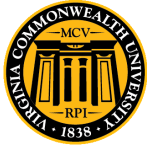

Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations

Graduate School

2021

Improving Space Efficiency of Deep Neural Networks

Aliakbar Panahi
Virginia Commonwealth University

Follow this and additional works at: https://scholarscompass.vcu.edu/etd

Part of the Artificial Intelligence and Robotics Commons, and the Data Science Commons

© Aliakbar Panahi

Downloaded from
https://scholarscompass.vcu.edu/etd/6757

This Dissertation is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

©Aliakbar Panahi, August 2021

All Rights Reserved.

IMPROVING SPACE EFFICIENCY OF DEEP NEURAL NETWORKS

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

ALIAKBAR PANAHI

Master of Science in Computer Science from Virginia Commonwealth University, 2017

Bachelor of Science in Mechanical Engineering from Azad University, 2012

Director: Dr. Tomasz Arodz,

Associate Professor, Department of Computer Science

Virginia Commonwealth University

Richmond, Virginia

August, 2021

TABLE OF CONTENTS

Chapter Page

Table of Contents i

List of Figures iii

Abstract v

1 Introduction 1

2 Background 4

2.1 Supervised Learning 4
2.2 Cross-Entropy Loss 7
2.3 Optimization 8
2.4 Neural Network Architectures 12
2.4.1 Fully Connected Neural Networks 12
2.4.2 Convolutional Neural Networks 14
2.4.3 Recurrent Neural Networks 16
2.5 Attention-based Networks 19
2.5.1 Transfer Learning 20
2.5.2 Self-Supervised Learning 20
2.5.3 Word Embeddings 21
2.5.4 Self-Attentional Neural Networks and Transformers 23

3 Improving Space Efficiency of Word Embeddings 25

3.1 Introduction 25
3.2 Our Contribution 26
3.3 From Tensor Product Spaces to *word2ket* Embeddings 27
3.3.1 Tensor Product Space 27
3.3.2 Entangled Tensors 28
3.3.3 The *word2ket* Embeddings 29
3.4 Linear Operators in Tensor Product Spaces and *word2ketXS* 32
3.4.1 Linear Operators in Tensor Product Spaces 32
3.4.2 The *word2ketXS* Embeddings 32
3.5 Experimental Evaluation of *word2ket* and *word2ketXS* in NLP Tasks 33

4 Improving Space Efficiency of Transformer Model 40

4.1 Introduction 40

4.2 Limitations of Factorized Matrices in Deep Networks 44
4.3 Stacked Kronecker Product-based Representations 44
4.3.1 Expressiveness of stacked Kronecker-product Layers 45
4.4 Experimental Results 52
4.4.1 Comparison with PHM Layers 53
4.4.2 Comparison with DeLighT 55
4.4.3 Comparison with Standard Low-rank Factorization 56

5 Conclusions 58

Bibliography 60

List of Algorithms

1 Stochastic Gradient Descent 10

LIST OF FIGURES

Figure Page

1 A schematic view of a feedforward neural network (also called Multilayer Perceptron) that maps inputs from $\mathbb{R}^2$ to $\mathbb{R}^2$. Each layer has trainable weights that are adjusted during the training phase using backpropagation algorithm. 13

2 A convolutional neural network with a convolutional layer, a ReLU layer, and a max pooling layer and 3 fully connected layers. In this example, the network maps an input image, a bar chart, to an output vector, representing the probability of each of the classes. 15

3 Architecture of the word2ket (left) and word2ketXS (right) embeddings. The word2ket example depicts a representation of a single-word 256-dimensional embedding vector using rank 5, order 4 tensor $\sum_{k=1}^{5} \bigotimes_{j=1}^{4} v_{jk}$ that uses twenty 4-dimensional vectors v_{jk} as the underlying trainable parameters. The word2ketXS example depicts representation of a full 81-word, 16-dimensional embedding matrix as $\sum_{k=1}^{5} \bigotimes_{j=1}^{4} F_{jk}$ that uses twenty 3×2 matrices F_{jk} as trainable parameters. 31

4 Dynamics of the test-set F1 score on SQuAD dataset using DrQA model with different embeddings: rank-2 order-2 word2ketXS, rank-1 order-4 word2ketXS, and regular embedding. 37

5 Test set questions and answers from DrQA model trained using rank-1 order-4 word2ketXS embedding that utilizes only 380 parameters (four 19×5 matrices F_{jk}, see eq. 3.4) to encode the full, 118,655-word embedding matrix. As each parameter is saved as a single-precision floating-point number, this translates to 1.5 Kilobytes memory usage. Here the True Answers are the manually labeled and the model is able to correctly answer the questions by a F1 measure of 70.65. The two points loss in F1 compared to the regular trained model is hardly noticeable in practice. We did not notice any trends in which types of questions either consistently remained correct or consistently were incorrect when using the word2ket embeddings as compared to the regular embeddings. 39

Abstract

IMPROVING SPACE EFFICIENCY OF DEEP NEURAL NETWORKS

By Aliakbar Panahi

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2021.

Director: Dr. Tomasz Arodz,

Associate Professor, Department of Computer Science

Language models employ a very large number of trainable parameters. Despite being highly overparameterized, these networks often achieve good out-of-sample test performance on the original task and easily fine-tune to related tasks. Recent observations involving, for example, intrinsic dimension of the objective landscape and the lottery ticket hypothesis, indicate that often training actively involves only a small fraction of the parameter space. Thus, a question remains how large a parameter space needs to be in the first place — the evidence from recent work on model compression, parameter sharing, factorized representations, and knowledge distillation increasingly shows that models can be made much smaller and still perform well. Here, we focus on factorized representations of matrices that underpin dense, embedding, and self-attention layers. We use low-rank factorized representation of a reshaped and rearranged original matrix to achieve fast, space efficient, and expressive embeddings and linear layers. We prove that stacking such low-rank layers increases their expressiveness, providing theoretical understanding for their effectiveness in deep networks. Our approach achieves a hundred-fold or more reduction in the space required to store the embeddings with almost no relative drop in accuracy in practical natural language processing tasks. In Transformer models, our approach leads to more than ten-fold reduction in the number of total trainable parameters, including embedding, attention, and feed-forward layers, with little degradation in on-task performance. The approach operates out-of-the-box, replacing each parameter matrix with its compact equivalent while maintaining the architecture of the network.

CHAPTER 1

INTRODUCTION

> *We may hope that machines will eventually compete with men in all purely intellectual fields.*
>
> –Alan Turing, Computing Machinery and Intelligence [1]

Artificial neural networks [2] are among the most successful methods for modeling complex systems. There have been numerous success stories of using neural networks in a diverse set of problems in fields like machine vision, natural language processing, physics, chemistry, medicine, bio-informatics, drug discovery, robotics, and transportation. Although artificial neural networks with their current format existed since the 80s, recent technological advancement in computation power and data acquisition opened the door for harnessing neural networks' immense power. As more data and computation becomes available for training deep networks, their size has grown rapidly: for example, BERT model for natural language understanding has 110 million trainable weights [3]. The need for methods that allow efficient storage of these models becomes critical.

Our work is focused on methods for reducing the space needed to represent a model not only once the model is trained, but also during training. Methods like parameter pruning, knowledge distillation, and quantization are only applicable where we already trained the models. We proposed a novel factorization method using a sum of Kroneckor products for reducing models' size. Though the method would add some extra computation, we hypothesized that the gain in memory decrease is much more valuable than the overhead in computation, and the model would still be able to get a comparable performance. We validated this approach for embedding matrices used in natural language processing (NLP) models and the weight matrix of linear layers in

Transformer [4] models and compared this method with other state-of-the-art methods for reducing the size of neural networks. The results show that the proposed method can achieve up to 100 folds reduction in the size of embedding matrices and up to 6 folds reduction in the size of the NLP models while maintaining the model's performance with less than 50% increase in training or inference time.

The main contributions are as follows:

- Proposed a novel factorization method using a sum of Kroneckor products for reducing size of embedding matrices in deep neural networks.
- Proposed a low-rank factorized representation of a reshaped and rearranged weight matrices to achieve fast, space efficient, and expressive embeddings and linear layers
- Proved that stacking low-rank factorized representation of a reshaped and rearranged weight matrices increases their expressiveness, providing theoretical understanding for their effectiveness in deep networks.
- Proposed a novel model based on transformer architecture, Shapeshifter, that leads to more than ten-fold reduction in the number of total trainable parameters, including embedding, attention, and feed-forward layers, with little degradation in on-task performance

This dissertation is organized as follows. Section 2 gives an introduction to the deep neural networks and attention based networks. Section 3 describes word2ket, our proposed approach for word embeddings with experimental evaluations on text summarization, machine translation, and question answering tasks. Section 4 discusses how factorized representations of matrices can be applied to all layers in Transformer architecture, and proves the expressiveness power of the proposed method. We also experimentally validated this approach by comparing the results with two state-of-the-art methods for reducing the memory of neural networks in machine translation

task. Finally, in Section 5 we summarize the proposed methods, identify the remaining challenges and discuss the path forward.

CHAPTER 2

BACKGROUND

In this chapter, we provide a brief introduction to the main concepts of neural networks. Interested readers can find further details in [2].

2.1 Supervised Learning

Consider the problem of recognizing if an image contains a cat or not. We can formulate this problem and many other similar practical problems as a mapping $f : X \to Y$, where X is the space of inputs, and Y is the space of the outputs. In the cat recognition problem, X is the space of all images, and Y is a number between 0 and 1, showing the probability of a cat being in the image. In most practical cases, it is challenging, if not impossible, to specify function f manually but it is easy to gather pairs of labeled examples $(x, y) \in X \times Y$ for this mapping. In our example, this means we have to collect a dataset of images, and for each image, a "supervisor" labels if they see a cat or not. The goal of supervised learning is to find the mapping f using the input-label pairs.

Supervised Learning Objective. More formally, the objective of supervised learning can be formulated in the following way. Given a *training dataset* of n independent and identically distributed (i.i.d.) samples $\{(x_1, y_1), ...(x_n, y_n)\}$ coming from an unknown distribution D over $X \times Y$ (i.e., $(x_i, y_i) \sim D$ for all i), our goal is to *learn* the mapping $f : X \to Y$. To achieve this goal, we search over a class of functions $\mathcal{F} : X \to Y$, known often as the space of possible hypotheses or space of possible models, and find a function f^* in $\mathcal{F}$ that approximates the unknown function f well. One way to measure how well f^* approximates f is to see if it is good at predicting the labels of the training examples.

Precisely, we select a scalar-valued loss function $L(\hat{y}, y)$ that measures the disagreement between the label y_i and the function output $\hat{y}_i = f(x_i)$ for any $f \in \mathcal{F}$. Our learning objective is to find $f^* \in \mathcal{F}$ that satisfies:

$$f^* = \arg\min_{f \in \mathcal{F}} \mathbb{E}_{(x,y)\sim D} L(f(x), y) \tag{2.1}$$

Once we can find the function f^* we can discard the training samples and only use it to map any elements from X to Y. In our visual recognition example, this means we can use this function on any unlabeled image to get its label.

Unfortunately, we can not evaluate Equation 2.1 or simplify it analytically without making any strong assumptions about the form of D, L, or f as we do not have access to all the elements in D. Since we assumed the samples are i.i.d., we can approximate the expected loss by averaging over the training dataset.

$$f^* = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L(f(x), y) \tag{2.2}$$

This approximation is only optimizing the loss on the training data, but the hope is that it serves as a good proxy to the actual objective in Equation 2.1.

Regularization. Optimizing Equation 2.2 instead of Equation 2.1 can lead to problems. For example, consider a function f that returns zero everywhere but returns corresponding y_i for each x_i in the training data. This function could be a solution to Equation 2.2, and for any sensible loss function, we expect a very high loss for all examples with $y \neq 0$ that are not in our training data. This means that the described function does not *generalize* to all (x, y) pairs. To increase the generalization capability of f, it is common to introduce a scalar-valued function R to the objective function as follows:

$$f^* = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} L(f(x), y) + R(f) \tag{2.3}$$

The new objective term R measures the complexity of functions regardless of their fit on the training data. This can be partly justified by the Occam's razor principle, which states: "given two explanations of the data, all other things being equal, the simpler explanation is preferable" [5]. With the addition of this regularization term, Equation 2.3 results in simple solutions that also fit well with the training data. This regularization term is intended to decrease the disparity between the objective in Equation 2.2 and Equation 2.1.

Linear Regression. To illustrate the formalization of regularized supervised learning outlined above, we focus on linear regression. Linear regression quantifies the relationship between one or more predictor variable(s) and one response variable. Linear regression is also known as multiple regression, multivariate regression, ordinary least squares (OLS), and regression. As a simple example of linear regression, assume a dataset of ($n = 100$) 2-dimensional samples such that each data point is represented by a vector from $X = \mathbb{R}^2$ and annotated with a scalar response value (i.e., $Y = \mathbb{R}$). Informally, linear regression consists of finding the best-fitting straight line (called a regression line) through the points. Linear regression line has an equation of the form of a linear function from X to Y, that is, the set of possible models is $\mathcal{F} = \{w^T x + b \mid w \in \mathbb{R}^2, b \in \mathbb{R}\})$. In this example, our set of hypothesis linear regression line has three trainable parameters (w_1, w_2, b), where $w = [w_1, w_2]$ is coefficients weights, and b is an intercept or bias. Training of the model translates to finding the values for these parameters through an optimization process.

To train the linear regression model, we need is a cost function so that we can start optimizing our weights. The most commonly used cost function for linear regression is mean squared error (MSE) that measures the average squared difference between an observation's actual and predicted values, $L(\hat{y}, y) = (\hat{y} - y)^2$. To avoid the risk of overfitting and increasing model interpretability, we use $R(w, b) = \lambda(w_1^2 + w_2^2)$ as the regularization term to discourage the parameters from being too large. The variable λ

controls the importance of the regularization term and is a hyperparameter - that is, it is chosen prior to training by the user, and not changed during training. When a regularizer is added to the loss function, the full optimization problem is to minimize the objective:

$$f^* = \underset{w_1,w_2,b}{\arg\min} \underbrace{\left[\frac{1}{n}\sum_{i=1}^{n}(w_1 x_{i_1} + w_2 x_{i_2} + b - y_i)^2\right]}_{\text{fit the training data}} + \underbrace{\left[\lambda(w_1^2 + w_2^2)\right]}_{\text{regularization}} \tag{2.4}$$

Interestingly, in most of the classification problems like linear regression, the bias term is ignored while regularizing because overfitting usually requires the model's output to be sensitive to small changes in the input data, and the bias parameters do not contribute to the curvature of the model, so there is usually little point in regularizing them as well.

