

RESEARCH ARTICLE

Padding Module: Learning the Padding in Deep Neural Networks

FAHAD ALRASHEEDI¹, XIN ZHONG¹, (Member, IEEE), AND PEI-CHI HUANG¹, (Member, IEEE)

Department of Computer Science, University of Nebraska at Omaha, Omaha, NE 68182, USA

Corresponding author: Xin Zhong (xzhong@unomaha.edu)

This work was supported in part by the National Science Foundation under Grant CIF/SaTC-2104267, and in part by the Nebraska University (NU) Collaboration Initiative Program.

ABSTRACT During the last decades, many studies have been dedicated to improving the performance of neural networks, for example, the network architectures, initialization, and activation. However, investigating the importance and effects of learnable padding methods in deep learning remains relatively open. To mitigate the gap, this paper proposes a novel trainable *Padding Module* that can be placed in a deep learning model. The *Padding Module* can optimize itself without requiring or influencing the model's entire loss function. To train itself, the *Padding Module* constructs a ground truth and a predictor from the inputs by leveraging the underlying structure in the input data for supervision. As a result, the *Padding Module* can learn automatically to pad pixels to the border of its input images or feature maps. The padding contents are realistic extensions to its input data and simultaneously facilitate the deep learning model's downstream task. Experiments have shown that the proposed *Padding Module* outperforms the state-of-the-art competitors and the baseline methods. For example, the *Padding Module* has 1.49% and 0.44% more classification accuracy than the zero padding when tested on the VGG16 and ResNet50.

INDEX TERMS Padding module, deep learning, neural networks, trainable padding.

I. INTRODUCTION

Deep Neural Networks (DNNs) have significantly improved the performance of a wide range of computer vision tasks to the extent of being comparable to or exceeding human-level in many domains [1], such as image classification [2], object recognition [3], and image segmentation [4]. DNNs for computer vision have been iteratively improving in different aspects such as network architecture [5], [6], [7], [8], network initialization [9], [10], optimization [11], [12], and activation [13], [14]. While it is intuitive that the salient foreground of an input image can control the results of a deep learning model [15], [16], researchers have also discovered that the input's borders and corners can dominate the model's performance recently [17], [18], [19]. The study on the importance and effects of image borders remains relatively open, and this paper focuses on a trainable padding method that process image borders for deep learning models.

Padding refers to the technique of adding extra data to the input's borders so that the input's width, height, or depth

can be manipulated. Padding is widely used in Convolutional Neural Networks (CNNs) to alter the output size of a convolutional layer. Without padding, convolutional filters will not process the input's borders and the output size will be reduced. The input size can be maintained with padding; we add an extra border before the convolution so that the original border can be processed [20].

Traditional padding techniques include zero padding, replication padding, and reflection padding. The reflection padding reflects the valid data over the borders; the replication padding uses the borders themselves as padding values; the zero padding specifies the use of zeroes as padding values. The replication and reflection padding methods extend the input with duplicate contents that may not be realistic; hence, they may destroy the original distribution [19]. The zero padding may outweigh the replication and reflection padding methods in terms of speed due to its computational simplicity. The major drawback of the traditional methods is that they are not dynamic. Thus, the padding values are always static and not optimized during the model training in a way that how they could be optimally predicted rationally to the input's borders.

The associate editor coordinating the review of this manuscript and approving it for publication was Jiju Poovancheri¹.

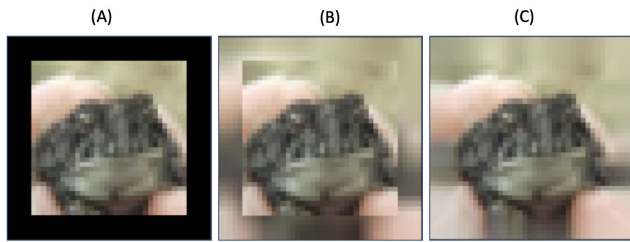


FIGURE 1. Five-pixel padding applied to a CIFAR-10 sampled image using three different padding methods: A) the zero padding, B) the local mean interpolation, and C) the proposed *Padding Module*.

More recently, padding methods have been studied aiming at a more related and realistic extension of the original input [18], [19]. For example, Liu et al. [18] proposed a padding method using partial convolution. Nguyen et al. [19] used a local mean of a sliding window over the input's borders so the local distributions at the borders before and after the padding are consistent. These state-of-the-art padding methods outperformed the traditional padding in several tasks such as image classification, image segmentation, and image style transfer. However, the major disadvantage of the state-of-the-art padding methods is that they are not trainable: the padding contents are still not optimized.

In this paper, we propose a trainable *Padding Module* that can be inserted into selected positions of a deep learning model to learn how to pad its inputs. The *Padding Module* can be trained simultaneously with a deep learning model, but, it is a self-learner in a way that will not require or influence the model's entire loss function. During the training, the *Padding Module* internally constructs a ground truth from the input's actual borders and trains a predictor considering the neighboring areas. The trained *Padding Module* can produce plausible results, as shown in Figure 1. The advantages of our work can be summarized as three-fold:

- The proposed *Padding Module* introduces a trainable method that automatically pads its inputs.
- The *Padding Module* extends its input with realistic new data that are related to the original data.
- The *Padding Module* improves the performance of a downstream task of a deep learning model and outperforms the state-of-the-art competitors, e.g., classification.

The remainder of this paper is organized as follows. In Section II, we review the related work that addressed the padding effects on the neural networks performance and discuss how the current study fills the gap in the related work. Section III discusses our approach for the *Padding Module* followed by evaluation results in Section IV. Finally, Section V concludes with the discussion on evaluation and highlights some of the future work in this sector.

II. LITERATURE REVIEW AND RELATED WORK

Many studies have tried to improve the performance of CNNs models from network architecture [13], [14], [21], [22], different variants of optimization [11], [12], [23],

activations [10], [24], [25], [26], regularization methods [27], [28] and so on. However, little attention has been paid to investigating the padding schemes during the convolution operation. To assist a kernel, i.e., features extractor, in extracting important features during image processing in CNNs, padding layers can be added to visit pixels of the images around the corners more times, and then increase accuracy. The previous padding methods are presented as follows: Section II-A presents the performance improvement of neural networks; Section II-B introduces the improvement of space design; and Section II-C describes our contributions.

A. PERFORMANCE IMPROVEMENT OF NEURAL NETWORKS

Several studies have proposed padding methods to improve the performance of the neural networks [17], [18], [19].

