MACHINE LEARNING PROJECT

MARIUS-CONSTANTIN DINU (E10715010)

Department of Information Engineering
NTUST
Professor Hsing-Kuo Pao

Abstract. Serrà et al. demonstrated in their paper *Overcoming Catastrophic Forgetting with Hard Attention to the Task (HAT)* [1] that they were able to counter catastrophic forgetting by applying layer-wise attention and slightly altering the loss function with an own regularization term. In their experiments they modified the AlexNet [2] architecture by adding attention embeddings and compared their results with other prominent solutions for overcoming catastrophic forgetting, such as *Elastic Weight Consolidation (EWC)* [3] or *Progressive Neural Networks (PNN)* [4] from DeepMind. In this report we will show how to apply the HAT method to a 18-layer ResNet [5] architecture and analyse the effects during training with focus on the performance of residual networks.

1. Introduction

The term catastrophic forgetting or catastrophic inference is used to refer to the issue occurring in traditional connectionist neural network models, when the performance drastically decreases while encountering a continuous learning problem. Connectionist models were not innately designed for extracting abstract associative features and acting as memory systems, which is one of the key requirements to solving sequential problems. In the past their development mainly focused on static data sets, optimizing statistical approximation functions transferring the input features to a pre-defined set of classes or regressive values; often referred to as supervised learning tasks. Since the comeback and outstanding success of neural networks in 2005 our tasks increasingly became more and more complex and we are now seeking the transition from handling statically pre-defined problems to dynamically changing, relational, unbounded problems. Such problems are much harder to stabilize during training and require a lot of effort in finding the right representation to constrain and guide the training progress for proper generalization. This report will explore the solution presented by Serrà et al. including their implementation based on AlexNet and continue their work by applying their solution to ResNet-18, which is another popular network architecture in the field of Deep Learning. First we will analyse their method in detail and then relate to their implementation. Afterwards we show how to transfer their approach to ResNet-18 and analyse the experimental results. In the last section 5 we will provide some insights gained during the development and experimental process.

Date: January 2nd, 2019.
Key words and phrases. Neural Networks, Catastrophic Forgetting, ResNet.

2. Hard Attention to the Task

2.1. **Architecture.** In the paper from Serrà et al. the training procedure is guided to learn identifying features by applying layer-wise attention to the output units of each layer l, $\mathbf{h}_l$, such that

$$\mathbf{h}'_l = \mathbf{a}_l \odot \mathbf{h}_l$$

for each task t respectively. In contrast to regular attention mechanism, where $\mathbf{a}_l^t$ forms a probability distribution over all tasks t, they apply a gated version of a single layer task embedding

$$\mathbf{a}_l^t = \sigma(s\mathbf{e}_l^t),$$

where σ defines an arbitrary gating mechanism and s a positive scaling factor for all layers $l = 1 \dots L-1$ equally and where the last layer L uses a binary hard-coded attention $\mathbf{a}_L^t$ (layer L is similar to a multi-head output).

For their experiments they defined σ to be a sigmoid gating function, such that the gating mechanism arises similar behavior as *inhibitory synapsis*, activating or deactivating different units for each layer. To preserve information over multiple tasks they conditioned the training by adapting the gradient updated, creating a cumulative attention vector by recursively computing the element-wise maximum

$$\mathbf{a}_l^{\leq t} = \max(\mathbf{a}_l^t, \mathbf{a}_l^{\leq t-1}),$$

including the all zero vector $\mathbf{a}_l^{\leq 0}$. This results in applying the reverse of the minimum of the cumulative update for the gradient update $g_{l,i,j}$ at layer l

$$g'_{l,i,j} = [1 - \min(a_{l,i}^{\leq t}, a_{l-1,j}^{\leq t})]g_{l,i,j},$$

with the indices i and j corresponding to the units of the output layers l and input layers $(l-1)$ respectively. These update masks prevent large updates to the weights for important features, which preserve the information of previous tasks.