Neural Network Regression. Neural networks are similar to regression models, except we make the hypothesis function space $\mathcal{F}$ more complex. For example, instead of linear forms for hypothesis, we can use more complex nonlinear functions like $f(x) = w\tanh(V^T x + b_1) + b_2$ where hyperbolic tangent is a nonlinear function that always has an output value in $[-1, 1]$, and the set of $\{V, w, b_1, b_2\}$ are all trainable parameters: V is a matrix of size $H \times 2$, w, and b_1 are vectors of size H, and b_2 has a scalar value. Then, the optimization problem is as follows:

$$f^* = \underset{V,b_1,w,b_2}{\arg\min} \underbrace{\left[\frac{1}{n}\sum_{i=1}^{n}(w\tanh(V^T x + b_1) + b_2 - y_i)^2\right]}_{\text{fit the training data}} + \underbrace{\left[\lambda(\|V\|_2 + \|w\|_2)\right]}_{\text{regularization}} \tag{2.5}$$

2.2 Cross-Entropy Loss

Mean squared error is the typical loss function in regression problems, but is not often used in classification problems, where we are predicting probabilities of the input belonging to one of several classes. Specifically, for a problem with k classes, the space

of targets $\mathcal{Y}$ is a space of k-dimensional vectors that represent probability distribution over k possibilities. One of the most commonly uses loss function in machine learning models and optimization is cross-entropy loss, or log loss. Cross-entropy measures the performance of a classification model whose output is a probability value between 0 and 1. There is a trade off between predicted probability and cross-entropy loss, when the predicted probability approaches 1, loss slowly decreases and if predicted probability decreases then loss increases rapidly. The cross entropy formula takes in two distributions, $p(c)$ and $q(c)$, with $c \in \{1, ..., k\}$ as the true distribution over possible classes c and estimated distribution, respectively and it is defined as follows:

$$H(p, q) = \sum_{\forall c=1,\ldots,k} p(c) \log(q(c)). \tag{2.6}$$

We often normalize the model's output, the k-dimensional vector $f(x)$, to form a distribution over possible k classes, and then directly use $q = f(x)$ as the estimated distribution.

In most cases, the true distribution $p(c)$ assigns full, unit probability to one class (i.e., $p(true_class) = 1$), and null probabilities to all other classes (e.g., the image for sure contains a cat). Then, only one term in the cross-entropy definition is non-zero, and the formula simplifies to $H(p, q) = \log(q(true_class))$, where $q(true_class)$ is the probability assigned by the model for the true class of the sample.

2.3 Optimization

Optimization is at the heart of almost all machine learning techniques. In the last section of the linear regression problem, we ended up with solving an optimization problem of the form $\theta^* = \arg\min_\theta g(\theta)$, where θ is the vector of parameters that can be learned by minimizing loss function and g is the sum of average loss of all training samples and the regularization term. Below we will cover some of the well-known optimization methods.

Derivative-free Optimization. Derivative-free optimization does not use derivative information in the classical sense to find optimal solutions: sometimes, information about the derivative of the objective function f is unavailable, unreliable, or impractical to obtain. For example, f might be non-smooth or defined over a discrete set, so that methods that rely on derivatives are of little use. Instead, we can evaluate $g(\theta)$ for any random values of θ and take the one that minimizes g, and this is a "trial and error" approach to solving this optimization problem. Another method is to give perturbations to θ and iteratively monitor the loss $g(\theta)$. However, this "guess-and-check" approach and perturbing weights approach are computationally intractable because we usually train neural networks with hundreds of thousands or millions of parameters.

First-order methods. First-order method is the most popular nowadays, and is suitable for large-scale data optimization due to relatively low computational cost of calculating first-order derivatives. Instead of the naive method like derivative-free optimization, we can improve optimization time by changing some assumptions about g to find the optimal value for parameters. For instance, if we assume g to be differentiable, we can compute gradient ($\nabla_\theta g$), the vector of partial derivatives, giving us the slope of g with respect to parameters θ. The gradient allows us to construct the first-order approximation in the Taylor expansion of g, and use it to change the parameters in a way that minimizes the approximation.

The gradient vector can be interpreted as the direction of locally steepest growth of the function. Thus, we can improve θ by adding to it a small amount of the negative gradient direction. This process is continued until we reach the value of θ for which the loss function is locally minimum. This gradient descent (GD) algorithm alternates the two steps: 1) evaluate the gradient with backpropagation (with respect to parameters at the current point) and 2) update the parameters by taking a small step in the direction opposite to the gradients. To make GD practically feasible for very large datasets, we divide samples to minibatch of data at a given time and then compute the gradients, and

then update the parameters. The resulting algorithm is Stochastic Gradient Descent (SGD) which is summarized in Algorithm 1.

Algorithm 1 Stochastic Gradient Descent

Given a starting point $\theta \in \mathbf{dom} g$

Given a step size $\epsilon \in \mathbb{R}^+$

repeat

1. Sample a minibatch of m examples $\{(x_1, x_2), ..., (x_m, x_m)\}$ from training data
2. Estimate the gradient $\nabla_\theta g(\theta)$ with backpropagation
3. Compute the update direction: $\Delta\theta := -\epsilon \nabla_\theta g(\theta)$
4. Update the parameter: $\theta := \theta + \Delta\theta$

until convergence

One of the most critical hyperparameters in the SGD algorithm is step size ϵ, also called *learning rate*. This hyperparameter controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging. If learning rate is set too low, training will progress very slowly as the model is making tiny updates to the weights. However, if learning rate is set too high, it can cause undesirable divergent behavior in the loss function and an unstable training process.

A simple example illustrates how learning rate can affect the optimizing process: consider the function $y = x^2$ and $\frac{dy}{dx} = 2x$ as its gradient with respect to x. Starting from $x = 1$ and $\epsilon > 1$ will cause the gradient descent update to oscillate and diverge to infinity. On the other hand, $\epsilon = 1$ will make the algorithm wobble between $x = 1$ and $x = -1$. Only if $\epsilon < 1$ it will cause to converge to the optimal minimum value, $x = 0$. Unfortunately, in practice an optimal learning rate cannot be analytically calculated for a given model on a given dataset. Instead, a good enough learning rate must be discovered via trial and error, or a good heuristic method is to binary search to find the lowest setting of the learning rate that makes the optimization diverge. Typical values

for a neural network with standardized inputs are less than 1 and greater than 10^{-6}, but we can not rely exclusively on this default value. There are several ways to automatically pick hyper-parameters, such as grid search and Population-based training. However, gradually decaying the learning rate over time is a good rule of thumb, which can be achieved by multiplying the learning rate by a small factor. The other option is Polyak Averaging, an optimization technique that sets final parameters to an average of recently updated parameters. Precisely, for parameters $\theta_1, ..., \theta_t$ in t iterations Polyak Averaging suggests setting $\theta_t = \frac{1}{t}\sum_i \theta_i$. Learning the hyperparameters of different prediction functions and testing them on the same data depends on the problem, so it is common to evaluate the learning rate performance using the cross-validation, a technique we discuss below.

Backpropagation. Neural networks could learn their hyperparameters using gradient descent algorithm. However, we did not discuss how to compute the gradient of the cost function and train the model. This training is usually associated with the term *backpropagation*, which essentially refers to chain rule of differentiation. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the model's parameters. As we explained, we are interested in computing the gradient of the cost function $g(\theta)$ ($\frac{dg}{d\theta}$ or $(\nabla_\theta g)$) with respect to the parameter θ. Also, we can compute the gradients for the inputs x_i with the same process.

By using the chain rule, evaluating the gradient of the output with respect to the input reduces to a product of Jacobian matrices. In neural networks, we first need to do a *forward pass* in which we take a mini-batch $\{(x_i, y_i)\}_{i=1}^m$ and current parameters of the network θ and calculate all intermediate values (saving them for later use) and the cost g. Then we do the *backward pass* in which we *chain* the local gradients by multiplying the next Jacobian matrix in the full product.

Cross-validation. Once we are done with training our model, we need some assurance of the accuracy of the predictions that our model is putting out. Cross-validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model and a test set to evaluate it. The most common variant of cross-validation is the k-fold cross-validation. The procedure has a single parameter called k that refers to the number of groups that a given data set is to be split. The model is trained using $k-1$ groups and the one remaining group is denoted as the validation set to evaluate the trained model. This process is repeated k times, with each of the k groups serving as the validation set. The average of their performance produces cross-validated performance.

2.4 Neural Network Architectures

In the previous sections, we explained how to define a differentiable cost function f to map each data point x in input to a response value $\hat{y}$ and then how to optimize the model using gradient descent. We now turn to the details of the function f, which we have so far left unexplained. We first introduce the basic architecture of fully-connected feed-forward neural networks, and then describe two specialized architectures: convolutional and recurrent networks.

2.4.1 Fully Connected Neural Networks

Neural networks are composed of input, hidden, and output layers; the input layer is the first, representing data that is fed into the network; the hidden layers include the "neurons" that adjust the weight and bias for each number of times machine learning model pass through the entire dataset; the last is the output layer where we receive a response from our model, based on the specified task (see figure 1). The network is fully connected, that is, each neuron receives input from all neurons from the preceding layer. As a simple example, a 2-layer neural network would be implemented

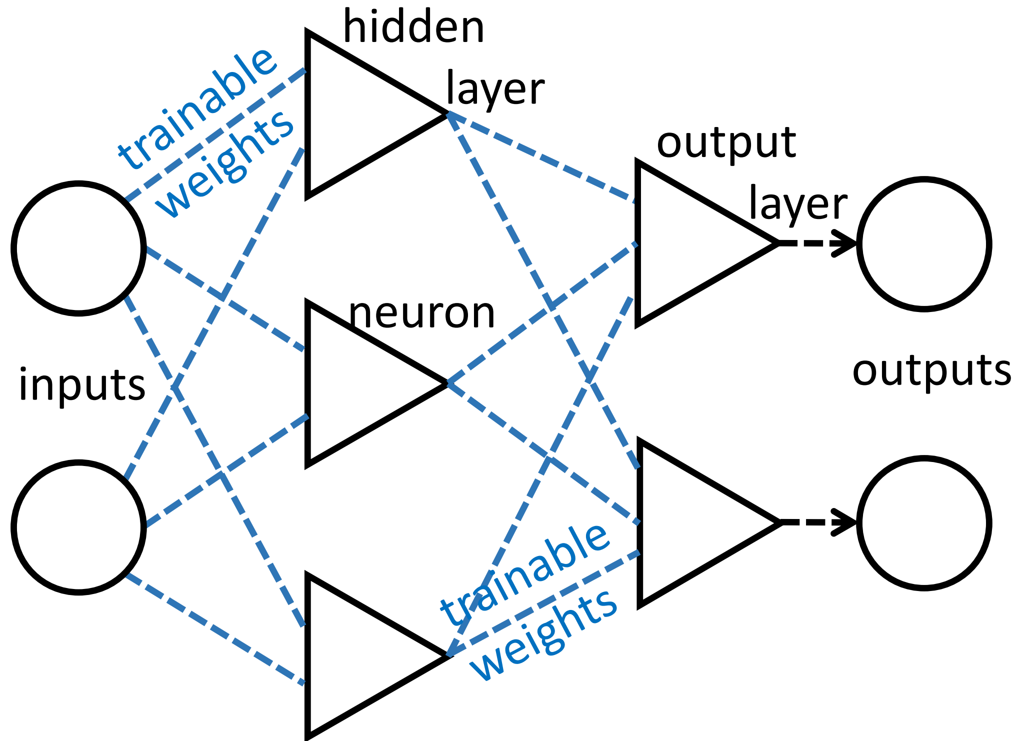

Figure 1: A schematic view of a feedforward neural network (also called Multilayer Perceptron) that maps inputs from $\mathbb{R}^2$ to $\mathbb{R}^2$. Each layer has trainable weights that are adjusted during the training phase using backpropagation algorithm.

as $f(x) = W_2\sigma(W_1x)$, where W_1, W_2 are matrices and σ is an element-wise non-linearity function. Historically, frequently used for the non-linearities were sigmoid functions $tanh$ or $\frac{1}{1+e^{-x}}$. More recently, the rectified linear unit (ReLU) defined as $max(0, x)$ is the standard choice. Note that the last layer of the neural network normally does not contain the non-linearity. Also, note that a 1-layer neural network is a simple linear function.

Biological inspiration. Although neural networks are very far from real biological neural networks, a crude model of biological neurons was the motivation behind neural network. Each row of the parameters corresponds to one neuron and its synaptic connection strengths to its inputs. The negative weights are inhibitory, the positive

weights relate to excitation, and a zero represents no dependence. After some computation process, the weighted sum of the inputs connects to a cell body, and then an activation function like sigmoid produce an output between $[0, 1]$, so it identifies the firing neurons. In the human body, the connection between two neurons represents weights in a computing neural network, and it helps to transfer activation signal to neurons in other layers.

2.4.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a class of deep neural networks used for inputs that form a regular lattice. The input x is a multi-dimensional array; such arrays are referred to in deep learning as a tensor. It can be an image, video or text. For example, a 256×256 color image is a $256 \times 256 \times 3$ tensor (for 3 color channels red, green, blue). A 112-character sentence could be represented on a character level as a 112×30, indicating which of 30 possible characters occupies any one of 112 positions in the sentence. Usually, the dimensionality of input is high and using fully connected layers is not efficient – the number of possible weights to be trained is very large and the network is prone to overfitting. Therefore, neural network architectures can be designed using specific local connectivity and parameter sharing schemes.

CNN are made of neurons that have trainable weights that describe filters, which are applied to various parts of the image. As shown in Figure 2, a single filter/neuron is applied not to all inputs as in a fully-connected network, but only to a small subset of inputs – an image region that matches the size of the filter. For example, a two-by-two filter only focuses on a two-by-two patch of the image, with one trainable weight in the filter corresponding to one pixel in the image patch. The same single filter is applied, in parallel, to all possible locations in the image, using the same set of trainable weights to produce multiple outputs. Essentially, the output of applying a single filter to an image is also an image, but altered by the filter. We can view a single filter as equivalent to a

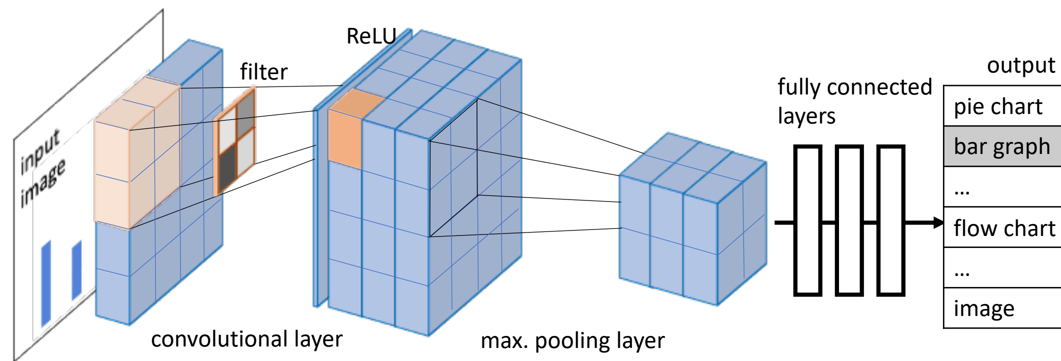

Figure 2: A convolutional neural network with a convolutional layer, a ReLU layer, and a max pooling layer and 3 fully connected layers. In this example, the network maps an input image, a bar chart, to an output vector, representing the probability of each of the classes.

large number of neurons, each producing one pixel of the output, but instead of each neuron having its own set of weights for its connections to pixels in the input, these weights are shared. This drastically reduces the number of trainable parameters, and can prevent overfitting.