Innamorati et al. [17] addressed the importance of the data at the borders of the input by proposing a convolution layer that dealt with corners and borders separately from the middle part of the image. They specifically designed filters for each corner and border to handle the data at the boundaries, including upper, lower, left, and right borders. The boundary filters used in the convolution were jointly learned with the filter used for the middle part of the image. However, the main issue of this study is that the number of filters used to deal with the boundaries increases linearly with the size of the receptive field.

Also, Nguyen et al. [19] proposed a padding method that could keep the local spatial distribution of the padded area consistent with the original input. The proposed method used the local means of the input at the borders to produce the padding values; they proposed two different variants of the padding method: mean-interpolation and mean-reflection. Both variants used filters with static values, based on the receptive field, in the convolution operation that is supposed to yield the padding values maintaining the same distributions as the original borders. However, the main issue with this method is that they are not learnable.

Liu et al. [18] proposed a padding layer that uses a partial convolution that mainly re-weighted the convolution operation based on the ratio of the number of parameters in the filter to the number of valid data in the sliding window. In other words, they dealt with the padded area as hole areas that need to be in-painted, while the data coming from the original image were seen as non-hole areas. The main issue of this study is that the padding process is not learnable.

B. IMPROVEMENT OF SPACES DESIGN

Also, some studies addressed the importance of the padding and data at the boundaries in the semantic representation learning and converting 360-degree space to 2-dimensional (2-D) space respectively [29], [30], [31].

Cheng et al. [31] showed the importance of the padding method when they converted the 360-degree video to 2-dimensional space. They converted the video to six faces.

Then, they used the reflection padding to connect them to form the 2-D space. The reflection padding naturally connected the faces compared to the zero-padding, which caused discontinuity.

Interesting works were provided by Islam et al. [29], [30] in which they showed the importance of zero padding along with the data at the borders in encoding the absolute position information in the semantic representation learning. They showed that the zero padding and the boundaries drove the CNN models to encode the spatial information that helped the filters where to look for a specific feature; the spatial information was eventually propagated over the whole image.

C. OUR CONTRIBUTIONS

The padding methods and their effects on a CNN model's performance are still open areas for researchers to investigate; hence, it is worth proposing new padding methods that could improve the performance of the CNN models. We propose a novel padding method, *Padding Module*, that could realistically extend the input with related data. It learns how to pad the input by using the inputs' borders as a ground truth and the neighboring areas of the borders as a predictor. Then, it uses a local loss function such as Mean Squared Error (MSE) and updates the filters using the local differentiation of the loss function with respect to the *Padding Module*'s filters. The following section explains the implementation of the *Padding Module*.

III. THE PROPOSED PADDING MODULE

This paper presents the *Padding Module*, a learnable padding method that can pad the input with related and realistic padding, as shown in Figure 1. The *Padding Module* can be used as a substitute for other padding methods in the convolution layer, such as the zero padding, the replication padding, and the reflection padding. This section shows how the padding procedure (Section III-A) and the backpropagation (Section III-B) of the *Padding Module* work.

A. PADDING PROCEDURE

Algorithms 1 and 2, respectively, give an overview of the forward pass and the back-propagation of the *Padding Module*. The *Padding Module* first constructs a ground truth and a predictor from the input (shown in step 1 to step 3 in Algorithm 1 and explained in Sections III-A1 and III-A2). Then, the *Padding Module* uses the filters being learned to produce the actual padding values using the input's borders as a predictor (shown in steps 4 to 13 in Algorithm 1 and explained in Section III-A4). Finally, the *Padding Module* uses the MSE as a loss function to compute the loss value and updates the filters during the model's back-propagation (shown in steps 1 to 2 in Algorithm 2 and explained in Sections III-A3 and III-B).

The *Padding Module* can pad the original input with any padding size, (e.g., one-pixel, two-pixels, etc). Indeed, the padding process in the *Padding Module* is iterative (shown in steps 4 to 13 in Algorithm 1). Assume the required padding

Algorithm 1 Forward Pass

Input: M_c^r , $size$, where r and c are the dimensions of a matrix, and $size$ is the padding size.

Output: $M_c^{r'}$, where $r' = r + 2 \times size$, and $c' = c + 2 \times size$.

```

1:  $T \leftarrow target(M_c^r)$  /* as in Eq.1 */
2:  $N \leftarrow neighbors(M_c^r)$  /* as in Eq.2 */
3:  $P \leftarrow pad_z(pad_r(N))$  /* as in Eq.3 */
4:  $M_c^{r'} \leftarrow M_c^r$  /* initial state for  $M_c^{r'}$  */
5: while  $size \neq 0$  do
6:    $Nout \leftarrow borders(M_c^{r'})$  /* as in Eq.6 */
7:    $Pout \leftarrow pad_z(pad_r(Nout))$  /* as in Eq.3 */
8:    $O \leftarrow f_\theta(Pout)$  /* as in Eq.7 */
9:    $M_c^{r'+2} \leftarrow O^0 // pad_z(M_c^{r'}) // O^1$ 
10:   $M_c^{r'+2} \leftarrow sides((O^2)^T, M_c^{r'+2}, (O^3)^T)$  /* as in Eq.8 */
11:   $M_c^{r'} \leftarrow corners(M_c^{r'+2})$  /* as in Eq.9 */
12:   $size \leftarrow size - 1$ 
13: end while
14: return  $M_c^{r'}$ 

```

Algorithm 2 Back Propagation

Input: $G_c^{r'}$, where c' and r' are the same dimensions of the output of the forward pass in Algorithm 1.

Output: G_c^r , where c and r are the same dimensions of the input of the forward pass in Algorithm 1.

```

1: Compute the local gradients. /* as in Eq.5 */
2: Update the filter weights
3:  $G_c^r \leftarrow strip(G_c^{r'})$  /* as in Eq.10 */
4: return  $G_c^r$ 

```

size is three pixels, the padding process will iterate three times as follows: (1) padding the original input with one-pixel along all the four borders; (2) padding the output of the 1st iteration with one-pixel along all the four borders; and (3) padding the output of the 2nd iteration with one-pixel along all the four borders. Here, to easily explain our method, a simple case of padding process was presented here, e.g., one-pixel padding. Also, the *Padding Module* is assigned filters as many as the number of channels in the input as explained in Section III-A3. Then, we explain the padding process considering a single channel. Here, the same procedure is separately applied to each channel in case of multiple channels.

