Signal Propagation analysis

CNS Project

Geremia Pompei

August 19, 2023

Abstract

Nowadays backpropagation is the main used algorithm to train a deep neural network. It is very good according prediction performance but on the other hand it pays the price of huge use of memory and processor. An alternative to this approach is Signal Propagation (SigProp) framework that is better in term of resource scaling, in fact, it uses a forward pass to train parameters. In this work there is a comparison with the SigProp original paper [2] results of section VI using a Convolutional Spiking Neural Network model on MNIST and FashionMNIST dataset. Then there is a comparison between backpropagation and sigprop on a VGG network to understand some metric difference in a DL base model.

1 Introduction

SigProp is a framework able to fit DL model parameters using a forward pass. It is different from backpropagation approach where two steps are used to fit parameters: one forward and one backward through all the layers of the network. In particular, given a dataset composed by inputs X and outputs Y, backpropagation work with the following steps:

- model computes predictions P given the inputs X (*forward propagation*)
- it's computed the *loss* between predictions P and outputs Y
- gradients of weights G are backpropagated from the last to the first layer of the network (*backward propagation*) and are stored to update and fit the weights
- according the chosen optimizer weights W are fitted using G

These steps can't be paralleled, so, backpropagation isn't good for scaling according computational power. In term of memory, after forward pass, neurons activations are stored at each layer of network to compute gradients. When gradients of weights are computed they are stored before to update the weights.

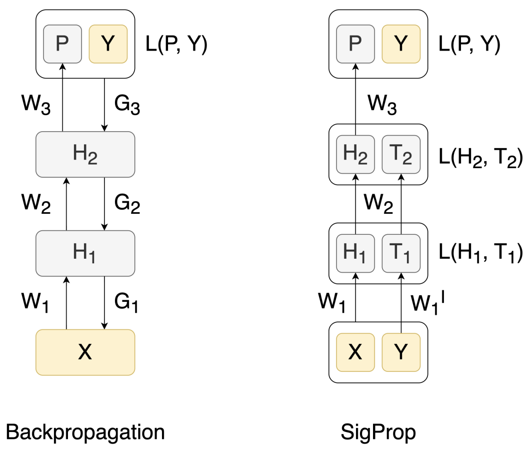


Figure 1: Backpropagation vs SigProp framework schema

So, during training, a big quantity of memory should be used to store needed data.

SigProp instead has a different approach with respect to backpropagation. For learning it uses a forward step for each layer computing the loss between the hidden representation of the inputs X and the outputs Y. As in backpropagation there is a dataset composed by inputs and related outputs (label or class in case of classification). In SigProp each layer l (with weights W_l) of the network (except the first layer where specific layers are used for X and Y because they have different dimensions) processes inputs representations H and outputs representations T (they are activations of previous layer $l-1$) concatenated together: $[H_l, T_l] = W_l[H_{l-1}, T_{l-1}]$. After that is computed the output of one layer the mapping of inputs and outputs ($[H_l, T_l]$) are separated and is computed a loss between them. After the computation of the loss the gradients of the weights of the current layer are computed and through an optimizer the weights are trained. In term of computation and memory this approach should be better than backpropagation because:

- for computation SigProp is parallelizable, so, scales better than backpropagation. During the update of the weights of a layer the next one can start to train
- about memory not all the gradients and activations of neurons are stored in the same time, but, only the part used for the current layer. In fact after the training of a layer its data (the ones used for its fitting) can be deallocated

The two approaches are represented in Figure 1.
The datasets used during this work are:

- *MNIST*: famous dataset composed by images of written digits from 0 to

9 as inputs and the related digits as outputs. There are 60000 example in training set and 10000 in test set and images are in black and white (1 channel) and 28x28 (width and height of images)

- *FashionMNIST*: this dataset is composed by fashion images of Zalando articles. Except the subject of images the features of this dataset are the same of MNIST (number of training and test set and dimensions of images)

The two analysis are executed using different models:

- *Convolutional Spiking Neural Network*: a convolutional neural network with a spiking layers that can have a surrogate function when is needed to backpropagate the gradients. This model is created on the description of the orignal paper [2] at the section VI
- *VGGb8*: the same model used from the original paper for a certain experiment. In this case is not emulated the same experiment of the original paper but it is chosen VGG model because is a common model used in literature