Edge Detection Example. Suppose we are given an image with many vertical and horizontal edges. The first question is, how do we detect these edges? To answer this question, let us take a 6×6 grayscale image with only one channel. Then, we convolve this 6×6 matrix with a 3×3 filter, by doing the convolution, we will get a 4×4 image. Higher pixel values represent the brighter portion of the image, and the lower pixel values represent the darker portions. The pixel values help to detect a vertical edge in an image. The values of the filter's matrix help to detect the vertical or horizontal edges. There are some commonly used filters to detect different edges, such as Sobel or Scharr filters. In the convolutional network, instead of relying on a pre-defined filter design, the matrix defining the filter is trained from data. The weights in the filter matrix are treated as parameters which the model will learn using backpropagation.

A simple neural network can detect the edges of an image at the early layers of the network. Then, subsequent layers might be able to detect the cause of the objects, and even deeper layers might detect the cause of complete objects.

CNN architecture is built by stacking three types of layers, including convolutional layers and possibly introducing pooling layers that are generally used to reduce the size of the inputs and hence speed up the computation. A series of convonlutional and pooling layers is typically followed by several fully connected layers. A typical CNN architecture that processes images might take the structure $[INPUT, [CONV, CONV, POOL] \times 3, FC, FC]$. Here, INPUT represents a tensor of a batch of images and CONV is a convolutional layer with 3×3 filters applied with the padding of 1 and stride of 1, POOL stands for a typical 2×2 filter max pooling layer with a stride of 2, and FC the last one computes the logits of different classes just before a softmax classifier.

2.4.3 Recurrent Neural Networks

Recurrent neural networks, also known as RNNs, are a class of artificial neural networks that allow previous outputs to be used as inputs while having hidden states, and the connections between nodes form a directed graph along a temporal sequence. While fully-connected and convolutional layers expect inputs of a fixed size, RNNs can use their internal memory to process variable-length sequences of inputs. This makes them applicable to many NLP tasks like commonsense reasoning, question/answering problems, and many others.

An RNN processes a sequence of vectors $\{x_1, ..., x_T\}$ using a recurrence formula of the form $h_t = f_\theta(h_{t-1}, x_t)$, where f and θ are a function and parameters, respectively. They are used at every time step and help to process sequences with an arbitrary number of vectors. f_θ is a connectivity pattern that can process the sequence of a vector with arbitrary length. The hidden vector h_t can be interpreted as a running summary of all vectors x until the current time step, and the recurrence formula updates the summary

based on the next vector ($x_{t\prime}$). For initialization, we can either treat h_0 as vector parameters and learn the starting hidden state or set $h_0 = \vec{0}$. The mathematical form of the recurrence $(h_{t-1}, x_t) \rightarrow h_t$ varies depending on the specific network architecture. Some of the formula in the computation process of RNN can be:

1. x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the first word of a sentence.

2. h_t is the hidden state at time step t. It is the "memory" of the network. h_t is calculated based on the previous hidden state and the input at the current step: $h_t = a(Ux_t + Wh_{t-1})$. The function a usually is a nonlinearity such as tanh or ReLU. The first hidden state, is typically initialized to all zeroes.

3. o_t is the output at step t. In some architectures $o_t = h_t$, but more generally these two can differ, with $o_t = softmax(Vh_t)$. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary.

RNN models are mostly used in the fields of natural language processing (NLP) and speech recognition. Different types of RNNs are applied for various applications. For example, one-to-one RNN is referred to as a traditional neural network, one-to-many RNN is mostly used for music generation; many-to-one RNN has high performance in sentiment classification, and many-to-many RNN is widely applied in name entity recognition and machine translation task. The advantages and disadvantages of a typical RNN architecture are summed up as follow:

Advantages

- Possibility of processing input of any length

- Model size not increasing with the size of the input

- Computation takes into account historical information
- Weights are shared across time

Drawbacks

- Computation being slow
- Difficulty of accessing information from a long time ago
- Cannot consider any future input for the current state

Long Short-Term Memory. Despite the incredible success of applying RNNs to a variety of problems, there are problems with Long-Term Dependencies and RNNs become unable to learn to connect the information. However, a special kind of RNN, called Long Short-Term Memory (LSTM) does not have this problem. LSTMs are explicitly designed to avoid the long-term dependency problem, and they remember information for long periods of time. Like any RNNs, LSTMs have a chain of repeating modules of the neural network, but the repeating module has a specific structure. Instead of having a single neural network layer, there are four that are interacting differently. Rather than go into the equations that control how LSTMs work, we explain operations within the LSTM's cells to quickly show how they work step by step.

The core concept of LSTM's are the cell state and its various gate units. The cell state is the memory of the network and transfers relative information down the sequence chain. Cell state reduces the side effect of short-term memory, and information from the earlier time steps can access later time steps. So, information is added or removed to the cell state via gates. The gates can learn what information is relevant to keep or forget during training. The next operation is gates that contain unipolar sigmoid activation function. As we mentioned before, sigmoid squishes values between 0 and 1. Zero value causes some values to be forgotten because multiplication by 0 is 0, and some

information is kept when values are multiplied by 1. Therefore, the sigmoid function helps the network to learn which data are more important than the others.

The other gates regulate information flow in an LSTM cell. Forget gate decides what information should be thrown away or kept. It is straightforward, and information from the previous hidden state and current input is passed through the sigmoid function. As we expect, values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep. The next is the input gate in order to update the cell state after every change. Bypassing the previous hidden state and current input into a sigmoid function, it decides which values will be updated. Null means throw away information, and 1 means keep them. Also, tanh function gets the hidden state and current input and gives us values between -1 and 1; then multiplication of tanh output with the sigmoid output helps to decide which information is essential to keep from the tanh output. Now it is time to calculate the new cell state with achieved information. The cell state gets pointwise multiplied by the forget vector. It drops values in the cell state if it gets multiplied by values near 0. Then we take the input gate's output and do a pointwise addition that updates the cell state to new values that the neural network finds relevant. The last is the output gate that decides what the next hidden state is. So we pass the previous hidden state and the current input into a sigmoid function. We pass the newly modified cell state to the tanh function. By multiplying the tanh output with the sigmoid output, we decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.

2.5 Attention-based Networks

Recently, an approach that relies on attention mechanism and a combination of transfer and self-supervised learning has been shown to outperform RNN-based networks. Below, we describe the components behind this approach in more detail.

2.5.1 Transfer Learning

Transfer learning is a problem in machine learning where a trained model on a large dataset for a specific task is used and applied to a different but related task. Typically, the second task the the one that the user is interested in, but scarcity of training data prevents training the network towards that task directly. Transfer learning focuses on storing achieved knowledge from the the related, data-rich task and then using that information as a starting point in training the network towards the desired task. Using a pre-trained model is a popular approach in deep learning as a starting point instead of designing neural network layers from scratch to learn features. Transfer learning was a game-changer in computer vision and natural language processing tasks.

2.5.2 Self-Supervised Learning

Deep learning can be applied to different learning paradigms, including supervised learning, reinforcement learning, as well as unsupervised or self-supervised learning. As we discussed, supervised learning is the category of machine learning algorithms that require annotated training data, which is one the limitations of deep learning. However, unsupervised learning is not easy and usually works much less efficiently than supervised learning. We can modify a supervised learning task to hide some parts of the input x and train the system to predict it, instead of requiring a separate label y. This is known as self-supervised learning. The idea behind self-supervised learning is to develop a deep learning system that can learn to fill in the blanks and buy doing so, learn to understand the structure of inputs it is being trained on. Once this is achieved, the model can be fine tuned towards a specific supervised learning task by training it on a much small dataset of labeled pairs (x, y). Therefore, self-supervised learning is typically used together with transfer learning to increased performance without needing a huge amount of labeled data.

Real-World Applications of Self-Supervised Learning. Many ideas have been proposed for self-supervised representation learning on images. One of the typical applications is to train a model on one or more pretext tasks with unlabeled images and then use one of the model's features layers as inputs of the logistic regression classifier on image classification task. The accuracy of the final model tells us the performance of the learned model. Another popular used self-supervised learning systems are the so-called "Transformers" based on the idea of attention, used primarily in natural language processing and designed to handle sequential data without the need for large labeled datasets. In the next section, we discuss word embeddings that form the basis of modern natural language processing networks, and then follow up with the description of Transformers in more details along with Self-Attentional Neural Networks.

2.5.3 Word Embeddings

Word embedding is a method to capture the meaning of the word, some of the semantics of the input, and represent it as a vector, so that it can be easily provided as input to the neural network. The most basic method for word representation is one-hot encoding, which encodes each word in the vocabulary by creating a zero vector with length equal to the vocabulary, then placing a one in the index that corresponds to the word. Despite its simple approach, one-hot encoding has limitations: the dimensionality of the transformed vector unmanageable in high-cardinality variables (e.g. for NLP, the length of the vector would have to be equal to the number of words in the English language) and also "similar" categories are not placed closer to each other in embedding space.

A more convenient way to represent words using vectors is to use low dimension real-valued vectors, with words that have the same or nearly the same meaning represented with a similar vector. Machine learning models take the advantages of such word embedding techniques to work with huge amount of input data like sparse vectors

representing words. On most of the challenging NLP tasks, word embedding play a key role for the impressive performance of deep learning methods. The basis technique to learn a word embedding from text data is Word2Vec, a statistical method using probabilities of groups of words for efficiently learning a standalone word embedding from a text corpus. For example, the male/female relationship is automatically learned, and with the induced vector representations, "King – Man + Woman" results in a vector very close to "Queen." [6] . Word2Vec can be obtained using two algorithms: Skip Gram model and Common Bag Of Words (CBOW). Both models are shallow neural networks and focus on learning about words given their neighborhood context, where the context is defined by a window over the sequence of words. This size of the window is a configurable parameter of the model. The CBOW model learns the embedding by predicting the next word based on previously seen context, however the skip-gram model learns by predicting the surrounding words given a current word. CBOW's memory usage is low because it does not need to store co-occurrence matrix and it performs superior in deterministic methods. Although, CBOW takes the average of the context of a word before predicting the center word and also training a CBOW can take very long if not properly optimized. On the other hand, in skip-gram model there is no averaging of embedding vectors so it can capture two semantics for a single word, therefore it needs more data so will learn to understand even rare words.

Word embeddings can also be constructed using GloVe (Global Vectors for Word Representation), an extension to the word2vec method for efficiently learning word vectors. GloVe has the best of both worlds of global statistics of matrix factorization techniques like LSA and the local context-based learning in word2vec. Training process in GloVe is performed on aggregated global word-word co-occurrence statistics from a corpus based on matrix factorization techniques on the word-context matrix. The result is a learning model that may result in generally better word embeddings in word analogy, word similarity, and named entity recognition tasks[7]. Also, GloVe adds some

more practical meaning into word vectors by considering the relationships between word pair, however, as the model is trained on the co-occurrence matrix of words it takes a lot of memory for storage.

2.5.4 Self-Attentional Neural Networks and Transformers

The paper 'Attention Is All You Need' [4] describes transformers, and a sequence-to-sequence architecture using them. Sequence-to-Sequence (or Seq2Seq) is a neural network that transforms a given sequence of words into another sequence, for example provides a summary sentence for an article. Before the introduction of Transformers and self-attention mechanism, most state-of-the-art NLP systems relied on RNNs, LSTMs and gated recurrent units (GRUs). However, these models are sensitive to the length of sentences so that when sentences are too long, they do not perform too well. The Transformer built on the attention mechanism for learning to focus on specific, potentially distant words in the sequence without using an RNN architecture. The idea behind self-attention the neural network is that in interpreting the information contained in a single specific word in the sentence, there might be relevant information in every other word in a sentence.

Self-attentional neural networks or Seq2Seq models consist of an Encoder and a Decoder. The Encoder takes the input sequence encoded using a pre-defined word embedding approach, and maps it into an n-dimensional output vector. In that sense, the encoded can be seen as a more elaborate way to provide word embeddings. That vector resulting from the Encoder is fed into the Decoder, which turns it into an output sequence. The output sequence can, for example, be in another language. The attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important. In other words, for each input that the Encoder receives, the attention-mechanism takes into account several other inputs at the same time and decides which ones are important by attributing different weights to those

inputs. Then the Decoder will take the encoded sentence and the weights provided by the attention-mechanism.

A transformer is a specific model that implements self-attention approach in an efficient manner. The Transformer consists of six encoders and six decoders. All encoders have the same architecture, and decoders share the same property. Each encoder consists of two parts: Self-attention layer and a small number of full-connected layers. Encoders and decoders can be stacked on top of each other multiple times. The inputs and outputs (target sentences) are first embedded in an n-dimensional space. The self-attention layer first takes the encoder's inputs, so during the encoding of a specific word, the encoder can see and have information about other words in the input sentence. The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence. After the multi-attention heads in both the encoder and decoder, there is a feed-forward layer. This feed-forward network has identical parameters for each position, described as a separate, identical linear transformation of each element from the given sequence.

The Transformers are behind Google's BERT and T5, OpenAI's GPT2 and GPT3, Facebook's RoBERTa, XLNet, and ALBERT. They outperform many other predecessors at different NLP tasks like Winograd Schema Challenge, answering questions, understanding human linguistics, machine translation and time series prediction.

CHAPTER 3

IMPROVING SPACE EFFICIENCY OF WORD EMBEDDINGS

3.1 Introduction

Modern deep learning approaches for natural language processing (NLP) often rely on vector representation of words to convert discrete space of human language into continuous space best suited for further processing through a neural network. For a language with vocabulary of size d, a simple way to achieve this mapping is to use one-hot representation – each word is mapped to its own row of a $d \times d$ identity matrix. There is no need to actually store the identity matrix in memory, it is trivial to reconstruct the row from the word identifier. Word embedding approaches such as word2vec [8] or GloVe [7] use instead vectors of dimensionality p much smaller than d to represent words, but the vectors are not necessarily extremely sparse nor mutually orthogonal. This has two benefits: the embeddings can be trained on large text corpora to capture the semantic relationship between words, and the downstream neural network layers only need to be of width proportional to p, not d, to accept a word or a sentence. We do, however, need to explicitly store the $d \times p$ embedding matrix in GPU memory for efficient access during training and inference. Vocabulary sizes can reach $d = 10^5$ or 10^6 [7], and dimensionality of the embeddings used in current systems ranges from $p = 300$ [8, 7] to $p = 1024$ [3]. The $d \times p$ embedding matrix thus becomes a substantial, often dominating, part of the parameter space of a learning model. Our goal is to explore approaches to reduce the size of the embedding matrix.

Given the current hardware limitation for training and inference, it is crucial to be able to decrease the amount of memory these networks requires to work. A number of approaches have been used in lowering the space requirements for word embeddings.

