Master Thesis

Analysis of Deep Neural Networks based on Feedback Alignment algorithms

Luca Di Pasquale

Supervisor

Prof. Maurizio Martina

Final Report for Master
Thesis in Electronic Engineering

Politecnico di Torino
Italia, Torino
Aprile 2019

Abstract

The thesis work deals with the implementation of three different algorithms to be used during the training of a network to replace the classic algorithm of backpropagation using one of them [15]. They are based on the Random Feedback Alignment, which reduces the computational work required to obtain the transposed of the forwarding weights necessary during the classical backpropagation using random matrices [13]. The error information is propagated from the last layer to the others using the random matrices. The three algorithms studied modify the path used to transport the error from the last layer to each previous one [15]. They are implemented on different networks to understand which algorithm is more indicated for a deep one as the Residual Neural Network (ResNet) and the performance differences reached are computed. There are many advantages using random propagation: the possibility to parallelize the computational work during backpropagation; the forward and backward steps performed simultaneously; reducing the memory accesses. Variations of the Feedback Alignment algorithm are studied to understand if they can give back better accuracies on a deep network as ResNet [18]. After simulations performed using the PyTorch framework, there is a theoretical study of the computational requirements differences using the various solutions considered during the work and it is analyzed which implementations can give a better alternative to backpropagation.

Contents

1 Introduction and targets to reach **15**

2 State of the art **17**

2.1 AI and Neural Networks 17

2.1.1 Linear and convolutional layers 18

2.1.2 Forward 18

2.1.3 Training of the network 19

2.1.4 Stochastic gradient descent 19

2.1.5 Backpropagation 20

2.1.6 Problem due to classic error propagation 21

2.2 Optimizations to backpropagation 22

2.2.1 Random Feedback Alignment 22

2.2.2 Alternative Direction Method of Multipliers approach 25

2.2.3 Moore-Penrose Pseudo Inverse 26

2.2.4 Invertible Networks 26

2.3 Deep Neural Network 26

2.3.1 Residual Neural Networks: ResNet 27

2.3.2 Batch Normalization 28

2.4 PyTorch: Framework for deep networks 28

2.4.1 Autograd function 28

2.4.2 Dataset 29

3 Work development **31**

3.1 Set-up of environment 31

3.1.1 PyTorch 31

3.1.2 CUDA 32

3.1.3 Nvidia Geforce 920m 32

3.1.4 Tesla K40 32

3.1.5 Docker 33

3.1.6 Google Colab with Tesla K80 33

3.2 Develop simple network with numpy 33

3.2.1 FA 35

3.2.2 DFA . . . 35
3.2.3 IFA . . . 36
3.2.4 Results . . . 37
3.3 Implementation in optimized networks . . . 40
3.4 Approch to PyTorch . . . 43
3.4.1 Most important PyTorch functions . . . 44
3.4.2 Custom module definition . . . 44
3.4.3 Register Hook . . . 45
3.4.4 Implementations of algorithms with PyTorch . . . 45
3.4.5 Results BP, FA, DFA and IFA . . . 49
3.4.6 Results BP, FA and DFA . . . 51
3.4.7 Convolutional network without module . . . 52
3.4.8 Results . . . 54
3.4.9 Convolutional network with module . . . 56
3.4.10 Results . . . 59
3.5 Developing of ResNet with FA and DFA algorithms . . . 61
3.5.1 Problem implementing custom modules . . . 62
3.5.2 Developing of ResNet switching weights values for FA . . . 62
3.5.3 Developing of ResNet using register hook function for DFA . . . 63
3.5.4 Results Back Propagation . . . 64
3.5.5 Results FA . . . 64
3.5.6 FA Nokland and Lillicrap . . . 65
3.5.7 Results DFA . . . 66
3.5.8 Failing FA and DFA and state of the art comparison: approach to sign concordance . . . 66
3.5.9 Batch Manhattan . . . 67
3.6 Simulations results and comparison with papers ones . . . 67
3.6.1 FA with initial sign concordance . . . 67
3.6.2 FA with Batch Manhattan and initial sign concordance . . . 68
3.6.3 DFA gradients signed . . . 69
3.6.4 FA with Batch Manhattan and sign concordance every epoch . . . 70
3.6.5 FA with Batch Manhattan and random weights generated every batch . . . 71
3.6.6 FA using only sign concordance every epoch . . . 72
3.6.7 ResNet34 with Batch Manhattan and sign concordance every epoch . . . 72
3.6.8 Error propagation using forward weights signs scaled by stdv and Batch Manhattan . . . 73
3.6.9 Error propagation using forward weights signs scaled by 0,001 and Batch Manhattan . . . 74

3.6.10 Error propagation using forward weights signs scaled by stdv . 74
3.6.11 Comparison with paper results 75
3.7 Computational efforts evaluation 76
3.7.1 Evaluation of operations required for algorithms 77
3.7.2 BP operations 79
3.7.3 FA operations 80
3.7.4 DFA operations 80
3.7.5 DFA+Signgrad operations 81
3.7.6 FAsign+Signgrad operations 82
3.7.7 FArand+Signgrad operations 83
3.7.8 FAsign operations 83
3.7.9 Sign+signgrad operations 84
3.7.10 Sign operations 85

4 Conclusions **87**
4.1 Discussion about results 87
4.1.1 Promising implementations 88

List of Figures

2.1 Schematic structure of an axon 17
2.2 Computation of partial derivatives through the network 20
2.3 Schematic of how backpropagation works 21
2.4 Schematic of how Feedback Alignment works 23
2.5 Schematic of how DFA works 24
2.6 Schematic of how IFA works 25
2.7 Basic Block of a ResNet 27

3.1 Test accuracy on a three hidden layer network 37
3.2 Test accuracy on a five hidden layer network 38
3.3 Test accuracy on a five hidden layer network with more neurons . . . 38
3.4 Test accuracy on a five hidden layer network with additional neurons 39
3.5 Test accuracy on optimized linear network with three hidden layers . 41
3.6 Test accuracy on optimized linear network with five hidden layers . . 41
3.7 Test accuracy on optimized linear network with more neuros 42
3.8 Test accuracy on a optimized linear network with increased neurons . 43
3.9 Test accuracy on a linear network classifiying Cifar10 images 49
3.10 Test accuracy on a linear network classifiying Cifar10 images with modified initializations 51
3.11 Test accuracy on a convolutional network classifiying Cifar10 images . 54
3.12 Test accuracy on a linear network classifiying Cifar10 images with different initialization 55
3.13 Test accuracy on a convolution network using custom modules to classify Cifar10 59
3.14 Test accuracy on a convolution network using only FA algorithms . . 60
3.15 CPU and GPU architectures comparisons [16] 76
3.16 How the matrix is indexed in optimized way [9] 78

List of Tables

3.1 Graphs of linear network classifying Cifar10 images 50
3.2 Graphs of linear network on Cifar10 using different initializations . . 51
3.3 Graphs of convolutional network on Cifar10 with (0, 0.001) initialization 54
3.4 Graphs of convolutional network on Cifar10 with (0, 0.005) initialization 55
3.5 Graphs of convolutional network on Cifar10 with (0, 0.005) initialization 59
3.6 Graphs of convolutional network on Cifar10 with (0, 0.005) initialization 60
3.7 Simulations ResNet18 with backpropagation algorithm 64
3.8 Simulations ResNet18 with Feedback Alignment 64
3.9 Nøkland weight initialization on ResNet18 with FA 65
3.10 Lillicrap weight initialization on ResNet18 with FA 65
3.11 First weight initialization on ResNet18 with DFA 66
3.12 Second weight initialization on ResNet18 with DFA 66
3.13 Results of ResNet18 with FA and sign concordance 67
3.14 Results of ResNet18 with FA signed scaled learning rate 68
3.15 Results of ResNet18 with the Batch Manhattan and initial sign concordance 68
3.16 Results of ResNet18 using DFA and Batch Manhattan 69
3.17 Results of ResNet18 using DFA and Batch Manhattan with different weights 69
3.18 DFA using weights scaling with depth and Batch Manhattan 69
3.19 Results of ResNet18 using FA, Batch Manhattan and sign concordance every epoch 70
3.20 Results of ResNet18 using FA and random weights generated every epoch 71
3.21 Results implementing only the sign concordance every epoch 72
3.22 Results implementing sign concordance every epoch and Batch Manhattan on ResNet34 72
3.23 Results using the signs to brought back the error information 73
3.24 Results implementing a different scale factor for forward signs 74
3.25 Results using only signs to brought back the error information 74
3.26 Operations Bandiwth comparison [9] 77
3.27 Operations Bandwidth comparison [9] 78

3.28 Summary of computational complexity and accuracies 86

Chapter 1

Introduction and targets to reach

One of the most efforts during the training of a network is due to the computational requirement for the transpose of the weights required during backward and the management of memory accesses since they increase the time required to perform the operations. The transpose operation is necessary to compute the different gradients of outputs and inputs of each layer. Once the output of the whole network is computed, the loss referred to the label assigned to each input is calculated and the gradients of the loss with respect the last output is used to train the network. This term is used to compute the partial gradients of inputs of each layer and it is required to multiply this term to the transposed weight used during forward for the error propagation. The possibility to replace the transposed forward weights with random ones has been proposed by Lillicrap [13]. Successively Nøkland proposed two additional algorithms to one of the previous paper, the Direct Feedback Alignment and Indirect Feedback Alignment [15]. These three algorithms are important since they expose the possibility to change operations required to train a network, with the possibility to reduce memory accesses or computations in the training phase of a network. The targets to reach in this thesis work concern the possibility to apply them to a Residual Neural Network (ResNet), to compute the final accuracy to understand which one of the three implementations are best suited for our purpose and to analyze the possible reduction in the computational cost achieved through the different solutions. The structure of the network has been chosen because it is one of the most accurate in the image recognition task and its structure is developed to an efficient error information transport, avoiding vanishing gradients without an increase of computational complexity. The work has been developed starting from a simple network to understand how the accuracy values will be modified by the algorithms and more complex networks are used on the solutions more promising. The ResNet has been studied in the last part to understand the results obtained with the different solutions adopted in the previous networks and to analyze the effect obtained in deeper networks. Other optimizations presented recently were considered to obtain a complete understanding of the propagation of the error in deep

networks [18] [20] [22]. An analysis of the computational cost required by the most important solutions studied is presented in the last chapter in order to understand if the backpropagation can be replaced with algorithms with less computational complexity or if it is possible reduce the memory required or the accesses.

Chapter 2

State of the art

2.1 AI and Neural Networks

In the last decades, there have been improvements in artificial intelligence (AI) field, due to the increase of the computational capability of the calculators and in the numbers of fields interested in the performances that can be achieved [23]. The application of AI has reached many different environments like medical diagnostic, images recognition, autonomous driving cars and others. They are all based on the ability of a system, not programmed to perform a specific work, to learn how to respond to inputs from the external world and to take decisions after a training step. This means the possibility that a system can learn and perform actions to respond to inputs not provided during the training. The neural network is the basic unit behind the artificial intelligence system and it is associated with the brain for its structure. The element similar to the neuron is the axon, which inputs are multiplied to weights and a non-linear function is applied to the sum of them:

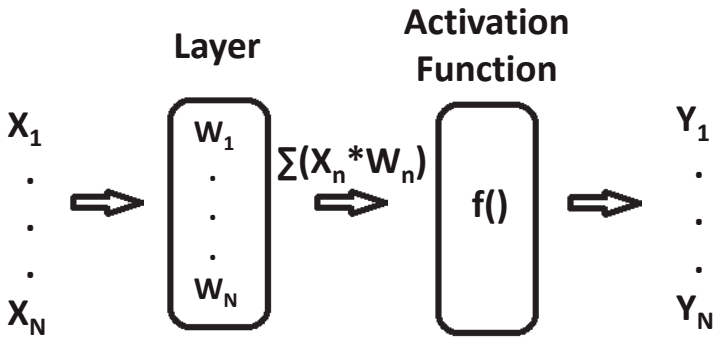

Figure 2.1: Schematic structure of an axon

The training phase is required since the weights are initialized randomly, with some distribution used to be faster, so at the beginning, the network is not able to classify inputs received.

This ability is acquired after that the weights values are updated, so the network can approximate complex non-linear functions and recognize inputs never seen before. There are different methods to train a network:

- Supervised
- Unsupervised
- Semi-supervised

The first one uses a dataset of images in which a label is associated with each input to indicate their class. This label is used at the end of the forward step to compute the loss term. The derivative of this term indicates the error committed during the forward phase and it is propagated back to each layer to update the values and get better accuracy. The second method uses a dataset without labels and the task of the machine is to be able to extract common features from the inputs and be able to distinguish them. An example can be a machine to check spam incoming. The third method is a mix of the previous two.

2.1.1 Linear and convolutional layers

A neural network is based on stacked layers and an activation function is performed at the output of each layer. The layers of a network can be different, due to the function that is applied to inputs. There are linear and convolutional layers for example. The first uses a linear function, where the inputs are multiplied for the weights and a term of bias can be added. The second type of layer performs product iteratively between the input images and a kernel of weights. This method is preferred for a deep network, in which there is more than one hidden layer. It allows to reduce the dimensions of the input for each successive layer and it is able to reach a higher level of abstraction, generating a features map [23].

2.1.2 Forward

During the forward phase, outputs are computed from inputs applying the right function of the layer. The output of the whole network is a probability term for every class and the higher one is the choice of the network for the class where the input image belongs. Since the weights are random the training phase is required. The equations that are used during forward are:

$$a_n = x_{n-1} * W_{n-1} + b_{n-1} \tag{2.1}$$

$$x_n = f(a_n) \tag{2.2}$$

where indicating with n the specific layer of the network:

- x_{n-1} indicates the input;
- W_{n-1} the weights;
- b_{n-1} the bias term added;
- a_n the output of the layer;
- f() the activation function applied;

The multiplication between weights and inputs can be replaced by convolution if there is a convolutional layer. There are different types of activation function that it is possible to apply, like Rectified Linear Unit, Tangent Hyperbolic or the Sigmoid. The activation function purpose is to obtain a probability term for each class and the higher term is the choice of the network referred to inputs received. Each activation function can be more adapted for a specific classification purpose, related to the subject that the network must classify.

2.1.3 Training of the network

The purpose of the training is obtaining probability values more accurate in the class where the input belongs. This phase requires computational efforts in relation to the number of layers and their weights that are related to the number of features requested to be extracted. A huge number of weights can have a negative effect, resulting in a network not able to learn due to overfitting, that occurs when a network learns too much from training dataset and it is not able to recognize different inputs. Weights and biases values will be updated using a training set of images, where the class is indicated by a vector in which the 1 (100% probability) indicates the right class and other are 0. After the first forward it is possible to evaluate the error or loss computed by the networks since its output will be a vector with dimensions like the number of classes and a probability value for each class is computed. The update is obtained in different steps: loss computing, obtaining the gradient of the loss referred to the output and propagation of this value through all the layers.

2.1.4 Stochastic gradient descent

The Stochastic Gradient Descent method allows the computation of partial derivatives of the loss referred to the different weights and biases of the whole network. In this way, all values can be tuned to obtain better accuracy. This method is based on the chain rule, the possibility to compute gradients of inputs and outputs of a function, performing iteratively multiplications between gradients of the last output

with the partial derivative of output with respect to input of a layer as can be seen in the figure.

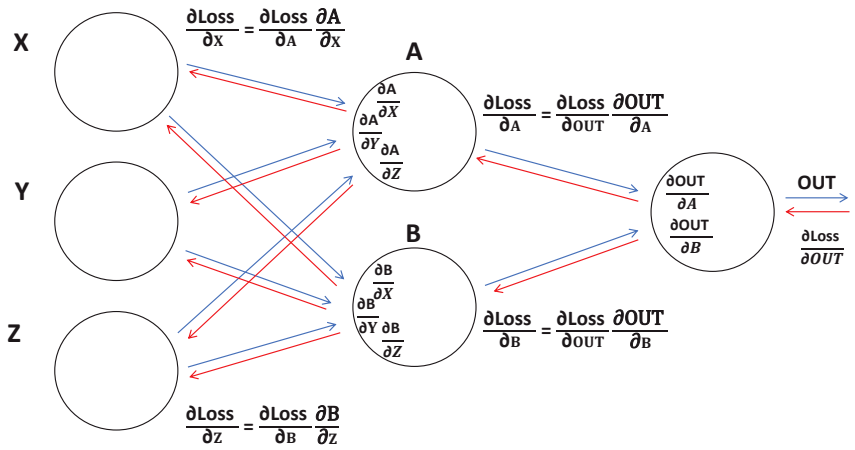

Figure 2.2: Computation of partial derivatives through the network

The purpose is to compute local gradients that indicates how the error affects a variable and this term will be used to reach a minimum of the function that can approximate in the better way the whole dataset information. The method tries to reach the minimum moving in the direction that allows obtaining a smaller loss value, using this equation:

$$W_n = W_{n-1} - lr * \delta W_{n-1} \tag{2.3}$$

where lr is the learning rate and δW$_{n-1}$ is the gradient of the loss referred to the weight. The learning rate is an important factor to consider during the training phase since it is the step size that allows moving in the direction of the minimum. Using a large learning rate it is hard to find the right direction, while if it is too small a huge number of steps is required and this is not an efficient way to reach the minimum.