1) GROUND TRUTH T

The *Padding Module* structures the ground truth T by extracting the input's borders and stacking them upon each other vertically to form a four-row matrix. However, to stack the left and right borders vertically in T , they are transposed from column vectors to row vectors. Formally, given M_c^r as an original input with r and c as the number of rows and columns respectively; henceforth, superscripts and subscripts represent the indexes in the row-wise traversal

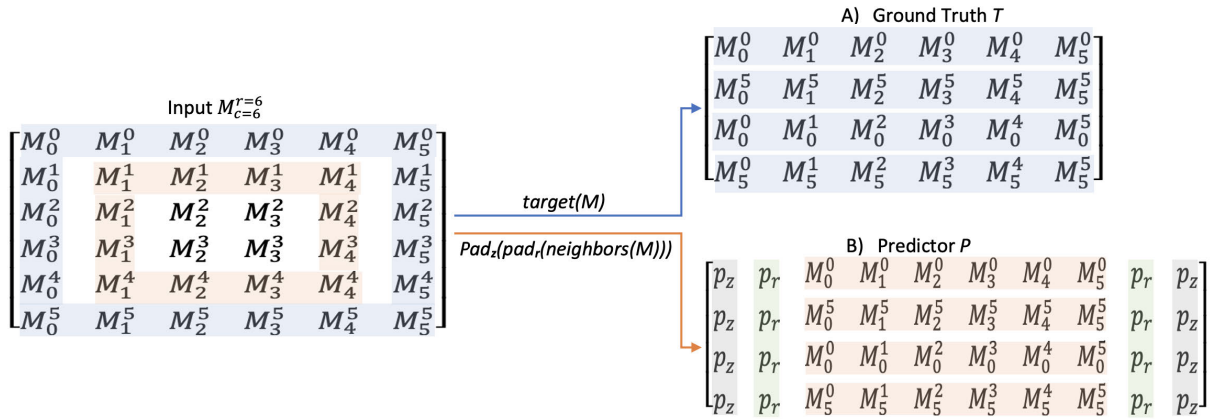


FIGURE 2. An example to illustrate the steps 1-3 in Algorithm 1. On the left: the input M_c^r with size of (6, 6) pixels; the superscripts are the indexes in the row-wise traversal while the subscripts are the indexes in the column-wise traversal of the input. On the right: (A) the ground truth T : a result of applying step 1 in Algorithm 1 which is a stack of the borders where the first row, second row, third row, and last row are the upper, lower, left, and right borders in the input respectively; and (B) the predictor P : a result of applying steps 2 and 3 in Algorithm 1 which is a stack of the neighbors where the first row, second row, third row, and last row are neighbors to the upper, lower, left, and right borders in the input respectively, and the stack is padded at the left and right sides with reflection padding (p_r) and with zero padding (p_z).

and the column-wise traversal of the input respectively. The following is T 's extracting function $target$ of the input M_c^r :

$$T = target(M_c^r) = \begin{bmatrix} M_{[:]}^0 \\ M_{[:]}^{r-1} \\ (M_0^{[:]})^T \\ (M_{c-1}^{[:]})^T \end{bmatrix}, \quad (1)$$

where $M_{[:]}^0$ is the entire row vector in M_c^r at index 0, $M_{[:]}^{r-1}$ is the entire row vector in M_c^r at index $r - 1$, $(M_0^{[:]})^T$ is the transpose of the entire column vector in M_c^r at index 0, and $(M_{c-1}^{[:]})^T$ is the transpose of the entire column vector in M_c^r at index $c - 1$. Figure 2. (A) is an example to visually illustrate how T is constructed where the first row represents the upper border in M_c^r , the second row represents the lower border in M_c^r , the third row represents the left border in M_c^r , and the last row represents the right border in M_c^r .

2) PREDICTOR (P)

To structure the predictor from the original input M_c^r , the *Padding Module* extracts the row vectors that neighbor the upper border and lower border in M_c^r and the transpose of the column vectors that neighbor the left border and right border in M_c^r . Then, the *Padding Module* stacks all the extracted neighbors vertically to form a four-row matrix. Formally, the predictor's (denoted as P) extracting function of M_c^r can be expressed in the following way:

First, the neighbors in M_c^r are selected and denoted as N as follows:

$$N = neighbors(M_c^r) = \begin{bmatrix} M_{[1:c-1]}^1 \\ M_{[1:c-1]}^{r-2} \\ (M_1^{[1:r-1]})^T \\ (M_{c-2}^{[1:r-1]})^T \end{bmatrix}. \quad (2)$$

The slice $[1 : c - 1]$ excludes the data in the row vectors at the borders due to overlapping with the T , whereas the slice $[1 : r - 1]$ excludes the data in the column vectors at the borders due to overlapping with T .

Then, the *Padding Module* pads the structure as follows:

$$P = pad_z(pad_r(N)). \quad (3)$$

First, the $pad_r(\cdot)$ function pads the structure with one pixel of the reflection padding horizontally (the left and right sides); then, with one pixel of the zero padding horizontally using the $pad_z(\cdot)$ function can get the final structure for P .

Each row in P will be used to predict the corresponding row in T . For example, the first row in P will be used to predict the first row in T representing the upper border in the input M_c^r . Figure 2 (B) is an example to visually illustrate how the *Padding Module* constructs the stack of the neighbors (as a predictor) where the right and left sides of the stack are padded with the reflection padding (named as p_r), and the zero padding (named as p_z).

3) FILTERS AND THE LOSS FUNCTION

The *Padding Module* uses as many filters as the channels in the input (*i.e.*, filter per channel). Also, each filter will be a row vector with a size of (1, 3) and a stride of (1, 1); that is because of having each row in P as a predictor for the corresponding row in T . Therefore, to predict T , the *Padding Module* convolutes the filters over P ; it uses its own loss function to optimize the prediction through the local differentiation of the loss function with respect to the filters.

The loss function used by the *Padding Module* is the MSE which computes the squared difference between the ground truth and the predicted value. The following equation is the

MSE's mathematical expression for a single data point:

$$MSE(f_{\theta}(P), T) = \sum_{a=1}^4 \sum_{j=1}^n (\theta^T \cdot P_j^a - T_j^a)^2, \quad (4)$$

where f is the convolutional operation parameterized by θ , P and T are the predictor and the ground truth extracted from the original input M_c^r , a represents the indexes for rows in the four-row matrices P and T , and j represents the indexes for both the slide windows and columns in P and T respectively. Hence, P_j^a is the j th slide window in the row indexed at a in P , and T_j^a is the corresponding value in T indexed at the a th row and j th column.