Dictionary learning [9], word embedding clustering [10] and Bit encoding [11] are among some of the approaches that have been proposed. An optimized method for uniform quantization of floating point numbers in the embedding matrix has been proposed recently by May et al[12]. To compress a model for low-memory inference, Han et al. [13] used pruning and quantization for lowering the number of parameters. For low-memory training, methods like sparsity [14] [15] [16] and low numerical precision [17] [18] have been proposed. In approximating matrices in general, Fourier-based approximation methods have also been used [19, 20]. None of these approaches can match the space saving factors achieved by word2ketXS. The methods based on bit encoding, such as the one proposed by Andrews et al. [10], Gupta et al. [11], and May et al. [12] are limited to space saving factor of at most 32 for 32-bit architectures. Other methods, for example based on parameter sharing [21] or based on PCA, can offer higher saving factors, but their storage requirement is limited by $d + p$, the vocabulary size and embedding dimensionality. In more distantly related work, tensor product spaces have been used in studying document embeddings, by using sketching of a tensor representing n-grams in the document [22].

3.2 Our Contribution

Here, we propose tensor product-based methods, [1] *word2ket* and *word2ketXS*, for storing word embedding matrix during training and inference in a highly efficient way. The first method operates independently on the embedding of each word, allowing for more efficient processing, while the second method operates jointly on all word embeddings, offering even higher efficiency in storing the embedding matrix, at the cost of more complex processing. Empirical evidence from three NLP tasks shows that the new *word2ket* embeddings offer high space saving factor at little cost in terms of accuracy of the downstream NLP model.

[1] PyTorch implementation available at `https://github.com/panaali/word2ket`

3.3 From Tensor Product Spaces to *word2ket* Embeddings

3.3.1 Tensor Product Space

Consider two separable[2] Hilbert spaces[3] $\mathcal{V}$ and $\mathcal{W}$. A *tensor product space* of $\mathcal{V}$ and $\mathcal{W}$, denoted as $\mathcal{V} \otimes \mathcal{W}$, is a separable Hilbert space $\mathcal{H}$ constructed using ordered pairs $v \otimes w$, where $v \in \mathcal{V}$ and $w \in \mathcal{W}$. In the tensor product space, the addition and multiplication in $\mathcal{H}$ have the following properties

$$\begin{aligned} c\left\{v \otimes w\right\} &= \left\{cv\right\} \otimes w = v \otimes \left\{cw\right\}, \\ v \otimes w + v' \otimes w &= \left\{v + v'\right\} \otimes w, \\ v \otimes w + v \otimes w' &= v \otimes \left\{w + w'\right\}. \end{aligned} \tag{3.1}$$

The inner product between $v \otimes w$ and $v' \otimes w'$ is defined as a product of individual inner products

$$\langle v \otimes w, v' \otimes w' \rangle = \langle v, v' \rangle \langle w, w' \rangle. \tag{3.2}$$

It immediately follows that $\| v \otimes w \| = \| v \| \; \| w \|$; in particular, a tensor product of two unit-norm vectors, from $\mathcal{V}$ and $\mathcal{W}$, respectively, is a unit norm vector in $\mathcal{V} \otimes \mathcal{W}$. The Hilbert space $\mathcal{V} \otimes \mathcal{W}$ is a space of equivalence classes of pairs $v \otimes w$; for example $\{cv\} \otimes w$ and $v \otimes \{cw\}$ are equivalent ways to write the same vector. A vector in a tensor product space is often simply called a *tensor*.

Let $\{\psi_j\}$ and $\{\phi_k\}$ be orthonormal basis sets in $\mathcal{V}$ and $\mathcal{W}$, respectively. From eq.

[2] That is, with countable orthonormal basis.

[3] We use here finite-dimensional Hilbert spaces over real numbers, that is real vector spaces endowed with an inner product.

3.1 and 3.2 we can see that

$$\left\{\sum_j c_j \psi_j\right\} \otimes \left\{\sum_k d_k \phi_k\right\} = \sum_j \sum_k c_j d_k \psi_j \otimes \phi_k,$$

$$\langle \psi_j \otimes \phi_k, \psi_{j'} \otimes \phi_{k'} \rangle = \delta_{j-j'} \delta_{k-k'},$$

where δ_z is the Kronecker delta, equal to one at $z = 0$ and to null elsewhere. That is, the set $\{\psi_j \otimes \phi_k\}_{jk}$ forms an orthonormal basis in $\mathcal{V} \otimes \mathcal{W}$, with coefficients indexed by pairs jk and numerically equal to the products of the corresponding coefficients in $\mathcal{V}$ and $\mathcal{W}$. We can add any pairs of vectors in the new spaces by adding the coefficients. The dimensionality of $\mathcal{V} \otimes \mathcal{W}$ is the product of dimensionalities of $\mathcal{V}$ and $\mathcal{W}$.

We can create tensor product spaces by more than one application of tensor product, $\mathcal{H} = \mathcal{U} \otimes \mathcal{V} \otimes \mathcal{W}$, with arbitrary bracketing, since tensor product is associative. Tensor product space of the form

$$\bigotimes_{j=1}^{n} \mathcal{H}_j = \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \ldots \otimes \mathcal{H}_n$$

is said to have tensor *order*[4] of n.

3.3.2 Entangled Tensors

Consider $\mathcal{H} = \mathcal{V} \otimes \mathcal{W}$. We have seen the addition property $v \otimes w + v' \otimes w = \{v + v'\} \otimes w$ and similar property with linearity in the first argument – tensor product is bilinear. We have not, however, seen how to express $v \otimes w + v' \otimes w'$ as $\phi \otimes \psi$ for some $\phi \in \mathcal{V}$, $\psi \in \mathcal{W}$. In many cases, while the left side is a proper vector from the tensor product space, it is not possible to find such ϕ and ψ. The tensor product space contains not only vectors of the form $v \otimes w$, but also their linear combinations, some of which cannot be expressed as $\phi \otimes \psi$. For example, $\sum_{j=0}^{1} \sum_{k=1}^{1} \frac{\psi_j \otimes \phi_k}{\sqrt{4}}$ can be decomposed as $\left\{\sum_{j=0}^{1} \frac{1}{\sqrt{2}} \psi_j\right\} \otimes \left\{\sum_{k=1}^{1} \frac{1}{\sqrt{2}} \phi_k\right\}$. On the other hand, $\frac{\psi_0 \otimes \phi_0 + \psi_1 \otimes \phi_1}{\sqrt{2}}$ cannot be decomposed

[4]Note that some sources alternatively call n a degree or a rank of a tensor. Here, we use tensor rank to refer to a property similar to matrix rank, see below.

as a tensor product of two matrices; no matter what we choose as coefficients a, b, c, d, we have

$$\frac{1}{\sqrt{2}}\psi_0 \otimes \phi_0 + \frac{1}{\sqrt{2}}\psi_1 \otimes \phi_1 \neq (a\psi_0 + b\psi_1) \otimes (c\phi_0 + d\phi_1)$$
$$= ac\psi_0 \otimes \phi_0 + bd\psi_1 \otimes \phi_1 + ad\psi_0 \otimes \phi_1 + bc\psi_1 \otimes \phi_0,$$

since we require $ac = 1/\sqrt{2}$, that is, $a \neq 0$, $c \neq 0$, and similarly $bd = 1/\sqrt{2}$, that is, $b \neq 0$, $c \neq 0$, yet we also require $bd = ad = 0$, which is incompatible with $a, b, c, d \neq 0$.

For tensor product spaces of order n, that is, $\bigotimes_{j=1}^{n} \mathcal{H}_j$, tensors of the form $v = \bigotimes_{j=1}^{n} v_j$, where $v_j \in \mathcal{H}_j$, are called *simple*. Tensor *rank*[5] of a tensor v is the smallest number of simple tensors that sum up to v; for example, $\frac{\psi_0 \otimes \phi_0 + \psi_1 \otimes \phi_1}{\sqrt{2}}$ is a tensor of rank 2. Tensors with rank greater than one are called *entangled*. Maximum rank of a tensor in a tensor product space of order higher than two is not known in general [23].

3.3.3 The *word2ket* Embeddings

A p-dimensional word embedding model involving a d-token vocabulary is[6] a mapping $f : [d] \to \mathbb{R}^p$, that is, it maps word identifiers into a p-dimensional real Hilbert space, an inner product space with the standard inner product $\langle \cdot, \cdot \rangle$ leading to the L_2 norm. Function f is trained to capture semantic information from the language corpus it is trained on, for example, two words i, j with $\langle f(i), f(j) \rangle \sim 0$ are expected to be semantically unrelated. In practical implementations, we represent f as a collection of vectors $f_i \in \mathbb{R}^p$ indexed by i, typically in the form of $d \times p$ matrix M, with embeddings of individual words as rows.

We propose to represent an embedding $v \in \mathbb{R}^p$ of each a single word as an entangled

[5]Note that some authors use rank to denote what we above called order. In the nomenclature used here, a vector space of $n \times m$ matrices is isomorphic to a tensor product space of order 2 and dimensionality mn, and individual tensors in that space can have rank of up to $\min(m, n)$.

[6]We write $[d] = \{0, ..., d\}$.

tensor. Specifically, in *word2ket*, we use tensor of rank r and order n of the form

$$v = \sum_{k=1}^{r} \bigotimes_{j=1}^{n} v_{jk}, \tag{3.3}$$

where $v_{jk} \in \mathbb{R}^q$. The resulting vector v has dimension $p = q^n$, but takes $rnq = O(rq \log p/q)$ space. We use $q \geq 4$; it does not make sense to reduce it to $q = 2$ since a tensor product of two vectors in $\mathbb{R}^2$ takes the same space as a vector in $\mathbb{R}^4$, but not every vector in $\mathbb{R}^4$ can be expressed as a rank-one tensor in $\mathbb{R}^2 \otimes \mathbb{R}^2$.

If the downstream computation involving the word embedding vectors is limited to inner products of embedding vectors, there is no need to explicitly calculate the q^n-dimensional vectors. Indeed, we have (see eq. 3.2)

$$\langle v, w \rangle = \langle \sum_{k=1}^{r} \bigotimes_{j=1}^{n} v_{jk}, \sum_{k'=1}^{r} \bigotimes_{j=1}^{n} w_{jk'} \rangle = \sum_{k,k'=1}^{r,r} \prod_{j=1}^{n} \langle v_{jk}, w_{jk'} \rangle.$$

Thus, the calculation of inner product between two p-dimensional word embeddings, v and w, represented via *word2ket* takes $O(r^2 q \log p/q)$ time and $O(1)$ additional space.

In most applications, a small number of embedding vectors do need to be made available for processing through subsequent neural network layers – for example, embeddings of all words in all sentences in a batch. For a batch consisting of b words, the total space requirement is $O(bp + rq \log p/q)$, instead of $O(dp)$ in traditional word embeddings.

Reconstructing a b-word batch of p-dimensional word embedding vectors from tensors of rank r and order n takes $O(brpn)$ arithmetic operations. To facilitate parallel processing, we arrange the order-n tensor product space into a balanced tensor product tree (see Figure 3), with the underlying vectors v_{jk} as leaves, and v as root. For example, for $n = 4$, instead of $v = \sum_k ((v_{1k} \otimes v_{2k}) \otimes v_{3k}) \otimes v_{4k}$ we use $v = \sum_k (v_{1k} \otimes v_{2k}) \otimes (v_{3k} \otimes v_{4k})$. Instead of performing n multiplications sequentially, we can perform them in parallel along branches of the tree, reducing the length of the sequential processing to $O(\log n)$.

Typically, word embeddings are trained using gradient descent. The proposed

embedding representation involves only differentiable arithmetic operations, so gradients with respect to individual elements of vectors v_{jk} can always be defined. With the balanced tree structure, *word2ket* representation can be seen as a sequence of $O(\log n)$ linear layers with linear activation functions, where n is already small. Still, the gradient of the embedding vector v with respect to an underlying tunable parameters v_{lk} involves products $\partial \left(\sum_k \prod_{j=1}^n v_{jk} \right) / \partial v_{lk} = \prod_{j \neq l} v_{jk}$, leading to potentially high Lipschitz constant of the gradient, which may harm training. To alleviate this problem, at each node in the balanced tensor product tree we use LayerNorm [24].

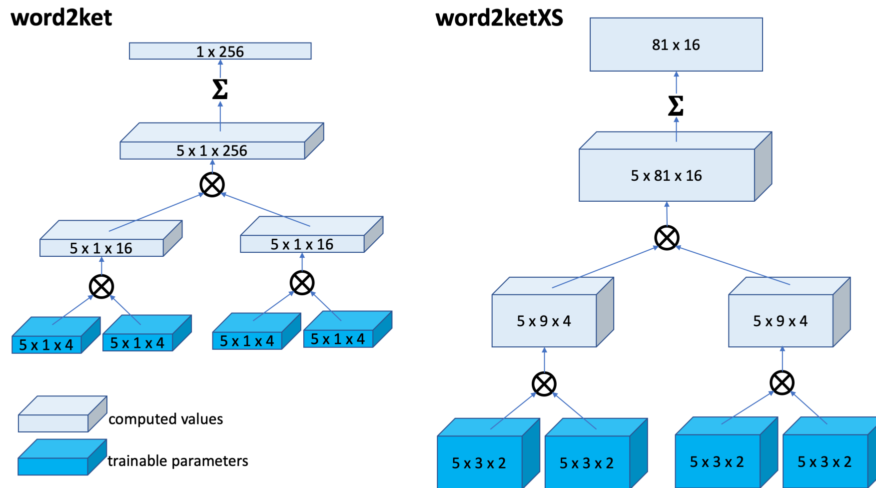

Figure 3: Architecture of the word2ket (left) and word2ketXS (right) embeddings. The word2ket example depicts a representation of a single-word 256-dimensional embedding vector using rank 5, order 4 tensor $\sum_{k=1}^{5} \bigotimes_{j=1}^{4} v_{jk}$ that uses twenty 4-dimensional vectors v_{jk} as the underlying trainable parameters. The word2ketXS example depicts representation of a full 81-word, 16-dimensional embedding matrix as $\sum_{k=1}^{5} \bigotimes_{j=1}^{4} F_{jk}$ that uses twenty 3×2 matrices F_{jk} as trainable parameters.

3.4 Linear Operators in Tensor Product Spaces and *word2ketXS*

3.4.1 Linear Operators in Tensor Product Spaces

Let $A : \mathcal{V} \to \mathcal{U}$ be a linear operator that maps vectors from Hilbert space $\mathcal{V}$ into vector in Hilbert space $\mathcal{U}$; that is, for $v, v', \in \mathcal{V}$, $\alpha, \beta \in \mathbb{R}$, the vector $A(\alpha v + \beta v') = \alpha A v + \beta A v'$ is a member of $\mathcal{U}$. Let us also define a linear operator $B : \mathcal{W} \to \mathcal{Y}$.