2.1.5 Backpropagation

The step to compute partial gradients starting from the last layer to the first one is called backpropagation. It refers to the computation of each value going along the network in direction opposite during the forward. This error respect to each weight or bias is used to update their values as can be seen in the Stochastic Gradient Descent equation, trying to minimize the function described by all features of the whole network.

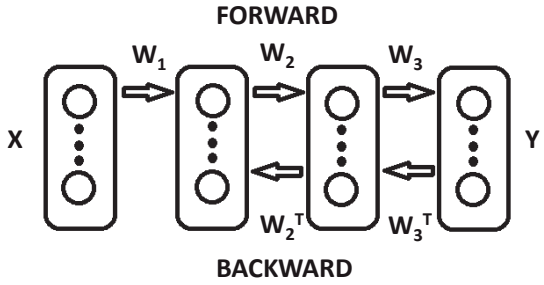

Figure 2.3: Schematic of how backpropagation works

The backpropagation is actually the most efficient way to train a network, allowing to reach results with higher accuracy. It is the slower part and the one that requires most computational efforts to load variables computed during forward, like the results of activation function, to obtain transposed of the weights used to compute gradients and to update the weights. The equations show the error computation, from the derivative of the loss function, and how the error is used to compute local gradients [15]:

$$e = \delta output = \frac{\Delta J}{\Delta output} \tag{2.4}$$

$$\delta a_2 = (W_3^T * e) \cdot f'(a_2) \tag{2.5}$$

$$\delta a_1 = (W_2^T * \delta a_2) \cdot f'(a_1) \tag{2.6}$$

$\delta \mathrm{a}_n$ refers to the gradient of the loss with respect to the input of the layer, while the gradient of the output is the term $\delta \mathrm{a}_{n+1}$ that is multiplied for the transposed weight.

2.1.6 Problem due to classic error propagation

The transposed computation requires an effort that increases the difference in time between the forward and backward phases during the training. In addition, there is a waste of time related to the error propagation through each layer that can slow down the entire backward phase. For the backpropagation it is also necessary to save all results of activation functions in memory because they must be used in the backward step. This requires memory accesses that slow down the operations and also a portion of memory is busied to save temporarily these values. For this reason, it is thought that other solutions can reach the same performances as the human brain. It is supposed that the learning phase of a brain is based on a parallel input elaboration and output evaluation, that allows faster learning than actually possible in neural networks.

2.2 Optimizations to backpropagation

There are many algorithms with the purpose to avoid problems that occur during the updating of network values using the classical backpropagation: for the transpose computation some hardware solution has been proposed, where the memories are able to save values of weights and give back automatically their transposed [6], but also software solutions that are based on algorithms where random matrices substitute transposed weights [15]. Other optimizations are based on different updating methods to Stochastic Gradient Descent, like the Alternative Direction Method of Multipliers (ADMM) or Moore-Penrose Pseudo Inverse [7] [19]. Some solutions as invertible networks try to provide better management of the memory required during the training phase with also speed up [21].

2.2.1 Random Feedback Alignment

The Random Feedback Alignment is based on the possibility to use fixed random matrices to transport error information in a network with one or two hidden layers [13]. These matrices replace the original ones avoiding transposed computation and obtaining the same performance of the classic method. This is an important step to understand how the training works since it breaks the link of using forward weights during the backpropagation. This algorithm is called Feedback Alignment (FA) and later Nøkland proposed other two algorithms following the Lillicrap's idea [15], where the error is brought back creating different paths between the last layer and each layer. This is important since in the classic backpropagation and FA the gradients of each layer are computed going through the network in the opposite direction of forwarding, while with the ones proposed by Nøkland the different paths allow to update simultaneously or compute gradients going in the same direction as forward. The two methods proposed are called Direct Feedback Alignment (DFA) and Indirect Feedback Alignment (IFA). A similar study is performed by Qianli Liao, where the FA method is applied but some modifications are performed and interesting results are shown [18].

FA

The difference between Feedback Alignment and backpropagation is the method used to bring back the error e to each layer replacing the transposed weights, referred as W^T, with fixed random weights, referred as B [15].

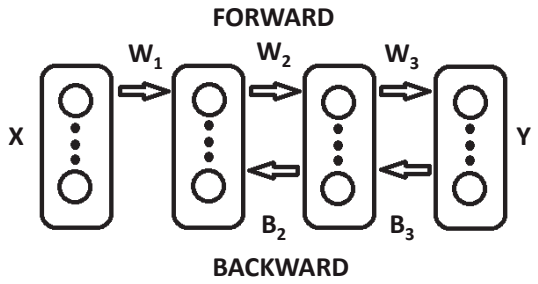

Figure 2.4: Schematic of how Feedback Alignment works

It is demonstrated that initially the training goes worse than the classic method, but the network is able to learn how to use the fixed random weights and to obtain the same perfomances [13]. The condition that must be respected is [13]:

$$e^T WBe > 0 \tag{2.7}$$

where W is the matrix used during forwarding, B the fixed random matrix used during backward and e is the error of the last layer [13]. The condition means to have the B matrix able to behave as the transpose of the W matrix, allowing the network to train in a similar way [13]. This is possible adjusting forwarding weights of the networks since they start to act like Pseudo Inverse and so the error information can be transported using fixed random matrices. The initializations of values are based on successive iterations with weights between 0.0001 to 0.01, while the random weights are initialized in the interval -0.5 to 0.5 in order to meet the best behaviour during the training [13].

The equations related to this algorithm are [15]:

$$\delta a_2 = (B_3 * e) \cdot f'(a_2) \tag{2.8}$$

$$\delta a_1 = (B_2 * \delta a_2) \cdot f'(a_1) \tag{2.9}$$

It is possible to see how the random matrices replace the transposed ones, but the computation direction is almost the same as backpropagation. The Qianli Liao's solution has results in contrast to Lillicrap'ones since it is demonstrated that the magnitude of feedback weights is not relevant if a concordance of signs is respected between forwarding weights and random ones [18]. Implementations with different percentage of concordace are discussed to highlight this behaviour [18]. It is applied the technique of Batch Normalization to obtain results even better than SGD ones. This result is also due to a different implementation of SGD, where the gradients values used during the update are substituted by their signs. In this way, it is possible to avoid the problems of exploding or vanishing gradients because the magnitudes are discarded.

DFA and IFA

Nøkland proposed two systems to transport the error of the network to each layer: the first one is based on paths that connect all hidden layers to the last and it is called Direct Feedback Alignment (DFA) [15]; the second has a unique path that links the last layer to the first hidden one and it is called Indirect Feedback Alignment(IFA) [15]. In the DFA there is the possibility to perform the computation of gradients independently from the next layer, so the computations of the whole network are parallelized. In the scheme it is shown as the last layer is linked to all previous ones:

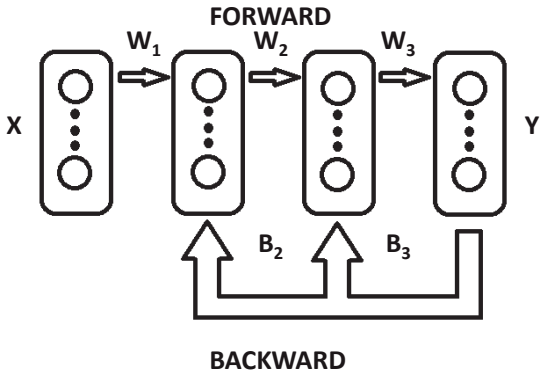

Figure 2.5: Schematic of how DFA works

The dimensions of these random matrices are different from the one used in FA since a dimension agreement in the gradient computation must be matched. The equations related to this algorithm are [15]:

$$\delta a_2 = (B_3 * e) \cdot f'(a_2) \tag{2.10}$$

$$\delta a_1 = (B_2 * e) \cdot f'(a_1) \tag{2.11}$$

As can be seen, the gradients of a layer are not propagated to others and this allows the parallelization of the process. The results are comparable to backpropagation ones and the DFA algorithm is able to train efficiently also deep networks, while the feedback implementation can't do. In the IFA the gradients computation is performed in the same order as during the forward and the major advantage is the possibility to substitute the random matrices with the one used during forward. This allows the possibility to reduce the memory requirements to store random matrices and at the same time to avoid transposed computation.

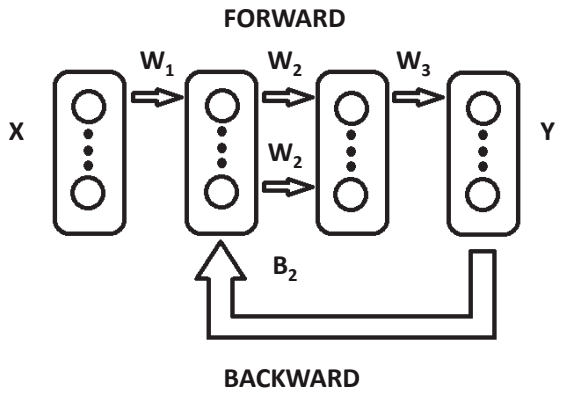

Figure 2.6: Schematic of how IFA works

In this manner it is possible to perform gradients computation while other inputs are evaluated, allowing a reduction of memory accesses for weights loading. The equations related to this algorithm are [15]:

$$\delta a_2 = (W_2 * \delta a_1) \cdot f'(a_2) \tag{2.12}$$

$$\delta a_1 = (B_2 * e) \cdot f'(a_1) \tag{2.13}$$

The results about this algorithm are not presented in the paper of Nøkland since it is only a theoretical approach and experiments are performed only on MNIST dataset, but results are not shown.

2.2.2 Alternative Direction Method of Multipliers approach

This method tries to solve the problem related to stochastic gradient descent, when there is a large dataset and using the classic method can occur phenomena of saturation, bad conditioning and be stuck in a minimum point. The aim is to allow the dataset to be divided into smaller steps and to find a solution globally in closed form [7]. In this way, the work is divided during the training phase between different cores, communicating via MPI, that analyze each a different substep, since the solution is found to be globally optimal. This means a speedup in the training phase and performance similar to the one obtained training networks on GPUs. This optimization is not applicable on the Stochastic Gradient Descent since it takes only a small portion of training dataset and performs the update of network values, incurring in the possibility of stuck in local minima. The alternating direction method is referred to the approach to the layers: these are divided in the relation between inputs and weights and a term referred to the non-linear activation function. It is possible to update networks values avoiding vanishing gradients and the updating can be performed simultaneously for every layer. The most important results are

the analysis of performance with ADMM: it reaches the same accuracies obtained with other implementations performed on GPUs, but in a short interval of time and results on different numbers of cores shown as the speedup of the training is huge.

2.2.3 Moore-Penrose Pseudo Inverse

An alternative to SGD method is the use of the Pseudo-Inverse, that is a generalization of the inverse of a matrix [19]. The difference with Extreme Machine Learning, where this matrix is used after the training phase of the network in replacement of the last layer, is that it is used to train the whole network. While SGD performs local optimizations due to limited input dimensions, being the dataset divided into batches, using the Pseudo Inverse it is possible to analyze all the n inputs together. Each input will be multiplied for its weight, added the bias term and obtained the output. So it is required to perform a division of the value of weights on the number n of inputs. The results showed the possibility for the network to be able to learn and a possible implementation is referred to cancer detection.

2.2.4 Invertible Networks

The invertible networks are based on the possibility to avoid the storage of the activations, used to compute the gradients as shown in the Stochastic Gradient Method, thanks to the ability of the layer to compute the inputs from the outputs [21]. This is possible because are used invertible functions, so during the backpropagation phase, it is performed additional computations to obtain activations and be able to compute the values of gradients. This gives the possibility to reduce the memory necessary to store net values. In the paper, it is presented a new framework based on PyTorch, called MemCNN [21]. It allows speeding up the training of the networks using invertible networks and thanks to the integration of reversible blocks and autograd functions. These invertible networks are called RevNet [21] and their results shown as can be possible train networks on Cifar10 and Cifar100 dataset with accuracies similar to ResNet, but the time required is less.

2.3 Deep Neural Network

The definition of Deep Neural Network (DNN) is applied when there is more hidden layers [23]. The increase of computation capability has allowed working with deeper networks in a reasonable amount of time and the DNN purpose is reaching a higher accuracy in the evaluation of inputs provided. This is possible because deep networks have the ability to extract more and complex features, combining them to obtain final probability values more accurate. The problem presented with

network based on a lot of hidden layers is the vanishing gradients, that is the unsuccessful transportation of the error information through the whole network. The error information is repeatedly multiplied for the weight values, usually lower than one. Different networks structures are proposed during the last years that are able to learn better than smaller ones, based on more than ten or hundreds of hidden layers, like VGG19 or ResNet101.

2.3.1 Residual Neural Networks: ResNet

The ResNet is born to solve the problem of vanishing gradients using a shortcut link between blocks of layers [11]. In this way, the output of shortcut connections are added to the output of stacked layers and it is possible to achieve better results implementing SGD, without requiring additional computational capability. The function is changed as can be seen [11]:

$$y = F(x, W_i) \tag{2.14}$$

$$y = F(x, W_i) + x \tag{2.15}$$

and the non-linear activation is applied after the addition operation. The effect is the reduction of the vanishing gradients since the shortcut connections provide information also in deep layers and the possibility to train networks of more than one hundred of layers. The ResNet is based on convolutional layers followed by Batch Normalitazion, while in other structures it is also possible to apply after the non-linear activation function [11]. A basic element that forms the ResNet can be seen:

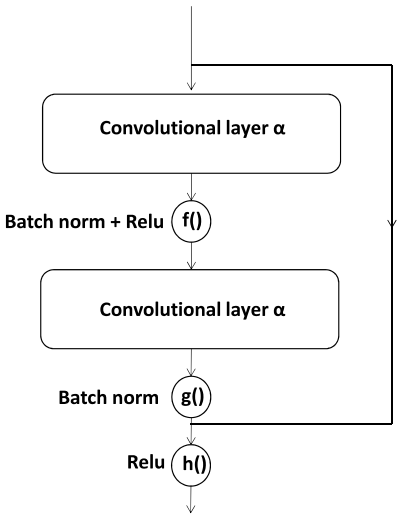

Figure 2.7: Basic Block of a ResNet

where α is the number of planes inside a layer and it can be: 64, 128, 256, 512 and others.

2.3.2 Batch Normalization

This technique has the effect to reduce the noisy elements present in the batch images taken as input, reducing the covariance between the images of a batch [18]. The Batch Normalization is very effective in the reduction of the vanishing gradients problem, that affects networks very deep. In addition, it avoids the phenomenon of Exploding Gradients that has the opposite effect [18]: the network initially starts training, the loss goes to infinite after some epochs and the network is not more able to learn. This solution allows recognizing images different from the one used for the training set, based on major freedom of the layers to each other. There is also the possibility to use a higher learning rate and the effect of overfitting is reduced, that occurs when a network learns too much from the training set and it is not able to recognize other inputs. The formula that characterizes this technique is taken from PyTorch documentation [3]:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{2.16}$$

where E[x] is the mean and Var[x] is the standard-deviation.

2.4 PyTorch: Framework for deep networks

There are many frameworks that allow a first approach on the world of machine learning, but PyTorch is one of the most efficient in memory savage and based on Python. This is a high-level programming language, with a variety of functions that allows an easy approach to machine learning through codes available also on Github repositories. This framework has some important built-in functions that allow to speed up training and different types of networks are already available.

2.4.1 Autograd function

The autograd function is one of the most important characteristics of PyTorch since it allows fast computations of the different gradients in an automatic manner, while in other frameworks this must be implemented manually [5]. This is possible since during the forward phase a dynamic graph is built, that will be used to compute gradients during the backward phase going in the reverse direction of the forward. This can be done in a faster way since the functions to compute gradients are defined at a low level during the forward phase. The autograd function is based on memory optimizations, so buffers are used and free to reduce memory requirement. Since PyTorch is based on C++ functions, it is extremely efficient since they are faster than custom functions defined in Python.

2.4.2 Dataset

A huge variety of dataset is available to train networks in supervised learning for image recognition. The most used are MNIST, Cifar10, Cifar100 and ImageNet. Usually, the dataset is divided into training images and testing images. There is the possibility of a third sub-dataset taken from the training images called validation test, used after training and before the testing. Images are associated with classes and the purpose of the neural network is to recognize the correct class to which the image belongs, giving in the output a higher probability in the right class.

MNIST

The MNIST dataset is based on images of handwritten digits, there are 10 classes, 60'000 training images and 10'000 testing. This is a simple dataset and a simple neural network based on linear layers is able to reach 80% on the test accuracy without most of the optimizations currently used. Since there are only handwritten numbers, the features to extract are relatively simple. The low error value (0.3%) obtained also implementing advanced networks is due to numbers that are difficult to recognize also for humans. The classes are divided in order to express them with a number between 0 and 9, so at class 10 corresponds the number 0. The dimension of the images are 28x28 pixels and they are in grey scale.

Cifar10

The Cifar10 dataset is based on 50'000 training images, 10'000 testing ones and 10 classes. This dataset is based on images of horses, cars, houses and other stuff. It is more difficult for a network the extraction of features with respect to the MNIST case since there is a higher complexity to be considered, like colours, ambient noise and others. The networks depth must be increased and more complex solutions must be used to reach higher accuracy. ResNet networks are able to reach accuracy about 90% using 18 layers and higher accuracy using deeper implementations. The classes contained in the dataset are plane, car, bird, cat, deer, dog, frog, horse, ship, truck. This dataset is very challenging for neural networks since there are very different objects to classify. The dimensions of the input images are 3x32x32, where 32x32 is the pixel resolution and 3 is the channels number since the images are in RGB colours.