2 Method

2.1 SigProp framework

SigProp framework is used in a forward pass for each layer of a DL model. During training X and Y are used in input of the network (at each level of the network there are H_l and T_l projection of them). Both matrices are mapped in an other space using a layer of the network. At layer l is computed:

$$[H_l, T_l] = W_l[H_{l-1}, T_{l-1}]$$

This is true except for the first level of the network where are used different layers to map X and Y. This is done because the two matrices have different dimensions

$$H_1 = W_1 X \quad T_1 = W_1^I Y$$

After this mapping all H and T will have the same shape, so, they can be concatenated and use the same layer to be projected. After the projection it's computed a loss between H_l and of T_l. In particular is used the L2 distance loss

$$L_{L2Loss}(H_l, T_l) = ||H_l - T_l||_2$$

for intermediate layers and a specific loss, according the given task, for the last layer. For example, usually for classification tasks is used the cross entropy loss. Loss is useful to compute the gradients of weights G_l and the fitting weights W_l using a given optimizer.

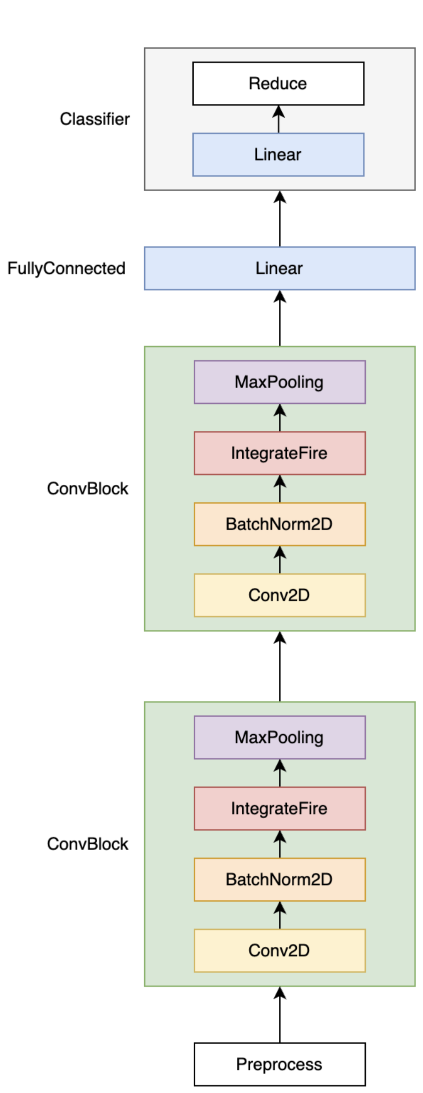
SigProp is implemented exploiting the pytorch framework to be modular with different architectures. In fact for each epoch the SigProp trainer is able to iterate each children (layer) of a pytorch module to compute the previous steps. The pseudocode of the implementation of SigProp trainer is available at Algorithm 1. Here the function $.detach()$ applied on H_{new} and T_{new} tensors is used to avoid that is applied the backpropagation for lower layers. In this way these tensors are detached from pytorch computational graph.

Algorithm 1 SigProp pseudocode

```
$X, Y$ training dataset
$epochs$ number of epochs used for training
$model$ model to train
$W^I$ layer to embed $Y$ matrix in $T_1$
$L$ loss function of last layer that is specific for the task of this dataset
$optim$ optimizer used to update weights

for $e \in epochs$ do
  $H, T \leftarrow X, Y$

  for $W_l \in model.layers$ do
    if $layer$ is the first of the model then
      $H_{new} \leftarrow W_l X$
      $T_{new} \leftarrow W_1^I Y$
    else
      $[H_{new}, T_{new}] \leftarrow W_l[H, T]$
    end if

    if $layer$ is the last of the model then
      $loss \leftarrow L(H_{new}, Y)$
    else
      $loss \leftarrow L_{L2Loss}(H_{new}, T_{new})$
    end if

    $optim.zero\_grad()$
    $loss.backward()$
    $optim.step()$

    $H, T \leftarrow H_{new}.detach(), T_{new}.detach()$
  end for
end for
```

2.2 Convolutional Spiking Neural Network

Convolutional Spiking Neural Network (ConvSpikeNN) is a model taken from the original paper of SigProp [2] at section VI. The model architecture, that is represented at Figure 2 is composed by 4 layers: 2 *ConvBlock* layers, a *FullyConnected* layer and the *Classifier*. The *ConvBlock* is composed by a convolutional