The local differentiation of the *Padding Module's* loss function and the filters' updates are achieved during the model's back-propagation; these local gradients are not propagated to the previous layer. Besides that, the *Padding Module* facilitates the back-propagation of the model's loss function going through it to the previous layer as explained in Section III-B. The following is the mathematical expression for the local gradients (*Padding Module's* loss function gradients with respect to a single filter for a single data point):

$$\frac{\varphi}{\varphi_{\theta_m}} MSE(f_{\theta}(P), T) = 2 \sum_{a=1}^4 \sum_{j=1}^n (\theta^T \cdot P_j^a - T_j^a) x_m, \quad (5)$$

where x_m is a single feature in the P_j^a slide window which was multiplied by the corresponding weight, namely θ_m , in the θ during the convolution.

4) PADDING PROCESS

The procedures in Sections III-A1, III-A2, and III-A3 are used to guide the *Padding Module* on learning how to predict the borders of the *original* input, M_c^r , based on the neighboring areas to the borders, and then the *Padding Module* can optimize its filters.

However, the padding process is shown in steps 4 to 13 in Algorithm 1; it uses the borders of the input, $M_c^{r'}$, as the predictor. In detail, the padding process iterates until the original input is padded with the required padding size. Hence, the original input M_c^r is assigned to $M_c^{r'}$ as an initial state in step 4 before the padding loop starts. Then, each iteration pads the input, $M_c^{r'}$, with one-pixel, and outputs a new $M_c^{r'}$ which will be used for the next iteration and so forth. The dimensions of an iteration's output, $M_c^{r'}$ in step 11, are two-pixel larger than the dimensions of that iteration's input, $M_c^{r'}$ in step 6.

Minutely, constructing the predictor in the padding process is similar to the way that constructs P in Section III-A2 with small modifications. To distinguish the notions of *neighbors*, N , and P , in Section III-A2, *borders*, $Nout$, and $Pout$ are denoted for the extracting function, the function's output, and the predictor, respectively. The following is the mathematical

expression for the extracting function *borders*:

$$Nout = borders(M_c^{r'}) = \begin{bmatrix} M_{[:]}^0 \\ M_{[:]}^{r'-1} \\ (M_0^{[:])}^T \\ (M_{c-1}^{[:])}^T \end{bmatrix}, \quad (6)$$

where $M_{[:]}^0$ and $M_{[:]}^{r'-1}$ mean extracting the entire upper and lower borders respectively. Whereas, $(M_0^{[:])}^T$ and $(M_{c-1}^{[:])}^T$ mean extracting the transpose of the entire left and right borders respectively. Then, the *Padding Module* pads the output $Nout$ using Equation 3 to get the final structure for $Pout$.

Consequently, convoluting the filters over the $Pout$ will produce the padding values for the iteration's input. The output can be expressed as follows:

$$O = f_{\theta}(Pout), \quad (7)$$

where f is the convolutional operation parameterized by θ , $Pout$ is the predictor, and the O is the output and comes as a matrix of four rows. Each row represents the padding values for the corresponding area in the iteration's input, $M_c^{r'}$, as follows: the first row (O^0), the second row (O^1), the third row (O^2), and the last row (O^3) represent the padding values for the upper, the lower, the left, and the right areas in the input respectively.

Then, the steps from 9 to 11 are how the produced padding values stick around the input $M_c^{r'}$. First, in step 9, the vertical concatenation operator $//$ is used to concatenate the first row (O^0) with $M_c^{r'}$, and then concatenates the resulted matrix with the second row (O^1). However, the rows from O are two-pixel wider than the rows of $M_c^{r'}$; therefore, to match the dimensions of these operands, the *Padding Module* uses $pad_z(\cdot)$ to pad the $M_c^{r'}$ horizontally with one pixel of the zero padding before the concatenation process. Hence, the output's dimensions in step 9, denoted as $M_c^{r'+2}$, are two-pixel larger than the input $M_c^{r'}$. Finally, the algorithm uses *sides* function which can be formally expressed as the following:

$$sides((O^2)^T, M_c^{r'+2}, (O^3)^T). \quad (8)$$

This function does not change the dimensions; however, it adds respectively the transpose of the third row (O^2) and last row (O^3) to the left and right columns of $M_c^{r'+2}$, the concatenated matrix with zero values at the left and right columns unless the corners already assigned values from the concatenation process. To resolve the double-count problem at the corners, the *Padding Module* takes the average of added values in the corners by dividing each corner by 2; this averaging function is step 12 in Algorithm 1:

$$M_c^{r'} = corners(M_c^{r'+2}). \quad (9)$$

Lastly, as mentioned early in this section that the dimensions of the iteration's output are two-pixel larger than the iteration's input. Hence, the output $M_c^{r'}$, in Equation 9, has