A desired outcome of the attention vector $\mathbf{a}_l^t$ would be to learn a binary attention mask. However, since the training of the embeddings $\mathbf{e}_l^t$ should remain fully differentiable, they use a scaling factor s to define a pseudo-step function emulating this behavior. During training s is incrementally linearly annealed, controlling the plasticity of the gradient flow and during testing $s = s_{\max}$, with $s_{\max} \gg 1$, defining $\mathbf{a}_l^t$ to behave almost like a step function. The following equation shows the definition of s:

$$s = \frac{1}{s_{\max}} + \left(s_{\max} - \frac{1}{s_{\max}}\right)\frac{b-1}{B-1},$$

whereas $b = 1 \dots B$ is the batch index and B defines the total number of batches in an epoch. If $s_{\max} \to \infty$ then the attention $a_{l,i}^t \to \{0, 1\}$ behaves like a step function, otherwise if $s_{\max} \to 0$ then this results in $a_{l,i}^t \to \frac{1}{2}$ to enable similar gradient flow as in a regular sigmoid function.

2.2. **Embedding Gradient Compensation.** After applying these measures and empirically evaluating the results they came to the conclusions that through the annealing scheme the embeddings showed little changes. Without annealing $s = 1$, after one epoch the cumulative gradient shows a bell shaped curve, spanning the entire sigmoid range. When setting $s = s_{\max}$ results in increasing the magnitude of the cumulative gradient, but in decreasing the spanned value range. To ensure an effective training process both properties are desired; having high magnitude and the full value range of the sigmoid function. For an illustration see figure 1.

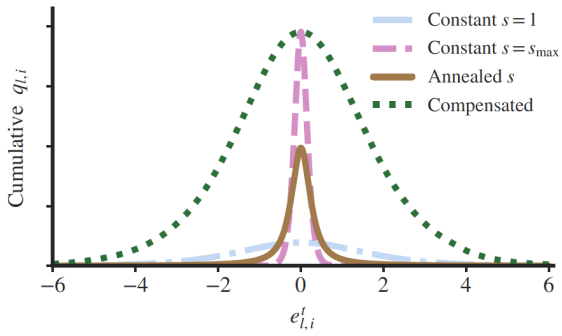

FIGURE 1. Illustration of the annealing effect of s on the gradients q of $\mathbf{e}^t$.

They achieve this by compensating the gradient before the actual update. By dividing the gradient $q_{l,i}$ by the derivative of the annealed gradient and multiply it by the desired compensation

$$q'_{l,i} = \frac{s_{\max}\sigma(e^t_{l,i})[1-\sigma(e^t_{l,i})]}{s\sigma(se^t_{l,i})[1-\sigma(se^t_{l,i})]}q_{l,i},$$

which we can rearranged into

$$q'_{l,i} = \frac{s_{\max}[\cosh(se^t_{l,i})+1]}{s[\cosh(e^t_{l,i})+1]}q_{l,i},$$

they were able to improve their training process. To also ensure numerical stability clamping $|se^t_{l,i}| \leq 50$ proved promising, which is keeping $e^t_{l,i}$ in an active range of the standard sigmoid function $e^t_{l,i} \in [-6, 6]$.

2.3. **Promoting Low Capacity.** To promote efficient usage of the available network capacity they added a regularization term to the loss function. Since $a^t_{l,i} \to 1$ directly determines the units that will be dedicated to task t, they use the set of attention vectors $A^t = \{\mathbf{a}^t_1, \dots \mathbf{a}^t_{L-1}\}$ and promote updates only on a sparse subset of units from the network. To prevent updates on important features of previous tasks the regularization term is taking the cumulative attention of tasks up to $t-1$ into account, defining the cumulative attention as $A^{<t} = \{\mathbf{a}^{<t}_1, \dots \mathbf{a}^{<t}_{L-1}\}$. This brings us to the following loss function:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}, A^t, A^{<t}) = \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) + cR(A^t, A^{<t}),$$

where $c \geq 0$ is the regularization constant, which controls the capacity spent on each task. The regularization term R can be defined as a weighted and normalized L1 regularization over A^t:

$$R(A^t, A^{<t}) = \frac{\sum_{l=1}^{L-1}\sum_{i=1}^{N_l} a^t_{l,i}(1-a^{<t}_{l,i})}{\sum_{l=1}^{L-1}\sum_{i=1}^{N_l} 1-a^{<t}_{l,i}},$$

with N_l defining the number of units in each layer l and where the cumulative attentions over the past tasks $A^{<t}$ represent a weight for the current task. If $a^{<t}_{l,i} \to 1$ then $a^t_{l,i}$ receives a weight close to 0 and vice versa.

