*, "Dsd: Dense-sparse-dense training for deep neural networks," *arXiv:1607.04381*, 2016.

[30] X. Dai, H. Yin, and N. K. Jha, "Grow and prune compact, fast, and accurate lstms," *arXiv:1805.11797*, 2018.

[31] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu, "Reshaping deep neural network for fast decoding by node-pruning," in *ICASSP*. IEEE, 2014, pp. 245–249.

[32] K. Murray and D. Chiang, "Auto-sizing neural networks: With applications to n-gram language models," *arXiv:1508.05051*, 2015.

[33] W. Pan, H. Dong, and Y. Guo, "Dropneuron: Simplifying the structure of deep neural networks," *arXiv:1606.07326*, 2016.

[34] S. Srinivas and R. V. Babu, "Data-free parameter pruning for deep neural networks," *arXiv:1507.06149*, 2015.

Task	Popular Datasets	References
Language modeling	Penn TreeBank Corpus, One billion word benchmark, Europarl, WikiText-103, text8, source code of Linux kernel, 2013 ACL Workshop Morphological Language Datasets (ACLW), Arabic news commentary corpus, 2013 ACL workshop on MT, enwik8 (from Wikipedia), Lambada	[20], [32], [36], [40], [44], [59], [60], [65], [94], [95], [96], [100], [101], [103], [105], [106], [114], [116], [118], [120], [121], [123], [124], [125], [128], [129], [130]
Neural Machine translation (NMT)	IWSLT German-English, IWSLT Thai-English, ASPEC English-Japanese, WMT English-German, WMT German-English, WMT English-Russian, IWSLT English Vietnamese, WMT English-Romanian, WMT English-Estonian, Ted Talk	[20], [24], [27], [32], [37], [38], [40], [73], [77], [78], [83], [99], [105], [113], [118], [120]
Sentiment Analysis	IMDB movie review, SST, SST-2, Elec (electronic product reviews)	[6], [28], [54], [60], [87], [88], [89], [91], [92], [93], [99], [104], [112], [113], [118], [122], [124], [125]
Question Answering	SQuAD1.1, SQuAD2.0, ELI5, SemEval, BABI	[28], [40], [45], [91], [104], [105], [131]
Natural Language Inference	SNLI, MNLI-m, MNLI-mm, QNLI, RTE, WNLI, XNLI	[6], [28], [40], [87], [88], [89], [91], [92], [104], [122], [124], [125]
Paraphrasing	QQP, STS-B	[6], [28], [88], [89], [91], [92], [104], [124]
Image captioning	MSCOCO	[23], [29], [30]
Handwritten character recognition	ICDAR	[132]
Part-of-speech (PO) tagging	Wall Street Journal of the Penn Treebank dataset, WikiAnn NER corpus	[93], [94]
Summarization	CNN-DailyMail, XSum	[40], [91]
Machine Reading Comprehension	Microsoft Research Paraphrase Corpus (MRPC), ReAding Comprehension from Examinations (RACE)	[6], [28], [40], [87], [88], [89], [91], [104]
Linguistic Acceptability	CoLA	[6], [28], [89], [91], [104]
Topic Classification	DbPedia, Ag News,20 Newsgroup	[93], [112]
Question Type Classification	TREC	[112]
Noun Phrase Bracketing	Lazaridou [133]	[112]
Word Similarity	SimLex-999, MEN, MTurk, RARE, SCWS, WSR, WSS	[102], [112]
Mathematical Language Understanding	Wangperawong's MLU [134]	[99]
Subject Verb Agreement	Linzen [135]	[99], [105]
Word Analogy	GSEM, GSYN, MSYN	[102]
Sentence Completion	MSC	[102]
Learning to execute	Zaremba and Sutskever Method [136]	[105]
Ad Click Through Rate Prediction	Criteo Kaggle	[118]

TABLE 2
Applications of Model Compression Methods for Text

[35] S. Narang, E. Undersander, and G. Diamos, "Block-sparse recurrent neural networks," *arXiv:1711.02782*, 2017.

[36] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *SIGDA Intl. Symp. on FPGA*. ACM, 2019, pp. 63–72.

[37] P. Michel, O. Levy, and G. Neubig, "Are sixteen heads really better than one?" *arXiv:1905.10650*, 2019.

[38] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov, "Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned," *arXiv:1905.09418*, 2019.

[39] Y. Ding, Y. Liu, H. Luan, and M. Sun, "Visualizing and understanding neural machine translation," in *ACL*, 2017, pp. 1150–1159.

[40] A. Fan, E. Grave, and A. Joulin, "Reducing transformer depth on demand with structured dropout," *arXiv:1909.11556*, 2019.

[41] S. Prasanna, A. Rogers, and A. Rumshisky, "When bert plays the lottery, all tickets are winning," *arXiv:2005.00561*, 2020.