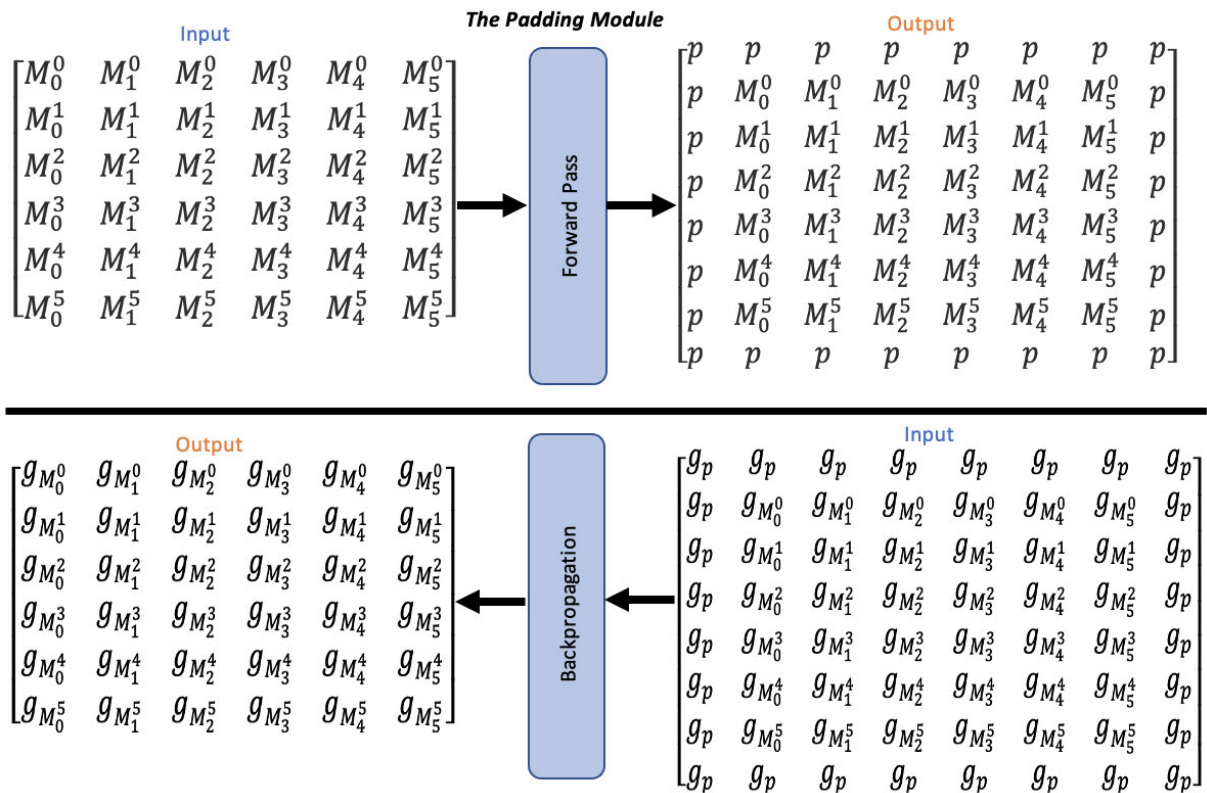


FIGURE 3. An example to illustrate the back propagation in Algorithm 2. On the top: the input to the *Padding Module* is of size (6, 6), the *Padding Module* uses one-pixel padding and produces an output of size (8, 8) where the borders p is the computed padding values. On the bottom: the back-propagation of the received gradients which is of size (8, 8) where the borders are gradients for the padding values g_p ; the *Padding Module* strips out g_p from the received gradients, and sends the remaining to the previous layer. The g stands for gradient; for example $g_{M_0^0}$ is the gradient for the pixel at index [0, 0] in the input of the Forward Pass.

dimensions r' and c' that are updated with the dimensions of $M_{c'+2}^{r'+2}$, namely $r' + 2$ and $c' + 2$ respectively.

B. BACK-PROPAGATION

As seen in Section III-A3, the *Padding Module* is not optimized based on the model’s main loss function; therefore, the model does not compute the gradients of its loss function with respect to the filters of the *Padding Module*. However, during the model’s backpropagation, the *Padding Module* achieves two key points as follows:

- 1) As shown in step 1 in Algorithm 2, the *Padding Module* optimizes its filters through computing the local gradients for its loss function with respect to the filters as explained in Section III-A3.
- 2) The process also receives $G_{c'}^{r'}$ which are the gradients of the model’s loss function with respect to the *Padding Module*’s output, the original input M_c^r after being padded. Therefore, the *Padding Module* strips out the gradients from $G_{c'}^{r'}$ that represent the gradients for the padded areas in the *Padding Module*’s output; the stripping-out process is step 3 in Algorithm 2, and formally expressed as follows:

$$G_c^r = \text{strip}(G_{c'}^{r'}). \tag{10}$$

Then, the *Padding Module* back-propagates to the previous layer the G_c^r , representing the gradients for the previous layer’s output. Figure 3 is an example to visually illustrate how the back-propagation process in the *Padding Module* is achieved.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

This section shows the design of the training and testing experiments on our *Padding Module* applied to a downstream task, *i.e.*, image classification. The experimental setup is presented in Sections IV-A. The quantitative and qualitative results are described in Section IV-B and IV-C respectively.

A. EXPERIMENT SETUP

The study used the premium service from Google Colaboratory where a GPU of Tesla T4 was assigned. The experiments and comparisons were conducted on the CIFAR-10 dataset for a classification task [32]. The CIFAR-10 dataset includes a training dataset of 50,000 images and a test dataset of 10,000 images. The images are of shape (32, 32, 3), distributed equally to ten classes of airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The *Padding Module* was applied to different networks namely: VGG16 [8] and ResNet50V2 [33]; to make the deeper layers

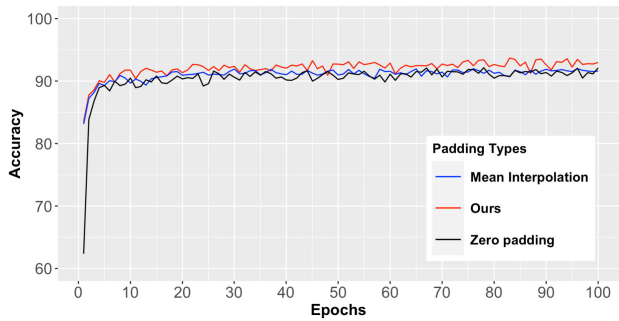


FIGURE 4. The comparison of three different padding methods on the test images: zero padding, mean interpolation padding, and the *Padding Module* when applied to the VGG16 model.

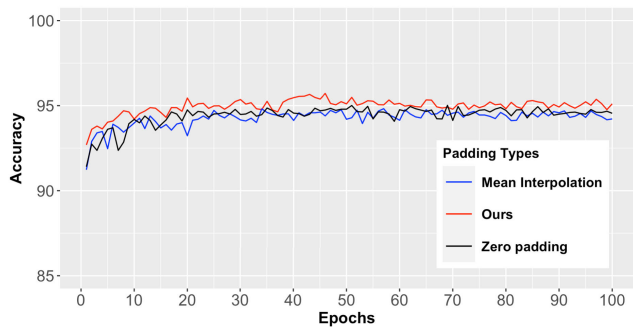


FIGURE 5. The comparison of three different padding methods on the test images: zero padding, mean interpolation padding, and the *Padding Module* when applied to the ResNet50V2.

in these networks carry out a valid convolution, the images were resized to (64, 64, 3) and (224, 224, 3) for the VGG16 and the ResNet50V2, respectively.