3. Implementation

3.1. **AlexNet.** The authors of the original paper based their implementation on a slightly altered version of AlexNet, consisting of three layers of convolutional operations, mixed with dropout and max pooling and three fully connected layers.

Since the dimensions of the input features may vary depending on the image size of the data set, they use a dynamic definition for calculating the kernel size, where an integer division operation **div** dynamically establishes the kernel size of the convolutional layers. Assuming we train on the CIFAR10 or CIFAR100 data set, we have an input size of $32 \times 32 \times 3$ pixels (3 corresponds to the RGB color channels) resulting in a 2D-kernel for the first layer of 4×4 (32 **div** $8 = 4$) and 3×3 (32 **div** $10 = 3$) for the second layer. To provide an overview of the layer operations see the following listing:

(1) Input: $x = 32 \times 32 \times 3$
(2) Convolution: in $= 3$, out $= 64$, $k = 4 \times 4$, $s = 1$, $p = 0$
(3) ReLU
(4) Dropout: $p = 0.2$
(5) Max Pooling: $k = 2 \times 2$
(6) Convolution: in $= 64$, out $= 128$, $k = 3 \times 3$, $s = 1$, $p = 0$
(7) ReLU
(8) Dropout: $p = 0.2$
(9) Max Pooling: $k = 2 \times 2$
(10) Convolution: in $= 128$, out $= 256$, $k = 2 \times 2$, $s = 1$, $p = 0$
(11) ReLU
(12) Dropout: $p = 0.5$
(13) Max Pooling: $k = 2 \times 2$
(14) Fully Connected: in $= 1024$, out $= 2048$
(15) ReLU
(16) Dropout: $p = 0.5$
(17) Fully Connected: in $= 2048$, out $= 2048$
(18) ReLU
(19) Dropout: $p = 0.5$
(20) Fully Connected: in $= 2048$, out $=$ number of classes per task t,

whereas *in* represents the channels input dimensions, *out* the channels output dimensions, k the kernel size, s the strides, p the paddings in the context of convolutional layers and in the context of dropout layers the dropout probability. The output dimension of the network is dependent on the defined number of classes for task t.

To apply the HAT approach they create an embedding of $t \times n_l$ for the number of units n in each trainable layer l for task t; except for the output layer, which is only task dependent. During the forward pass, the embedding $\mathbf{e}_l^t$ is multiplied by the scaling factor s and gated by the activation function σ. The following listing will provide an illustrative example of the first embedding operation:

Listing 1. Applying HAT to convolution layer

```
# definition
self.c1 = torch.nn.Conv2d(ncha, 64, kernel_size=size // 8)
self.ec1 = torch.nn.Embedding(len(self.taskcla), 64)
self.drop1 = torch.nn.Dropout(0.2)
```

```
self.maxpool = torch.nn.MaxPool2d(2)
self.relu = torch.nn.ReLU()
self.gate = torch.nn.Sigmoid()
...
# forward pass
gc1 = self.gate(s * self.ec1(t))
h = self.maxpool(self.drop1(self.relu(self.c1(x))))
h = h * gc1.view(1, -1, 1, 1).expand_as(h)
```

As we see above, the gating can be simply applied without having to change the overall network architecture, such as in *Progressive Neural networks (PNN)* or other approaches. Furthermore, the entire method is fully differentiable and self-contained, requiring no heuristics, such as in PathNet [6] or two-step update procedures as in EWC. During the backward pass the gradient compensation is also applied before updating as illustrated in figure 2, where we have to compute the cumulative attention by iterating over all embedding layers $l = 1 \dots L-1$.