[42] T. M. Bartol, C. Bromer, J. Kinney, M. A. Chirillo, J. N. Bourne, K. M. Harris, and T. J. Sejnowski, "Hippocampal spine head sizes are highly precise," *bioRxiv*, p. 016329, 2015.

[43] D. J. Linden, *Think Tank: Forty Neuroscientists Explore the Biological Roots of Human Experience*. Yale University Press, 2018.

[44] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *JMLR*, vol. 18, no. 1, pp. 6869–6898, 2017.

[45] M. Lam, "Word2bits-quantized word vectors," *arXiv:1803.05651*, 2018.

[46] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *NIPS*, 2015, pp. 3123–3131.

[47] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[48] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv:1510.03009*, 2015.

[49] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *NIPS*, 2016, pp. 4107–4115.

[50] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*. Springer, 2016, pp. 525–542.

[51] L. Hou, Q. Yao, and J. T. Kwok, "Loss-aware binarization of deep networks," *arXiv:1611.01600*, 2016.

[52] J. D. Lee, Y. Sun, and M. A. Saunders, "Proximal newton-type methods for minimizing composite functions," *J. Optimization*, vol. 24, no. 3, pp. 1420–1443, 2014.

[53] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu, and Y. Bengio, "Recurrent neural networks with limited numerical precision," *arXiv:1608.06902*, 2016.

[54] M. Z. Alom, A. T. Moody, N. Maruyama, B. C. Van Essen, and T. M. Taha, "Effective quantization approaches for recurrent neural networks," in *IJCNN*. IEEE, 2018, pp. 1–8.

[55] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv:1605.04711*, 2016.

[56] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *SiPS*. IEEE, 2014, pp. 1–6.

[57] S. Lloyd, "Least squares quantization in pcm," *Tran. on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[58] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv:1612.01064*, 2016.

[59] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, "Hitnet: hybrid ternary recurrent neural network," in *NIPS*, 2018, pp. 604–614.

[60] Q. He, H. Wen, S. Zhou, Y. Wu, C. Yao, X. Zhou, and Y. Zou, "Effective quantization methods for recurrent neural networks," *arXiv:1611.10176*, 2016.

[61] S.-C. Zhou, Y.-Z. Wang, H. Wen, Q.-Y. He, and Y.-H. Zou, "Balanced quantization: An effective and efficient approach to quantized neural networks," *J. of Computer Science and Technology*, vol. 32, no. 4, pp. 667–682, 2017.

[62] L. K. Muller and G. Indiveri, "Rounding methods for neural networks with low resolution synaptic weights," *arXiv:1504.05767*, 2015.

[63] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv:1412.6115*, 2014.

[64] Y. Guo, A. Yao, H. Zhao, and Y. Chen, "Network sketching: Exploiting binary structure in deep cnns," in *CVPR*, 2017, pp. 5955–5963.

[65] C. Xu, J. Yao, Z. Lin, W. Ou, Y. Cao, Z. Wang, and H. Zha, "Alternating multi-bit quantization for recurrent neural networks," *arXiv:1802.00150*, 2018.

[66] T. Mikolov, K. Chen, G. Corrado, J. Dean, L. Sutskever, and G. Zweig, "word2vec," *URL https://code. google. com/p/word2vec*, vol. 22, 2013.

[67] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-bert: Hessian based ultra low precision quantization of bert," *arXiv:1909.05840*, 2019.

[68] J. Ba and R. Caruana, "Do deep nets really need to be deep?" in *NIPS*, 2014, pp. 2654–2662.

[69] B. B. Sau and V. N. Balasubramanian, "Deep model compression: Distilling knowledge from noisy teachers," *arXiv:1610.09650*, 2016.

[70] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv:1503.02531*, 2015.

[71] W. M. Czarnecki, S. Osindero, M. Jaderberg, G. Swirszcz, and R. Pascanu, "Sobolev training for neural networks," in *NIPS*, 2017, pp. 4278–4287.

[72] A. Mishra and D. Marr, "Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy," *arXiv:1711.05852*, 2017.

[73] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *arXiv:1802.05668*, 2018.

[74] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," *arXiv:1412.6550*, 2014.

[75] J. Yim, D. Joo, J. Bae, and J. Kim, "A gift from knowledge distillation: Fast optimization, network minimization and transfer learning," in *CVPR*, 2017, pp. 4133–4141.

[76] P. McClure and N. Kriegeskorte, "Representational distance learning for deep neural networks," *Frontiers in computational neuroscience*, vol. 10, p. 131, 2016.

[77] Y. Kim and A. M. Rush, "Sequence-level knowledge distillation," *arXiv:1606.07947*, 2016.

[78] M. Freitag, Y. Al-Onaizan, and B. Sankaran, "Ensemble distillation for neural machine translation," *arXiv:1702.01802*, 2017.