A mapping $A \otimes B$ is a linear operator that maps vectors from $\mathcal{V} \otimes \mathcal{W}$ into vectors in $\mathcal{U} \otimes \mathcal{Y}$. We define $A \otimes B : \mathcal{V} \otimes \mathcal{W} \to \mathcal{U} \otimes \mathcal{Y}$ through its action on simple vectors and through linearity

$$(A \otimes B) \left(\sum_{jk} \psi_j \otimes \phi_k \right) = \sum_{jk} (A\psi_j) \otimes (B\phi_k),$$

for $\psi_j \in \mathcal{V}$ and $\phi_k \in \mathcal{U}$. Same as for vectors, tensor product of linear operators is bilinear

$$\left(\sum_{j} a_j A_j \right) \otimes \left(\sum_{k} b_k B_k \right) = \sum_{jk} a_j b_k \left(A_j \otimes B_k \right).$$

In finite-dimensional case, for $n \times n'$ matrix representation of linear operator A and $m \times m'$ matrix representing B, we can represent $A \otimes B$ as an $mn \times m'n'$ matrix composed of blocks $a_{jk}B$.

3.4.2 The *word2ketXS* Embeddings

We can see a p-dimensional word embedding model involving a d-token vocabulary as a linear operator $F : \mathbb{R}^d \to \mathbb{R}^p$ that maps the one-hot vector corresponding to a word into the corresponding word embedding vector. Specifically, if e_i is the i-th basis vector in $\mathbb{R}^d$ representing i-th word in the vocabulary, and v_i is the embedding vector for that word in $\mathbb{R}^p$, then the word embedding linear operator is $F = \sum_{i=1}^{d} v_i {e_i}^T$. If we store the word embeddings a $d \times p$ matrix M, we can then interpret that matrix's transpose, M^T, as the matrix representation of the linear operator F.

Consider q and t such that $q^n = p$ and $t^n = d$, and a series of n linear operators

$F_j : \mathbb{R}^t \to \mathbb{R}^q$. A tensor product $\bigotimes_{j=1}^{n} F_j$ is a $\mathbb{R}^d \to \mathbb{R}^p$ linear operator. In *word2ketXS*, we represent the $d \times p$ word embedding matrix as

$$F = \sum_{k=1}^{r} \bigotimes_{j=1}^{n} F_{jk}, \tag{3.4}$$

where F_{jk} can be represented by a $q \times t$ matrix. The resulting matrix F has dimension $p \times d$, but takes $rnqt = O\left(rqt \max(\log p/q, \log d/t)\right)$ space. Intuitively, the additional space efficiency comes from applying tensor product-based exponential compression not only horizontally, individually to each row, but horizontally and vertically at the same time, to the whole embedding matrix.

We use the same balanced binary tree structure as in *word2ket*. To avoid reconstructing the full embedding matrix each time a small number of rows is needed for a multiplication by a weight matrix in the downstream layer of the neural NLP model, which would eliminate any space saving, we use lazy tensors [25, 26]. If A is an $m \times n$ matrix and matrix B is $p \times q$, then ij^{th} entry of $A \otimes B$ is equal to

$$(A \otimes B)_{ij} = a_{\lfloor (i-1)/p \rfloor + 1, \lfloor (j-1)/q \rfloor + 1} b_{i - \lfloor (i-1)/p \rfloor p, j - \lfloor (j-1)/q \rfloor q}.$$

As we can see, reconstructing a row of the full embedding matrix involves only single rows of the underlying matrices, and can be done efficiently using lazy tensors.

3.5 Experimental Evaluation of *word2ket* and *word2ketXS* in NLP Tasks

In order to evaluate the ability of the proposed space-efficient word embeddings in capturing semantic information about words, we used them in three different downstream NLP tasks: text summarization, language translation, and question answering. In all three cases, we compared the accuracy in the downstream task for the proposed space-efficient embeddings with the accuracy achieved by regular embeddings, that is, embeddings that store p-dimensional vectors for d-word vocabulary using a single $d \times p$ matrix.

Table 1.: Results for the GIGAWORD text summarization task using Rouge-1, Rouge-2, and Rouge-L metrics. The space saving factor is defined as the total number of parameters for the embedding divided by the total number of parameters in the corresponding regular embedding.

Embedding	**Order/Rank**	**Dim**	**RG-1**	**RG-2**	**RG-L**	**#Params**	**Space saving factor**
Regular	1/1	256	35.80	16.40	32.47	7,789,568	1
word2ket	4/1	256	33.65	14.87	30.47	486,848	16
word2ketXS	2/10	256	34.59	15.90	31.35	56,000	139
word2ketXS	4/1	256	34.05	15.39	30.75	224	34,775
Regular	1/1	8,000	36.71	17.48	33.37	243,424,000	1
word2ketXS	3/10	8,000	35.17	16.35	31.72	19,200	12,678

In text summarization experiments, we used the GIGAWORD text summarization dataset [27] using the same preprocessing as [28], that is, using 200K examples in training. We used an encoder-decoder sequence-to-sequence architecture with bidirectional forward-backward RNN encoder and an attention-based RNN decoder [29], as implemented in PyTorch-Texar [30]. In both the encoder and the decoder we used internal layers with dimensionality of 256 and dropout rate of 0.2, and trained the models, starting from random weights and embeddings, for 20 epochs. We used the validation set to select the best model epoch, and reported results on a separate test set. We used Rouge-1, Rouge-2, and Rouge-L [31] as evaluation metrics. Rouge-1 and Rouge-2 measures the number of unigram and bigram that overlaps between the model predictions and the true labels respectively. Rouge-L measures the number of longest matching sequence of words using longest commons subsequence method.

Table 2.: Results for the IWSLT2014 German-to-English machine translation task. The space saving factor is defined as the total number of parameters for the embedding divided by the total number of parameters in the corresponding regular embedding.

Embedding	**Order/Rank**	**Dimensionality**	**BLEU**	**#Params**	**Space saving factor**
Regular	1/1	256	26.44	8,194,816	1
word2ketXS	2/30	400	25.97	214,800	38
word2ketXS	2/10	400	25.33	71,600	114
word2ketXS	3/10	1000	25.02	9,600	853

In addition to testing the regular dimensionality of 256, we also explored 8000, but kept the dimensionality of other layers constant. Increasing the dimension from 256 to 8,000 leads to a better performance but incur additional memory and computation costs. The word2ketXS is able to gain this performance increase while keeping the memory footprints very low.

The results in Table 1 show that word2ket can achieve 16-fold reduction in trainable parameters at the cost of a drop of Rouge scores by about 2 points. This roughly translates to 2 percent less number of overlapping words between the machine generated summary and the reference. As expected, word2ketXS is much more space-efficient, matching the scores of word2ket while allowing for 34,000 fold reduction in trainable parameters. More importantly, it offers over 100-fold space reduction while reducing the Rouge scores by only about 0.5. Thus, in the evaluation on the remaining two NLP tasks we focused on word2ketXS.

The second task we explored is German-English machine translation, using the IWSLT2014 (DE-EN) dataset of TED and TEDx talks as preprocessed in [32]. We used the same sequence-to-sequence model as in GIGAWORD summarization task above.

Table 3.: Results for the Stanford Question Answering task using DrQA model. The space saving factor is defined as the total number of parameters for the embedding divided by the total number of parameters in the corresponding regular embedding.

Embedding	**Order/Rank**	**F1**	**#Params**	**Space saving factor**
Regular	1	72.73	35,596,500	1
word2ketXS	2/2	72.23	24,840	1,433
word2ketXS	4/1	70.65	380	93,675

We explored embedding dimensions of 100, 256, 400, 1000, and 8000 by using different values for the tensor order and the dimensions of the underlying matrices F_{jk}.

We used BLEU score [33] to measure test set performance. The BLEU metric is defined for a range of 0 to 1 and BLEU score of 1 shows a perfect translation. BLEU evaluates how similar a machine translation is to a human reference translation, taking into account translation length, word choice, and word order. The results in Table 2 show a drop of about 1 point on the BLEU scale for 100-fold reduction in the parameter space, with drops of 0.5 and 1.5 for lower and higher space saving factors, respectively. Increasing the dimensions leads to better performance. We noticed that for experiments with the same number of parameters, i.e., same memory footprints, models with larger dimensions generally lead to better performance. However, this increase in dimension might introduce some additional computation time.

The third task we used involves the Stanford Question Answering Dataset (SQuAD) dataset. We used the DrQA's model [34], a 3-layer bidirectional LSTMs with 128 hidden units for both paragraph and question encoding. We trained the model for 40 epochs, starting from random weights and embeddings, and reported the test set F1 score. DrQA uses an embedding with vocabulary size of 118,655 and embedding dimensionality of

300. As the embedding matrix is larger, we can increase the tensor order in word2ketXS to four, which allows for much higher space savings.

Results in Table 3 show a 0.5 point drop in F1 score with 1000-fold saving of the parameter space required to store the embeddings. For order-4 tensor word2ketXS, we see almost a 10^5-fold space saving factor, at the cost of a drop of F1 by less than two points, that is, by a relative drop of less than 3%. We also investigated the computational overhead introduced by the word2ketXS embeddings. While the training time increased, as shown in Fig. 4, the dynamics of model training remains largely unchanged. For tensors order 2, the training time for 40 epochs increased from 5.8 for the model using regular embedding to 7.4 hours for the word2ketXS-based model. Using tensors of order 4, to gain additional space savings, increased the time to 9 hours. Though we see a trade-off between memory usage and increased training time, for most practical purposes, the gain is much more than the time we lose. For example, for tensor order 2, we get a 99.93% decrease in memory usage by increasing training time by 27%. We

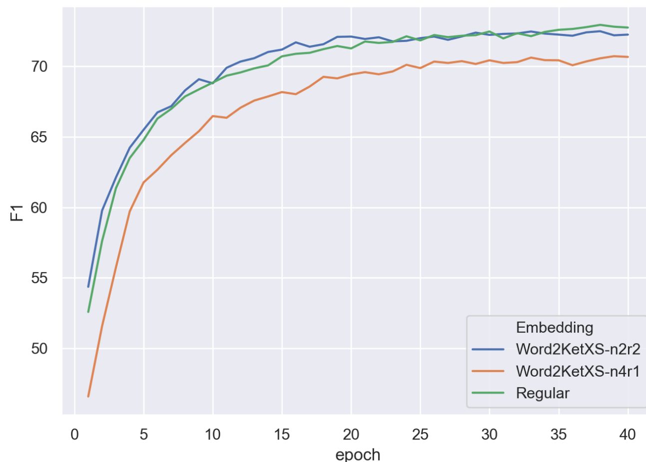

Figure 4: Dynamics of the test-set F1 score on SQuAD dataset using DrQA model with different embeddings: rank-2 order-2 word2ketXS, rank-1 order-4 word2ketXS, and regular embedding.

estimate the increase in inference time is in the same order as the increase in training time. Each run was executed on a single NVIDIA Tesla V100 GPU card, on a 2 Intel Xeon Gold 6146 CPUs, 384 GB RAM machine.

The results of the experiments show substantial decreases in the memory footprint of the word embedding part of the model, used in the input layers of the encoder and decoder of sequence-to-sequence models. These also have other parameters, including weight matrices in the intermediate layers, as well as the matrix of word probabilities prior to the last, softmax activation, that are not compressed by our method. During inference, embedding and other layers dominate the memory footprint of the model. Recent successful transformer models like BERT by [3], GPT-2 by [35], RoBERTa by [36] and Sparse Transformers by [37] require hundreds of millions of parameters to work. In RoBERTa$_{\text{BASE}}$, 30% of the parameters belong to the word embeddings.

During training, there is an additional memory need to store activations in the forward phase in all layers, to make them available for calculating the gradients in the backwards phase. These often dominate the memory footprint during training, but one can decrease the memory required for storing them with e.g. gradient checkpointing [38] used recently in [37].

CONTEXT:
The 8- and 10-county definitions are not used for the greater Southern California Megaregion, one of the 11 megaregions of the United States. The megaregion's area is more expansive, extending east into Las Vegas, Nevada, and south across the Mexican border into Tijuana.

Question	True Answers	Model Prediction
What is the name of the region that is not defined by the eight or 10 county definitions?	Southern California Megaregion, the greater Southern California Megaregion	greater Southern California Megaregion
How many megaregions are there in the United States?	11	11
What is the name of the state that the megaregion expands to in the east?	Nevada	Southern California Megaregion
Which border does the megaregion extend over?	Mexican	Tijuana
What is the name of the area past the border that the megaregion extends into?	Tijuana	Tijuana

CONTEXT:
In 1900, the Los Angeles Times defined southern California as including "the seven counties of Los Angeles, San Bernardino, Orange, Riverside, San Diego, Ventura and Santa Barbara." In 1999, the Times added a newer county—Imperial—to that list.

Question	True Answers	Model Prediction
Which newspaper defined southern California?	Los Angeles Times, the Los Angeles Times	Los Angeles Times
In which year did the newspaper define southern California?	1900	1900
In which year did the newspaper change its previous definition?	1999	1900
What was the newer county added to the list?	Imperial	Imperial
How many counties initially made up the definition of southern California?	seven	seven

Figure 5: Test set questions and answers from DrQA model trained using rank-1 order-4 word2ketXS embedding that utilizes only 380 parameters (four 19×5 matrices F_{jk}, see eq. 3.4) to encode the full, 118,655-word embedding matrix. As each parameter is saved as a single-precision floating-point number, this translates to 1.5 Kilobytes memory usage. Here the True Answers are the manually labeled and the model is able to correctly answer the questions by a F1 measure of 70.65. The two points loss in F1 compared to the regular trained model is hardly noticeable in practice. We did not notice any trends in which types of questions either consistently remained correct or consistently were incorrect when using the word2ket embeddings as compared to the regular embeddings.

CHAPTER 4

IMPROVING SPACE EFFICIENCY OF TRANSFORMER MODEL

4.1 Introduction

Natural language models involve large number of parameters. A single encoder-decoder Transformer [4] in its base variant has about 44 million parameters, not counting the word embedding matrix, which adds another 10 million or more, depending on the chosen vocabulary or tokenization scheme. Base variant of encoder-only BERT [3], including the embedding, has about 108 million parameters. GPT-3 [39] has about 175 billion parameters, and the largest of the Switch Transformer [40] models has 1.5 trillion. This explosion in the model size has led to increased interest in approaches for reducing the number of parameters in the model.

Models with high-dimensional parameter space have much lower intrinsic dimension [41], that is, training trajectory can be successfully restricted to a random, smaller-dimensional subspace, even though training a small-parameter architecture is often less successful. These observations have been recently extended to the fine-tuning trajectories of language models [42]. Lottery ticket hypothesis [43, 44], recently demonstrated to hold also for language model fine-tuning [45, 46], shows that smaller subnetworks can be selected from a large model, re-trained in isolation, and perform as well as the large model; what those subnetworks are is not known a prior, in absence of the trained large model, though. Some approaches for reducing model size build on this observation to train a smaller model based on an existing large model, for example a 66 million parameter DistillBERT [47] student has been distilled from 108 million parameter BERT-base [3] teacher with little loss in quality. Many other approaches train a reduced-parameter model de novo, without relying on an already trained large model. For example, DeLighT

[48] uses an alternative parameterization of the multi-headed self-attention based on group linear transform to reduce a 62 million parameter Transformer to 22 million.

