Neural Tensor Networks for image classification

Artem Vazhentsev
Artur Bulanbaev
Lev Kosolapov
Dmitry Galkin
Alexey Voskoboinikov

Skoltech

NLA, 2020

Problem statement

Our project includes several key-tasks:

1. To implement one of the recent neural tensor networks architectures: a CNN with higher-order Taylor terms **(Neural Taylor Network)**, and check the efficacy of this approach in comparison to usual CNNs.
2. Apply tensor decompositions to Neural Taylor networks in order to speed them up.

Skoltech

Main ideas

- Implementing Taylor layers for NTN.
- Low-rank tensor decompositions (namely **CP**) to speed up the forward pass of the higher-order Taylor terms in the network.
- Training and evaluating deep NN's for image classification problem on **CIFAR10** and **CIFAR100.**
- To compare Deep NN's performance with NTN (is there some outperformance?)

Skoltech

Models and methods

Skoltech

Neural Taylor Network

$f(x) = b + W_1 x$ - simple fully-connected layer

$f(x) = b + W_1 x + \alpha_1 W_2 \times_1 x \times_2 x,$ - 2nd order NTN

$f(x) = b + W_1 x + \alpha_1 W_2 \times_1 x \times_2 x + \alpha_2 W_3 \times_1 x \times_2 x \times_3 x,$ - 3nd order NTN

where W_1 - matrix $m \times n$,

W_2 - tensor $n \times n \times m$,

W_3 - tensor $n \times n \times n \times m$,

x - input vector $n \times 1$,

Skoltech

Taylor series expansion of function of d variables

$$T(x_1,\ldots,x_d) = \sum_{n_1=0}^{\infty}\cdots\sum_{n_d=0}^{\infty}\frac{(x_1-a_1)^{n_1}\cdots(x_d-a_d)^{n_d}}{n_1!\cdots n_d!}\left(\frac{\partial^{n_1+\cdots+n_d}f}{\partial x_1^{n_1}\cdots\partial x_d^{n_d}}\right)(a_1,\ldots,a_d)$$
$$= f(a_1,\ldots,a_d) + \sum_{j=1}^{d}\frac{\partial f(a_1,\ldots,a_d)}{\partial x_j}(x_j-a_j) + \frac{1}{2!}\sum_{j=1}^{d}\sum_{k=1}^{d}\frac{\partial^2 f(a_1,\ldots,a_d)}{\partial x_j\partial x_k}(x_j-a_j)(x_k-a_k)$$
$$+\frac{1}{3!}\sum_{j=1}^{d}\sum_{k=1}^{d}\sum_{l=1}^{d}\frac{\partial^3 f(a_1,\ldots,a_d)}{\partial x_j\partial x_k\partial x_l}(x_j-a_j)(x_k-a_k)(x_l-a_l)+\cdots$$

In our case, the 3rd order derivatives correspond to the tensor W_3 . Thus we obtain the third-order Tensor Neural Network Slice.

Skoltech

Factorization for Tensor Neural Network

To make training and storing more efficient we store tensor layers in CPD format.

Storage optimizing:

from mn^3 to $(3n + m)r + r$

Speed improvement approximately:

$\frac{max(n, m)}{r}$

where r - decomposition rank

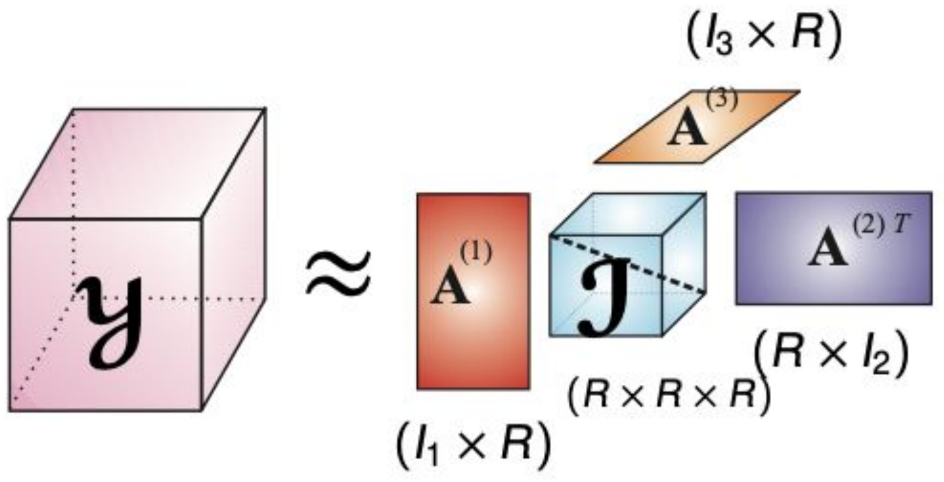


Fig. 1: CP decomposition of tensor layers Y.

Experiments

```python
class TensorLayer2order(torch.nn.Module):
    def __init__(self, n_input, n_output, alpha, no_diag=False):
        super(TensorLayer2order, self).__init__()
        self.M = torch.nn.Parameter(torch.randn(n_input, n_input, n_output, requires_grad=True, dtype=torch.float64), requires_grad=True)
        self.V = torch.nn.Parameter(torch.randn(n_input, n_output, requires_grad=True, dtype=torch.float64), requires_grad=True)
        self.B = torch.nn.Parameter(torch.randn(1, n_output, requires_grad=True, dtype=torch.float64), requires_grad=True)
        self.E = None
        self.alpha = alpha
        self.no_diag = no_diag

    def forward(self, E):
        M = self.M.to(device)
        # print(M.shape)
        # print(torch.eye(M.shape[0]).shape)
        # print(M[:,:,0])
        if self.no_diag:
          M=M-torch.diag_embed(M.diagonal(dim1=0,dim2=1), dim1=0, dim2 =1)
          # M.requires_grad=True
          # M = torch.autograd.Variable(M.data, requires_grad=True)
        # print(M[:,:,0])

        V = self.V.to(device)
        B = self.B.to(device)

        self.E = E.clone().detach().requires_grad_(True)
        self.out = B + E@V + self.alpha*tl.tenalg.mode_dot(tl.tenalg.mode_dot(M, E, 0), E, 1)
        self.out = self.out[0]
        return self.out

    def params(self):
        return {'M': self.M, 'E': self.E, 'B': self.B}
```

Skoltech

```python
class TensorLayer3order(torch.nn.Module):
    def __init__(self, n_input, n_output, rank = 50, alpha2=1e-1, alpha3=1e-2):
        super(TensorLayer3order, self).__init__()
        self.rank = rank
        self.V = torch.nn.Parameter(torch.randn(n_input, n_output, requires_grad=True, dtype=torch.float32), requires_grad=True)
        self.B = torch.nn.Parameter(torch.randn(1, n_output, requires_grad=True, dtype=torch.float32), requires_grad=True)

        weight1 = torch.randn(rank, dtype=torch.float32)
        factors1 = torch.randn(n_input, rank, dtype=torch.float32)
        factors2 = torch.randn(n_input, rank, dtype=torch.float32)
        factors3 = torch.randn(n_output, rank, dtype=torch.float32)
        factors1 = [factors1, factors2, factors3]

        self.weight1 = torch.nn.Parameter(weight1, requires_grad=True)
        self.factors1 = torch.nn.Parameter(factors1[0], requires_grad=True)
        self.factors2 = torch.nn.Parameter(factors1[1], requires_grad=True)
        self.factors3 = torch.nn.Parameter(factors1[2], requires_grad=True)

        weight2 = torch.randn(rank, dtype=torch.float32)
        factors21 = torch.randn(n_input, rank, dtype=torch.float32)
        factors22 = torch.randn(n_input, rank, dtype=torch.float32)
        factors23 = torch.randn(n_input, rank, dtype=torch.float32)
        factors24 = torch.randn(n_output, rank, dtype=torch.float32)
        factors2 = [factors21, factors22, factors23, factors24]

        self.weight2 = torch.nn.Parameter(weight2, requires_grad=True)
        self.factors21 = torch.nn.Parameter(factors2[0], requires_grad=True)
        self.factors22 = torch.nn.Parameter(factors2[1], requires_grad=True)
        self.factors23 = torch.nn.Parameter(factors2[2], requires_grad=True)
        self.factors24 = torch.nn.Parameter(factors2[3], requires_grad=True)

        self.alpha2 = alpha2
        self.alpha3 = alpha3

        self.E = None
```

