Tensor Train Format for Recurrent Neural Networks

Philip Blagoveschensky, Daniil Vankov,
Ivan Golovatskikh, Maria Sindeeva

March 21, 2019

1 Introduction

Neural networks have huger numbers of parameters. Because of this, training and prediction is often slow, they need a lot of RAM, and sometimes it causes bad generalization. What if we could compress weights in a way which reduced the number of parameters, allowed us to efficiently calculate linear algebraic functions on them, didn't make predictions much worse, and increased regularity of learned neural network? Perhaps there is a way – tensor decompositions.

In this project we investigated use of Matrix Tensor Train decomposition in Recurrent Neural Networks for video classification in hope that we would achieve the aforementioned benefits. The main plan was to replicate *Tensor-Train Recurrent Neural Networks for Video Classification* by Yang et al. Usually RNNs are not used directly on video frames data, because there are a lot of pixels in a frame, hence the dimensionality is very high. Sometimes people use feature extraction algorithms on frames and feed them into RNN. Tensor Train allows us not to do any feature extraction.

1.1 Theory

Definition 1.1. A **tensor** of **order** n is an n-dimensional array of real numbers. For example, vector $x \in \mathbb{R}^I$ is an **order**-1 tensor, it has one **mode** of length I. A matrix $A \in \mathbb{R}^{I \times J}$ is an **order**-2 tensor, it has two **modes** of lengths I and J.

It is known that matrices can be approximated by low rank matrices (e.g. SVD), in which case we don't need to store as many elements.

Tensors of order greater than 2 can be approximated in similar ways as well. These approximations allow us to perform linear algebra operations and tensor calculus operations on them without calculating unfolded tensor. Tensor train is a type of tensor decomposition. matrix tensor train is tensor train decomposition for large matrices.

Definition 1.2. Suppose $M \in \mathbb{R}^{I \times J}$ is a matrix with $I = I_1 I_2 \cdots I_N$, $J = J_1 J_2 \cdots J_N$. Let $X \in \mathbb{R}^{I_1 \times J_1 \times I_2 \times J_2 \times \cdots \times I_N \times J_N}$ be the order-$2N$ reshape tensor of M. Then its rank-$[R_0 = 1, R_1, R_2, \ldots, R_{N-1}, R_N = 1]$ **matrix tensor train** decomposition is represented as N order-4 **"core" tensors** $G^{(1)}, G^{(2)}, \ldots, G^{(N)}$, where $\forall n\, G^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times J_n \times R_n}$. The components of X are given by

$$X_{i_1, j_1, i_2, j_2, \ldots, i_N, j_N} = \sum_{r_1=1}^{R_0} \sum_{r_2=1}^{R_2} \cdots \sum_{r_N=1}^{R_N} \prod_{n=1}^{N} G^{(n)}_{r_{n-1}, i_n, j_n, r_n}$$

See visualization in fig. 1.

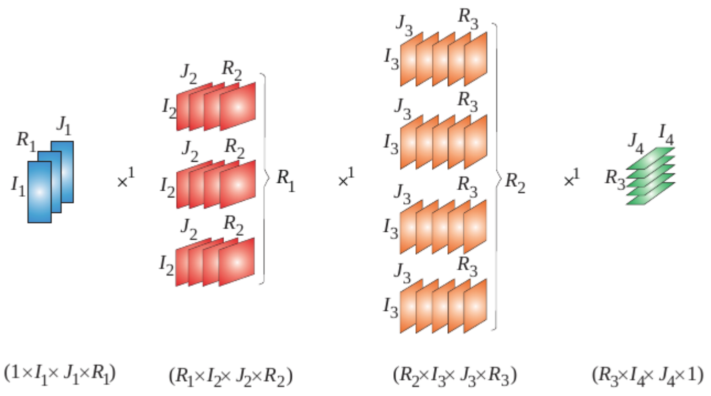
R_1
J_1
I_1
$\times^1$
J_2 R_2 I_2
R_1
$\times^1$
J_3 R_3 I_3
R_2
$\times^1$
J_4 I_4 R_3
$(1 \times I_1 \times J_1 \times R_1)$
$(R_1 \times I_2 \times J_2 \times R_2)$
$(R_2 \times I_3 \times J_3 \times R_3)$
$(R_3 \times I_4 \times J_4 \times 1)$