The VGG16 is a vanilla-based architecture where the network shape is wider at the beginning of the network and narrowed down as going deep in the network. The pre-trained VGG16 was obtained from the keras¹ without the top layers (the last three dense layers including the original softmax layer). Then, we added two fully-connected layers each with 512 neurons and followed by a dropout layer. On the other hand, the ResNet50V2 is made up of blocks where each block sends the block's input through the block itself, and also uses a skip connection to directly add the block's input to the output of the input's flow coming through the block. The process is known as the identity function that could help deep layers to improve the model's accuracy. ResNet50V2 is a modified version of the ResNet50 [13]. The modification mainly is in the arrangement of the block layers; batch normalization [11] and ReLU activation [34] are applied to the data flow before the convolutional layer in the block. These changes enabled the ResNet50V2 to outperform the ResNet50 on the image classification task. The ResNet50V2 was downloaded from the keras² without the top layer (the last dense layer which is the original softmax layer). Then, two fully-connected layers with 1024 and 512 neurons were added.

¹VGG16 from the keras: <https://keras.io/api/applications/vgg>

²ResNet50V2 from the kera: <https://keras.io/api/applications/resnet>

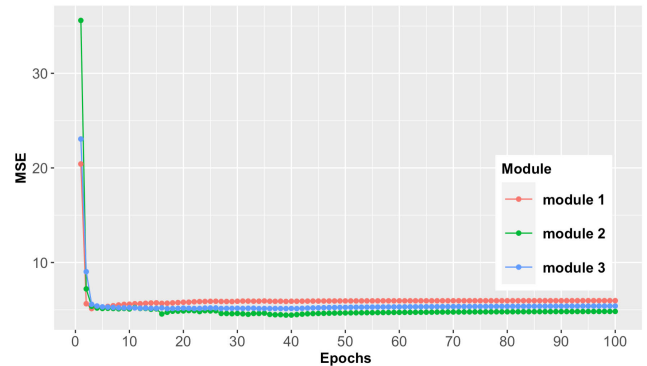


FIGURE 6. MSEs for three *Padding Modules* placed at different positions in the VGG16: 1) *module 1*: at the beginning, 2) *module 2*: at the middle, and 3) *module 3*: at the end.

Moreover, we added a softmax layer with ten outputs for both models of VGG16 and ResNet50V2, and then used the Adam optimizer [12] for the back-propagation of the gradients. Finally, the *Padding Module* was used before every convolutional layer in the VGG16; whereas, we replaced every zero padding layer in the ResNet50V2 with the *Padding Module*.

B. QUANTITATIVE RESULTS

Section IV-B1 compares the proposed *Padding Module* and state-of-art padding solutions by performing the image classification task, and then Section IV-B2 discusses an ablation study based on our solution.

1) IMAGE CLASSIFICATION TASK

We considered the zero padding method as a baseline to compare the *Padding Module* with. Moreover, we used the mean interpolation padding method [19] as the state-of-art since it outperformed the partial convolution padding method [18] in the image classification task [19]. The main goal of this study, which aligns with the literature, is to investigate the padding effect on the accuracy of DNN models. Therefore, the accuracy is used as a comparison metric between the performance of the *Padding Module* and the benchmark. The accuracy is the percentage of correctly classified images over the total number of images in the dataset.

Each model was trained with 100 epochs using the training dataset, and tested in each epoch using the test dataset. In Figure 4, the *Padding Module* outperforms both the baseline method and the mean interpolation padding method when using the VGG16; also, we found that the baseline is comparable to the mean interpolation method. As for the ResNet50, the *Padding Module* also outperforms the other two paddings as shown in Figure 5. We also noticed that the baseline method is comparable to the mean interpolation method. Moreover, Table 1 summarizes the average of the last five epochs for the three different padding methods and the margin between the highest and the second-highest accuracies for the two models. Also, the Area Under the

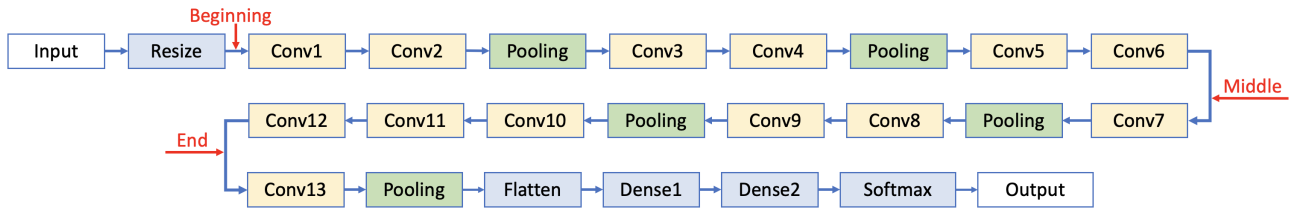


FIGURE 7. The selected positions in the VGG16 for the *Padding Module*: at the beginning, middle, and the end.

TABLE 1. The ACC is the average accuracy of the last five epochs for three different padding methods used in VGG16 and ResNet50V2. The AUC is the Area Under the Curve of the ROC considering the highest-accuracy epoch for each padding method. The margin shows the difference between the highest and second-highest values in both ACC and AUC for the two models.

| Model | VGG16 | | ResNet50V2 | |
|--------------------|-------------|-------------|-------------|-------------|
| | AUC | ACC | AUC | ACC |
| Padding Module | 98.18 | 92.92 | 99.39 | 95.08 |
| Zero Padding | 97.87 | 91.43 | 99.31 | 94.64 |
| Mean Interpolation | 97.94 | 91.69 | 99.25 | 94.44 |
| Margin | 0.24 | 1.23 | 0.08 | 0.44 |

Curve, AUC, of the ROC is used as another perspective to show the superiority of the *Padding Module* over the other padding methods, as shown in Table 1.

As mentioned, the study investigated the effects of the *Padding Module* on the accuracy of DNN models. Importantly, we showed that the related and realistic padding could improve the accuracy of DNN models; therefore, the *Padding Module* was able to produce such padding by minimizing the MSE in Algorithms 1 and 2. Figure 6 illustrates MSEs for three cases of the *Padding Module* applied in different places (at the beginning, in the middle, and at the end) in the VGG16; it is evident that the MSEs significantly decreased after only two epochs and then stayed flat till the end of the experiment for the three cases.