Chapter 3

Work development

3.1 Set-up of environment

The first part of the thesis work is performed on a notebook Asus with a GPU Nvidia Geforce 920m, since it has a compute capability of 3.5 and according to Nvidia tutorials this factor is sufficient to use this GPU to compute faster than using a CPU. A partition with Ubuntu 16.04 has been installed to work with Python, in order to have a faster environment than a Windows IDE and to avoid compatibility problems related to libraries. Ubuntu has Python as a native language so it is requested only to install other libraries to have an easier approach to machine learning using specific packages. PyTorch, CUDA and Torchvision are installed. The first one is a framework built to work on machine learning with Python and it is an optimized library for machine learning. CUDA is the specific architecture and it consists of a set of C functions that allows the use of the GPU to compute faster than CPU can do. Torchvision is a package for the management of images and used to normalize the inputs before the training step.

3.1.1 PyTorch

PyTorch is a framework developed for deep learning purposes with interesting features like Python integration and an optimized management of the memory. These factors allow an easy approach to machine learning field, since also deep networks can be trained in a reasonable amount of time. The possibility to have a great number of examples on Github repositories is a good starting point to approach the study of neural networks and to implement different algorithms as in this work is done. The installation guide is provided on PyTorch site and the package can be installed through a simple command. This command is based on the preferences for the installation like using Conda or Pip package [2]. An example of the command is provided:

```
pip3 install torch torchvision
```

It is possible to work with Tensor, that is one of the main features of PyTorch. These elements are multidimensional arrays used to work with CUDA and they can be used to work with CPU or GPU. Flags can be set to track operations computed on the Tensor and in this way using the function torch.autograd the backward step is automatically computed.

3.1.2 CUDA

CUDA is the architecture that allows the GPU to perform parallel computing [16]. In the last years, many applications have been developed on CUDA GPU in order to speed up elaborations and simulations like machine learning or other scientific purposes. The number of applications is increasing and nowadays a higher number of Nvidia devices are based on CUDA architectures. In the machine learning field it allows the possibility to perform all computations on GPU, using the higher number of ALU, and many operations in parallel using threads. This is possible since different programming languages, based on C, Python, Matlab or others, allow to use the CUDA architectures for computation through native instructions.

3.1.3 Nvidia Geforce 920m

The Nvidia GPU of the Asus Notebook used in the first part of the work is a GPU with medium performances launched in March 2015, but it is sufficient to start working with simple neural networks. It has an FP32 theoretical performance of 732.7 GFLOPS.

3.1.4 Tesla K40

The next machine used to train networks is provided by the VLSI laboratory of the Politecnico di Torino and it is a workstation with a GPU Nvidia Tesla K40. The work is performed remotely using the connection to the network of VLSI laboratory where the workstation shares access to the internet. The configurations of the workstation are related to the docker installed on the machine in order to allow different users to work in a safe space and avoiding to interrupt or corrupt others' systems [10]. This machine has been used in the last part of the simulation work since the timing requested by the GPU of the notebook performs too poorly with a network deep as ResNet18. It has an FP32 theoretical performance of 4'291 GFLOPS, so a speed up of 5 times is expected referred to Geforce 920m.

3.1.5 Docker

Docker is a container used to isolate a portion of software from the whole computing machine [10]. It is provided with all the libraries necessary to develop code and it is independent of the system, in order to have a safe environment for developing codes. It is a similar situation to virtual machines, but the advantage of the docker is the operating system virtualized, while with virtual machine different hardware are virtualized and on each VM an operating system must be installed [10]. The major benefit is the performances obtained by the docker and the OS is shared by other dockers so better management of the instructions can be obtained through kernel scheduling.

3.1.6 Google Colab with Tesla K80

Google Colab is a free platform provided by Google, where every user can simulate Jupyter notebooks linking Google Drive account to the platform [8]. The hardware of the emulated computers are really performant and the GPU is an Nvidia Tesla K80. The notebook starts with a lot of libraries already installed, such as PyTorch and CUDA. The limit is a runtime lower than 12 hours consecutively and if this limit is exceeded the runtime will interrupt and the user is disconnected from the platform. This service can be linked to Google Drive account in order to save files produced by code running. There are cells where the code can be written and executed without the need to set-up the libraries since the most used ones are already installed. In a Jupyter notebook, there are different cells and the variables are shared between all of them after the execution. One of the greater features is the possibility to share a notebook through links or downloading from the Google Drive account.

3.2 Develop simple network with numpy

The starting point to analyze the algorithms is to approach the simplest possible network and to modify it using the FA, DFA and IFA implementations. The dataset chosen is the MNIST, since the first results provided in the paper of Nøkland refers to this one. The code is obtained by a free online book that is the first one used to approach machine learning [14]. The network is described by a sequence of linear layers and the number of them is simply editable changing the dimensions of a variable called sizes. Optimizations are not applied, such as cross entropy cost function, regularization and the weights are initialized in a simple way. The cost function used in this code is the quadratic cost:

$$J(x) = \alpha(l - y(x))^2 \tag{3.1}$$

where α is a constant value, l is the label of the image taken as input and y(x) is the output of the neural network. The activation function used is the sigmoid, since it is one of the first utilized to approach machine learning. The code is based on the Numpy library, so the use of GPU is avoided for these first codes. The forward part of the code used is shown:

```
activation = x
activations = [x]
zs = []
for b, w in zip(self.biases, self.weights):
    z = np.dot(w, activation) + b
    zs.append(z)
    activation = sigmoid(z)
    activations.append(activation)
```

The forward part consists of multiplications between inputs and weights of the layer and in the addition of a bias term. After the sigmoid activation function application, the output is used as input for the new layer. The activations and zs arrays are required since in the backward part it is necessary to recall these values for the gradients computations. The backward part is reported to understand the computations performed and how they can be modified successively to analyze the different algorithms:

```
delta = self.cost_derivative(activations[-1], y)* \
        sigmoid_prime(zs[-1])
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
for l in range(2, self.num_layers):
  z = zs[-l]
  sp = sigmoid_prime(z)
  delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
  nabla_b[-l] = delta
  nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
return (nabla_b, nabla_w)
```

In this part of the code the delta is the error of the last layer at the beginning and it is computed using the function self.cost_derivative that performs the function in equation 2.4, that in this case is:

$$\delta J = l - x \tag{3.2}$$

This value is multiplied with the derivative of the activation function of the last output of the network and successively it is multiplied again with the transposed of the weight to obtain the gradient referred to the input. In this way, the error information is propagated to each layer. The gradients referred to bias and weight terms

expressed as nabla_b and nabla_w are obtained as shown: the first is equal to the gradient of the output and the second is obtained multiplying with the transposed of the activation.

3.2.1 FA

The Feedback Alignment algorithm differs from the classic backpropagation for the use of random matrices instead of the transposed of weights used during forward as can be seen in equation 2.8. The code has been modified as shown:

```
delta = self.cost_derivative(activations[-1], y)* \
        sigmoid_prime(zs[-1])
nabla_b[-1] = delta
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
for l in range(2, self.num_layers):
  z = zs[-l]
  sp = sigmoid_prime(z)
  delta = np.dot(self.coeff[-l+1], delta) * sp
  nabla_b[-l] = delta
  nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
return (nabla_b, nabla_w)
```

The variable self.coeff is initialized in the same way as self.weight, but it has dimensions trasposed in order to avoid additional computations.

3.2.2 DFA

In the Direct Feedback Alignment, there is a different path to compute the gradients of the output for each layer. It is performed the product between the random matrices and the gradient of the last layer as shown in equation 2.10:

```
delta_0 = self.cost_derivative(activations[-1], y)* \
          sigmoid_prime(zs[-1])
nabla_b[-1] = delta_0
nabla_w[-1] = np.dot(delta_0, activations[-2].transpose())
for l in range(2, self.num_layers):
  z = zs[-l]
  sp = sigmoid_prime(z)
  delta = np.dot(self.coeff[-l+1], delta_0) * sp
  nabla_b[-l] = delta
  nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
return (nabla_b, nabla_w)
```

In the FA it is possible to see that the difference is only in the matrices used to propagate the error in each layer, but in the DFA the code has been modified in a deeper way. The delta coefficient is different from the one computed in the last layer (delta_0). This factor must be considered as the error performed by the network in the classification task and in the FA it is updated going through the layers of the network, while in the DFA it is used always the same value. The advantage of DFA is the possibility to have an error factor independent between the hidden layers, but related only to the last layer.

3.2.3 IFA

In the Indirect Feedback Alignment, the error is not reported in each layer as in the DFA, but a random matrix is used to link the last layer to the first hidden one [15]. The gradients of the output are computed going in the same direction of the forward, using as matrices the weights in their normal form as shown in equation 2.12. The code modified is shown:

```
delta_0 = self.cost_derivative(activations[-1], y)* \
          sigmoid_prime(zs[-1])
nabla_b[-1] = delta_0
nabla_w[-1] = np.dot(delta_0, activations[-2].transpose())
delta = np.dot(self.coeff, delta_0) * sigmoid_prime(zs[0])
nabla_b[0] = delta
nabla_w[0] = np.dot(delta, activations[0].traspose())
for l in range(1, self.num_layers-2):
  delta = np.dot(self.weights[l], delta) * sigmoid_prime(zs[l])
  nabla_b[l] = delta
  nabla_w[l] = np.dot(delta, activations[l].transpose())
return (nabla_b, nabla_w)
```

The code has been modified computing the gradients of the first hidden layer at the beginning, after it is computed the last layer gradient. This changes the direction of computation for the gradients in each layer and the indices inside the for cycle has an opposite sign with respect to previous cases. The advantage is the use of the same weights during forwarding, without performing the transpose operation. In addition, it can give the opportunity to perform unique access in memory, using the weights both for forward and backward at the same time.

3.2.4 Results

The number of hidden layers has been changed to check how the different implementations modify their behaviours with increased depth. In the paper of Nøkland there are not results about IFA implementations but only related to BP, FA and DFA. The four algorithms are performed on linear networks to make comparisons between all of them. At the beginning only 3 hidden layers are used, with respectively 10, 20 and 10 neurons. The output layer has 10 neurons because the dataset to classify is the MNIST that has 10 classes.

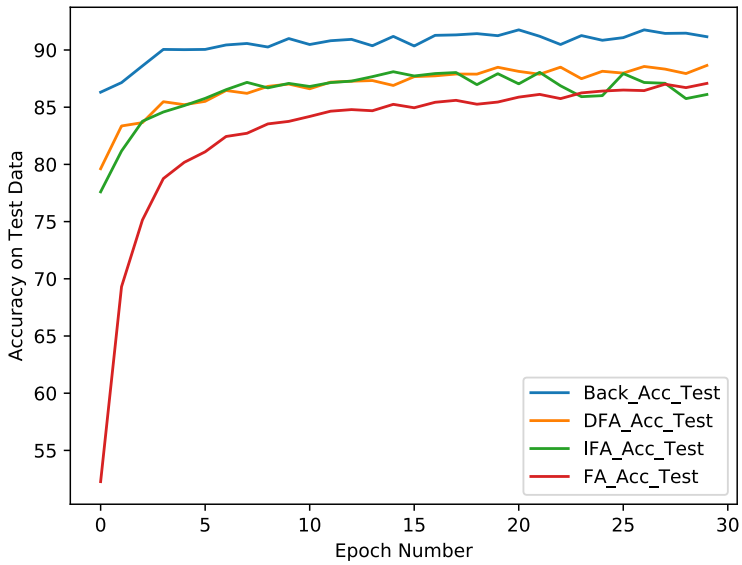

Figure 3.1: Test accuracy on a three hidden layer network

The picture shows how the four different algorithms have the same behaviour and it is interesting to note as the DFA and IFA algorithms perform slightly better than the FA algorithm. In the FA it is possible to note as the accuracy is about 20% lower than the other cases in the first epochs and this is coherent with the alignment theory about the initial phase of adapting the backward weight for the learning phase. The backpropagation seems to learn in a faster way, since at the end of the first epoch it has a gain of 5-6% with respect to other algorithms. Due to the simplicity of the dataset and the network used, it is not possible to make considerations valid for these algorithms so the network depth has been modified. Increasing the depth of the network, five hidden layers are used with 10 neurons everyone and what it is expected is the DFA ability to propagate in a better way the error in each layer since the hidden layers are adjusted all with the error value instead of propagating this error through the whole network.

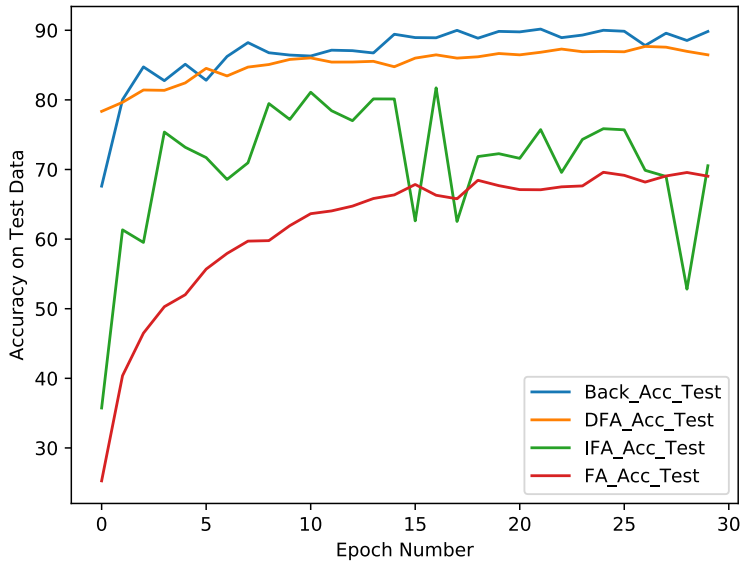

Figure 3.2: Test accuracy on a five hidden layer network

The DFA algorithm shows an initial accuracy value higher than the backpropagation algorithm, while the FA and IFA implementations have accuracies lower than the others. The IFA algorithm seems to learn with difficulty since the accuracy values don't increase in a uniform way. The FA algorithm instead seems to learn correctly, but it reaches lower accuracy values, probably due to the structure of the network. The dimensions of the five hidden layers are modified increasing the number of neurons to 10, 20, 50, 20 and 10 respectively.

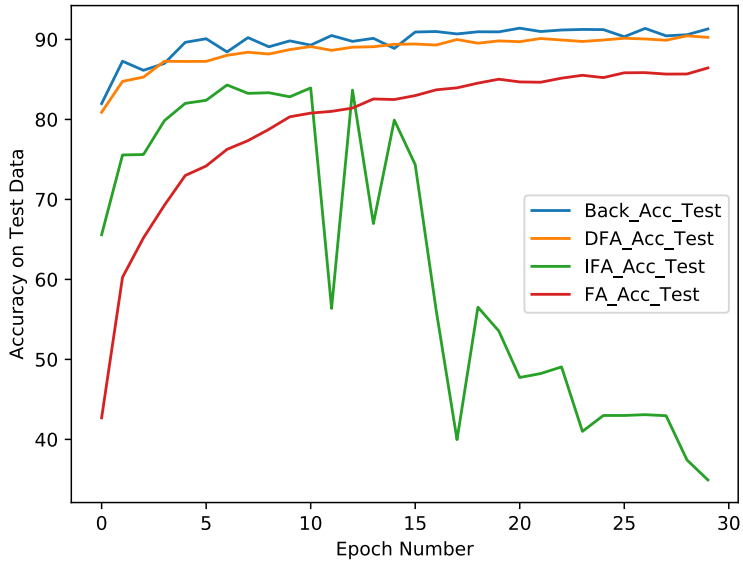

Figure 3.3: Test accuracy on a five hidden layer network with more neurons

The results confirm as the backpropagation and DFA have similar behaviour, the functions of accuracy are pretty similar and the difference is about 1-2%. The FA shows a behaviour similar to the previous case, reaching in the last epochs an accuracy value 5% lower than BP and DFA. The IFA algorithm performs very badly since at beginning its learning is slightly lower respect previous case, but the problem is the drop in accuracy after 10 epochs. This phenomenon can be linked to exploding gradients. The gradients values instead of fitting the curve that can describe the dataset start to diverge and the consequence is a loss value that goes to infinite. The network is not able to learn anymore starting to deteriorate its performance. Another experiment has been performed increasing a second time the number of neurons in the last layers to check if the behaviour of the last results can be confirmed. So there are five hidden layers with 10, 20, 50, 100 and 20 neurons:

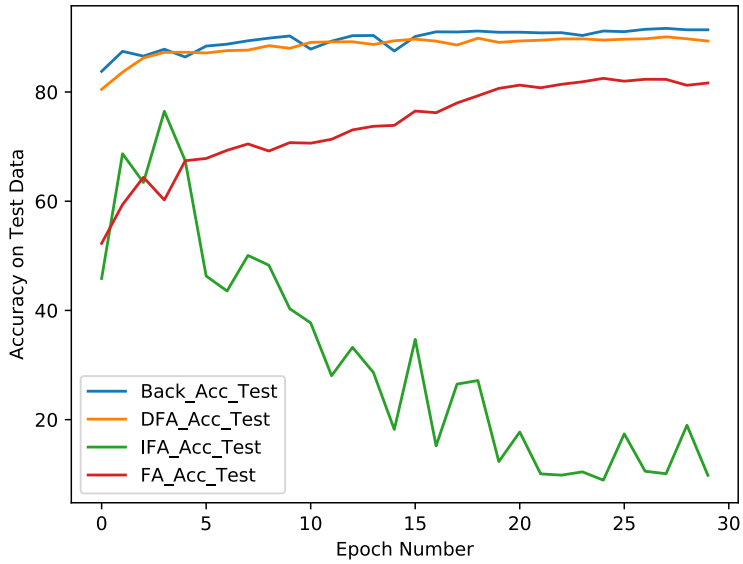

Figure 3.4: Test accuracy on a five hidden layer network with additional neurons

The results confirm that increasing the complexity of the network each of the four algorithms has a different answer. The FA has a 10% less accuracy with respect to backpropagation and it is possible to see how the accuracy of the first epochs is smaller than the BP case. This phenomenon is due to the fixed random matrices used during the backpropagation. In the beginning, the training of the network performs worse than using the classic algorithm, but the updating of the forwarding weights allow learning for the network. The forwarding weights start to act as Pseudo Inverse of fixed random matrices and they became able to bring back the error information in an efficient way. The DFA seems to train the networks as well as the backpropagation, so it is the best replacement that can be used in the training phase for networks with this complexity. The IFA algorithm, instead, seems to be able to train only very small networks. This can be the effect of a bad initialization

resulting in the impossibility for the network to train with reduced error information. In fact, the error is brought back to the first layer in the same entity as in the last layer, resulting in a different magnitude of information for the first hidden layer. The three algorithms have been tested on a neural network with some optimizations to understand which solution can give better performance, avoiding bad initialization that can occur in the previous networks studied.