Figure 1: Matrix of shape $I_1 I_2 I_3 I_4 \times J_1 J_2 J_3 J_4$ in matrix tensor train format. From: *Low-Rank Tensor Networks for Dimensionality Reduction and Large-Scale Optimization Problems: Perspectives and Challenges. Part 1* by Cichocki et al.

1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a family of neural networks for processing sequential data, e.g. texts – sequence of words, medical history of a patient – sequence of medical records, video – sequence of images.

A simple RNN with hidden state $h^{(t)}$, input $x^{(t)}$, parameter θ is defined by equation

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}\right).$$

RNNs can be visualized as folded and unfolded computation graphs, see fig. 2.

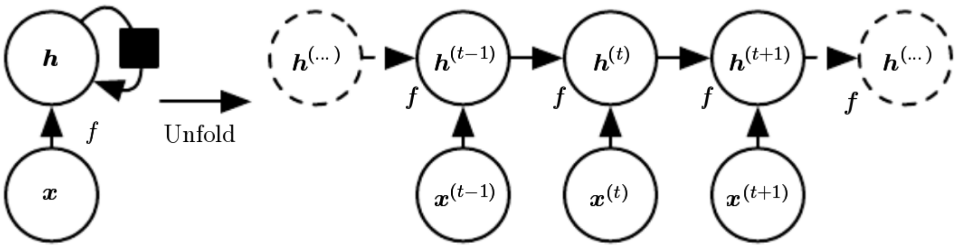

Figure 2: A recurrent network with no outputs. This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time. (Left) Circuit diagram. The black square indicates a delay of a single time step. (Right) The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance. From: *Deep Learning* by Goodfellow et al.

Gated Recurrent Unit (GRU) is a popular RNN with layers which allow the neural network to decide whether to update (u) or to reset (r) the hidden state (h). It is defined with the following equations:[1] [2]

$$h^{(t)} = u^{(t-1)} \odot h^{(t-1)} + (\vec{1} - u^{(t-1)}) \odot \tanh\left(b + Ux^{(t)} + W(r^{(t-1)} \odot h^{(t-1)})\right)$$

$$u^{(t)} = \sigma\left(b^{(u)} + U^{(u)}x^{(t)} + W^{(u)}h^{(t)}\right)$$

$$r^{(t)} = \sigma\left(b^{(r)} + U^{(r)}x^{(t)} + W^{(r)}h^{(t)}\right)$$

The main goal of this project was to make popular RNNs called Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) suitable for video processing by storing weights matrices of some linear layers in tensor train format. We such layers TTL layers. Optimization is performed as usual by any neural network optimizer, the parameters of these layers which we optimize are components of core tensors. In Tensor Train GRU is defined with the following equations, which are slightly modified equations above:

$$h^{(t)} = u^{(t-1)} \odot h^{(t-1)} + (\vec{1} - u^{(t-1)}) \odot \tanh\left(\mathrm{TTL}(x^{(t)}) + W(r^{(t-1)} \odot h^{(t-1)})\right)$$

$$u^{(t)} = \sigma\left(\mathrm{TTL}^{(u)}(x^{(t)}) + W^{(u)}h^{(t)}\right)$$

$$r^{(t)} = \sigma\left(\mathrm{TTL}^{(r)}(x^{(t)}) + W^{(r)}h^{(t)}\right)$$

LSTM is similar to GRU, and one doesn't need to know its definition to understand our work.