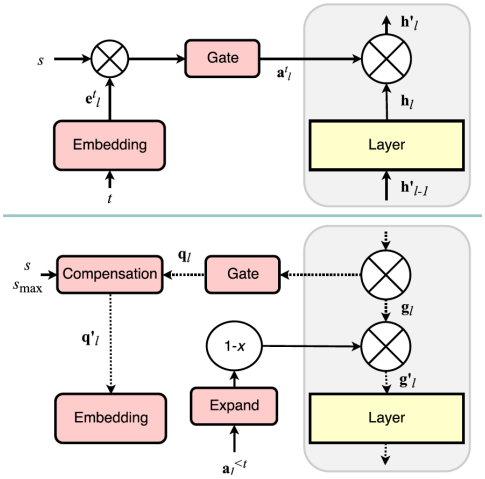

FIGURE 2. Schematic illustration of the forward (top) and backward pass (bottom).

The results will be shown in section 4.

3.2. **ResNet.** This section will show how to apply the HAT approach to the ResNet-18 architecture, since we want to see the effects of HAT in deep residual networks and compare them to the AlexNet architecture. Due to the much greater depth of layers, the developers of ResNet had to add batch normalization layers [7] after each convolutional operation to renormalize the activations and hence stabilize the training procedure. According to the paper from Sergey et al. batch normalization layers prevent the so called internal covariate shift between layers. ResNet also applies skip connections to enable a continuous gradient flow down to the lowest layers during the backward pass, which in exchange also allows them to build deeper models. Furthermore, the ResNet-team emphasizes that these skip connections allows them to learn residual transformations of the original representation function, easing the problem of adapting the layers during training.

A residual block consists of convolutional layers, batch normalization layers, skip connections and non-linearity operations, which are stacked into blocks. To visualize a residual block, figure 3 will provide an overview. In the actual implementation

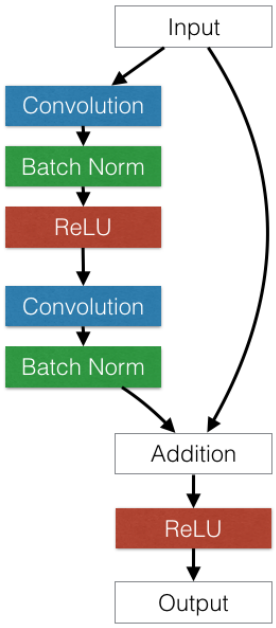

FIGURE 3. Residual block illustration.

for ResNet-18, the skip connections can be optionally applied and two blocks are stacked over each other, whereas the first block has no skip connections and the second block uses skip connections. A stack of two residual blocks, one without skip connections and one with skip connections on top of each other, is defined as being a layer in the current implementation. If a skip connection is applied, then the input of the residual block is mapped to the output via a linear transformation, applied by a convolutional operation without non-linearity operations. Before applying the first residual layer a convolutional layer is applied to transform the input channel size $x = 32 \times 32 \times 3$ into $x' = 32 \times 32 \times 64$, for ease of further computations. The following listing shows the operations of the first layer, which consists of two residual blocks, whereas the second one includes skip connections:

(1) Block-1 Input: $\text{in}_1 = x' = 32 \times 32 \times 64$
(2) Block-1 Convolution: in $= 64$, out $= 64$, $k = 3 \times 3$, $s = 1$, $p = 1$
(3) Block-1 Batch Normalization: $in = 64$ per task t
(4) Block-1 ReLU
(5) Block-1 Convolution: in $= 64$, out $= 64$, $k = 3 \times 3$, $s = 1$, $p = 1$
(6) Block-1 Batch Normalization: $in = 64$ per task t
(7) Block-1 ReLU
(8) Block-1 Output: out_1

---

(9) Block-2 Input: $\text{in}_{\text{2-bn}} = \text{out}_1$

(10) Block-2 Convolution: in = 64, out = 64, $k = 3 \times 3$, $s = 1$, $p = 1$
(11) Block-2 Batch Normalization: $in = 64$ per task t
(12) Block-2 ReLU
(13) Block-2 Convolution: in = 64, out = 64, $k = 3 \times 3$, $s = 1$, $p = 1$
(14) Block-2 Batch Normalization: $in = 64$ per task t
(15) Block-2.1 Output: $\text{out}_{\text{2-bn}}$

---

(16) Block-2 Skip Conn. Input: $\text{in}_{\text{2-skip}} = \text{out}_1$
(17) Block-2 Skip Conn. Convolution: in = 64, out = 64, $k = 3 \times 3$, $s = 1$, $p = 1$
(18) Block-2 Skip Conn. Batch Normalization: $in = 64$ per task t
(19) Block-2.2 Skip Conn. Output: $\text{out}_{\text{2-skip}}$
(20) Block-2 Output: $\text{out}_2 = \text{out}_{\text{2-bn}} + \text{out}_{\text{2-skip}}$

The entire ResNet-18 architecture is now constructed by stacking four such layers together and doubling the number of output channels from one layer to the next.