One natural drawback of the current *Padding Module* was the extra running time caused by constructing the data structures and optimizing the filters. Table 2 shows that VGG16 and ResNet50V2, on average, doubled the epoch’s time when applying the *Padding Module* (i.e., placing the *Padding Module* before every convolutional layer in the VGG16 and replacing every zero padding layer in the ResNet50V2 with the *Padding Module*). On the other side, the accuracy for the VGG16 and ResNet50V2, respectively, gained margins of 1.49% and 0.44% when applying the *Padding Module* compared to the zero padding (no *Padding Module*). One remedy to lessen the running time problem may be to stop training the *Padding Module* when it significantly decreases the MSE after the first two epochs. However, improving the current *Padding Module* including the time complexity can be a further direction for future research.

2) ABLATION STUDY

The experiments in this section were conducted as an ablation study where the *Padding Module* was empirically placed

TABLE 2. On average, the running time doubles for one epoch when applying the *Padding Module* to the VGG16 and ResNet50V2 compared to the case of the zero padding (no *Padding Module* applied). Times are shown in a minute-scale. The margin is the accuracy difference between the case of applying the *Padding Module* and the zero padding.

| Model | Zero Padding | | Padding Module | | Margin |
|----------|--------------|------|----------------|------|-------------|
| | Accuracy | Time | Accuracy | Time | |
| VGG16 | 91.43 | 2 | 92.92 | 4 | 1.49 |
| ResNet50 | 94.64 | 5 | 95.08 | 9 | 0.44 |

TABLE 3. Placing the *Padding Module* at different positions in the VGG16: 1) at the beginning 2) at the middle 3) at the end 4) combination of beginning, middle, end 5) before every convolutional layer 6) VGG16 with no *Padding Module* (zero padding instead).

| No | Different places in the VGG16 | Accuracy |
|----|-------------------------------------|-------------|
| 1 | At the beginning | 91.98 |
| 2 | At the middle | 91.8 |
| 3 | At the end | 91.97 |
| 4 | Combination of 1, 2, and 3 together | 92.18 |
| 5 | All positions | 92.8 |
| 6 | VGG16 with no <i>Padding Module</i> | 91.4 |

at different positions in the VGG16 model, as shown in Figure 7: at the beginning of the model, in the middle, at the end, and the combination of all the three places together. We also compared the four scenarios with two other scenarios: (1) where the *Padding Module* was placed in all positions (before each convolutional layer) in the model; and (2) where the *Padding Module* was not used but the zero padding was used instead. We ran each scenario 100 epochs using the training dataset for training and the test dataset for evaluation, and averaged the test accuracies of the last five epochs for each scenario; Table 3 illustrates the summary of the comparison of the models. We noticed that using a single *Padding Module* with the shallow layers outperformed the case of using it with the deep layers. Also, the combination scenario showed a superiority over the scenario of a single *Padding Module*. However, the best performance was when the *Padding Module* applied in the scenario of all positions. Finally, all the scenarios of applying the *Padding Module* outperformed the scenario of the model with no *Padding Module*.

C. QUALITATIVE RESULTS

Different padding sizes, such as one-pixel, three-pixel, and five-pixel, were used to illustrate how the *Padding Module* can extend the input with related and realistic extensions.

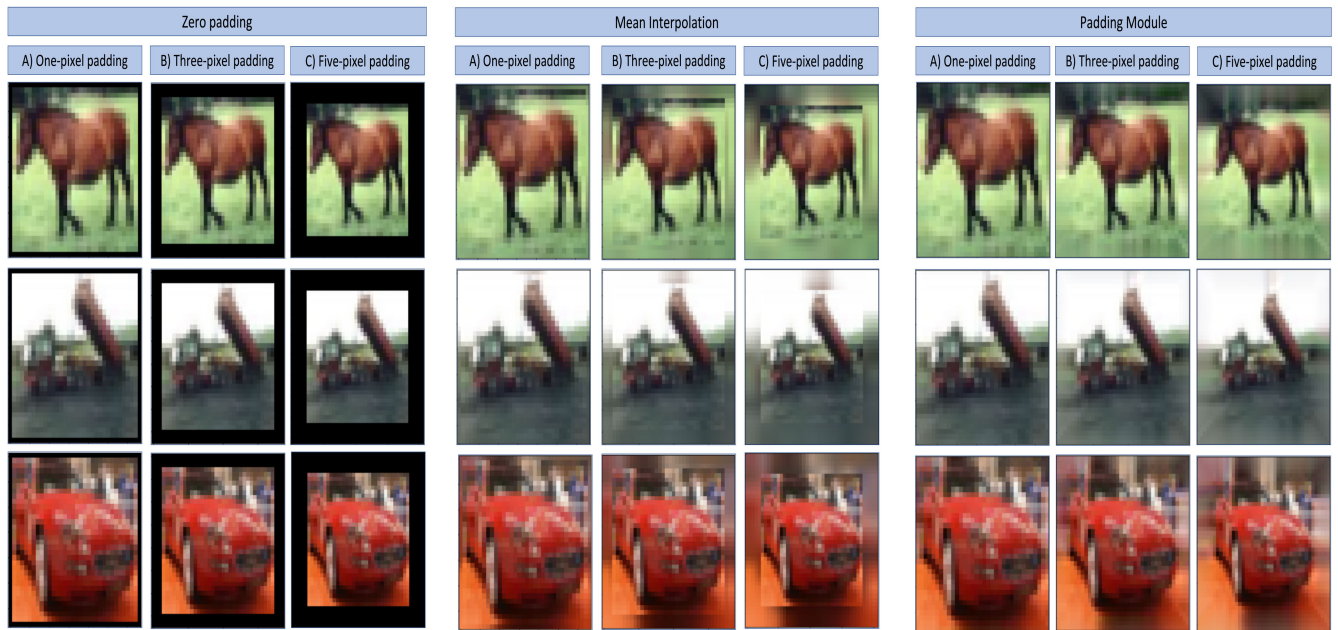


FIGURE 8. Three images sampled from CIFAR-10 and padded by different padding methods: zero-padding, mean interpolation, and the *Padding Module*. Each padding method uses three different padding sizes: A) one-pixel, B) three-pixel, C) five-pixel.

Also, we compared these different padding sizes with the other two methods, namely the zero padding and the mean interpolation padding. As shown in Figure 8, the *Padding Module* can learn how to pad the input with related data and natural extension; this finding becomes more evident as the padding size increases.