3.3 Implementation in optimized networks

After a first analysis of the results obtained using a simple network, it is used another code taken from the book cited before [14], in which optimizations such as regularization, cross entropy cost and a proper initialization of weights have been applied. The effect of a proper weight initialization is the increase in speed during the learning phase [14]. Since the network training consists of weights values updating, if they are initialized in an easier way to be modified, the learning speeds up. This implies fewer epochs to reach the same accuracy and to avoid iterations over the same set of images multiple times. In the previous code, the quadratic cost function is used to compute the loss between the label associated with the dataset images and network output. Using the cross entropy cost there is an advantage since its learning is associated with the entity of the error [14]. So the importance of this cost function is related to the speed that allows the network to learn since it is slower with the quadratic cost and bad initialization conditions. Using the cross entropy cost instead, there is a faster learning effect, even with bad initialization conditions. For this reason, the network can be less affected by overfitting, since it has to be trained for fewer epochs. Applying this technique exploding gradients can be reduced, decreasing the learning factor that occurs during the implementation of IFA but also to improve results for BP, DFA and FA. There are different techniques of regularization such as weight decay and modifying the cost function with a regularization term.

$$J = J_0 + \frac{\lambda}{2n} * \sum_w w^2 \tag{3.3}$$

The structures of the four networks with applied optimizations are the same as in the first type of code since the scope is to understand the effect of classical optimizations on these algorithms.

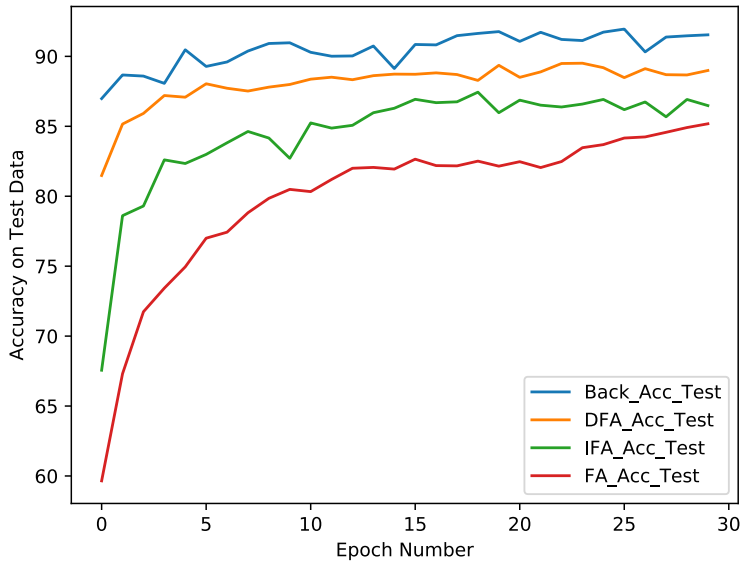

Figure 3.5: Test accuracy on optimized linear network with three hidden layers

The optimizations seem to be more effective on the DFA than on other algorithms. In the simple network, the results of DFA and IFA are comparable, while in this case there is a difference of 5% between the two algorithms. The FA solution seems to start with faster learning since at first epoch there is a gain of 5% with respect to the simple network, but after five epochs of iterations the accuracy reached is about 75%, that is less than the 80% of the simple network case. The BP doesn't seem to be affected by the optimizations and this can be related to the simplicity of the network, where a low overfitting occurs. The next network is based on more hidden layers to check if the optimizations give better results:

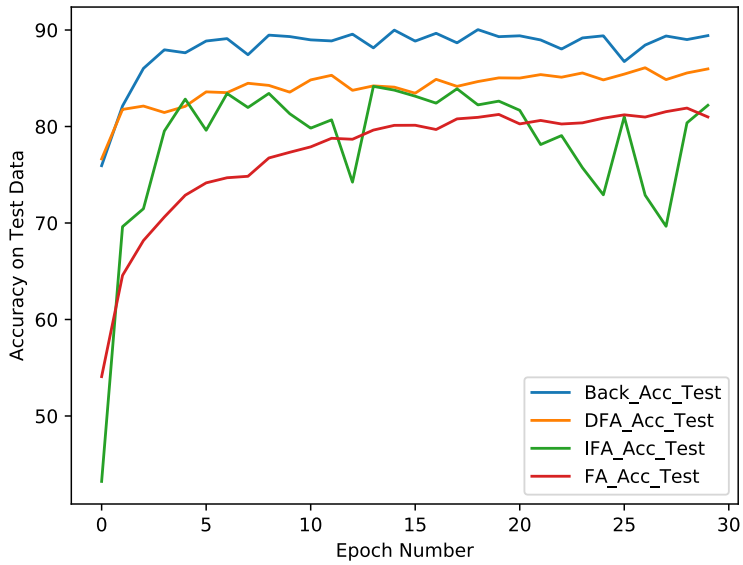

Figure 3.6: Test accuracy on optimized linear network with five hidden layers

The BP, DFA and FA seem to have similar behaviour as the previous result, with reduced accuracy for the FA about 5%. The IFA curve has the same problem shown in the simple network case, where the learning curve is irregular with respect to the others. The third simulation performed with the simple network is repeated on the network with optimizations to confirm the results obtained:

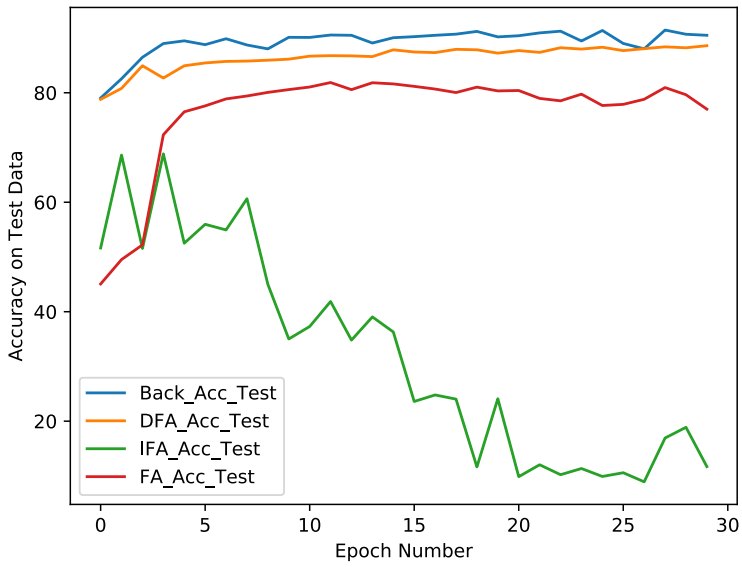

Figure 3.7: Test accuracy on optimized linear network with more neuros

The results show how the BP, DFA and FA have similar behaviour with respect to the previous case, but the IFA performs very badly compared to the simple network. This phenomenon is related to the difficulty for the IFA to train deeper networks since the effect of the error is completely changed compared to the BP. While in the BP the error has a higher effect on the last layers going to a lower effect on the first hidden ones, in the IFA this is completely changed. The effect influences too much the first layers of the network and the last layers are unable to change the prediction based on the first features extracted. If the matrix values used to propagate the error in the first hidden layer are chosen extremely low, the network is not able to train properly since the next hidden layers are updated with very small values. To confirm the inability of the IFA to train deep networks, the training performed on the last analyzed structure was repeated with 5 hidden layers made of 10, 20, 50, 100 and 20 neurons respectively.

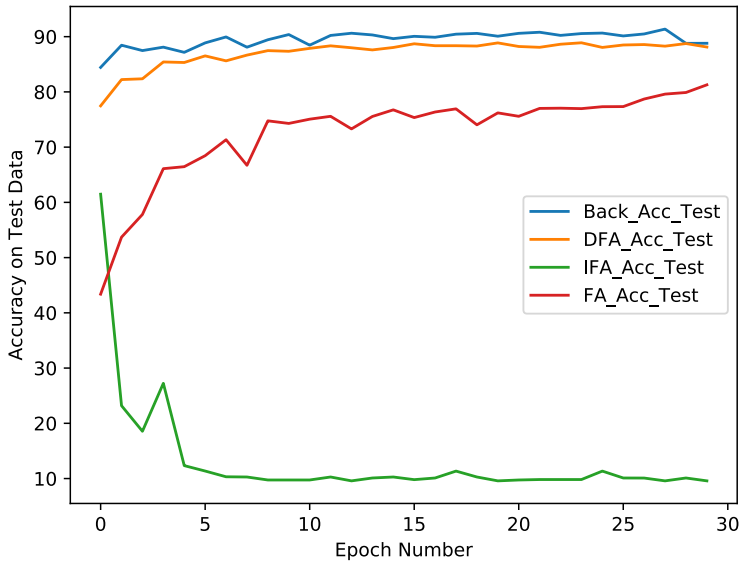

Figure 3.8: Test accuracy on a optimized linear network with increased neurons

The results show as the IFA algorithm is completely unable to train a network so deep. The difference for IFA implementations between the performance obtained in the simple network and the one of the optimized version can be related to difficult starting conditions in the training phase, but the optimizations have given worst results only for the IFA algorithm, while BP, DFA and FA seem to perform in the same way.

3.4 Approch to PyTorch

PyTorch is used because the aim of the work is implementing these different algorithms on a complex network as ResNet. It is an optimized framework for memory management, resulting in faster training. This is an important factor working with deep networks as ResNet because the number of free parameters is very large and an optimized framework can give better performances. The basic code that has been used to approach PyTorch is found on the site of the framework and it uses the Cifar10 dataset [4]. The first part of the code is about the normalization of the input images using torchvision. This library allows easy loading of the training and testing set to prepare the inputs images to be classified. The network defined in the original code is a convolutional network. It is based on two convolutional layers with 6 and 16 output channels respectively. The rectified linear unit and max-pooling are applied to the output of the convolutional layers. The max-pooling effect is to take a number of elements of the matrix and the lower is discarded, so there is a reduction of the inputs dimensions. In this network has been used a max-pooling between four values each time. It is possible to set also the stride factor, allowing

the windows of values to overlap. The filter shape of the convolutional layer is 5x5 for both layers. The networks have also three linear layers, with 120, 84 and 10 neurons respectively and at each output it is applied the rectified linear unit. The cost function used is the cross entropy loss, since its performances are better than the quadratic cost.

3.4.1 Most important PyTorch functions

The most important packages of PyTorch are the Autograd and Optimizer since these are used during the training phase to compute and update the weights values. The function autograd is called using the backward() function and gradients of all the network are computed iteratively. The greater benefit of this function is related to the speed of computation, due to the efficient memory management of PyTorch. During the forward step, the functions related to the backpropagation step are saved and in this way, it is not necessary to perform two times the translation from Python to C++. The gradients are saved in the .grad of every variable and using the optimizer function .step() the weights and biases terms are updated. The values to update, the weight decay, the learning rate and momentum factors are chosen through this function. Different algorithms for updating the weights are available such Stochastic Gradient Descent, Adam, L-BFGS and others. After the updating the optimizer library is called through .zero_grad() function to set null the values of gradients because the function autograd accumulates the value in the .grad performing iteratively the addition between values saved and the computed ones. The optimization method used is the Stochastic Gradient Descent, discussed in the state of the art chapter.

3.4.2 Custom module definition

In PyTorch, a module represents a class of data and functions to be saved. Usually, a module has a function to be performed during the forward step. There are modules defined in the libraries and there is the possibility to define custom modules to have a specific behaviour. In order to modify the backpropagation step of the training, it is necessary to modify how the gradients are computed. So it is required to use custom modules and a custom autograd function for the specific module must be defined. The operations to be performed both in forward and in backward are specified through the definition of the custom function. The difference is in the organization of the GPU instructions to perform the backpropagation: there is the dynamic method, used in a classical way, and the static method, used defining custom module. Using the dynamic method there is a faster computation performed since the values are maintained in memory and the steps required to compute during the backward are defined at a low level in the forward phase, so the code interpretation is avoided in

the backward step. Using the static method instead there is an additional loss of time due to the translation of the code to machine language and how it is implemented by the hardware.

3.4.3 Register Hook

The register_hook is used to have a different approach to modify the gradients value as requested by the implementations to be analyzed. Since during simulations there is a huge slow down in the computation using the custom modules, it has been decided to try other solution. In the Autograd package, there is the possibility to modify the gradients computed with respect to the Tensor using the function register_hook [5]. This is an in-place operation and it is executed when the value of the gradients has been computed. During the backward phase, the register_hook function allows modifying the gradients values in a way that is faster than using custom modules.

3.4.4 Implementations of algorithms with PyTorch

The original network has been obtained by PyTorch tutorials and it is modified to have better comparisons with the previous networks analyzed. At the beginning it is used only linear layers and successively convolutional layers are restored, but the max-pooling ones are avoided. The network is based on linear layers with a number of neurons higher than the MNIST network classifier since the features that must be extracted in a dataset as the Cifar10 are more complicated and the dimensions of the images are bigger.

BP

The custom module is implemented also on the classic backpropagation to understand better the modifications applied to the code. The forward function is shown, but it is not modified for the other algorithms:

```
def forward(ctx, input, weight, bias=None):
        ctx.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
          output += bias.unsqueeze(0).expand_as(output)
        return output
```

During the forward part, the input variable is saved with weights and biases in the context variable (ctx.save_for_backward) beacause they must be used again during the backward step. The output is computed performing the product between input and the transposed of the weights. In the previous codes, the product between

weights and inputs is performed without the transpose computation in the forward step. PyTorch defines the weights in transposed dimensions for linear layers with the purpose to perform the transposition in the forward and to use the matrices in the backward avoiding loss of time with additional computation. In this way, it is possible to have a symmetric time for computation between the forward and backward. Also the bias term is added to the output. The backward part is shown to analyze how the different gradients values are computed and the error information is propagated through the layers:

```
def backward(ctx, grad_output):
        input, weight, bias = ctx.saved_tensors
        grad_input = grad_weight = grad_bias = None
        if ctx.needs_input_grad[0]:
      grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
          grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.needs_input_grad[2]:
          grad_bias = grad_output.sum(0).squeeze(0)
        return grad_input, grad_weight, grad_bias
```

It is possible to see how the gradients of the loss with respect to input and weights are computed in the custom module. The grad_output variable is the gradient of the loss referred to the output of the specific layer and it is obtained by the autograd function to be propagated through all the layers using the dynamic graph that PyTorch builds during the forward. The gradients referred to inputs are obtained as the product between the gradients with respect to output and the weights used during the forward. It is important to note as the weights matrices are not transposed thanks to PyTorch initialization. The gradients referred to weights are computed multiplying the gradient with respect to output with the input variable saved during the forward, that will be the output of the previous layer where the activation function has been applied. The transpose operation applied to the gradients term is the difference with the codes seen before, because it is applied to activation before. The gradients of the loss referred to bias are computed in order to update this value during the optimizer step as it is done with the other parameters.

FA

The backward part for the Feedback Alignment method is shown and the replacement of forward weights with random matrices is applied as can be seen:

```
def backward(ctx, grad_output):
  input, weight, weight_fa, bias = ctx.saved_tensors
  grad_input = grad_weight = grad_weight_fa = grad_bias = None
  if ctx.needs_input_grad[0]:
    grad_input = grad_output.mm(weight_fa)
  if ctx.needs_input_grad[1]:
    grad_weight = grad_output.t().mm(input)
  if bias is not None and ctx.needs_input_grad[2]:
    grad_bias = grad_output.sum(0).squeeze(0)
  return grad_input, grad_weight, grad_bias
```

The grad_input computation is modified using the weight_fa variable instead of weight as in the BP case. The weight_fa variable is defined during the class definition and the weights are initialized in the same way as done with the forward weights.

```
self.weight.data.uniform(-stdv, stdv)
self.weight_fa.data.uniform(-stdv, stdv)
```

The modification has only the purpose to check how the results will change in this case. In the implementation of this algorithm, the dimensions must be inverted in the definition step of the forward weights, to remove the transpose operation in the forward. The gradients computation for both grad_weight and grad_bias is not modified.