Skoltech

```python
def forward(self, E):
    weight1 = self.weight1.to(device)
    factors1 = self.factors1.to(device)
    factors2 = self.factors2.to(device)
    factors3 = self.factors3.to(device)

    weight2 = self.weight2.to(device)
    factors21 = self.factors21.to(device)
    factors22 = self.factors22.to(device)
    factors23 = self.factors23.to(device)
    factors24 = self.factors24.to(device)

    batch_size = E.shape[0]
    tensor_output_2 = torch.zeros(size=(E.shape[0], factors3.shape[0])).to(device)
    tensor_output_3 = torch.zeros(size=(E.shape[0], factors3.shape[0])).to(device)

    P_cpd = (weight2, [factors21, factors22, factors23, factors24])
    for batch in range(batch_size):
        M1 = tl.cp_mode_dot((weight1, [factors1, factors2, factors3]), E[np.newaxis, batch, :], 0, copy=True)
        M1 = tl.cp_tensor.cp_to_unfolded(M1, 1)
        tensor_output_2[batch] = (E[np.newaxis, batch,:] @ M1)[0].to(device)

        P1 = tl.cp_mode_dot(tl.cp_mode_dot(tl.cp_mode_dot(P_cpd, E[np.newaxis, batch,:], 0, copy=True), E[np.newaxis, batch,:], 1), E[np.newaxis, batch,:], 2)
        P1 = tl.cp_tensor.cp_to_unfolded(P1, 1)
        tensor_output_3[batch] = P1[0].to(device)

    V = self.V.to(device)
    B = self.B.to(device)
    self.E = E.clone().detach().requires_grad_(True)
    self.out = B + E@V + self.alpha2*tensor_output_2.to(device) + self.alpha3*tensor_output_3.to(device)
    return self.out
```