One simple way to reduce model size involves factorized matrix representations. ALBERT [49] employs a rank r decomposition of a $d \times n_{vocab}$ embedding matrix storing d-dimensional embedding vectors for each of the n_{vocab} tokens by using a stack of two linear layers, $d \times r$ on top of $r \times n_{vocab}$. Similar low-rank decomposition is also used implicitly in the multi-headed self-attention in the generic Transformer [4] with hidden dimension d and n_{heads} self-attention heads. In each Transformer head, a $r = d/n_{heads}$-rank factorized representation involving $d \times d/n_{heads}$ key (K) and query (Q) matrices are used, with the pairwise self-attention scores for sequence x calculated using $x^T K^T Q x$, instead of $x^T W x$ involving a full general attention $d \times d$ trainable matrix W as originally considered in trainable-attention encoder-decoder LSTM models [29]. In both cases, the models are trained from a random initialization of the factorized parameter space, instead of attempting to find the lowest-error factorized representation of an already trained original model.

We explore here a Transformer model that uses an alternative way to decompose a matrix into two smaller matrices. Instead of standard low-rank factorization as above, it involves reshaping and reordering matrix dimensions prior to the decomposition, and is equivalent to a sum of Kronecker products with an efficient implementation. For non-square matrices, the approach allows for increased reduction in parameters for the same decomposition rank. For square matrices, the benefits come from increased expressiveness of the decomposition for the same rank, allowing for reducing the rank needed to preserve model accuracy, and thus reducing the model size. Our main contribution is proving that stacking multiple linear layers decomposed this way increases the expressiveness of the network, unlike stacking multiple low-rank layers factorized in the standard way, which can only map into a subspace of dimensionality equal to the rank. Empirically, we show that the decomposition can reduce the size of a simple encoder-decoder Transformer to

as little as 4 million parameters, including 2 million for the model and 2 million for the embeddings. The technique can be employed automatically to any matrix, including embedding, dense, attention, and output layers, without requiring any modification of the model architecture.

Parameterized hypercomplex multiplication (PHM) linear layers [50] has an approach that most closely related to ours. PHM linear layers [50] arise from generalizing hypercomplex number layers [51] to arbitrary dimensionality r, with the arithmetic over the numbers learned during training. For a given dimensionality r, the arithmetic takes form of a sum of r Kronecker products. Irrespective of the original matrix size $n \times m$, each Kronecker product involves a small $r \times r$ matrix and a larger $n/r \times m/r$ matrix, which together have $r^3 + nm/r$ trainable parameters. For r used in practice, the second term dominates, resulting in r-fold reduction in size. PHM layers with r as large as 16 have been demonstrated to be effective, leading to a reduction of a 44 million parameter Transformer to a 2.9 million parameter PHM-Transformer. These numbers do not include the embedding matrix for token embeddings which is not reduced in size by the PHM approach; and which for example for $32,000$ BPE tokens requires about 16 million parameters.

Our word2ket results from Section 3, as well as results from PHM [50], Kronecker-based convolutional [52] and recurrent networks [53], demonstrate that a sum of Kronecker products leads to very compact representations. Here, we advance theoretical understanding of why this low-rank representation has advantages in terms of expressivity compared to a standard matrix factorization with the same rank. We also provide a more efficient, more general Kronecker-based representation. This approach can be used in linear transformations anywhere in the model and unlike PHM it also applies to embedding matrices. To represent an $n \times m$ matrix, our approach requires $2r\sqrt{mn}$ instead of $r^3 + nm/r$ parameters required by PHM. For example, for $r = 16$, the highest used by PHM, for a Transformer with hidden dimension of 512, each 512×2048 feed-forward

matrix in the self-attention head would be reduced from 1 million parameters to 69,632 parameters by PHM, but to 16,384 by our approach.

Kronecker product of two matrices is a concrete-basis representation of a finite-dimensional linear operator defined by a tensor product of two underlying linear operators. Thus, Kronecker-product-based representations, including PHM layers and ours, can be seen as special case of a more general family of higher-order tensor-product-based representations. Kronecker-based representation unfolds a $n \times m$ matrix from two smaller matrices, $n_1 \times m_1$ and $n_2 \times m_2$, with $m = m_1 m_2$ and $n = n_1 n_2$. In tensor-based representations, this is generalized to $m = \prod_{j=1}^{o} m_j$ and $n = \prod_{j=1}^{o} n_j$ for some tensor order o, leading to unfolding the $n \times m$ matrix from an o-way tensor that is then decomposed using low rank leading to reduction in parameters. For example, the tensor-train representation [54] represents a matrix using a series of core tensors with matching ranks. Tensor decomposition has been used successfully in convolutional networks [55, 56] and recurrent networks [57]. In the context of language models, tensor-train representation has been used to construct tensorised embeddings [58], which use order-three tensor representation to obtain 60-fold reduction in embedding in a sequence-to-sequence Transformer, and also use order-six tensors to achieve almost 400-fold reduction in an LSTM model for sentiment analysis. Beyond embeddings, Tensorized Transformer [59] uses block-term tensor decomposition – a combination of CP and Tucker decompositions – to reduce the multi-headed self-attention layer with n_{heads} by a factor $1/n_{heads}$, leaving other layers intact; this allowed for reducing a 52M Transformer down to 21M trainable parameters. So far, none of the Kronecker- or tensor-product-based approaches have been applied comprehensively to all components of a Transformer model – embeddings, attention, and feed-forward layers.

4.2 Limitations of Factorized Matrices in Deep Networks

A simple approach for reducing the size and complexity of a model involves replacing each linear transformation $n \times m$ matrix W with its factorized, low-rank representation involving an $n \times r$ matrix A and an $r \times m$ matrix B, with the rank $r \ll m, n$. During training and inference, the product AB is used instead of W, essentially replacing W with a two linear layers, A stacked on B. In this way, the number of parameters reduces from nm needed to represent W to $r(n+m)$ needed to represent A and B.

One potential approach to use the factorized representation is to take an already trained model, and factorize each matrix W_j into its low-rank approximation $\widehat{W}_j = A_j B_j$ using a method such as SVD. However, analyzing an idealized deep model with linear activations shows that errors from the layers add up [60], $\|\prod_j W_j - \prod_j \widehat{W}_j\| = \sum_j \|W_j - \widehat{W}_j\|$. Instead, a typical approach is to train a model de novo in its factorized form, using randomly initialized matrices A_j and B_j, without attempting to approximate individual linear transformations of the non-factorized model. This is the approach we take here.

Representing an $n \times m$ matrix W as a low-rank product AB limits the linear transformation – the output is constrained to be a low-dimensional subspace of $\mathbb{R}^n$. Stacking rank-r factorized linear transformations into an idealized deep network with linear activations does not improve the expressiveness, the dimensionality of the space of possible outputs is still constrained by the smallest of the decomposed layers rank. While this may have regularizing effect for medium values of the rank, it may prevent from using very small ranks and thus from creating compact models.

4.3 Stacked Kronecker Product-based Representations

For $n = m$, the standard factorized $(n \times r)\,(r \times m) \rightarrow n \times m$ representation is optimal among pairwise-product-based representations in the sense that all elements of a column of A are multiplied with each element of a row of B. For non-square matrices, for example matrix A is smaller than B, a more efficient all-pairs product representation

can be made by increasing the size of A and decreasing the size of B.

In Shapeshifter, the approach we propose here, instead of the standard $(n \times r)\,(r \times m) \rightarrow n \times m$ factorized representation, we rearrange the mn parameters of the original matrix W into a $\sqrt{mn} \times \sqrt{mn}$ matrix W', and then use a factorized representation of W' as a $\sqrt{mn} \times r$ matrix A and an $r \times \sqrt{mn}$ matrix B:

$$\left(\sqrt{nm} \times r\right)\left(r \times \sqrt{nm}\right) \xrightarrow{\text{multiply}} \sqrt{nm} \times \sqrt{nm} \xrightarrow{\text{reshape}} \sqrt{n} \times \sqrt{m} \times \sqrt{n} \times \sqrt{m}$$
$$\xrightarrow{\text{transpose}} \sqrt{n} \times \sqrt{n} \times \sqrt{m} \times \sqrt{m} \xrightarrow{\text{reshape}} n \times m$$

This representation uses $2r\sqrt{mn}$ parameters instead of nm.

We apply this representation to all matrices in attention and feed-forward layers in all multi-head attention blocks, and to the embedding matrices. For non-square matrices, the representation results in parameter saving compared to standard factorization with the same rank r. For square matrices, no parameter saving is achieved. However, as we show below, the way the matrix is decomposed allows for increased expressive power.

For $r = 1$, the $n \times m$ matrix W arises from taking products of each element from $\sqrt{nm} \times 1$ matrix A with each element from $1 \times \sqrt{nm}$ matrix B. It is known to be equivalent [61] to a Kronecker product $A \otimes B$ involving $\sqrt{n} \times \sqrt{m}$ matrices A, B resulting from reshaping the single-column and single-row matrices, respectively. Beyond $r = 1$, the representation is equivalent to representing W as sum of r Kronecker products, $W = \sum_{j=1}^{r} A_j \otimes B_j$. Compared to the equivalent sum of Kronecker-products representation, using the matrix factorization representation avoids the explicit summation of r terms, resulting in simpler, more efficient implementations.

4.3.1 Expressiveness of stacked Kronecker-product Layers

Certain types of matrices can easily be expressed as a sum of Kronecker products, but not in a r-rank factorized matrix form. For example, identity mapping on $\mathbb{R}^n$ does not admit low-rank factorization, but it can be decomposed as a single Kronecker

product of two smaller identity matrices. More broadly, the output of a sum of Kronecker products layer is not limited to r-dimensional subspace; a Kronecker product of two orthogonal matrices is an orthogonal matrix [62], thus even a single $A \otimes B$ term can model isomorphisms $\mathbb{R}^n \to \mathbb{R}^n$. The converse is not true, most matrices require Kronecker representation with a high rank, and the question of approximating various types of matrices with a sum of Kronecker products has received ample attention [63, 64, 62, 65].

Here, we focus on a different question – how does the expressive power of a stack of layers, each involving a sum of Kronecker products instead of generic linear transformation, grow with the network depth. We analyze an idealized network formed by a stack of Kronecker layers with linear activations. Given high-enough rank, a single layer is enough to represent any matrix. We first analyze what the depth needs to be if rank is restricted to two, and then investigate how quickly the depth can be reduced as the rank increases.

Below, we use the following notation. Support $\text{supp}(x)$ of vector x is a set of indices at which x is not null. We use $[n] = \{1, ..., n\}$. I is identity matrix. $\mathbb{1}_{jk}$ is a matrix with unity at row j and column k and null elsewhere; $\mathbb{1}_k$ is a shorthand for $\mathbb{1}_{kk}$. To avoid ambiguity, we use square brackets for individual entries of a matrix or a vector; thus $A[j,:]$ represents j-th row of matrix A, while A_j represents j-th matrix out of a series, as used above. For index set Z of cardinality k and a square matrix A, by $A[Z]$ we represent a $k \times k$ submatrix of A obtained by taking the intersection of columns and rows from set Z.

We first define k-variant matrices, which will be a useful tool in exploring stacked factorized reshaped layers.

Definition 1 (k-variant matrix)**.** *For an $n \times n$ orthogonal matrix U and an index set $Z \subset [n]$, by writing U^Z we indicate that both U and its inverse U^T act as identity on all the dimensions not in set Z. That is, for each $x \in \mathbb{R}^n$, $\text{supp}(Ux - x) \subseteq Z$ and $\text{supp}(U^T x - x) \subseteq Z$. If Z has cardinality at most k, we call U^Z a k-variant matrix.*

The notion of k-variant matrices generalizes Givens rotation matrices [66] used for example in QR decomposition. A Givens rotation in n dimensions is any rotation that acts on a plane spanned by two of the n coordinate axes – it is thus 2-variant. QR decomposition can be used to show that each matrix can be represented as a product of 2-variant matrices. We will then show that 2-variant matrices, or more broadly, k-variant matrices for $k \leq \sqrt{n}$, can be represented using sum of Kronecker products.

Theorem 2 (Decomposition into layers involving Kronecker products)**.** *Let $n = m^2$ for some $m \in \mathbb{N}$. Any orthogonal $n \times n$ matrix U can be represented as*

$$U = \prod_{i=1}^{L} \sum_{j=1}^{2} A_{ij} \otimes B_{ij},$$

where A_{ij} and B_{ij} are $\sqrt{n} \times \sqrt{n}$ matrices, and $L = O\left(n^2\right)$.

Proof. QR decomposition using Givens matrices will result in Q being a product of up to $n(n-1)/2$ Givens rotations, which are 2-variant, and R an upper triangular matrix which for orthogonal matrices is a diagonal matrix with $+1$ and -1 elements. The sign of any pair of -1 diagonal entries can be negated by a 2-variant matrix; at most $n/2$ such matrices are needed to convert the diagonal into identity, since there are at most $n-1$ negative elements in R. In total, up to $n^2/2$ 2-variant matrices are needed to represent any given orthogonal matrix.

Next, we set to show that any 2-variant matrix can be represented as a product of at most three matrices, each of the form $A_1 \otimes B_1 + A_2 \otimes B_2$, using $m \times m$ matrices.

Let $U^{\{k,q\}}$ be an arbitrary 2-variant orthogonal $m^2 \times m^2$ matrix for arbitrary $k, q \in [m^2]$. That is, $y = U^{\{k,q\}}x$ acts only on $x[k]$ and $x[q]$ to produce $y[k]$ and $y[q]$, for $l \neq k, q$ we have $y[l] = x[l]$.

Consider two vectors, $x_A, x_B \in \mathbb{R}^m$. Kronecker product of $x = x_A \otimes x_B$ of these

two vectors is a vector $x \in \mathbb{R}^n$ with entries defined through products

$$x[k] = x_A[\kappa]x_B[\lambda] \qquad \text{for} \quad k = (\kappa - 1)m + \lambda,$$
$$x[q] = x_A[\pi]x_B[\rho] \qquad \text{for} \quad q = (\pi - 1)m + \rho$$

for $\kappa, \lambda, \pi, \rho \in [m]$.