DFA

In the DFA algorithm the gradients are not propagated through the layers, so it is requested to interrupt the propagation of grad_input values:

```
def backward(ctx, grad_output):
  input, weight, bias = ctx.saved_tensors
  grad_input = grad_weight = grad_bias = None
  if ctx.needs_input_grad[1]:
    grad_weight = grad_output.t().mm(input)
  if bias is not None and ctx.needs_input_grad[2]:
    grad_bias = grad_output.sum(0).squeeze(0)
  return grad_input, grad_weight, grad_bias
```

As can be seen, the grad_input computation has been removed and the value of grad_output is modified using the register_hook function. This function performs

the replacement when the variable grad_input is declared None and it is propagated to the next layer. The values of grad_input will be changed from the autograd function performing the product with the derivative of the activation function.

```
delta = delta_cross_entropy(zs[-1], labels)
zs[-1].register_hook(lambda grad: delta)
activations[-2].register_hook(lambda grad: delta.mm(Wfc4))
activations[-3].register_hook(lambda grad: delta.mm(Wfc3))
activations[-4].register_hook(lambda grad: delta.mm(Wfc2))
activations[-5].register_hook(lambda grad: delta.mm(Wfc1))
loss.backward()
```

The output gradients of the last layer are computed using the custom function that performs the derivative of the loss referred to the last output of the whole network, called delta_cross_entropy. The function used to compute this term is related to the cost function chosen that can be the quadratic cost or the cross entropy one, because also the values of the gradients are related to the function used. These multiple calls to register_hook modify the gradients values referred to each variable. The zs vector is filled with the output of each layer and the activations vector is filled with the output values after the activation is applied. In this way using the register_hook function on zs[-1] the gradients referred to the last output are modified and the product with derivative of activation function is avoided. For the other layers, it is used the activation vector in order to avoid the manual multiplication between the product of the gradient delta and the different random matrices with the derivative of the activation function. This product will be performed by the autograd function in an automatic way, using the dynamic graph that has tracked all the operations performed on the Tensors.

IFA

For the IFA algorithm, the backward() function has been avoided since it computes the gradients going from the last layer to the first hidden one and this has an opposite direction to the one used by the IFA. This function is replaced by writing the backward steps manually and using defined variables to save gradients values:

```
for i, data in enumerate(trainloader, 0):
  inputs, labels = data
  optimizer.zero_grad()
  activations, zs = net.forward(inputs)
  loss = criterion(zs[-1], labels)
  delta0 = delta_cross_entropy(zs[-1], labels)
  fc5_weight_grad = delta0.t().mm(activations[-2])
  delta = delta0.mm(Wfc2)*relu_prime(zs[-5])
```

```
fc1_weight_grad = delta.t().mm(activations[-6])
delta = delta.mm(net.fc2.weight.t())*relu_prime(zs[-4])
fc2_weight_grad = delta.t().mm(activations[-5])
delta = delta.mm(net.fc3.weight.t())*relu_prime(zs[-3])
fc3_weight_grad = delta.t().mm(activations[-4])
delta = delta.mm(net.fc4.weight.t())*relu_prime(zs[-2])
fc4_weight_grad = delta.t().mm(activations[-3])

net.fc5.weight.data -= 0.01*fc5_weight_grad
net.fc4.weight.data -= 0.01*fc4_weight_grad
net.fc3.weight.data -= 0.01*fc3_weight_grad
net.fc2.weight.data -= 0.01*fc2_weight_grad
net.fc1.weight.data -= 0.01*fc1_weight_grad
```

The gradients variables are obtained by the product between the previous delta and the weights, but it is requested to perform the multiplication with the derivative of the rectified linear unit function applied to the output. This is not required to be performed manually with the custom module because the element by element multiplication is executed by the autograd function automatically using the dynamic graph.

3.4.5 Results BP, FA, DFA and IFA

The four codes explained previously have been simulated to understand which one is more indicated to replace the backpropagation to classify a dataset complex as the Cifar10 because the MNIST dataset is not considered a challenging one:

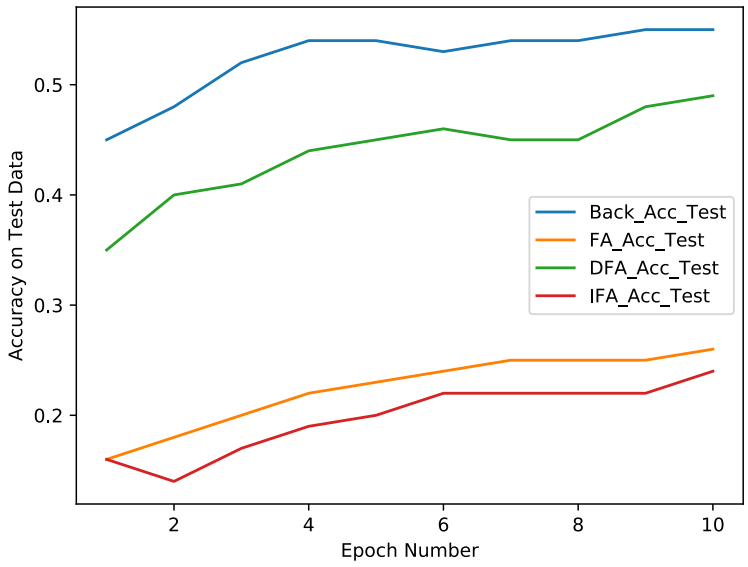

Figure 3.9: Test accuracy on a linear network classifiying Cifar10 images

Table 3.1: Graphs of linear network classifying Cifar10 images

Algorithm	**lr**	**Best Test Accuracy**
BP	0.001	55%
DFA	0.001	49%
FA	0.001	26%
IFA	0.001	24%

Many solutions where the network was not able to learn are obtained using the different algorithms since replacing the forward weights with random ones causes exploding gradients phenomena. This effect is relevant in the IFA implementation, but it is present also in the FA case. In order to obtain solutions comparable, different weights initializations are used and it is considered the accuracies value obtained as the best solutions to achieve with the defined structure of the network. The weights initialization for the random ones in the FA and DFA cases are a uniform distribution in the interval (0, 0.01), while for the IFA case it is a uniform distribution in the interval (0, 0.001). The BP and DFA perform pretty good reaching 60% and 50% of accuracy respectively. This value is lower than the ones shown before since there is a complex dataset as Cifar10 and a simple linear network is not able to classify in a better way. The FA seems to confirm the 20% lower accuracy referred to DFA as obtained in the previous networks. The IFA algorithm instead trains the network very difficult. This must be due to how the error of the last layer is reported.

Problem of Exploding Gradients on IFA

Since with the IFA implementation the error is reported from the last layer to the first hidden one, the random matrix must be defined with a uniform distribution in a smaller interval. This must be done in relation to the entity of the correction factor used during the backpropagation. The error is multiplied for several weights matrices before obtaining the correction factor for the first layer. If a matrix without proper values is used the problem of exploding gradients will be dominant. This is not relevant in the DFA probably due to the ability of the other layers to attenuate the phenomenon, while in the IFA the other layers will be less trained since the error goes in the forward direction. The several experiments have shown bad results and the best one is obtained using a random weight matrix for the IFA with uniform distribution in the interval (0, 0.001) as mentioned before. The IFA algorithm has been discarded due to the difficult training shown, considering this algorithm a bad solution for deep networks.

3.4.6 Results BP, FA and DFA

The initialization of the weights is an important factor in networks using random matrices and different solutions are implemented to understand if best ones can be found for the FA and DFA. For the IFA case, the solution shown before is the the best one obtained on this network structure. The initialization of the random weights has been changed to a uniform distribution in the interval (0, 0.03), but it is possible to see how the FA and DFA have irregular behaviour with respect to the previous case. This is due to the difficulty of fitting the curve that is used to make predictions, since the initialization of the weights used to transport back the error is not optimal.

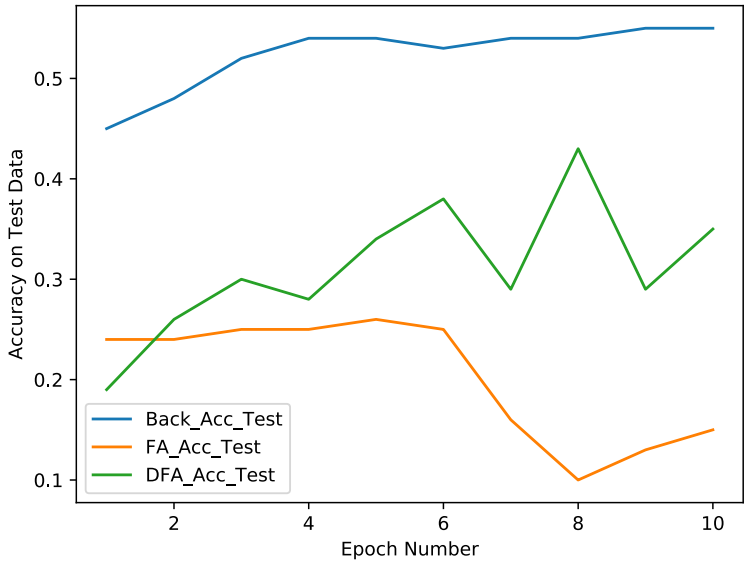

Figure 3.10: Test accuracy on a linear network classifiying Cifar10 images with modified initializations

Table 3.2: Graphs of linear network on Cifar10 using different initializations

Algorithm	**lr**	**Best Test Accuracy**
BP	0.001	55%
DFA	0.001	43%
FA	0.001	26%

3.4.7 Convolutional network without module

The next network trained on the Cifar10 dataset is the same as the one provided by PyTorch, but the max-pooling layers are removed in order to have a better understanding of the elements to be modified for a correct implementation [4]. It is based on two convolutional layers with 6 and 16 output channels and three linear layers with 120, 84 and 10 neurons respectively. It is used the CrossEntropyLoss() function to compute the loss and the Stochastic Gradient Descent to update the weights values with a learning rate equal to 0.001. The optim.SGD function is used for the FA e DFA implementations thanks to the register_hook function that is used to modify the gradients values. The solution of the custom modules and custom autograd function is not applied to replace convolutional layers since PyTorch does not expose the source of code for the backward step to the user, so the register_hook function is used to understand if FA and DFA implementations work correctly also on convolutional networks. The convolutional layer must give better results with a complex dataset as Cifar10 since it is able to extract more complex features and it is able to perform faster training since the kernel used for the convolution is always the same, so it is not requested to load a different value for every input.

DFA

The code for the DFA implementation is reported to understand how the functions register_hook and view are used to replace the gradients with the ones computed following the DFA algorithm:

```
delta = delta_cross_entropy(zs[-1], labels)
zs[-1].register_hook(lambda grad: delta)
activations[-2].register_hook(lambda grad: delta.mm(Wfc3))
activations[-3].register_hook(lambda grad: delta.mm(Wfc2))
delta1 = delta.mm(Wfc1)
delta1 = delta1.view(4,16,24,24)
activations[-4].register_hook(lambda grad: delta1)
delta2 = delta.mm(Wconv2)
delta2 = delta2.view(4,6,28,28)
activations[-5].register_hook(lambda grad: delta2)
loss.backward()
optimizer.step()
```

As can be seen, the function delta_cross_entropy has been used to compute the gradients of the output of the last layer and this value is used to derive all others output gradients. It is interesting how the variables delta1 and delta2 need to be modified using the view() function. The delta variable has two dimensions, but the gradients have four dimensions using convolutional layers due to two dimensions of the input

image and the other two related to output channels and batch dimension. A random matrix with concordant dimensions has been defined and before the computation of the gradient it is necessary to define the matrix in two dimensions. Since the function register_hook replaces the gradients only during the backward, it is necessary to define multiple delta variables. If it is used only one variable that will be updated, the register_hook function will consider only the last value.

FA

The FA code is reported, but it is similar to the DFA one, since only the error information propagation has a different path:

```
delta0 = delta_cross_entropy(zs[-1], labels)
zs[-1].register_hook(lambda grad: delta0)
delta1 = delta0.mm(Wfc3)
activations[-2].register_hook(lambda grad: delta1)
delta2 = delta1.mm(Wfc2)
activations[-3].register_hook(lambda grad: delta2)
delta3 = delta2.mm(Wfc1)
delta3_1 = delta3.view(4,16,24,24)
activations[-4].register_hook(lambda grad: delta3_1)
delta4 = delta3.mm(Wconv2)
delta4_1 = delta4.view(4,6,28,28)
activations[-5].register_hook(lambda grad: delta4_1)
loss.backward()
optimizer.step()
```

The function delta_cross_entropy is used also in this code. The error is propagated through the different delta variables used. The difference respect the DFA case is how the delta values are computed since the error information is propagated in each layer, while in the DFA the error uses skipping connections.

3.4.8 Results

After the explanation of the codes, two different simulations are shown where the initialization of the random weights is varied for both the FA and DFA algorithms. The first one has a uniform distribution in the interval (0, 0.001), the second in the interval (0, 0.005):

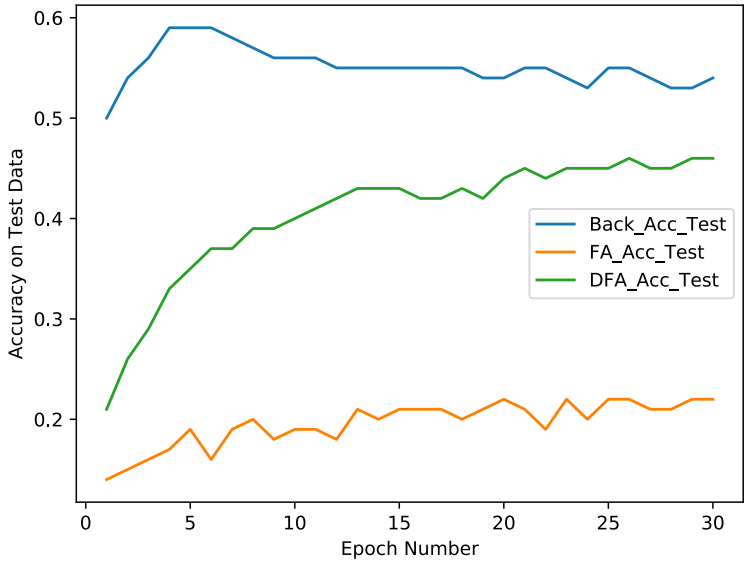

Figure 3.11: Test accuracy on a convolutional network classifiying Cifar10 images

Table 3.3: Graphs of convolutional network on Cifar10 with (0, 0.001) initialization

Algorithm	**lr**	**Best Test Accuracy**
BP	0.001	59%
DFA	0.001	46%
FA	0.001	22%

The results show the possibility to reach good accuracies also using a simple convolutional network with few hidden layers, reaching 59% of accuracy with the backpropagation. The DFA shows an increase in the accuracies difference with the backpropagation case, because it is about 15% lower than the one obtained with linear networks, that it is around 5%. The FA algorithm shows a lite degradation, increasing the drop in accuracy to 35%. The first conclusion with these results is the inefficiency for the FA algorithm to train convolutional networks. The number of hyper-parameters can be strongly influenced by many factors.

Implementations with different initialization are performed in order to understand how the network behaviour is modified for all the algorithms:

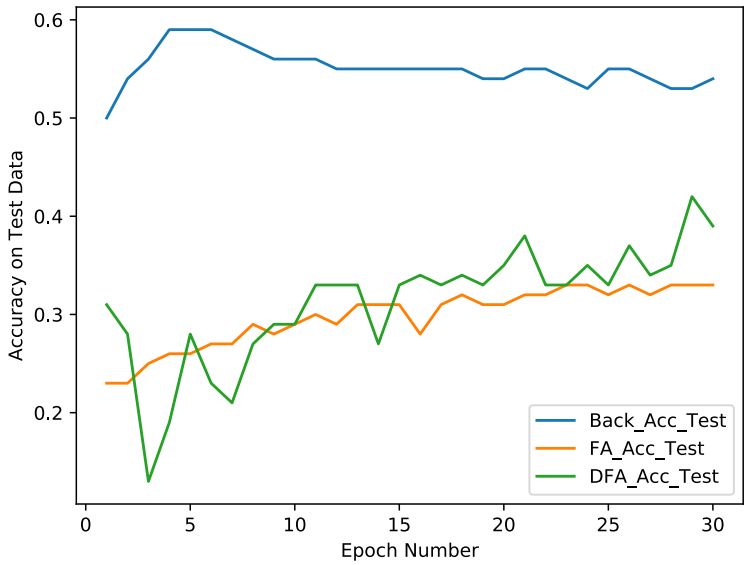

Figure 3.12: Test accuracy on a linear network classifiying Cifar10 images with different initialization

Table 3.4: Graphs of convolutional network on Cifar10 with (0, 0.005) initialization

Algorithm	**lr**	**Best Test Accuracy**
BP	0.001	59%
DFA	0.001	42%
FA	0.001	33%

The initialization of the weights is fixed in the interval (0, 0.005) with uniform distribution and both the FA and DFA have different behaviour. The FA improves the accuracy of 10%, while the DFA has a drop of 4%, that is not huge, but the accuracy curve seems to be very irregular respect the previous case. This factor gives a particular clue to the importance of the initialization of the random weights used to propagate back the error in each layer.