[79] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep mutual learning," in *CVPR*, 2018, pp. 4320–4328.

[80] R. Anil, G. Pereyra, A. Passos, R. Ormandi, G. E. Dahl, and G. E. Hinton, "Large scale distributed neural network training through online distillation," *arXiv:1804.03235*, 2018.

[81] S.-I. Mirzadeh, M. Farajtabar, A. Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh, "Improved knowledge distillation via teacher assistant," *arXiv:1902.03393*, 2019.

[82] S. You, C. Xu, C. Xu, and D. Tao, "Learning from multiple teacher networks," in *KDD*, 2017, pp. 1285–1294.

[83] X. Tan, Y. Ren, D. He, T. Qin, Z. Zhao, and T.-Y. Liu, "Multilingual neural machine translation with knowledge distillation," *arXiv:1902.10461*, 2019.

[84] B. Heo, M. Lee, S. Yun, and J. Y. Choi, "Knowledge distillation with adversarial samples supporting decision boundary," in *AAAI*, vol. 33, 2019, pp. 3771–3778.

[85] Z. Xu, Y.-C. Hsu, and J. Huang, "Training shallow and thin networks for acceleration via knowledge distillation with conditional adversarial networks," *arXiv:1709.00513*, 2017.

[86] X. Wang, R. Zhang, Y. Sun, and J. Qi, "Kdgan: Knowledge distillation with generative adversarial networks," in *NIPS*, 2018, pp. 775–786.

[87] S. Zhao, R. Gupta, Y. Song, and D. Zhou, "Extreme language model compression with optimal subwords and shared projections," *arXiv:1909.11687*, 2019.

[88] S. Sun, Y. Cheng, Z. Gan, and J. Liu, "Patient knowledge distillation for bert model compression," *arXiv:1908.09355*, 2019.

[89] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv:1909.10351*, 2019.

[90] X. Liu, P. He, W. Chen, and J. Gao, "Improving multi-task deep neural networks via knowledge distillation for natural language understanding," *arXiv:1904.09482*, 2019.

[91] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *arXiv:2002.10957*, 2020.

[92] R. Tang, Y. Lu, L. Liu, L. Mou, O. Vechtomova, and J. Lin, "Distilling task-specific knowledge from bert into simple neural networks," *arXiv:1903.12136*, 2019.

[93] S. Mukherjee and A. H. Awadallah, "Xtremedistil: Multi-stage distillation for massive multilingual models," in *ACL*, 2020, pp. 2221–2234.

[94] W. Ling, T. Luís, L. Marujo, R. F. Astudillo, S. Amir, C. Dyer, A. W. Black, and I. Trancoso, "Finding function in form: Compositional character models for open vocabulary word representation," *arXiv:1508.02096*, 2015.

[95] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," *arXiv:1602.02410*, 2016.

[96] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, "Character-aware neural language models," in *AAAI*, 2016.

[97] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," in *ICML*, 2015, pp. 2285–2294.

[98] Z. Lu, V. Sindhwani, and T. N. Sainath, "Learning compact recurrent neural networks," in *ICASSP*. IEEE, 2016, pp. 5960–5964.

[99] Y. Tay, A. Zhang, L. A. Tuan, J. Rao, S. Zhang, S. Wang, J. Fu, and S. C. Hui, "Lightweight and efficient neural natural language processing with quaternion networks," *arXiv:1906.04393*, 2019.

[100] Y. Chen, L. Mou, Y. Xu, G. Li, and Z. Jin, "Compressing neural language models by sparse word representations," *arXiv:1610.03950*, 2016.

[101] X. Li, T. Qin, J. Yang, and T.-Y. Liu, "Lightrnn: Memory and computation-efficient recurrent neural networks," in *NIPS*, 2016, pp. 4385–4393.

[102] J. Suzuki and M. Nagata, "Learning compact neural word embeddings by parameter space sharing." in *IJCAI*, 2016, pp. 2046–2052.

[103] Z. Li, R. Kulhanek, S. Wang, Y. Zhao, and S. Wu, "Slim embedding layers for recurrent neural language models," in *AAAI*, 2018.

[104] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv:1909.11942*, 2019.

[105] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and Ł. Kaiser, "Universal transformers," *arXiv:1807.03819*, 2018.

[106] S. Bai, J. Z. Kolter, and V. Koltun, "Deep equilibrium models," *arXiv:1909.01377*, 2019.

[107] I. V. Oseledets, "Tensor-train decomposition," *SIAM J. on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[108] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of eckart-young decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[109] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[110] R. Prabhavalkar, O. Alsharif, A. Bruguier, and L. McGraw, "On the compression of recurrent neural networks with an application to lvcsr acoustic modeling for embedded speech recognition," in *ICASSP*. IEEE, 2016, pp. 5970–5974.