A matrix $A \otimes B$ acts on vectors $x = x_A \otimes x_B$ as $(A \otimes B)x = (Ax_A) \otimes (Bx_B)$. For example, let $B^{\{\lambda,\rho\}}$ be a 2-variant matrix action on λ, ρ, then $\mathbb{1}_\nu \otimes B^{\{\lambda,\rho\}}$ is a acting on dimensions $k = (\nu - 1)m + \lambda$ and $q = (\nu - 1)m + \rho$ in the same way as $B^{\{\lambda,\rho\}}$ acts on dimensions λ and ρ, and producing either identity or null elsewhere, as shown below:

$$\begin{bmatrix} 0 & & \\ & 1 & \\ & & 0 \end{bmatrix} \otimes \begin{bmatrix} \alpha & & -\beta \\ & 1 & \\ \beta & & \alpha \end{bmatrix} = \begin{bmatrix} 0 & & & & & & \\ & \ddots & & & & & \\ & & \alpha & & -\beta & & \\ & & & 1 & & & \\ & & \beta & & \alpha & & \\ & & & & & \ddots & \\ & & & & & & 0 \end{bmatrix}.$$

First, consider k, q such that $\kappa = \pi = \nu$ for some $\nu \in [m]$. Define a 2-variant $B^{\{\lambda,\rho\}}$ such that $B[\{\lambda, \rho\}] = U[\{k, q\}]$, that is, B acts on λ, ρ in the same way as U acts on k, q. We then have

$$U^{\{k,q\}} = \mathbb{1}_\nu \otimes B^{\{\lambda,\rho\}} + (I - \mathbb{1}_\nu) \otimes I.$$

A symmetric construction holds if $\lambda = \rho = \nu$. In both cases, we can represent a 2-variant orthogonal matrix by $A_1 \otimes B_1 + A_2 \otimes B_2$.

The general case of $U^{\{k,q\}}$ for $\kappa \neq \pi$ and $\lambda \neq \rho$ can be addressed by converting it into the case $\kappa = \pi$ via transforming κ into π, applying the special case above, and then transforming π back into κ.

We show that an orthogonal matrix V of the form $V = C \otimes D + C' \otimes D'$ exists such that

$$U^{\{k,q\}} = V^T \left(\mathbb{1}_\pi \otimes B^{\{\lambda,\rho\}} + (I - \mathbb{1}_\nu) \otimes I \right) V,$$

where $B[\{\lambda,\rho\}] = U[\{k,q\}]$, the λ, ρ columns/rows of B are formed by taking submatrix of U defined by columns/rows k, q.

To construct V, set $C = P_{\kappa\to\pi}$, a permutation matrix that moves κ to π, and $D = \mathbb{1}_\lambda$. Also set $C' = I$ and $D' = I - \mathbb{1}_\lambda$. Columns of $C \otimes D$ and $C' \otimes D$ are linearly independent, and each of the two matrices is orthogonal, thus V is orthogonal.

Consider $x = x_A \otimes x_B$, we then have

$$x' = Vx = P_{\kappa\to\pi} x_A \otimes \mathbb{1}_\lambda x_B + I x_A \otimes (I - \mathbb{1}_\lambda) x_B.$$

The first term will use the permutation to move $x_A[\kappa]$ to position π, where it will be multiplied by $x_B[\lambda]$, and placed at $x'[k']$ for $k' = (\pi - 1)m + \lambda$; all other entries $x_A[\cdot]x_B[\lambda]$ will be rearranged as well by the permutation. For $m = 3$, $\lambda = 2$, $\kappa = 1$, $\pi = 2$, the transformation that moves $x[k = 2] = x[(\kappa - 1)m + \lambda] = \alpha_1\beta_2$ into $x'[k' = 5] = x'[(\pi - 1)m + \lambda]$ and thus brings it into position ready for the special case, is illustrated below

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} \alpha_2 \\ \alpha_1 \\ \alpha_3 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ \beta_2 \\ 0 \end{bmatrix} =$$
$$= \begin{bmatrix} 0 & \alpha_2\beta_2 & 0 & 0 & \alpha_1\beta_2 & 0 & 0 & \alpha_3\beta_2 & 0 \end{bmatrix}^T.$$

The second term will preserve $x_A[\cdot]x_B[\cdot]$ for all entries of x_B except λ; this includes ρ, since in the general case $\lambda \neq \rho$. The k, q entries of x will be now at k', q entries of x', and applying $\mathbb{1}_\pi \otimes B^{\{\lambda,\rho\}} + (I - \mathbb{1}_\nu) \otimes I$ on x' will be equivalent to acting on positions k, q of x. The inverse orthogonal transformation V^T will restore the transformed values

at positions k', q, as well as all other entries $x_A[\cdot]x_B[\lambda]$, back to their original positions. The transformation V^T is of the same form as V, except using $P^{-1}_{\kappa\to\pi}$ instead of $P_{\kappa\to\pi}$, since the other matrices involved in V are diagonal with unit-magnitude entries. In summary, in the general case, $m \times m$ matrices A_{ij} and B_{ij} exist such that any 2-variant orthogonal matrix can be represented as $\prod_{i=1}^{3} \sum_{j=1}^{2} A_{ij} \otimes B_{ij}$.

Together, any orthogonal $n = m^2$ dimensional matrix U can be decomposed into $U = \prod_{i=1}^{3n^2/2} \sum_{j=1}^{2} A_{ij} \otimes B_{ij}$. □

Corollary 3. *Any linear $n \times m$ matrix W can be represented as a product of a finite sequence of sums of two Kronecker products of $\lceil n \rceil \times \lceil m \rceil$ matrices followed by selecting an $n \times m$ submatrix.*

Proof. The result immediately follows by considering an orthogonal matrix with dimension $\max(\lceil n \rceil^2 \times \lceil m \rceil^2)$, observing that an arbitrary, possibly rank-deficient diagonal matrix D can be decomposed as a product of a series of matrices of the form $\mathbb{1}_k \otimes (I - (D[j,k] - 1)\mathbb{1}_j) + (I - \mathbb{1}_k) \otimes I$, one per diagonal entry $D[j,k]$, which allows for producing arbitrary rank-deficient square matrices, from which non-square matrices can be selected. □

The above theorem shows that stacking layers represented compactly as $(\sqrt{nm} \times r)\,(r \times \sqrt{nm})$ for $r = 2$ increases the expressive power of the network. In the idealized case with linear activation function, adding such layers allows the network to eventually produce arbitrary $\mathbb{R}^m \to \mathbb{R}^n$ linear transformations, unlike stacking standard factorized layers $(n \times r)\,(r \times m)$, which always maps into an r-dimensional subspace irrespective of the number of layers.

With rank $r = 2$, we may need as many as $3n^2/2$ layers to represent an orthogonal $n \times n$ matrix. On the other extreme, only one layer is needed if the rank is n, since arbitrary $n \times n$ matrix can be seen as a concatenation of n tiles of size $\sqrt{n} \times \sqrt{n}$. As long as we define n indicator $\sqrt{n} \times \sqrt{n}$ matrices $\mathbb{1}_{jk}$ and use them as tile position selectors,

we can use a sum of Kronecker products of the selection with the corresponding tile to produce the matrix. In between these extremes, the number of layers needed to increase the expressiveness of the network to allow arbitrary transformations depends on the rank r in the Shapeshifter representation in the following way.

Theorem 4 (Depth-rank tradeoff for deep models involving Kronecker products)**.** *Let $n = m^2$ for some $m \in \mathbb{N}$, and let $r \leq m$. Any orthogonal $n \times n$ matrix U can be represented as a product of a sequence of sums of two Kronecker products of $m \times m$ matrices,*

$$U = \prod_{i=1}^{L} \sum_{j=1}^{r} A_{ij} \otimes B_{ij},$$

where A_{ij} and B_{ij} are $\sqrt{n} \times \sqrt{n}$ matrices, and $L = O\left(n^2/r\right)$.

Proof. Instead of 2-variant orthogonal matrices, we use r-variant ones. Consider $r = 3$, with $U^{\{k,q,t\}}$, and with $k = (\kappa - 1)m + \lambda$, $q = (\pi - 1)m + \rho$, and $t = (\tau - 1)m + \upsilon$. We now have two groups of indices, κ, π, τ and λ, ρ, υ. If $\kappa = \pi = \tau = \nu$ for some ν, we have a special case as in Theorem 2

$$U^{\{k,q,s\}} = \mathbb{1}_\nu \otimes B^{\{\lambda,\rho,\upsilon\}} + (I - \mathbb{1}_\nu) \otimes I;$$

we have a similar special case if $\lambda = \rho = \upsilon$.

The general case involves no equality in either of the two groups. Then, we can proceed similarly as for 2-variant general case, but we need a permutation matrix $P_{\kappa\to\tau}$ and separately a permutation matrix $P_{\pi\to\tau}$, which allow us to define

$$x' = Vx = P_{\kappa\to\tau} x_A \otimes \mathbb{1}_\lambda x_B + P_{\pi\to\tau} x_A \otimes \mathbb{1}_\rho x_B + I x_A \otimes (I - \mathbb{1}_\lambda - \mathbb{1}_\rho) x_B.$$

Here, V is an orthogonal transformation that uses a sum of three Kronecker products as a transformation that maps the general case to the $\kappa = \pi = \tau$ case. The construction naturally extends into higher r as long as $r \leq m$, the number of diagonal entries in matrices A: we need V to be a sum of r Kronecker products involving distinct dimensions

of A. In summary, we need less than $\prod_{i=1}^{3} \sum_{j=1}^{r} A_{ij} \otimes B_{ij}$ to represent arbitrary r-variant orthogonal matrix. A product of two k-variant matrices is at most $2k$-variant. Grouping $r/2$ Givens rotations or, more generally, 2-variant matrices can result in at most a r-variant matrix, thus we need at most $L = O\left(3n^2/r\right)$ layers, each involving a Kronecker product of rank at most r. □

As is common in theoretical work on deep networks, the results are limited to idealized multilayer models with linear activations. However, they can provide insights into networks with nonlinearities; for example, activations such as ReLU that are linear on large part of their input will lead to piece-wise linear network, and the results above hold in a piece-wise fashion.

4.4 Experimental Results

We validate the proposed Shapeshifter approach on machine translation using sequence-to-sequence models. The experiments were performed on a single V100 (longest run: 6 days) or A100 GPU (longest run: 1 day). We used Transformer [4], with embedding and encoder/decoder layers replaced with compact representations. We compare our approach to two recently proposed approaches for reducing model size: DeLighT [48] and PHM Layers [50]. DeLighT uses separate vocabulary for the source language encoder embeddings, and a different vocabulary for the target language decoder embedding and output layer. PHM Layers, like its predecessor Quaternion Transformer [51], uses the same Byte-Pair Encoding [67] in both encoder and decoder. To facilitate comparisons, we follow both approaches and train two Shapeshifter variants, one with separate embeddings and one with single embedding. In Shapeshifter, we use small rank for all matrices in the multi-head self-attention blocks. For embedding matrices, which are much larger and can be reduced more effectively, we use higher rank, while aiming to keep the total size of factorized embeddings below the total size of the rest of the factorized encoder/decoder.

Table 4.: **Results on the WMT'18 English-to-Romanian (En-Ro) dataset [72].** Model performance measured using BLEU [33] on dev set. We also list total number of model parameters, its breakdown into number of parameters in the encoder-decoder multi-headed self-attention and feed-forward layers, and separately in the all the embedding in the model. We also express the parameter reduction as a percentage and as a fold change compared to the size of the corresponding Transformer model. Shapeshifter $r_{encoder}$ / $r_{embedding}$ denotes encoder/decoder and embedding ranks.

Model	Total	Enc./Dec.	Emb.	Pct.	Fold	BLEU
Transformer [4], from [50]	–	44M	–	–	–	22.79
PHM-Tm n = 16 [50]	–	2.9M	–	–	–	19.63
*Transformer [4]	60.5M	44M	16.5M	100.0	1	24.30
*PHM-Tm n = 16 [50]	19.6M	3.2M	16.5M	32.4	3.08	22.38
*Shapeshifter 24/256	5.2M	3.1M	2.1M	8.62	11.59	22.56
*Shapeshifter 16/256	4.2M	2.1M	2.1M	6.95	14.39	22.12

* denotes our experimental results, other results taken from the referenced paper.

– denotes the information was not available in the referenced paper.

4.4.1 Comparison with PHM Layers

In comparisons with PHM layers [50], we use T5-small Transformer [68], which shares a single embedding matrix in both the encoder and the decoder. We use Huggingface Transformers [69] for PyTorch [70] implementation. We evaluate the approach on IWLSLT'14 German-to-English (De-En) [71] and WMT'18 English-to-Romanian (En-Ro) [72] datasets. For the first dataset we use learning rate 2e-3 with batch size of 128, while for the larger En-Ro dataset we use 3e-3 with batch size of 192. We used LAMB optimizer [73] with inverse square root scheduler, dropout 0.1, no weight decay, and 0.1 label-smoothed cross entropy loss.

Table 5.: **Results on the IWSLT'14 German-to-English (De-En) dataset [71].**

Model	Total	Enc./Dec.	Emb.	Pct.	Fold	BLEU
Transformer [4], from [50]	–	44M	–	–	–	36.68
PHM-Tm n = 16 [50]	–	2.9M	–	–	–	33.89
*Transformer [4]	60.5M	44M	16.5M	100.0	1	36.72
*PHM-Tm n = 16 [50]	19.6M	3.2M	16.5M	32.4	3.08	36.49
*Shapeshifter 16/256	4.2M	2.1M	2.1M	6.95	14.39	36.36
Transformer [4], from [48]	42M	–	–	100.0	1	34.3
DeLighT [48]	14M	–	–	35.5	2.82	33.8
*Transformer [4]	39.5M	31.5M	7.9M	100.0	1	35.43
*Shapeshifter 64/256	9.3M	7.2M	2.1M	23.6	4.24	35.43

To facilitate comparison of our results with PHM Layers approach under the same conditions, we reimplemented the approach and applied it to T5-Small Transformer that we used as basis for experiments with our method. The small increase in size of the PHM Transformer observed in our re-implementation of PHM layers compared to the parameter size provided in the literature [50] (3.2M vs 2.9M parameters) results from the fact that the underlying T5 Transformer that we build on uses three separate $d_{hid} \times d_{hid}$ matrices for self-attention query, key, and value weights for all heads, while the Transformer used in [50] stores all three as a single, concatenated $d_{hid} \times 3d_{hid}$ matrix.

The results presented in Tables 4 and 5 indicate that while our full models are four times smaller than full PHM models, they offer very similar on-task performance. The parameter reduction comes not only from factorized embeddings in Shapeshifter, but also from matrices inside the attention blocks in encoder/decoder: both Shapeshifter 16 / 256 and PHM $n = 16$ use a sum of 16 Kronecker products to represent a matrix, yet PHM results in a 1.5 times larger encoder/decoder when applied to the same exact base

Table 6.: **Results on the WMT'16 English-to-Romanian (En-Ro) dataset [75] .**

Model	Total	Enc./Dec.	Emb.	Pct.	Fold	BLEU
Transformer [4], from [48]	62.0M	44.1M	17.9M	100.0	1	34.3
DeLighT [48]	22.0M	–	–	35.5	2.82	34.3
*Transformer [4]	62.0M	44.1M	17.9M	100.0	1	33.89
*Shapeshifter 24/256	5.4M	3.1M	2.2M	8.6	11.63	32.48

architecture.