3.4.9 Convolutional network with module

After the implementation of the convolutional network with a manually defined backward, the custom modules have been used also in this type of network. The problem has been the definition of the backward step, since PyTorch does not give access to the backward autograd function of the convolutional layer to the users. The problem has been solved using the two functions torch.nn.grad.conv2d_input() and torch.nn.grad.conv2d_weight(). Another solution can be the definition of custom functions that in the backward step compute the partial gradients using concatenated for cycles, but this solution is not optimal due to times requested to perform the computations. The backpropagation function has been modified inserting the custom module to check if it works properly with the two functions mentioned previously, so the forward step and the backward are reported:

```
def forward(ctx, input, weight, bias=None, stride=1, \
            padding=0, dilation=1, groups=1):
    ctx.save_for_backward(input, weight, bias)
    return F.conv2d(input, weight, bias, stride, \
                    padding, dilation, groups)
```

The forward part of the code is used only to recall the classic function also inside the normal module of the convolutional layer, since the forward part is not required to be modified. In the backward part the two functions are called:

```
def backward(ctx, grad_output):
  input, weight, bias = ctx.saved_tensors
  grad_input = grad_weight = grad_bias = grad_stride = None
  grad_padding = grad_dilation = grad_groups = None
  if ctx.needs_input_grad[0]:
    grad_input = torch.nn.grad.conv2d_input(input.shape, /
    weight, grad_output)
  if ctx.needs_input_grad[1]:
    grad_weight = torch.nn.grad.conv2d_weight(input, /
    weight.shape, grad_output)
  if bias is not None and ctx.needs_input_grad[2]:
    grad_bias = grad_output.sum(0).squeeze(0)
  return grad_input, grad_weight, grad_bias, grad_stride, /
    grad_padding, grad_dilation, grad_groups
```

The function torch.nn.grad.conv2d_input computes the gradients referred to input and this function is based on the use of the transposed convolutional function. In the library torch.grad where this function is defined, there is a com-

putation for the dimensions of the parameter related to the gradients required and the call to the function conv_transpose2d of the library torch. The function torch.nn.grad.conv2d_weight computes the gradients with respect to weights and its definition is slightly different from the previous one. The function contiguous is called many times and the convolution is performed using the conv2d function of torch. The results must be modified with view and contiguous in order to match the right dimensions necessary for the computation. These functions are inserted in the grad package of the library torch to allow a different computation of the gradients instead of the classic autograd, avoiding a critical speed down of the computation. In this way, there is a code similar to the one for the linear layer, defining how the variables grad_input, grad_weight and grad_bias are computed.

FA

The backward part is reported to understand the differences with the backpropagation code:

```
def backward(ctx, grad_output):
  input, weight, weight_fa, bias = ctx.saved_tensors
  grad_input = grad_weight = grad_weight_fa = None
  grad_bias = grad_stride = grad_padding = None
  grad_dilation = grad_groups = None
  if ctx.needs_input_grad[0]:
   grad_input = torch.nn.grad.conv2d_input(input.shape, /
                weight_fa, grad_output)
  if ctx.needs_input_grad[1]:
   grad_weight = torch.nn.grad.conv2d_weight(input, /
                 weight.shape, grad_output)
  if bias is not None and ctx.needs_input_grad[2]:
   grad_bias = grad_output.sum(0).squeeze(0)
  return grad_input, grad_weight, grad_weight_fa, grad_bias, /
      grad_stride, grad_padding, grad_dilation, grad_groups
```

How the grad_input is computed has been modified only in the FA code, replacing the variable weight with the weight_fa. The variable grad_weight can be computed replacing the weight.shape with weight_fa.shape, but the dimensions are the same.

DFA

In the DFA the grad_input variable computation is not required since the error information is not propagated through all the layers as seen before.

```
def backward(ctx, grad_output):
  input, weight, bias = ctx.saved_tensors
  grad_input = grad_weight = grad_bias = grad_stride = None
  grad_padding = grad_dilation = grad_groups = None
  if ctx.needs_input_grad[1]:
     grad_weight = torch.nn.grad.conv2d_weight(input, /
                   weight.shape, grad_output)
  if bias is not None and ctx.needs_input_grad[2]:
     grad_bias = grad_output.sum(0).squeeze(0)
  return grad_input, grad_weight, grad_bias, grad_bias, /
         grad_stride, grad_padding, grad_dilation, grad_groups
```

In this code, it is possible to see how the computation of grad_input has been removed and the grad_output is modified using the register hook as seen before. This is done because in the FA there is an agreement of dimensions between the random matrices and the weights matrices used during forwarding, but these dimensions are not respected in the DFA case. The gain respect the previous code is the reduction of computation since the values of grad_input are not computed two times.

3.4.10 Results

The codes exposed are simulated with the same conditions used previously on the network without custom modules: the FA and DFA have random weights initialized with uniform distribution in the interval (0, 0.001), while the BP has a uniform distribution in the interval (-stdv, stdv), since it is not useful to modify the initial conditions for this algorithm.

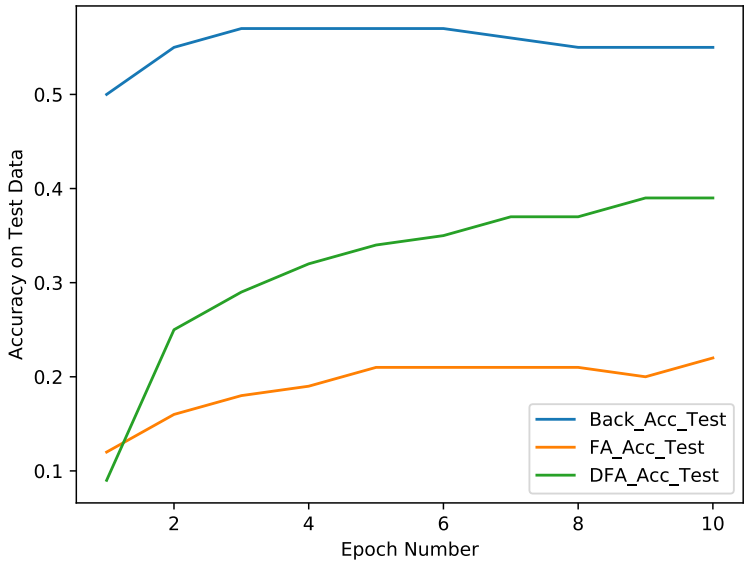

Figure 3.13: Test accuracy on a convolution network using custom modules to classify Cifar10

Table 3.5: Graphs of convolutional network on Cifar10 with (0, 0.005) initialization

Algorithm	**lr**	**Best Test Accuracy**
BP	0.001	57%
DFA	0.001	39%
FA	0.001	22%

The results show as the DFA and FA have functions similar to Figure 3.9 and this is due to the same initialization used. The accuracy value obtained is lower because the number of iterations has been reduced to analyze better the different results and their behaviour. The BP seems to perform as in the case without modules, so the custom module implementation is considered good. The FA algorithm has achieved a lower accuracy, so another initialization is applied and the weights have a uniform distribution in the interval (-stdv, stdv). The simulation is performed several times because the results show particular behaviours:

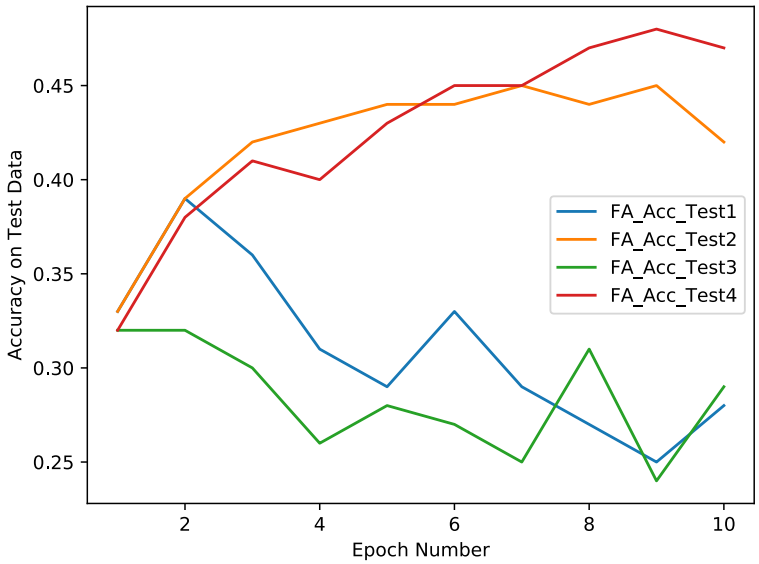

Figure 3.14: Test accuracy on a convolution network using only FA algorithms

Table 3.6: Graphs of convolutional network on Cifar10 with (0, 0.005) initialization

Algorithm	**lr**	**Best Test Accuracy**
FA1	0.001	39%
FA2	0.001	45%
FA3	0.001	32%
FA4	0.001	48%

The four simulations have two different behaviours: the red and orange lines are the simulations where the network is able to train in a proper way and the results are similar to the previous ones for the DFA implementations. The blue and green lines show the phenomenon of exploding gradients since there is a network starting to learn, but after two epochs the degradation occurs.

3.5 Developing of ResNet with FA and DFA algorithms

The ResNet code has been taken from Github repository and the structure is derived from the paper of Kaiming and others [12] [11]. For this work, ResNet18 has been chosen, such network has 18 layers with shortcuts that provide the identity function as required by the model. Some shortcut is formed by layers to provide an increase of dimensions because the hidden layers have an increasing number of output channels going deeper [11]. The whole network has a convolutional layer as first, followed by blocks of convolutional and batch normalization ones. The last layer is the linear one that allows the classification with 10 output channels, one for each class. The blocks that form the ResNet are called BasicBlock and they are based on convolutional layers and batch normalization. This structure is valid only for ResNet18 and ResNet34 models because increasing the number of layers the BasicBlock is replaced by the Bottleneck module formed with a different combination of layers. The convolutional and linear modules with their relative functions are replaced by the custom ones to implement the DFA and FA. The custom functions have been called MyConv and MyLinear. The codes are similar to the ones used for the convolutional and linear layers of the previous simulations. The backward step is reported to understand the implementations:

```
def backward(ctx, grad_output):
  input, weight, weight_fa, bias = ctx.saved_tensors
  grad_input = grad_weight = grad_weight_fa = grad_bias = None
  grad_stride = grad_padding = grad_dilation = grad_groups = None
  if ctx.needs_input_grad[0]:
    grad_input = torch.nn.grad.conv2d_input(input.shape, /
                 weight_fa, grad_output)
  if ctx.needs_input_grad[1]:
    grad_weight = torch.nn.grad.conv2d_weight(input, /
                 weight.shape, grad_output)
  if bias is not None and ctx.needs_input_grad[2]:
    grad_bias = grad_output.sum(0).squeeze(0)
  return grad_input, grad_weight, grad_weight_fa, grad_bias, /
        grad_stride, grad_padding, grad_dilation, grad_groups
```

The computation of grad_input is modified replacing the variable weight with the weight_fa one.

DFA

In the DFA code the grad_input computation steps have been modified, since the function torch.nn.grad.conv2d_input() is not able to work with an output gradient of different dimensions:

```
if ctx.needs_input_grad[0]:
 a = weight_dfa.shape[1]
 b = weight_dfa.shape[2]
 c = weight_dfa.shape[3]
 grad_input = grad.nn(weight_dfa.view(-1, a*b*c))
 grad_input = grad_input.view(-1, a, b, c)
```

The grad variable is the result of the function delta_cross_entropy seen previously and this parameter is passed using custom functions added to the convolutional layer function. In this way it is possible to pass the error value to all layers and to modify the grad_input computation. It is requested to modify the weight_dfa variable to have a concordance of dimensions between the product and the gradient.

3.5.1 Problem implementing custom modules

The results of these implementations are not available because the codes have requested too much time to simulate and the first simulations usually give only some configuration hyperparameter to be set as weight initialization or magnitude for the learning rate. The codes have been modified in order to have faster simulations and an idea has been taken from PyTorch forum [17], where a function was inserted to change the values of the weights used for the backward.

3.5.2 Developing of ResNet switching weights values for FA

The function to perform the switching of the weights has been inserted in the main code, where the training is performed. The function update is called after the output values computation from the network. The backpropgation is executed through the function backward() and the function update is called again to switch the weights since the ones to be updated using gradients are the forward ones. The code is shown to understand how the switch is performed [17]:

```
def update(self, mode):
  if mode == 'backward':
    self.forward_weight.copy_(self.weight)
    self.weight.data.copy_(self.backward_weight)
  elsif mode == 'forward':
    self.weight.data.copy_(self.forward_weight)
  return
```

The function .copy_ is an in-place operation that will be computed in a single step and the variables are maintained separated. The forwarding weights must be saved every time in a variable, since they must be restored to perform the optimization and used again during the forward step.

3.5.3 Developing of ResNet using register hook function for DFA

In the DFA algorithm, the dimensions of the random weights are different from the forwarding ones, so it is impossible to perform the switching as in the FA code. The solution provided uses the register_hook function passing the delta variable.

```
def grad_update(self, delta):
  a = self.weight_dfa.shape[1]
  b = self.weight_dfa.shape[2]
  self.grad_dfa = (delta.mm(self.weight_dfa.view(-1, a*b*b)))
  self.output.register_hook(lambda grad: /
                                self.grad_dfa.view(-1, a, b, b))
  return
```

The gradients computation is similar to the codes seen before since the gradient is computed using the function delta_cross_entropy and the product with a random matrix. The dimensions of the weight_dfa variables must be adjusted with the function view. The problem of this code is found in the error propagation, due to CUDA limitation. This is due to the memory allocable by the GPU, but a solution is provided using the following function:

```
def free_space(self):
    del self.grad_dfa
```

Since the DFA algorithm requires gradients matrices for every layer and the CUDA error occurs different times, the matrices of gradients are deleted after the backward step in order to free space in memory and to avoid exceeding the limit. The GPU memory has to maintain different values as the dynamic graph information and other variables, so the addition of other variables to be saved has generated this error.

3.5.4 Results Back Propagation

The backpropagation code is not modified, in order to obtain results as accurate as the original ones presented by the author of the code. In the instruction provided it is used a learning rate of 0.1, but it is requested too much time to reach values as the one presented by the author [12]. So it is used a learning rate that is two magnitudes smaller and the code is iterated two times over 200 epochs. The accuracy obtained is 92,58% and it is quite similar to the one of the author of the code (93,05%).

Table 3.7: Simulations ResNet18 with backpropagation algorithm

Iteration	**epochs**	**lr**	**Test Accuracy**
1	200	0.001	91,56%
2	200	0.0001	92,58%

The code allows to train over a variable number of epochs and when a higher accuracy value is found, the network parameters are saved in a file such that it is possible to iterate again the training phase with modified hyperparameters, but using the variables found previously and using them from the beginning.

3.5.5 Results FA

The FA implementation shows the limit of this algorithm to be applied on a network complex as the ResNet. A huge number of iterations is required to reach an accuracy of 64,47% and this is a poor solution, because the same results can be obtained with simpler networks and there is no gain using a deeper network. The main motivation for this result can be the inability for the FA to perform a good alignment in the first layers and the complexity of the convolutional network is an obstacle for this algorithm.

Table 3.8: Simulations ResNet18 with Feedback Alignment

Iteration	**epochs**	**lr**	**Test Accuracy**
1	150	0.1	50,64%
2	150	0.05	53,99%
3	200	0.01	60,09%
4	200	0.005	64,04%
5	200	0.001	64,74%

The number of epochs to reach the final accuracy value is over 800 epochs, so the time requested to perform this training is much greater than the one requested

by the simpler networks seen before.

3.5.6 FA Nokland and Lillicrap

In order to understand if the weight initialization used is correct, different experiments has been performed. The first one is the same initialization of Nøkland [15]. So the weights used during the forward are initialized with uniform distribution in the interval (-stdv, stdv), where stdv is obtained in this way:

$$stdv = \frac{1}{\sqrt{n_1}} * \sum_w w^2 \tag{3.4}$$

where n_1 is the product between the dimensions of the kernel and the input channels. For the random weights instead the uniform distribution is in the interval (-$stdv_2$, $stdv_2$), where $stdv_2$ is obtained in this way:

$$stdv_2 = \frac{1}{\sqrt{n_2}} * \sum_w w^2 \tag{3.5}$$

where n_2 is the product between the dimensions of the kernel and the output channels. The Nøkland implementation shows an initial training similar to the previous case.

Table 3.9: Nøkland weight initialization on ResNet18 with FA

FA Nøkland

Iteration	epochs	lr	Test Accuracy
1	150	0.1	49,60%
2	175	0.05	52,61%

The training steps are similar to the previous case and an accuracy of 52,61% is reached. In the Lillicrap method instead a different initilization is used with weights uniform distributed in the interval (-0.0001, 0.0001) and the random weights in the interval (-0.5, 0.5) [13].

Table 3.10: Lillicrap weight initialization on ResNet18 with FA

FA Lillicrap

Iteration	epochs	lr	Test Accuracy
1	none	0.1	10%

The results show a network not able to learn and this can be due to the different depth of the ResNet respect the networks used by Lillicrap.