Finally, to apply HAT to ResNet, we need to create attention embeddings for each trainable layer, except for the output layer; similar to the AlexNet implementation of the previous section.

4. Experiments

To establish a baseline for our experiments we will reference to the original paper from Serrà et al. with the AlexNet implementation. In their paper they evaluated their model on a sequence of multiple tasks, which are formed of different classification data sets. Here we will mainly focus on the CIFAR10 and CIFAR100 data sets and break multiple tasks from them based on a random seed. Since CIFAR10 consists of 10 different image classes and CIFAR100 of 100, and we desire a good task partitioning of both data sets, we divide the CIFAR10 data set into five two-element classes and the CIFAR100 into five 20-element classes. This gives us a set of 10 tasks on which to perform our training and evaluation respectively. Furthermore, to receive a chartable scalar value for the forgetting quantity after each sequentially learned task, they introduced the *forgetting ratio*:

$$\rho^{\tau \leq t} = \frac{A^{\tau \leq t} - A_R^{\tau}}{A_J^{\tau \leq t} - A_R^{\tau}} - 1,$$

where $A^{\tau \leq t}$ is the accuracy measured on task τ after sequentially learning task t, A_R^{τ} is the accuracy of the random stratified classifier using the class information of task τ, $A_J^{\tau \leq t}$ is the accuracy measured on task τ after jointly learning t tasks in a multitask fashion. In other words, in order to obtain $\rho^{\tau \leq t}$ (1) we need to train our models w.r.t. the approach we want to evaluate, (2) train the basic architecture on a joint data set version using SGD and (3) create randomly generated stratified values. To receive a single scalar value per task we then compute:

$$\rho^{\leq t} = \frac{1}{t} \sum_{\tau=1}^{t} \rho^{\tau \leq t}.$$

4.1. **Results.** To receive a good performance estimate of our trained models - ResNet-SGD, ResNet-HAT, ResNet-HAT-BN - we compare them to the AlexNet-SGD, AlexNet-HAT, AlexNet-PNN version from the original paper. When we plot the forgetting ration (figure 4) for all evaluated approaches, we can observe that AlexNet with HAT still seems to perform best. Although it is important to state

that the ResNet-18 model with HAT and batch normalization has not yet been fine-tuned by exhaustive hyperparameter search. Optimizing the scaling factor s for the embeddings or readjusting the regularization constant c might give some improvements. The results may also vary if the training set becomes more complex, since ResNet-18 uses roughly 11.2M parameters and the Serrà et al. version of AlexNet has only 6.7M parameters, allowing the ResNet version to model a higher degree of complexity. Nevertheless, as we can clearly state the ResNet-HAT-BN version in comparison with plain ResNet-SGD shows significant improvements. In addition, the ResNet-HAT without batch normalization results demonstrate how important task dependent batch normalization becomes when applying HAT; which interestingly performs even worse compared to the plain ResNet-SGD version. The AlexNet-PNN version serves as further baseline in comparison to another popular method known as *Progressive Neural Networks*, which per design is innate to forgetting, but shows a higher bias in adapting to the tasks.