In our project, to classify a video using RNN, its frames are treated as a sequence, which is fed into the RNN. The last hidden state $h^{(T)}$ of RNN for the given sequence goes into another linear layer, which produces logits, which go into softmax and cross entropy, or some other loss function.

1.3 Related Work

- *Tensorizing Neural Networks* by Novikov et al. (2015) – they use tensor train layers instead of fully connected layers in convolutional networks.
- *Tensorized Embedding Layers for Efficient Model Compression* by Khrulkov et al. (2019) – they use tensor train in embedding layer.
- *Tensor Decomposition for Compressing Recurrent Neural Network* by Tjandra et al. (2018) – they also investigate using tensor decompositions in RNNs, not only Matrix Tensor Train, but also CP and Tucker decompositions.

2 Experiments

Since linear `TTLayer` is at the core of the whole idea of TT-RNN, we started our experiments with simple linear models.

[1] $\odot$ is elementwise (Hadamard) product. $\vec{1}$ is a vector with all components equal to one.
[2] Upper subscripts of matrices indicate whether they produce reset gate, update gate, or hidden state.

2.1 SVHN Dataset and TTLayer

SVHN (Street View House Numbers) is a real-world image dataset obtained from house numbers in Google Street View images. It is characterized by:

- 32 × 32 colorful pictures of digits from house numbers
- 73 257 digits for training, 26 032 digits for testing
- 10 classes, one for each digit
- ”distracting” digits are present on the images to the sides of the digit of interest

Figure 3: Examples of digits from the dataset

On this dataset we train two Multi Layer Perceptron with one hidden layer. In the first one we use a dense hidden layer, and in the other one - a `TTLayer` hidden layer. For both of these models we had:

- Approximately 75:25 train/test split given by this split
- Categorical cross-entropy as a loss function
- Adam optimizer with learning rate of `10e-3`

For `TTLayer` with both input and output shapes being $(32 * 32 * 3,)$ we chose to use 4 core tensors; the factorization of the inputs of shape $32 \times 32 \times 3$ to be $4 \times 8 \times 8 \times 12$; with tensor train ranks being $[1, 3, 3, 3, 1]$.

Under these settings our `TTLayer` will have 1632 parameters to optimize, whereas a dense layer of the same input and output dimensions will have 9437184 parameters.

After fitting the models to the dataset, the first thing we noticed was the training time difference. While the TTL model took only 6:51 minutes to train, the dense model took 13:51 minutes - almost two times longer! On average, one epoch of TTL MLP took 4 seconds to fit, compared to 9 seconds per epoch for a regular dense MLP.

Let’s take a look at the resulting training curves:

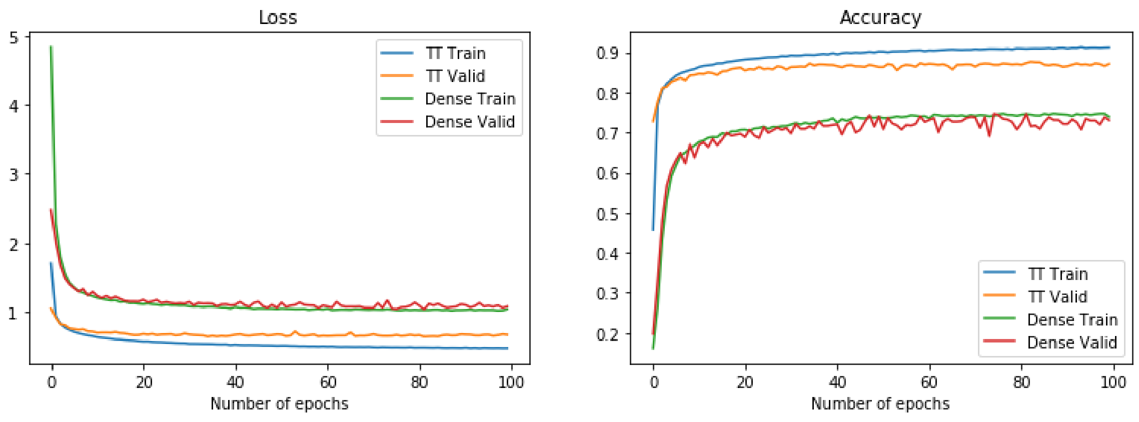

Figure 4: Resulting loss and accuracy curves

We can see that not only did TTL MLP take less time to fit, but it also provided lower loss and higher accuracy. This is most likely due to the fact that this model had lower number of parameters to be fit, making it a kind of model regularization, resulting in this model’s generalizing ability being better than that of a regular dense model.

On the test sample the accuracy of a TTL MLP was 0.827 - much higher than that of a dense MLP, which was only 0.607.

2.2 Youtube Celebrities Faces Dataset (Kim et al., 2008)

This dataset consists of 1910 Youtube video clips featuring 47 different persons (mostly movie stars and politicians). Each clip consists of 7 to 350 frames. See fig. 5 to see what video clips are like.

Figure 5: Youtube Celebrities Faces. Two video clips featuring Al Pacino and Emma Thompson.

For this dataset we implemented TT GRU using PyTorch and Valentin Khrulkov's TTL layer implementation `https://github.com/KhrulkovV/tt-pytorch`. We cut or padded clips to make each of them 85 frames long. We resized them to 80 by 60 pixels. We randomly chose 1710 training and 200 validation clips and performed classification.

We used the following hyperparameters: hidden state length = 256; 40% dropout for hidden state updated value; tensor train degree = 4 (i.e. 4 core tensors); tensor train ranks [1, 3, 3, 3, 1]; core tensor shapes were chosen automatically by TTL; Adam optimizer, amsgrad = True, learning rate = 10^{-3}, weight decay = 10^{-2}; early stopping when best validation accuracy doesn't increase for 50 epochs. And then some more training with smaller learning rate.

We got **88.5%** accuracy on this dataset. The authors of the paper we wanted to replicate, which was uploaded to Arxiv in 2017, claimed that state of the art accuracy on this dataset is **80.8%** accuracy (*Face recognition in movie trailers via mean sequence sparse representation-based classification* by Ortiz et al.). Their own result was **80%**. This implies that we **beat state of the art accuracy**.

We must make a remark here that the authors of that paper used approximately 392 validation samples, while we used only 200. Also we are not sure their comparison with other "classification accuracies" is valid, because it seems most people use this dataset for face recognition. Hence other models for this dataset probably tried to not just predict person, but predict person based on face and not on everything else. Also it seems that previous state of the art result with **80.8%** accuracy performed validation differently, although it's not clear whether their way was easier or more difficult for the model. Our training curves can be seen in fig. 6.