The Nøkland initialization seems to be the best to work with ResNet, because the magnitude of the weights is scaled going to layers with an higher number of output and input channels and the error propagation has a different effect.

3.5.7 Results DFA

The DFA algorithm has shown good results also in the implementation of convolutional networks, but the results obtained with the ResNet are not good to consider this algorithm a replacement for the backpropagation. The weights values used for the random matrices are different: the first time they are initialized with uniform distribution in the interval (-0.001, 0.001) and the second one they are initialized with uniform distribution in the interval (-0.0001, 0.0001). Better results are obtained reducing the magnitude of the maximum values in the interval, since an higher value of test accuracy is reached (32% in epoch 2).

Table 3.11: First weight initialization on ResNet18 with DFA

DFA with weight in [-0,001; 0,001]

Iteration	epochs	lr	Test Accuracy
1	5	0.1	10%

Table 3.12: Second weight initialization on ResNet18 with DFA

DFA with weight in [-0,0001; 0,0001]

Iteration	epochs	lr	Test Accuracy
1	22	0.1	10%

Both the iterations have given a final test accuracy of 10%, due to the phenomenon of exploding gradients. The effect is similar to the one occurred using the IFA algorithm on deeper networks, since there is an initial phase of learning, but the cost function goes to infinite and the network starts to perform worse every epoch.

3.5.8 Failing FA and DFA and state of the art comparison: approach to sign concordance

The results shown before are the demonstration that applying these algorithms on the ResNet is not an optimal solution due to the depth that characterizes this network. It is searched what has been proposed in the state of the art for the implementation of random matrices on networks so deep. The first hint is in the paper of Qianli where it is exposed the importance of sign concordance between forward and backward, that allows discarding the magnitude information of the weights reporting back the error [18]. This solution seems interesting because using the random

propagation many times the problem of the exploding gradients occurred. In the same paper, the Batch Manhattan is presented an additional optimization to limit the exploding gradients phenomenon [18].

3.5.9 Batch Manhattan

This operation is applied using the sign function on the gradients values before the weights values updating. The optimization is applied discarding the magnitudes for the gradients and only the signs information is preserved. Good results are obtained limiting the exploding gradients phenomenon, but a reduction of the learning rate is necessary to have valid results.

3.6 Simulations results and comparison with papers ones

3.6.1 FA with initial sign concordance

The first modification to the codes is applied in the initialization phase. The sign concordance is imposed when the forwarding weights and random weights are initialized. Different learning rates are tried in order to understand if the network fails to obtain higher accuracy due to stuck points. The first iteration is performed starting from a learning rate set to 0.01, while in the second iteration it is set to 0.001.

Table 3.13: Results of ResNet18 with FA and sign concordance

Iteration	**epochs**	**lr**	**Test Accuracy**
1	200	0.1	51,57%
2	200	0.05	54,32%
3	200	0.01	60,37%
4	200	0.005	62,48%
5	200	0.001	66,83%

The first result shows a similar behaviour as the experiment without sign concordance, reaching an accuracy of 66,83%.

The learning rate is scaled an order of magnitude:

Table 3.14: Results of ResNet18 with FA signed scaled learning rate

Iteration	epochs	lr	Test Accuracy
1	200	0.001	68,26%
2	200	0.0001	No improve

The second experiment shows that the number of epochs required to reach the same accuracy is lower. This result can be related to the sign concordance that has the effect to reduce the initial error training that characterizes the FA algorithm so a lower learning rate can be used since the learning is faster.

3.6.2 FA with Batch Manhattan and initial sign concordance

In order to apply the Batch Manhattan optimization the main code has been modified inserting a function called sign_grad that has the role to perform this optimization in all the layers of the ResNet. The function is called in the different modules that form the ResNet, so the BasicBlock, the convolutional layer and the linear layer:

```
def sign_grad(self):
    self.weight.grad = self.weight.grad.sign()
    return
```

In this way it is possible to see how the values saved in the variable grad are modified mantaining only the sign information.

Table 3.15: Results of ResNet18 with the Batch Manhattan and initial sign concordance

Iteration	epochs	lr	Test Accuracy
1	200	0.00001	72,16%
2	200	0.000001	No improve

The first implementations have performed very poorly and they are not reported. This is due to the sign_grad function applied to gradients. A reduction of the learning rate by different magnitude orders is necessary to make this algorithm work. Solutions very interesting are found reducing the learning rate to 0.00001. The accuracy has grown up to 72,16% and this is important because it means that

the algorithm is able to learn without the magnitude information and this algorithm can be applied also on network deep as the ResNet.

3.6.3 DFA gradients signed

The use of Batch Manhattan have improved the results for the FA algorithm so this optimization is implemented also on the DFA in the same way. This is done because the first simulations using the DFA on a ResNet are characterized by the exploding gradients phenomenon. In order to understand if the DFA algorithm needs some particular weight initialization, three different simulations are performed. In the first one the random weights are initialized with uniform distribution in the interval [0; 0.0000001]:

Table 3.16: Results of ResNet18 using DFA and Batch Manhattan

Iteration	**epochs**	**lr**	**Test Accuracy**
1	200	0.0001	45,86%
2	200	0.00005	No improve

The network is able to reach an accuracy value about 45,86% using this configuration, but the simulation seems to stuck in a local minima and the accuracy doesn't grown anymore also after scaling the learning rate. The next simulation has a modified interval set to [0; 0.0001]:

Table 3.17: Results of ResNet18 using DFA and Batch Manhattan with different weights

Iteration	**epochs**	**lr**	**Test Accuracy**
1	215	0.0001	46,84%

In the last case the interval is set equal to [-stdv, stdv] as the one used for the FA case, when it give back the best accuracy:

Table 3.18: DFA using weights scaling with depth and Batch Manhattan

Iteration	**epochs**	**lr**	**Test Accuracy**
1	133	0.0001	39,66%

The first simulations have similar results, because the effect of the value initialization is almost the same. In the third solution the best accuracy is reached at epoch 40 and after that the improvements are not consistent.

3.6.4 FA with Batch Manhattan and sign concordance every epoch

The next step has been to apply the sign concordance between forwarding weights and random ones every epoch in addition to Batch Manhattan. The code has been modified in this way:

```
def update(self, mode):
    if mode == 'backward':
        self.forward_weight.copy_(self.weight)
        self.weight.data.copy_(self.backward_weight.abs() /
                               * self.forward_weight.sign())
    elsif mode == 'forward':
        self.weight.data.copy_(self.forward_weight)
    return
```

As can be seen in the codes there is the multiplication between the signs of the forwarding weights and the absolute values of the weights used during the backward. This product is copied in the variable backward_weight. It is necessary to apply the abs() function to discard the signs randomly generated.

Table 3.19: Results of ResNet18 using FA, Batch Manhattan and sign concordance every epoch

Iteration	epochs	lr	Test Accuracy
1	200	0.00001	85,12%
2	100	0.000001	86,86%

The results show an increase of 10% of the accuracy reaching 86,86%. This value is near the one obtained implementing the classical backpropagation. The importance of the sign concordance is highlighted by the results obtained with these simulations and the next experiment aims to apply some modification in order to understand if it is possible to replace the backpropagation with some solution that has a reduced computational effort.

3.6.5 FA with Batch Manhattan and random weights generated every batch

Another experiment that has been tried is to generate the random matrix used for the backpropagation every epoch instead to initialize them during the module definition for the convolutional layer. This solution requires the sign concordance between weights generated and ones used in the forward step.

```
def update(self, mode):
    if mode == 'backward':
        self.forward_weight.copy_(self.weight)
        self.backward_weight = torch.FloatTensor(self.weight.size())
        self.backward_weight.uniform(-self.stdv, self.stdv)
        self.weight.data.copy_(self.backward_weight.abs() \
                               * self.forward_weight.sign())
    elsif mode == 'forward':
        self.weight.data.copy_(self.forward_weight)
    return
```

The definition of the values for the variable self.backward_weight is performed every time before the backward step. This solution is derived from the concept proposed by Qianli where the magnitude of the values is not more important if the sign concordance is respected [18]. In the FA algorithm proposed by Lillicrap is required to save the random matrices in order to have these values fixed, while using the sign concordance the matrices can be locally generated, used to propagate the error and discarded.

Table 3.20: Results of ResNet18 using FA and random weights generated every epoch

Iteration	**epochs**	**lr**	**Test Accuracy**
1	150	0.00001	83,58%
2	96	0.000001	86,14%

The results are comparable to the ones presented before, since the test accuracy is similar. In the next implementation the sign concordance is used avoiding the Batch Manhattan optimization, in order to understand if it is sufficient to limit the phenomenon that causes the degradation in the previous codes.

3.6.6 FA using only sign concordance every epoch

The Batch Manhattan is removed avoiding the step where the gradients are replaced by their signs. The learning rate has been set to 0.001, because the Batch Manhattan solution requires a scaled learning rate:

Table 3.21: Results implementing only the sign concordance every epoch

Iteration	epochs	lr	Test Accuracy
1	200	0.001	85,43%
2	200	0.0001	87,05%

The accuracy value reached is slightly higher than in the previous cases and this is possible since using the Batch Manhattan the magnitude information of the gradients is discarded to remove the exploding phenomenon that occours, but the sign concordance is sufficient to avoid its presence.

3.6.7 ResNet34 with Batch Manhattan and sign concordance every epoch

An implementation on the ResNet34 is tried in order to understand if this algorithm can be applied to deeper networks. The Batch Manhattan is used and the sign concordance is performed every epoch. These choices have been done because the ResNet34 is a deeper network and the phenomenon of exploding gradients must be avoided:

Table 3.22: Results implementing sign concordance every epoch and Batch Manhattan on ResNet34

Iteration	epochs	lr	Test Accuracy
1	200	0.00001	84,97%
2	130	0.000001	87,14%

The higher accuracy value obtained indicates the possibility to work with this implementation also on deeper networks. The accuracy difference between this solution and the same on a ResNet18 is very small, only 0.28%. This result is reasonable since the accuracy difference between the ResNet18 and ResNet50 using the backpropagation is 0.6%.

3.6.8 Error propagation using forward weights signs scaled by stdv and Batch Manhattan

The last implementation uses the matrices obtained by the forward ones after the sign function is performed. The matrix of signs is used to propagate the error obtained in the last layer and this solution avoids the use of the forward weights and also the need to define new matrices:

```
def update(self, mode):
    if mode == 'backward':
      self.forward_weight.copy_(self.weight)
      self.weight.data.copy_(self.forward_weight.sign() /
                             * self.stdv)
    elsif mode == 'forward':
      self.weight.data.copy_(self.forward_weight)
    return
```

The update function has been modified and it is possible to see how the matrices obtained by the sign function need to be scaled and the self.stdv factor is used.

Table 3.23: Results using the signs to brought back the error information

Iteration	**epochs**	**lr**	**Test Accuracy**
1	150	0.00001	88,96%
2	200	0.000001	90,25%

The results show an accuracy comparable to the one obtained using the back-propagation. This is an interesting result because with respect to the last algorithm there is a safe in the memory occupied by the random matrices and the multiplication between the backward and forward matrices is avoided.

3.6.9 Error propagation using forward weights signs scaled by 0,001 and Batch Manhattan

The previous result shows as it is possible to have a network able to learn without the magnitudes information of the weights used during the forward step. The simulation performed next is based on a different scaling factor, using 0.001 instead of stdv.

Table 3.24: Results implementing a different scale factor for forward signs

Iteration	epochs	lr	Test Accuracy
1	150	0.00001	78,25%
2	200	0.000001	80,51%

It is obtained a network able to learn better than the FA algorithm, but it has a 10 % drop of accuracy respect the previous algorithm.

3.6.10 Error propagation using forward weights signs scaled by stdv

The last experiment performed avoids the Batch Manhattan optimization, so the sign() function on the gradients is not performed.

Table 3.25: Results using only signs to brought back the error information

Iteration	epochs	lr	Test Accuracy
1	200	0.001	88,18%
2	200	0.0001	90,03%
3	200	0.00001	90,05%

The results are similar to the previous case but there is the advantage that the sign function is not applied on the gradients, so the algorithm has a reduced computational cost and the same accuracy value has been reached.

3.6.11 Comparison with paper results

The performances of the different solutions obtained are compared to the state of the art about the use of random matrices to transport back the error in the network. These papers are almost recent, published between the end of November 2018 and January 2019 [20] [22] [24] [1].

Moskovitz's considerations

In the paper of Moskovitz, it is noted as using random matrices a condition of asymmetry between forward and backward is imposed resulting in the inability for complex networks to learn properly [22]. The use of the sign concordance has been presented as a solution to limit the phenomenon of exploding gradients or vanishing ones [22]. The network used is simpler than the ResNet of this work, but the results are interesting for the study of the FA algorithm. For a network with 8 hidden layers, 9 convolutional ones and one linear, the difference between BP and FA is about 25% lower, while the solution using the sign concordance is 1.6%. The DFA is implemented only in a smaller network and it performs really bad, similar to FA, with a 10% difference respect the backpropagation [22]. This factor is obtained on a network with two convolutional layers followed by max-pooling and two linear layers. The DFA algorithm is not performed on more complex networks due to memory requirements [22].

Bartunov's considerations

In the paper of Bartunov, there are experiments on MNIST and Cifar10 datasets. The importance of this paper is in the difference in performance reached by FA, DFA and backpropagation algorithms. With the MNIST dataset there are good results as ones presented at the beginning of this work, but using the Cifar10 the DFA performs worse than the other two solutions, probably due to the use of the complex dataset. In order to extract more features, a deeper network is required, but the DFA is not able to provide learning.

Xiao's considerations

Solutions similar to this work are reported in this paper, it is analyzed the results obtained implementing the Lillicrap's FA, the Qianli's sign concordance and the BP in a ResNet18 using PyTorch to classify the ImageNet dataset [24]. The solution adopted to change the value of the weights is different to the one used in this work, since it is used the function torch.legacy.nn.SpatialConvolution [25]. In this way, during the backward, it is used directly another variable instead to copy values between variables. It is not possible to make direct comparisons between the solutions

of this work and the paper one since the ImageNet is quite different from the Cifar10, but it is possible to analyze how the three algorithms perform on a ResNet18 to understand if they are able to classify in a comparable manner as the solutions presented by this work. The results show as the FA has an accuracy lower than BP and sign concordance case. While the FA implementation reaches a test error of 90,52%, the BP and sign concordance reach 33,14% and 37,91% respectively [24]. It must be considered that ImageNet has a wider dataset and there are 1'000 classes, but the difference in performances between the signs concordance and the random matrices of FA confirms the great importance of the first one.

3.7 Computational efforts evaluation

The computational efforts required by the different solutions implemented are evaluated considering the simulation results, in order to understand which one can have a lower computational cost during the training of the network reaching a good accuracy value. This evaluation is performed to understand if the different solutions can replace the classic backpropagation algorithm. The GPUs architectures used during the simulations are Kepler ones since the last part of the work is performed on Nvidia Tesla K40 and K80. The difference between the CPU and GPU architectures is the number of ALU inside the GPU and the reduction of cache memory [16].

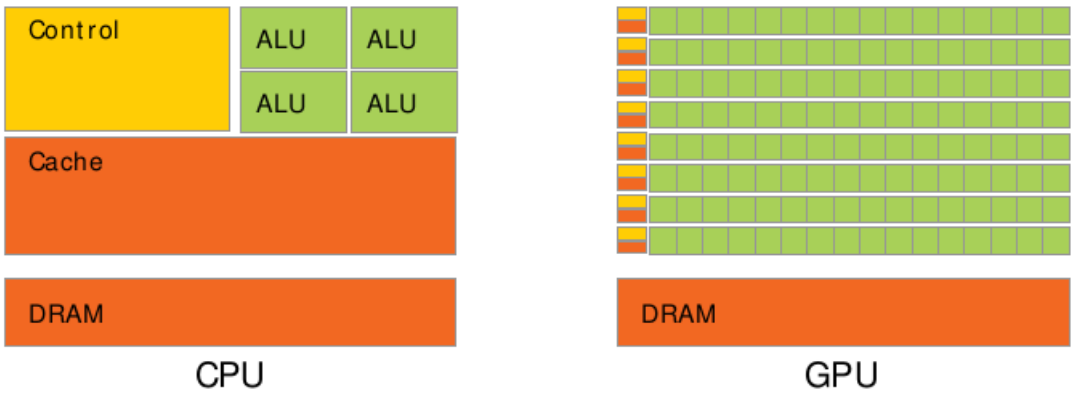

Figure 3.15: CPU and GPU architectures comparisons [16]

Using the huge ALU number it is possible with a GPU to work with parallel threads, despite the smaller cache memory inside the processor. This factor allows a faster computation performed by GPUs, but an higher number of accesses in the DRAM memory are required and this can be a loss of time. It is required a good management of memory to reduce the number of accesses in DRAM and the PyTorch framework has been chosen for this reason.

3.7.1 Evaluation of operations required for algorithms

The evaluation of the computational efforts is based on the analysis of the algorithms used during simulations to understand which are the basic operations of each algorithm in order to hypothesize the cost required by the single operation. The basic operations considered are:

- Copy data from DRAM to cache
- Transpose the matrix
- Product between matrices
- Saving values in DRAM
- Function sign() on matrix elements

The complexity of these operations are related to the instructions performed by the calculator, so it is checked details provided by Nvidia documents.