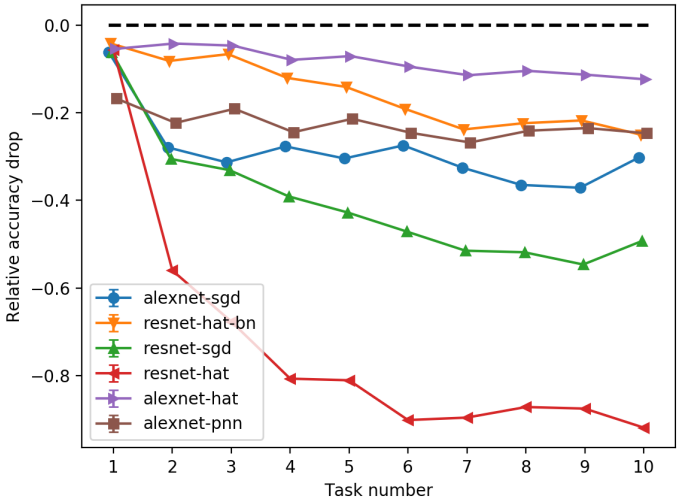

FIGURE 4. resnet-hat (ResNet-18 with HAT but without batch normalization); resnet-hat-bn (ResNet-18 with HAT and batch normalization); resnet-sgd (ResNet-18 with stochastic gradient descent); alexnet-sgd (AlexNet with stochastic gradient descent); alexnet-hat (AlexNet with HAT); alexnet-pnn (AlexNet with Progressive Neural Networks method)

4.2. **Batch Normalization per Task.** Since ResNet-18 uses batch normalization to renormalize the convolutional computations, and each trained batch normalization layer corresponds to the data distribution of the used data set, we need to define a batch normalization layer for each task t respectively. This issue was not obvious from the beginning since the original paper uses the AlexNet architecture, which requires no batch normalization, due to the shallow layer depth. Moreover, the mismatch of batch normalization layers were not that significant when optimizing

ResNet-18 in sequential manner with plain SGD and no HAT enhancements. After naively applying HAT to ResNet-18, the results were quite catastrophic, leading to exploding 35-digit numbers for losses of previously trained tasks, during optimization on new tasks. This also means, that the deeper a network becomes, the more batch normalization layers have to be trained for each task respectively. The current ResNet-18 implementation has $n_{\text{bn}} = 20$ batch normalization layers and $n_{\text{tasks}} = 10$ different tasks, resulting in $n_{\text{bn}} * n_{\text{tasks}} = 200$ trainable batch normalization layers.

4.3. **Performance Gap.** We can clearly see that adding HAT to ResNet-18 reduces catastrophic forgetting between tasks, but what we also observe is that the obtained results for task t are always sightly worse compared to the plain SGD version. This can of course be the effect of the applied regularization term with an un-tuned cost parameter c, being chosen too high, which is enforcing too much sparsity, and hence constraining the learning process too heavily. On the other hand the cumulative attention built over multiple layers used for performing the gradient compensation of the embeddings could also cause this effect, such that, by increasing the layer depth, the performance gap might increase as well. As these questions required intensive experimenting and testing with different ResNet layer depths and hyperparameters search, we leave these questions open for future work.

5. Conclusions

We have shown in this report how HAT helps prevent catastrophic forgetting and provided some key implementation insights. Further, we have shown how to migrate the attention embeddings from AlexNet to a ResNet architecture, and analysed some effects occurring, such as task dependent batch normalization. Although there remain some open questions regarding the effects of attention sparsity with increasing layer depth or ways to apply optimized hyperparameters, we showed that the improvements obtained by using HAT are unquestionable; reducing catastrophic forgetting significantly, regardless of the underlying architecture.

References

1. Joan Serrà, Dídac Surís, Marius Miron, Alexandros Karatzoglou - *Overcoming catastrophic forgetting with hard attention to the task, 2018*, **arXiv:1801.01423**.
2. Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton - *ImageNet Classification with Deep Convolutional Neural Networks, 2012*, Advances in Neural Information Processing Systems 25, page 1097-1105, **Curran Associates, Inc.**.
3. James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, Raia Hadsell - *Overcoming catastrophic forgetting in neural networks, 2017*, **arXiv:1612.00796**.
4. Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, Raia Hadsell - *Progressive Neural Networks, 2016*, **arXiv:1606.04671**.
5. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - *Deep Residual Learning for Image Recognition, 2015*, **arXiv:1512.03385**.
6. Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, Daan Wierstra - *PathNet: Evolution Channels Gradient Descent in Super Neural Networks, 2017*, **arXiv:1701.08734**.
7. Sergey Ioffe, Christian Szegedy - *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015*, **arXiv:1502.03167**.

Department of Information Engineering, NTUST
E-mail address: dinu.marius-constantin@hotmail.com