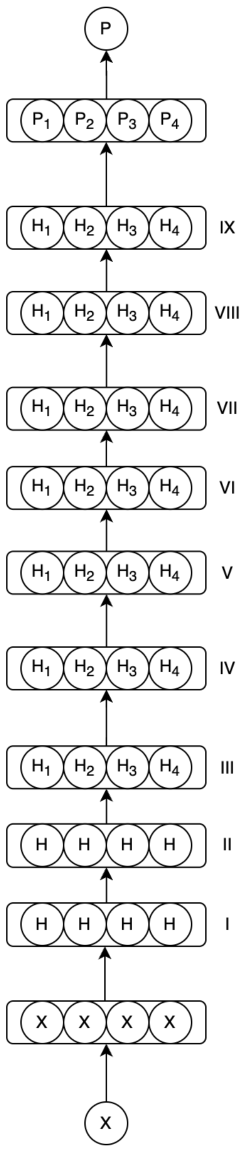

P
P_1 P_2 P_3 P_4
H_1 H_2 H_3 H_4 IX
H_1 H_2 H_3 H_4 VIII
H_1 H_2 H_3 H_4 VII
H_1 H_2 H_3 H_4 VI
H_1 H_2 H_3 H_4 V
H_1 H_2 H_3 H_4 IV
H_1 H_2 H_3 H_4 III
H H H H II
H H H H I
X X X X
X

Figure 2: On the left there is the Convolutional Spiking Neural Network architecture. On the right there is the representation of activations for each layer of the network

layer followed by a batch normalization layer, followed by a spiking layer (in particular this is an Integrate and Fire layer) and at the end there is a max-pooling layer. The *FullyConnected* layer is a simple linear layer that project the flattened representation of the output of last ConvBlock in another high dimensional representation. The *Classifier* is composed by a linear layer that is able to map the high dimensional representation in input in logits related to output classes and a *Reduce* layer that is able to take the matrices of each time of spiking layers and computing the mean. *Preprocess* and *Reduce* layers are used respectively to replicate and reduce a matrix n times. This is useful to work with spiking layers to provide the input current I for each time of integration. The *IntegrateFire* layer works as described in the pseudocode of the Algorithm 2.

Algorithm 2 SigProp pseudocode

```
$X$ inputs activations
$n$ number of times used for integration
$spike\_thr$ spiking threshold to establish the value of a spike
$spiking\_function$ function able to map for each activation of V if there is a
spike (1) of not (0)

$V \leftarrow 0$
$outputs \leftarrow []$
for $I \in X.split\_tensor(times)$ do
    $V \leftarrow V + I - spike\_thr$
    $fire \leftarrow spiking\_function(V)$
    $outputs.append(fire)$
end for
return $outputs$
```

The *spiking_function* is associated to a surrogate one to allow the propagation of gradients when is used a training criterion that requires it (e.g. back-propagation). The original spiking function is

$$fire = V > 0$$

The surrogate spiking function instead is

$$fire = \frac{1}{\pi} arctan(\pi V) + \frac{1}{2}$$

and its derivative is

$$\frac{1}{1 + (\pi x)^2}$$

When is used the surrogate spiking function is applied the function $V > 0$ in forward pass while the derivative of the surrogate $\frac{1}{1+(\pi x)^2}$ in backward pass as the implamentation of the code of the article [1].

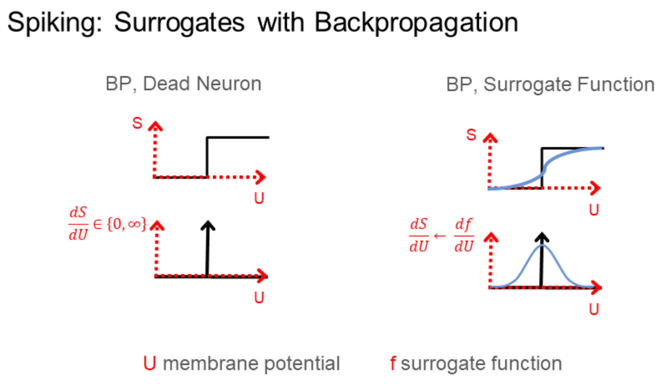

Spiking: Surrogates with Backpropagation
BP, Dead Neuron
BP, Surrogate Function
S
U
$\frac{dS}{dU} \in \{0, \infty\}$
U
S
U
$\frac{dS}{dU} \leftarrow \frac{df}{dU}$
U
U membrane potential
f surrogate function

Figure 3: On the left there is the original spiking function on top and its derivative below. On the right side there is the representation of the surrogate spiking function in blue color. Below there is its derivative

2.3 Configurations and metrics

Being a comparison with the original paper is not used a model selection phase. Best hyperparameters suggested from the papers are selected and used on all the following analysis (both for convolutional spiking model and vgg).