[111] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," *arXiv:1402.1128*, 2014.

[112] M. Faruqui, Y. Tsvetkov, D. Yogatama, C. Dyer, and N. Smith, "Sparse overcomplete word vector representations," *arXiv:1506.02004*, 2015.

[113] R. Shu and H. Nakayama, "Compressing word embeddings via deep compositional code learning," *arXiv:1711.01068*, 2017.

[114] Z. Wang, J. Wohlwend, and T. Lei, "Structured pruning of large language models," *arXiv:1910.04732*, 2019.

[115] W. Chen, D. Grangier, and M. Auli, "Strategies for training large vocabulary neural language models," *arXiv:1512.04906*, 2015.

[116] E. Variani, A. T. Suresh, and M. Weintraub, "West: Word encoded sequence transducers," in *ICASSP*. IEEE, 2019, pp. 7340–7344.

[117] A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," in *IJCNN*. IEEE, 2017, pp. 4451–4458.

[118] V. Khrulkov, O. Hrinchuk, L. Mirvakhabova, and I. Oseledets, "Tensorized embedding layers for efficient model compression," *arXiv:1901.10787*, 2019.

[119] J. Ye, L. Wang, G. Li, D. Chen, S. Zhe, X. Chu, and Z. Xu, "Learning compact recurrent neural networks with block-term tensor decomposition," in *CVPR*, 2018, pp. 9378–9387.

[120] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, D. Song, and M. Zhou, "A tensorized transformer for language modeling," *arXiv:1906.09777*, 2019.

[121] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv:1904.10509*, 2019.

[122] Q. Guo, X. Qiu, P. Liu, Y. Shao, X. Xue, and Z. Zhang, "Star-transformer," in *NAACL-HLT*, 2019, pp. 1315–1325.

[123] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv:2001.04451*, 2020.

[124] S. Wang, B. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv:2006.04768*, 2020.

[125] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan, "Sparse sinkhorn attention," *arXiv:2002.11296*, 2020.

[126] Z. Shen, M. Zhang, H. Zhao, S. Yi, and H. Li, "Efficient attention: Attention with linear complexities," *arXiv:1812.01243*, 2018.

[127] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," *arXiv:2006.16236*, 2020.

[128] S. Kapur, A. Mishra, and D. Marr, "Low precision rnns: Quantizing rnns without losing accuracy," *arXiv:1710.07706*, 2017.

[129] L. Hou and J. T. Kwok, "Loss-aware weight quantization of deep networks," *arXiv:1802.08635*, 2018.

[130] A. M. Grachev, D. I. Ignatov, and A. V. Savchenko, "Compression of recurrent neural networks for efficient language modeling," *Applied Soft Computing*, vol. 79, pp. 354–362, 2019.

[131] S. Damani, K. N. Narahari, A. Chatterjee, M. Gupta, and P. Agrawal, "Optimized transformer models for faq answering," in *PAKDD*, 2020, p. To appear.

[132] Y. Yang, K. Liang, X. Xiao, Z. Xie, L. Jin, J. Sun, and W. Zhou, "Accelerating and compressing lstm based model for online handwritten chinese character recognition," in *ICFHR*. IEEE, 2018, pp. 110–115.

[133] A. Lazaridou, E. M. Vecchi, and M. Baroni, "Fish transporters and miracle homes: How compositional distributional semantics can help np parsing," in *EMNLP*, 2013, pp. 1908–1913.

[134] A. Wangperawong, "Attending to mathematical language with transformers," *arXiv:1812.02825*, 2018.

[135] T. Linzen, E. Dupoux, and Y. Goldberg, "Assessing the ability of lstms to learn syntax-sensitive dependencies," *TACL*, vol. 4, pp. 521–535, 2016.

[136] W. Zaremba and I. Sutskever, "Learning to execute," *arXiv:1410.4615*, 2014.

Manish Gupta (*Homepage*) is a Principal Applied Researcher at Microsoft AI and Research. He is also an Adjunct Faculty at IIIT-Hyderabad and a visiting faculty at the Indian School of Business. He received his Masters from IIT-Bombay in 2007 and his Ph.D. from Univ. of Illinois at Urbana-Champaign in 2013. His research interests include deep learning, NLP, web mining and IR. He has also co-authored two books on "Outlier Detection for Temporal Data" and "Information Retrieval with Verbose Queries".

Puneet Agarwal (*LinkedIn*) is a Principal Software Engineering Manager at the Bing team in Microsoft AI and Research at Hyderabad, India. He received his BTech in Computer Science from Indian Institute of Technology (IIT) Delhi in 2008. He interned at Yahoo! in 2007, and has been with Microsoft for the past 12 years. His research interests include web search, natural language processing, deep learning, and conversational agents.