4.4.2 Comparison with DeLighT

Following DeLighT [48], we use Transformer models as implemented in fairseq [74]. We benchmark the Shapeshifter models on four datasets: IWSLT'14 German-to-English (De-En) [71], WMT'16 English-to-Romanian (En-Ro) [75], WMT'14 English-to-German (En-De) [76], and WMT'14 English-to-French (En-Fr) [76]. The specific Transformer architecture used by DeLighT, and also in our experiments, varies depending on the dataset. For the WMT'14 En-De we used 12 transformers blocks with 4 attention heads, 512 as the embedding dimension and 1024 for the fan out fully-connected layer. For the other three datasets we used 12 transformers blocks with 8 attention heads, 512 as the embedding dimension and 2048 for the fan out fully-connected layer. For the training setup we have used Adam optimizer, learning rate of 5e-4, inverse square root scheduler with 15K warmup steps, dropout 0.3, weight decay of 1e-4, and 0.1 label-smoothed cross entropy loss.

Compared to the smallest of the DeLighT models available for each dataset, as summarized in Tables 5–7, the Shapeshifter models have between 1.5 and 4 times fewer parameters, yet offer similar quality; Shapeshifter achieves higher score than DeLighT on two datasets, and lower score on two other datasets.

Table 7.: **Results on the WMT'14 English-to-German and -French datasets [76].**

Model	Total	Enc./Dec.	Emb.	Pct.	Fold	BLEU	
						En-De	En-Fr
Transformer [4], from [48]	62.0M	44.1M	17.9M	100.0	1	27.3	39.2
DeLighT [48]	37M	–	–	59.6	1.68	27.6	39.6
*Shapeshifter 64/256	10.7M	8.2M	2.5M	17.2	5.7	26.6	40.78

4.4.3 Comparison with Standard Low-rank Factorization

To validate experimentally the theoretical results from Section 4.3.1 concerning the increased expressiveness of the factorization type used in Shapeshifter compared to standard $n \times r$, $r \times m$ low-rank factorization, we trained models factorized this way using the same encoder/decoder rank of 16, and the same embedding rank of 256 as in Shapeshifter. We used one small dataset, IWSLT'14 German-to-English, and one large, WMT'18 English-to-Romanian. The results in Table 8 show that the size of the model is reduced substantially in the Kronecker-based approach compared to standard low-rank representation. The reduction comes mostly from the embedding, which involve non-square matrices. Despite more than two-times larger number of parameters, standard low-rank representation shows much higher loss on the training set, with no significant difference in terms of generalization outside of the training set, confirming that the difference between the two representations results from increased expressiveness of a stack of low-rank Kronecker-product-based layers.

Table 8.: **Comparison with standard low-rank representation.** Factorized models are trained de novo on a small (IWSLT'14 De-En) and a large (WMT'18 En-Ro) dataset.

Dataset	Factorization type	Total	Enc/Dec	Emb.	BLEU	Train Loss	Dev Loss
De-En	Shapeshifter 16 / 256	4.2M	2.1M	2.1M	36.36	2.59	2.67
De-En	Low-rank 16 / 256	10.6M	2.2M	8.4M	26.68	2.97	2.99
Ro-En	Shapeshifter 16 / 256	4.2M	2.1M	2.1M	22.12	2.19	3.09
Ro-En	Low-rank 16 / 256	10.6M	2.2M	8.4M	17.47	2.32	3.37

CHAPTER 5

CONCLUSIONS

The goal of our work is to reduce the space required for storing the weights in large, deep learning models. The problem of large model size is especially pronounced in deep networks for natural language processing, which can have hundreds of millions or even billions of trainable parameters.

Towards addressing our goal, we first considered one layer of the deep model for NLP tasks: the matrix of parameters capturing word embeddings. Deep neural networks for natural language processing often use a large embeddings matrix to represent words. A discrete sequence of words can be much more easily integrated with downstream neural layers if it is represented as a sequence of continuous vectors. Also, semantic relationships between words, learned from a text corpus, can be encoded in the relative configurations of the embedding vectors. However, storing and accessing embedding vectors for all words in a dictionary requires a large amount of space and may strain systems with limited GPU memory. To address this problem, we used tensor-products to propose two methods, *word2ket* and *word2ketXS*, for storing word embedding matrix during training and inference in a highly efficient way. Our approach achieved a hundred-fold or more reduction in the space required to store the embeddings with almost no relative drop in accuracy in machine translation, question answering, and text summarization tasks.

After addressing the embedding layer, we proposed a general method that applies for all linear and self-attentional layers in a neural network. It uses a similar factorization as in word2ketXS, but by limiting the tensor product to only two factors, we could use a more computationally efficient representation involving simple matrix multiplication instead of Kronecker or tensor product.

Our theoretical and experimental results show that deep models composed of stacked

low-rank Kronecker-product-based representations are more expressive, yet smaller, than equivalent models that use standard low-rank factorized matrix representations. These observations help solve a puzzle of how recent methods such as word2ket [77] and PHM layers [50] achieve their impressive parameter reduction rates for embeddings and for encoder/decoder layers, respectively, despite training models de novo, without the help of an existing large model serving as a blueprint for compression or pruning, or as a teacher for knowledge distillation. Building on these results, we provide a comprehensive approach that works both for the embeddings and for the encoder/decoder, using more effective way to exploit Kronecker products for factorized representations. Experimental comparisons with state-of-the-art model reduction approaches, not only PHM but also recently proposed DeLighT [48], on a range of machine translation problems, show that the approach offers similar translation quality with much fewer parameters.

BIBLIOGRAPHY

[1] A. M. TURING. “I.—COMPUTING MACHINERY AND INTELLIGENCE”. In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: `10.1093/mind/LIX.236.433`. eprint: `https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf`. URL: `https://doi.org/10.1093/mind/LIX.236.433`.

[2] Ian Goodfellow et al. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[3] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[4] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[5] Anselm Blumer et al. “Occam’s razor”. In: *Information processing letters* 24.6 (1987), pp. 377–380.

[6] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig. “Linguistic regularities in continuous space word representations”. In: *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*. 2013, pp. 746–751.

[7] Jeffrey Pennington, Richard Socher, and Christopher Manning. “GloVe: Global vectors for word representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*. 2014, pp. 1532–1543.

[8] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in Neural Information Processing Systems*. 2013, pp. 3111–3119.

[9] Raphael Shu and Hideki Nakayama. "Compressing word embeddings via deep compositional code learning". In: *International Conference on Learning Representations*. 2018, arXiv:1711.01068.

[10] Martin Andrews. "Compressing word embeddings". In: *International Conference on Neural Information Processing*. Springer. 2016, pp. 413–422.

[11] Suyog Gupta et al. "Deep Learning with Limited Numerical Precision". In: *Proceedings of the 32nd International Conference on Machine Learning*. 2015, pp. 1737–1746.

[12] Avner May et al. "On the Downstream Performance of Compressed Word Embeddings". In: *Advances in Neural Information Processing Systems*. 2019, arXiv:1909.01264.

[13] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[14] Hesham Mostafa and Xin Wang. "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization". In: *International Conference on Machine Learning*. 2019, pp. 4646–4655.

[15] Trevor Gale, Erich Elsen, and Sara Hooker. "The State of Sparsity in Deep Neural Networks". In: *CoRR* abs/1902.09574 (2019).

[16] Nimit Sharad Sohoni et al. "Low-Memory Neural Network Training: A Technical Report". In: *CoRR* abs/1904.10631 (2019).

[17] Christopher De Sa et al. "High-accuracy low-precision training". In: *arXiv preprint arXiv:1803.03383* (2018).

[18] Paulius Micikevicius et al. "Mixed precision training". In: *arXiv preprint arXiv:1710.03740* (2017).

[19] Jian Zhang et al. “Low-precision random Fourier features for memory-constrained kernel approximation”. In: *arXiv preprint arXiv:1811.00155* (2018).

[20] Haim Avron et al. “Random Fourier features for kernel ridge regression: Approximation bounds and statistical guarantees”. In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 253–262.

[21] Jun Suzuki and Masaaki Nagata. “Learning Compact Neural Word Embeddings by Parameter Space Sharing.” In: *International Joint Conference on Artificial Intelligence*. 2016, pp. 2046–2051.

[22] Sanjeev Arora et al. “A compressed sensing view of unsupervised text embeddings, bag-of-n-grams, and LSTMs”. In: *International Conference on Learning Representations*. 2018.

[23] Jarosław Buczyński and Joseph M Landsberg. “Ranks of tensors and a generalization of secant varieties”. In: *Linear Algebra and its Applications* 438.2 (2013), pp. 668–689.

[24] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[25] Jacob Gardner et al. “GPyTorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 7576–7586.

[26] Benjamin Charlier, Jean Feydy, and Joan Glaunes. *KeOps: Calcul rapide sur GPU dans les espaces à noyaux*. 2018.

[27] David Graff et al. “English gigaword”. In: *Linguistic Data Consortium, Philadelphia* 4.1 (2003), p. 34.

[28] Liqun Chen et al. “Improving sequence-to-sequence learning via optimal transport”. In: *arXiv preprint arXiv:1901.06283* (2019).

[29] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. “Effective approaches to attention-based neural machine translation”. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 2015, 1412—1421.

[30] Zhiting Hu et al. “Texar: A Modularized, Versatile, and Extensible Toolkit for Text Generation”. In: *arXiv preprint arXiv:1809.00794* (2018).

[31] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. In: *Text summarization branches out*. 2004, pp. 74–81.

[32] Marc’Aurelio Ranzato et al. “Sequence level training with recurrent neural networks”. In: *International Conference on Learning Representations*. 2016, arXiv:1511.06732.

[33] Kishore Papineni et al. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.

[34] Danqi Chen et al. “Reading Wikipedia to answer open-domain questions”. In: *arXiv preprint arXiv:1704.00051* (2017).

[35] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI Blog* 1.8 (2019).

[36] Yinhan Liu et al. “Roberta: A robustly optimized bert pretraining approach”. In: *arXiv preprint arXiv:1907.11692* (2019).

[37] Rewon Child et al. “Generating long sequences with sparse transformers”. In: *arXiv preprint arXiv:1904.10509* (2019).

[38] Tianqi Chen et al. “Training deep nets with sublinear memory cost”. In: *arXiv preprint arXiv:1604.06174* (2016).

[39] Tom B Brown et al. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

[40] William Fedus, Barret Zoph, and Noam Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *arXiv preprint arXiv:2101.03961* (2021).

[41] Chunyuan Li et al. "Measuring the Intrinsic Dimension of Objective Landscapes". In: *International Conference on Learning Representations*. 2018.

[42] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. "Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning". In: *arXiv preprint arXiv:2012.13255* (2020).

[43] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: *International Conference on Learning Representations*. 2018.

[44] Jonathan Frankle et al. "Linear mode connectivity and the lottery ticket hypothesis". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3259–3269.

[45] Tianlong Chen et al. "The lottery ticket hypothesis for pre-trained BERT networks". In: *Advances in Neural Information Processing Systems*. 2020.

[46] Sai Prasanna, Anna Rogers, and Anna Rumshisky. "When BERT Plays the Lottery, All Tickets Are Winning". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 3208–3229.

[47] Victor Sanh et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv preprint arXiv:1910.01108* (2019).

[48] Sachin Mehta et al. "DeLighT: Very Deep and Light-weight Transformer". In: *International Conference on Learning Representations ICLR'2021*. 2020.

[49] Zhenzhong Lan et al. "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations". In: *International Conference on Learning Representations ICLR'2020*. 2019.

[50] Aston Zhang et al. "Beyond Fully-Connected Layers with Quaternions: Parameterization of Hypercomplex Multiplications with $1/n$ Parameters". In: *International Conference on Learning Representations ICLR'2021*. 2020.

[51] Yi Tay et al. "Lightweight and Efficient Neural Natural Language Processing with Quaternion Networks". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 1494–1503.

[52] Shuchang Zhou et al. "Exploiting local structures with the Kronecker layer in convolutional networks". In: *arXiv preprint arXiv:1512.09194* (2015).

[53] Cijo Jose, Moustapha Cissé, and Francois Fleuret. "Kronecker recurrent units". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2380–2389.

[54] Ivan V Oseledets. "Tensor-train decomposition". In: *SIAM Journal on Scientific Computing* 33.5 (2011), pp. 2295–2317.

[55] Emily Denton et al. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1269–1277.

[56] Alexander Novikov et al. "Tensorizing Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 32. 2015.

[57] Charles C Onu, Jacob E Miller, and Doina Precup. "A Fully Tensorized Recurrent Neural Network". In: *arXiv preprint arXiv:2010.04196* (2020).

[58] Valentin Khrulkov et al. "Tensorized embedding layers for efficient model compression". In: *arXiv preprint arXiv:1901.10787* (2019).

[59] Xindian Ma et al. "A tensorized transformer for language modeling". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 2232–2242.

[60] Thomas Frerix and Joan Bruna. "Approximating orthogonal matrices with effective Givens factorization". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 1993–2001.

[61] Harold V Henderson and Shayle R Searle. "The vec-permutation matrix, the vec operator and Kronecker products: A review". In: *Linear and multilinear algebra* 9.4 (1981), pp. 271–288.

[62] Charles F Van Loan. "The ubiquitous Kronecker product". In: *Journal of Computational and Applied Mathematics* 123.1-2 (2000), pp. 85–100.

[63] Charles F Van Loan and Nikos Pitsianis. "Approximation with Kronecker products". In: *Linear algebra for large scale and real-time applications*. Springer, 1993, pp. 293–314.

[64] Julie Kamm and James G Nagy. "Kronecker product and SVD approximations in image restoration". In: *Linear Algebra and its Applications* 284.1-3 (1998), pp. 177–192.

[65] Eugene Tyrtyshnikov. "Kronecker-product approximations for some function-related matrices". In: *Linear Algebra and its Applications* 379 (2004), pp. 423–437.

[66] Wallace Givens. "Computation of plain unitary rotations transforming a general matrix to triangular form". In: *Journal of the Society for Industrial and Applied Mathematics* 6.1 (1958), pp. 26–50.

[67] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1715–1725.

[68] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67.

[69] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *arXiv e-prints* (2019), arXiv–1910.

[70] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 8026–8037.

[71] Mauro Cettolo et al. "Report on the 11th IWSLT Evaluation Campaign, IWSLT 2014". In: *IWSLT-International Workshop on Spoken Language Processing*. Marcello Federico, Sebastian Stüker, François Yvon. 2014, pp. 2–17.

[72] Ond Onrej Bojar et al. "Findings of the 2018 Conference on Machine Translation (WMT18)". In: *Proceedings of the Third Conference on Machine Translation*. Vol. 2. 2018, pp. 272–307.

[73] Yang You et al. "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes". In: *International Conference on Learning Representations ICLR'2020*. 2019.

[74] Myle Ott et al. "fairseq: A Fast, Extensible Toolkit for Sequence Modeling". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. 2019, pp. 48–53.

[75] Ondřej Bojar et al. "Findings of the 2016 Conference on Machine Translation". In: *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*. 2016, pp. 131–198.

[76] Ondřej Bojar et al. "Findings of the 2014 Workshop on Statistical Machine Translation". In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. 2014, pp. 12–58.

[77] Aliakbar Panahi, Seyran Saeedi, and Tom Arodz. "word2ket: Space-efficient Word Embeddings inspired by Quantum Entanglement". In: *International Conference on Learning Representations ICLR'2020*. 2019.