Metrics chosen to compare different methods on training and test sets are:

- *MSE*: the same error of the original paper on test set
- *Time*: the mean time used to compute a training epoch
- *Memory*: the mean of memory usage detected after each training epoch
- *Accuracy*: accuracy metric

3 Experimental results

There are two analysis run on MNIST and FashionMNIST dataset:

1. Comparison of metrics computed after training of ConvSpikeNN model using shallow trainer (Shallow, trains only the classifier), backpropagation trainer with surrogate spiking function (BP Surrogate), sigprop trainer with surrogate function (SP Surrogate) and sigprop trainer without surrogate function (SP Voltage). MSE value computed on test set are compared also with the values of original paper (PaperMSE). Results are shown on Table 1 for MNIST and on Table 2 for FashionMNIST

Trainer	**MSE**	**PaperMSE**	**Time**(s)	**Memory**(MB)	**Accuracy**
Shallow	2.3357	7.24	0.37	9.6	15.58 %
BP Surrogate	0.4612	0.84	1.11	21.26	84.95 %
SP Surrogate	2.3013	1.01	1.65	20.4	11.35 %
SP Voltage	2.3004	2.63	0.47	20.33	11.35 %

Table 1: Convolutional Spiking Neural Network results on MNIST dataset

Trainer	**MSE**	**PaperMSE**	**Time**(s)	**Memory**(MB)	**Accuracy**
Shallow	2.4524	16.42	0.27	18.14	12.04 %
BP Surrogate	0.4691	6.7	1.1	21.26	83.23 %
SP Surrogate	2.3025	9.51	1.65	20.4	10 %
SP Voltage	2.3025	10.68	0.47	20.33	10 %

Table 2: Convolutional Spiking Neural Network results on FashionMNIST dataset

2. Comparison of metrics computed after training of VGG model using shallow trainer, backpropagation trainer and sigprop trainer. Results are shown on Table 3 for MNIST and on Table 4 for FashionMNIST

4 Conslusion

According final results, both for MNIST and FashionMNIST, in analysis related to ConvSpikingNN, SP is slightly better then Shallow and worse than BP in term of MSE. SP Surrogate and SP Voltage seem to behave more or less in the same way according MSE. Values of PaperMSE are different from the ones of this analysis but the ranking remains the same (BP is the better followed by SP and Shallow).

In VGG analysis, both for MNIST and FashionMNIST, in term of MSE Shallow and SP behaves in the same way while backpropagation is different and reaches very good results.

According to Time in general the worst trainer is SP because is not parallel, so it is expected to have this kind of result. On the other hand Shallow trainer is the best because is trained only the classifier. BP is better than SP because is optimized in pytorch framework while SP is implemented in a custom and

Trainer	**MSE**	**Time**(s)	**Memory**(MB)	**Accuracy**
Shallow	2.3003	1.09	111.41	13.27 %
BP	7.69e-07	3.48	430.02	100 %
SP	2.3014	9.76	332.82	11.35 %

Table 3: VGG Neural Network results on MNIST dataset

Trainer	**MSE**	**Time**(s)	**Memory**(MB)	**Accuracy**
Shallow	2.3168	1.09	120.58	8.78 %
BP	1.17e-06	3.49	430.15	100 %
SP	2.303	9.83	332.73	10 %

Table 4: VGG Neural Network results on FashionMNIST dataset

not optimized way. SP can reach better performance than BP in term of time parallelizing it.

According Memory, as expected, Shallow trainer uses less memory because stores resources to train only the classifier. SP is better than BP because each time allocate data to train the current layer and then deallocate them when trains the next layer while BP should store all resources according each layer of the network, so use a lot of memory.

In term of accuracy BP is very good while Shallow and SP trainers reach very bad results that are more or less the same. Due to these results SP is not a good algorithm that can replace BP according this kind of implementation.

References

[1] Jason K Eshraghian et al. “Training spiking neural networks using lessons from deep learning”. In: *arXiv preprint arXiv:2109.12894* (2021).

[2] Adam Kohan, Edward A. Rietman, and Hava T. Siegelmann. *Signal Propagation: A Framework for Learning and Inference In a Forward Pass*. 2022. arXiv: `2204.01723 [cs.LG]`.