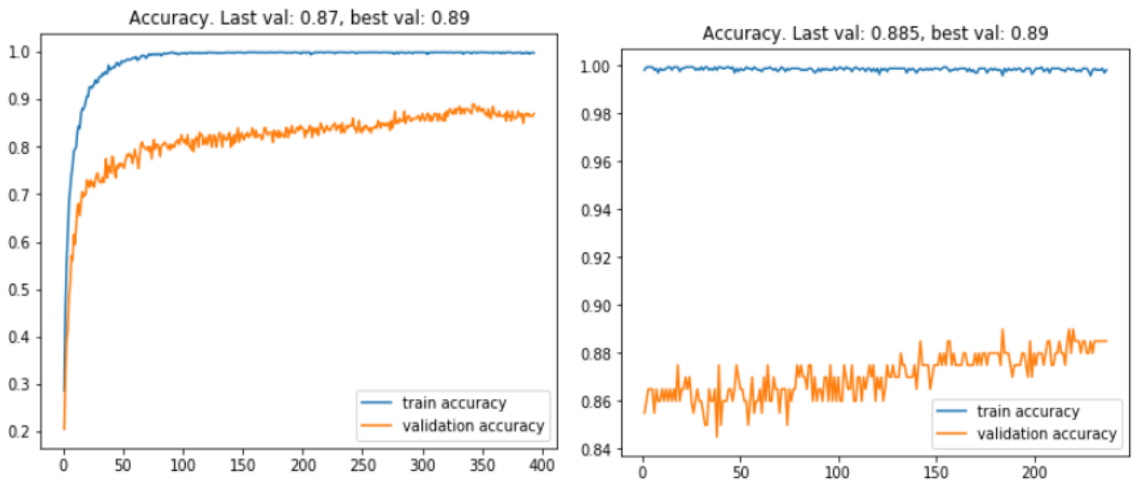

Figure 6: Two training curves. First we ran training (on the left) until early stopping. Then we continued training the same model (on the right) until the computer crashed for some reason. Best validation accuracy: 89%, last epoch validation accuracy: 88.5%. One epoch took approximately one minute.

2.3 Sentiment analysis and TT-Embedding

One of the main advantages of using TT is to reduce the number of trained parameters. To solve problems with large dictionaries, you need to train a huge number of parameters for word embedding. To reduce the number of trained parameters in the article "Tensorized Embedding Layers for Efficient Model Compression", it is proposed to use TT-Embedding layer instead of the usual Embedding layer. Sentiment analysis is a task of understanding an opinion of statement. We reproduced result from paper above with Valentin Khrulkov's TT-Embedding realization.

- IMDB dataset
- BCEWithLogitsLoss as a loss function

	Gygli et al., 2017			**TT LSTM**		
	Continous	Smooth	Sharp	Continous	Smooth	Sharp
Precision	0.94	0.97	0.95	0.89	0.93	0.93
Recall	0.88	0.95	0.99	0.90	0.92	0.93
F1-score	0.91	0.96	0.97	0.90	0.92	0.93
Support	14623	28073	33648	14623	28073	33648

- Adam optimizer with learning rate of `10e-5`
- Dropout = 0.5
- Embedding dimension = 256

2.4 Shot Boundary Detection Dataset (Hassanien et al., 2017)

DeepSBD is the large dataset of shot transitions. Shot boundary detection can be nontrivial problem in case of fades and gradual transitions between frames. Many methods address this problem, starting from classic image processing algorithms, such as Optical Flow, to neural networks -based. In this experiment we compared modern convolutional network architecture with simple Tensor Train LSTM layer, followed by one linear layer. It turned out, that even such simple RNN architecture can compete complex convolutional net. It worth noting that mentioned RNN architecture takes raw (normalized) video as input, without any feature extraction. Based on the original paper, we employed the following hyperparameters:

- $112 \times 112 \times 3 = 7 \times 16 \times 12 \times 28$ factorization.
- Hidden state length = 256.
- Batch size = 600
- 25% dropout for hidden state updated value
- Tensor train degree = 4 (i.e. 4 core tensors)
- Tensor train ranks [1, 4, 4, 4, 1]
- Adam optimizer, amsgrad = True, learning rate = 10^{-3}

After approximately 30 epochs of training we got the following results (in comparison with SOTA convolutional network):

- IMDB dataset
- BCEWithLogitsLoss as a loss function
- Adam optimizer with learning rate of `10e-5`
- Dropout = 0.5
- Embedding dimension = 256

3 Conclusions

4 Contributions

Each experiment was performed by one member of our team:

- Experiment Youtube Celebrities Faces Dataset (Kim et al., 2008) was carried out by Philip Blagoveschensky
- Experiment SVHN Dataset and TTLayer was carried out by Maria Sindeeva
- Experiment Sentiment analysis and TT-Embedding was carried out by Daniil Vankov
- Experiment Shot Boundary Detection Dataset (Hassanien et al., 2017) was carried out by Ivan Golovatskikh

5 Source code

Code for our experiments with some instructions about how to reproduce it can be found here: `https://github.com/philip-bl/tensor_train_rnn`.