Skoltech

Experimenter's starter-pack

Models

- ResNet18
- MobileNet

Data

- CIFAR10
- CIFAR100

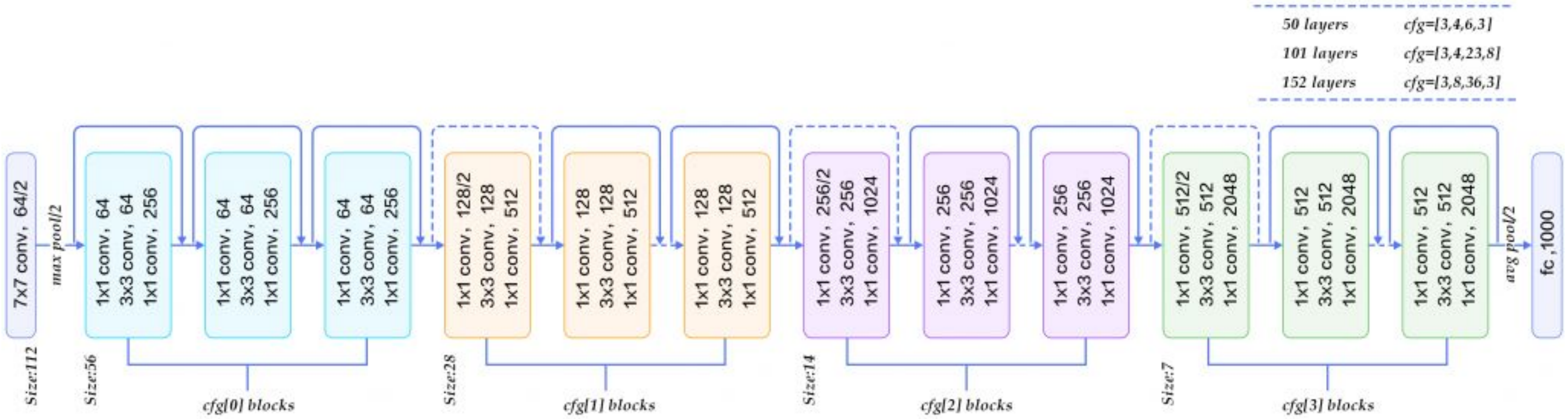
Parameters

- NTN Order
- CPD rank
- alpha
- no_diag

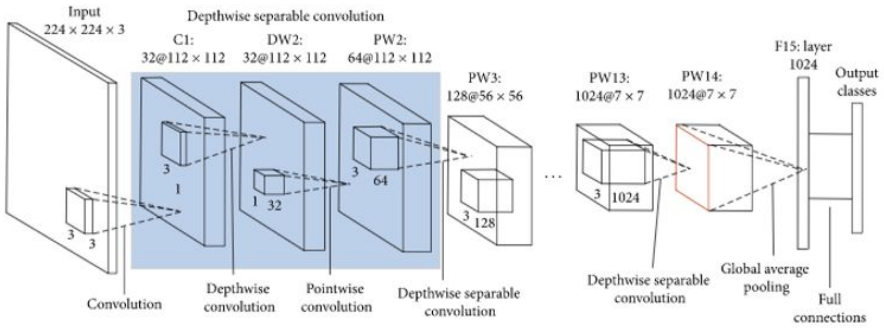
Skoltech

Pre-trained architectures

ResNet architecture



MobileNet architecture



MobileNet-V2 Architecture

Chiung-Yu Chen

Skoltech

Results: ResNet18

NTN Order	CPD rank	alpha	additional params	accuracy
1st (FC-layer)	-	-	-	0.41
2nd	full	1	-	0.37
2nd	200	0.01	-	0.47
2nd	full	0.01	diagonal subtraction	0.48
2nd	full	0.01	-	0.46
3rd	50	0.01	alpha2 = 0.001	**0.49**

Dataset	CIFAR-100
N epoch	50
batch size	32
Optimizer	Adam
Learning rate	0.01

Skoltech

Results: MobileNet

NTN Order	CPD rank	alpha	accuracy
1st	-	-	0.72
2nd	200	0.01	0.75
3rd	50	alpha = 0.01 alpha2 = 0.001	**0.76**

Dataset	CIFAR-10
N epoch	30
batch size	100
Optimizer	Adam
Learning rate	0.01

NTN Order	CPD rank	alpha	accuracy
1st	-	-	0.51
2nd	200	0.01	**0.53**
3rd	50	alpha = 0.01 alpha2 = 0.001	0.53

Dataset	CIFAR-100
N epoch	40
batch size	100
Optimizer	Adam
Learning rate	0.01

Skoltech

Methods comparison

- In general, 2nd order NTN allows to surpass accuracy of NN with FC layer.
- hyperparameter $\boldsymbol{\alpha}$ significantly affects quality of the model, and $\boldsymbol{\alpha}$ ***= 0.01*** for 2nd and $\boldsymbol{\alpha}$ ***= 0.01*** for 3rd order NTN give best results
- lowering rank with CPD allows to reduce training time without loss of model accuracy and to use 3nd order NTN, which further improves quality of the model.
- for 2nd order NTN subtracting a diagonal from tensor W_2 increased accuracy in comparison with ordinary NTN layer

Skoltech

Summary

- We planned to improve quality of existing deep models by applying to them NTN and decreasing ranks with CPD;

- We tested 2nd and 3rd ordered NTN as a final layer for different NN architectures for image classification problem on CIFAR10 and CIFAR100 datasets;

- In comparison with the original construction, modified architectures (for the majority of models) improved classification score and CPD improved training speed for 3rd order NTN;

- Since there is a wide variety of tensor's applications, it seems natural to proceed the investigation of NTN performance in some other popular machine learning tasks.