Table 3.26: Operations Bandiwth comparison [9]

Operation	**Effective Bandwidth(GB/s)**
Simple Copy	96.9
Naïve Transpose	2.2

The copy and transpose operations are similar, but the second one has a computational cost higher than the simple copy due to the partition camping of the memory [9]. This is related to the division of the memory in banks, that gives problems not allowing to use all banks in an optimal way. For this reason Nvidia systems adopted solutions to reduce the performance differences, going to use a diagonal indexing for the elements of the matrices.

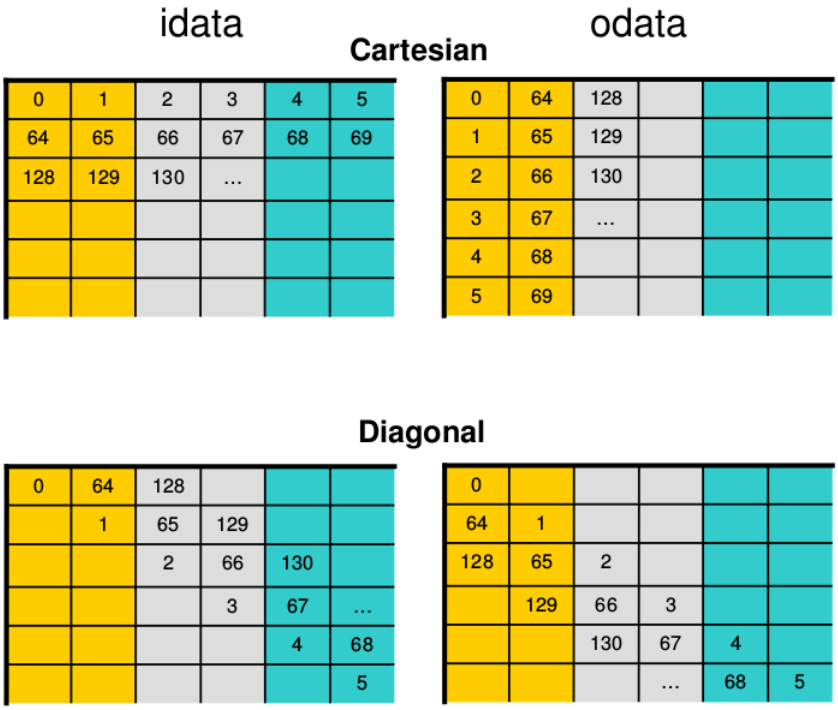

Figure 3.16: How the matrix is indexed in optimized way [9]

Table 3.27: Operations Bandwidth comparison [9]

Operation	**Effective Bandwidth(GB/s)**
Simple Copy	96.9
Shared Memory Copy	80.9
Naïve Transpose	2.2
Diagonal	69.5

Implementing the diagonal matrix solution the transpose operation is faster as can be seen from the table. Since the GPU works with parallelized operations performed by threads, also the product operation between matrices is divided using different threads. Every thread will perform the operation related to one element for each matrix, the product will be parallelized and the results matrix is obtained by the different products provided by threads. The analysis cost will be performed referring to the whole elements of the matrices and not to the cost of the operations between the single elements performed by every thread. Considering the operations required during the training phase, an evaluation of the computational load of every operation has been performed, in relation to data found on Nvidia documents. Since the values shown are referred to different systems and these values are varied also by the computer components, it is used a unit only with the purpose to compare the higher complexity of an operation respect another. The easier operation considered

is the multiplication, indicated with 1, while the transpose is referred with 3. The algorithms that will be considered are:

- Back Propagation (**BP**)
- Feedback Alignment (**FA**)
- Direct Feedback Alignment (**DFA**)
- Direct Feedback Alignment using Batch Manhattan (**DFA+Signgrad**)
- Feedback Alignment with initial sign concordance and Batch Manhattan (**FAsign+Signgrad**)
- Feedback Alignment with random matrices every epoch and Batch Manhattan (**FArand+Signgrad**)
- Feedback Alignment with sign concordance every epoch(**FAsign**)
- Signs of forward weights used to propagate error in each layer, after the scaling for constant value and Batch Manhattan (**Sign+Signgrad**)
- Signs of forward weights used to propagate error in each layer, after the scaling for a constant value (**Sign**)

3.7.2 BP operations

- Copying weights from memory to registers (**2**)
- Transposition and saving of transposed weights in memory (**3**)
- Copying transposed weights from memory to registers (**2**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between transposed weights and gradients of Loss referred to layer output to obtain gradients of input that will be the output of next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for the next layer (**2**)

- Total cost: **17**

The accuracy obtained is 92,58% and it is the result of the code implementing Back Propagation, so the original code taken from Github repository is not modified [12]. The computational cost is an index of the elemental operations that the calculator must perform to execute a specific algorithm. In order to have a more efficient training phase, a lower computational cost must be obtained from the other algorithms considered.

3.7.3 FA operations

- Copying random weights from memory to registers (**2**)
- It is not required to transpose weights matrices since these are defined with right dimensions (**0**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between random weights and gradients of Loss referred to layer output to obtain gradients of input that will be the output of the next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for the next layer (**2**)
- Total cost: **12**

The final cost is lower than using **BP**, but there is a lower accuracy value too. The random matrices that replace the transposed ones have to be saved in DRAM since they must be fixed during the whole training phase. In the case of **BP** the transposed are deleted after the Loss gradients are computed. For this reason the implementation will have a computational cost lower, but there is a higher memory requirement that can slow down the training phase due to DRAM accesses. The accuracy obtained is 68,26%.

3.7.4 DFA operations

- Copying random weights from memory to registers (**2**)

- It is not required to transpose weights matrices since these are defined with right dimensions (**0**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between random weights and gradients of Loss referred to last layer output to obtain gradients of the output of the layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Writing weights gradients in memory (**2**)
- It is not required to compute input gradients to propagate (**0**)
- Total cost: **10**

The final cost is very low, but there is the problem of a higher memory requirement, since the random matrices must be fixed. In addition during the simulation there are problems working with ResNet18, since the phenomenon of exploding gradients makes the training impossible.

3.7.5 DFA+Signgrad operations

- Copying random weights from memory to registers (**2**)
- It is not required to transpose weights matrices since these are defined with right dimensions (**0**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between random weights and gradients of Loss referred to last layer output to obtain gradients of the output of the layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Application of sign function on the weights gradients (**1**)
- Writing weights gradients in memory (**2**)
- It is not required to compute input gradients to propagate (**0**)
- Total cost: **11**

This case is similar to the previous one, but using the Batch Manhattan the phenomenon of exploding gradients is avoided. The maximum accuracy obtained is 46,84%. This solution has an accuracy value too low to consider it as a replacement for the Back Propagation algorithm.

3.7.6 FAsign+Signgrad operations

- Copying random weights from memory to registers (**2**)
- Copying weights from memory to registers (**2**)
- Perform the sign() function on weights matrices (**1**)
- Perform transposed of sign matrices and writing in memory (**3**)
- Copying transposed signs matrices from memory to registers (**2**)
- Multiplication between the transposed signs matrices and random weights matrices (**1**)
- Copying gradients of Loss referred to layer output in register (**2**)
- Multiplication between random+sign weights and gradients of Loss referred to last layer output to obtain gradients of input that will be the output of the next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Application of sign function on the weights gradients (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for next layer (**2**)
- Total cost: **22**

This solution has good accuracy results reaching a value of 86,86%. The problem is the higher DRAM memory requirement to save the random weights and the transpose operation must be performed to apply the sign concordance. The computational cost is higher than the **BP** ones and this makes the solution not optimal to replace the Back Propagation.

3.7.7 FArand+Signgrad operations

- Random weights generation (**2**)
- Copying weights from memory to registers (**2**)
- Perform the sign() function on weights matrices (**1**)
- Perform transposed of sign matrices and writing in memory (**3**)
- Copying transposed sign matrices from memory to registers (**2**)
- Multiplication between the transposed sign matrices and random weights matrices (**1**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between random+sign weights and gradients of Loss referred to last layer output to obtain gradients of the output of the next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Application of sign function on the weights gradients (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for the next layer (**2**)
- Total cost: **22**

The problem of a higher DRAM memory requirement is solved generating the random weights every epoch. This operation is more complex than the multiplication between matrices. The accuracy is about 86,14%. There is lower memory occupied, but there is an increase in the computational cost due to the required reading and transpose operation to be performed on the forward weights.

3.7.8 FAsign operations

- Copying random weights from memory to registers (**2**)
- Copying weights from memory to registers (**2**)
- Perform the sign() function on weights matrices (**1**)
- Perform transposed of sign matrices and writing in memory (**3**)

- Copying transposed sign matrices from memory to registers (**2**)
- Multiplication between the transposed sign matrices and random weights matrices (**1**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between random+sign weights and gradients of Loss referred to last layer output to obtain gradients of the output of the next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for next layer (**2**)
- Total cost: **21**

Respect the previous case the use of the Batch Manhattan optimization has been removed, obtaining anyway good results of accuracy about 87,05%. The higher memory requirement and the increased computational cost makes this solution not optimal to replace the Back Propagation.

3.7.9 Sign+signgrad operations

- Copying weights from memory to registers (**2**)
- Copying the scale factor STDV from memory to registers (**2**)
- Perform the sign() function on weights matrices (**1**)
- Perform transposed of signs matrices and writing in memory (**3**)
- Copying transposed signs matrices from memory to registers (**2**)
- Multiplication between the transposed signs matrices and STDV (**1**)
- Copying gradients of Loss referred to layer output in registers (**2**)
- Multiplication between scaled signs and gradients of Loss referred to last layer output to obtain gradients of the output of the next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)

- Application of sign function on the weights gradients (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for next layer (**2**)
- Total cost: **22**

The solution reduces the memory requirement using only the signs of the weights matrices scaled by the variable self.stdv. The accuracy is about 90,25%, but there is a computational cost higher.

3.7.10 Sign operations

- Copying weights from memory to registers (**2**)
- Copying the scale factor STDV from memory to registers (**2**)
- Perform the sign() function on weights matrices (**1**)
- Perform transposed of signs matrices and writing in memory (**3**)
- Copying transposed signs matrices from memory to registers (**2**)
- Multiplication between the transposed signs matrices and STDV (**1**)
- Copying gradients of Loss referred to layer output in register (**2**)
- Multiplication between scaled signs and gradients of Loss referred to last layer output to obtain gradients of the output of the next layer (**1**)
- Copying activations in registers (**2**)
- Multiplication between gradients of Loss referred to layer output and activations to obtain gradients of weights (**1**)
- Writing weights gradients in memory (**2**)
- Writing in memory the input gradients that will be used for next layer (**2**)
- Total cost: **21**

This solution is the best one achieved through the implementations adopted since its computational cost is the same for the **FAsign** solution, but there is an higher accuracy value comparable to the one of the **BP**, reaching a value of 90,05%. The memory requirement is reduced since there are not random matrices to be saved, but the signs of the forward weights are used to propagate back the error. The importance of this result is the possibility to use values different from the forward

one, in order to propagate back the error information, but the network is able to learn only if a strictly concordance of signs is respected. The results about the computational cost evaluation and the accuracies used to compare each solution to the Back Propagation are reported:

Table 3.28: Summary of computational complexity and accuracies

	Computational Complexity	Accuracy
BP	17	$92,58\%$
FA	12	$68,26\%$
DFA	10	10%
DFA+Signgrad	11	$46,84\%$
FAsign+Signgrad	22	$86,86\%$
FArand+Signgrad	22	$86,14\%$
FAsign	21	$87,05\%$
Sign+signgrad	22	$90,25\%$
Sign	21	$90,05\%$

Chapter 4

Conclusions

4.1 Discussion about results

The results obtained in this thesis work have shown as the implementation of algorithms using random matrices to propagate back the error in deep neural network as ResNet are not able to train in a proper way. This factor is in contrast to the ideas exposed by Lillicrap and Nøkland asserting the ability to use fixed matrices to train networks since this factor is limited only to the first hidden layers. Working with small networks the gain that can be achieved using these solutions is pretty good since the transposition is avoided and a higher parallelization is possible. If these solutions are applied on complex networks as ResNet the accuracies values are lower and sign concordance is required between forward and backward weights. The sign concordance is not considered initially in order to work with matrices completely independent from the ones used during the forward step, but successively this concordance must be applied to obtain valid results. The best alternative to backpropagation can be achieved using only the signs to report back the error, but the computational cost is higher anyway. In this thesis work, it is highlighted how the use of random matrices can give back vanishing gradients or exploding ones, resulting in a network completely unable to work. The sign concordance importance is evident through the results obtained, but the computational efforts to take the values from the memory, to apply the function to extract the signs and to use them in the backward phase don't allow this solution to give a reduction of computational cost.

4.1.1 Promising implementations

The architectures able to work with a network complex as the ResNet are the GPUs of the last years, allowing the parallelization of operations using multithreads and enough memory storage for all the parameters required as weights, biases, activations and gradients values. During the simulations on a GPU with 2 GB of memory the problem of exceeding memory error is encountered many times, so it is required to work with GPU more performant and with more memory allocable. The possibility to introduce in the CUDA the operation of an optimized extraction of signs values, avoding the multiple operations required by the one proposed in this work, can give back a reduction in the computational efforts. The values 1 and -1 should be saved in cache during the whole training and an optimized alternative of the copy of values operations can be used to build the matrices used during the backward phase.

Bibliography

[1] Arild Nøkland and Lars H. Eidnes. "Training Neural Networks with Local Error Signals". In: *ArXiv e-prints* (Jan. 2019). ArXiv: 1901.06656 (cs.CV).

[2] PyTorch Contributors. *Get Started*. 2018. URL: https://pytorch.org/get-started/locally/1.

[3] PyTorch Contributors. *Torch.nn*. 2018. URL: https://pytorch.org/docs/stable/nn.html.

[4] PyTorch Contributors. *Training a classifier*. 2017. URL: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-pyl.

[5] Torch Contributors. *Automatic Differentiation package - torch.autograd*. 2018. URL: https://pytorch.org/docs/stable/autograd.html.

[6] M. El-Hadedy et al. "Performance and area efficient transpose memory architecture for high throughput adaptive signal processing systems". In: (2010), pp. 113–120. DOI: 10.1109/AHS.2010.5546272.

[7] Gavin Taylor et al. "Training Neural Networks Without Gradients: A Scalable ADMM Approach". In: *ArXiv e-prints* (May 2016). ArXiv: 1605.02026 (cs.CV).

[8] Google. *Welcome to Colaboratory!* 2019. URL: https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=xitplqMNk_Hc.

[9] Greg Ruetsch. *Optimizing Matrix Transpose in CUDA*. 2009.

[10] Docker Inc. *What is a Contatiner?* 2019. URL: https:/www.docker.com/resources/what-contatiner.

[11] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *ArXiv e-prints* (Dec. 2017). ArXiv: 10.1109 (cs.CV).

[12] kuangliu. *pytorch-cifar*. 2018. URL: https://githb.com/kuangliu/pytorch-cifarl.

[13] Timothy P. Lillicrap et al. "Random feedback weights support learning in deep neural networks". In: *ArXiv e-prints* (Nov. 2014). arXiv: 1411.0247v1 [cs.CV].

[14] Micheal A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[15] Nøkland Arild. "Direct Feedback Alignment Provides Learning in Deep Neural Networks". In: *ArXiv e-prints* (Dec. 2016). ArXiv: 1609.01596 (cs.CV).

[16] NVIDIA. *CUDA 8 PERFORMANCE OVERVIEW*. 2016.

[17] Iacopo Poli. *Custom filters in conv2d backward*. 2018. URL: https://discuss.pytorch.org/t/custom-filters-in-conv2d-backward/20899.

[18] Qianli Liao, Joel Z. Leibo, and Tomaso Poggio. "How Important Is Weight Symmetry in Backpropagation?" In: *ArXiv e-prints* (Feb. 2016). ArXiv: 1510.05067 (cs.CV).

[19] Ranganathan Varun and Natarajan S. "A New Backpropagation Algorithm without Gradient Descent". In: *ArXiv e-prints* (Jan. 2018). ArXiv: 1802.00027 (cs.CV).

[20] Sergey Bartunov et al. "Assessing the Scalability of Biologically-Motivated Deep Learning Algorithms and Architectures". In: *arXiv e-prints* (Nov. 2018). eprint: 1807.04587 (cs.CV).

[21] Sil C. van de Leemput, Jonas Teuwen, and Reshindra Manniesing. "MemCNN: a Framework for Developing Memory Efficient Deep Invertible Networks". In: *Workshop track - ICLR 2018* (2018).

[22] Theodore H. Moskovitz, Ashok Litwin-Kumar, and L.F. Abbott. "Feedback Alignment in deep convolutional networks". In: *ArXiv e-prints* (Dec. 2018). ArXiv: 1812.06488 (cs.CV).

[23] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* (Dec. 2015). IEEE: 1512.03385 (cs.CV).

[24] Will Xiao et al. "Biologically-Plausible Learning Algorithms Can Scale to Large Datasets". In: *ArXiv e-prints* (Dec. 2018). ArXiv: 1811.03567 (cs.CV).

[25] willwx. *sign-simmetry*. 2018. URL: https://github.com/willwx/sign-symmetry.