V. FUTURE RESEARCH DIRECTIONS AND CONCLUSION

This paper proposed a novel padding method: *Padding Module*; that can learn how to pad an input from the input's borders; hence, the input can be realistically extended with related data. The *Padding Module* is a self-learning of its weights. To train itself, the *Padding Module* constructs a ground truth and a predictor from the inputs by leveraging the underlying structure in the input data for supervision. The *Padding Module* uses convolutional operation over the predictor to produce a predicted value that is, in turn, compared with the ground truth. The *Padding Module* uses a local loss function, independent from the model's main loss function, to minimize the difference between the predicted value and the ground truth. Therefore, the *Padding Module* updates its convolutional filters locally during the model's back-propagation. Besides that, the *Padding Module* back-propagates the model's gradients with respect to the *Padding Module*'s output after stripping out the gradients for the padded areas to the previous layer.

The experimental results showed that the *Padding Module* outperformed the zero-padding and the state-of-art padding in the image classification task. In the ablation study, we also observed that using a single *Padding Module* with the shallow layers improved the performance slightly better than using

it with the deep layers in the VGG16 network. On the other hand, using three of the *Padding Module* placed in different positions (at the beginning, at the middle, and at the end) in the VGG16 outperformed the scenario of a single *Padding Module*. Moreover, placing the *Padding Module* in all positions (before every convolutional layer) in the VGG16 outperformed all other scenarios as shown in Table 3.

Our experiments applied the *Padding Module* to the two well-known networks: VGG16 and ResNet50, for the image classification task. The VGG16 and ResNet50 networks were chosen to represent small and large networks, respectively. They, also, were used by the literature; hence, we used them to compare the *Padding Module* with the previous work. Although two different networks are only used in one task, we shall extend the *Padding Module* to improve such networks in different tasks, including object detection, style transfer, and image inpainting. We leave investigating the *Padding Module* in a wide range of tasks for future research.

Also, the *Padding Module* learned how to pad the input independently of the model's loss function. However, it is possible to optimize the *Padding Module*'s filters based on optimizing the model's main loss function; this approach will be entirely different. Hence, one research direction may be to implement a padding method that can optimize its padding filters based on the model's main loss function.

REFERENCES

- [1] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 1625–1634.

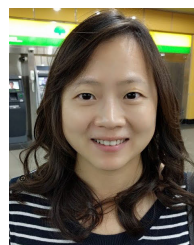
- [2] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convolutional neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 558–567.
- [3] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7310–7311.
- [4] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis.*, Oct. 2017, pp. 2961–2969.
- [5] J. Kim, H. Zeng, D. Ghadiyaram, S. Lee, L. Zhang, and A. C. Bovik, "Deep convolutional neural models for picture-quality prediction: Challenges and solutions to data-driven image quality assessment," *IEEE Signal Process. Mag.*, vol. 34, no. 6, pp. 130–141, Nov. 2017.
- [6] Y. Li, C. Fang, J. Yang, Z. Wang, X. Lu, and M.-H. Yang, "Universal style transfer via feature transforms," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.
- [7] A.-D. Nguyen, S. Choi, W. Kim, and S. Lee, "A simple way of multimodal and arbitrary style transfer," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 1752–1756.
- [8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
- [9] D. Arpit, V. Campos, and Y. Bengio, "How to initialize your network? Robust initialization for weightNorm & ResNets," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–10.
- [10] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.
- [11] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 448–456.
- [12] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
- [13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
- [14] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 4700–4708.
- [15] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," 2013, *arXiv:1312.6034*.
- [16] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2921–2929.
- [17] C. Innamorati, T. Ritschel, T. Weyrich, and N. J. Mitra, "Learning on the edge: Explicit boundary handling in CNNs," 2018, *arXiv:1805.03106*.
- [18] G. Liu, K. J. Shih, T.-C. Wang, F. A. Reda, K. Sapra, Z. Yu, A. Tao, and B. Catanzaro, "Partial convolution based padding," 2018, *arXiv:1811.11718*.
- [19] A.-D. Nguyen, S. Choi, W. Kim, S. Ahn, J. Kim, and S. Lee, "Distribution padding in convolutional neural networks," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2019, pp. 4275–4279.
- [20] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *Proc. Int. Conf. Eng. Technol. (ICET)*, Aug. 2017, pp. 1–6.
- [21] T. Takikawa, D. Acuna, V. Jampani, and S. Fidler, "Gated-SCNN: Gated shape CNNs for semantic segmentation," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 5229–5238.
- [22] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2020, pp. 10428–10436.
- [23] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," 2019, *arXiv:1908.03265*.
- [24] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. 30th Int. Conf. Mach. Learn.*, Atlanta, GA, USA, vol. 30, no. 1, 2013, pp. 1–6.
- [25] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017, *arXiv:1710.05941*.
- [26] M. Adriana Mercioni and S. Holban, "P-swish: Activation function with learnable parameters based on swish activation function in deep learning," in *Proc. Int. Symp. Electron. Telecommun. (ISETC)*, Nov. 2020, pp. 1–4.
- [27] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 648–656.
- [28] G. Larsson, M. Maire, and G. Shakhnarovich, "FractalNet: Ultra-deep neural networks without residuals," 2016, *arXiv:1605.07648*.
- [29] M. A. Islam, S. Jia, and N. D. B. Bruce, "How much position information do convolutional neural networks encode?" 2020, *arXiv:2001.08248*.
- [30] M. A. Islam, M. Kowal, S. Jia, K. G. Derpanis, and N. D. B. Bruce, "Position, padding and predictions: A deeper look at position information in CNNs," 2021, *arXiv:2101.12322*.
- [31] H.-T. Cheng, C.-H. Chao, J.-D. Dong, H.-K. Wen, T.-L. Liu, and M. Sun, "Cube padding for weakly-supervised saliency prediction in 360° videos," 2018, *arXiv:1806.01320*.
- [32] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, Dept. Comput. Sci., Univ. Toronto, Toronto, ON, Canada, 2009.
- [33] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Proc. Eur. Conf. Comput. Vis.* Amsterdam, The Netherlands: Springer, 2016, pp. 630–645.
- [34] A. F. Agarap, "Deep learning using rectified linear units (ReLU)," 2018, *arXiv:1803.08375*.



FAHAD ALRASHEEDI received the M.E. degree in information assurance from the University of Colorado at Colorado Springs, in 2015. He is currently pursuing the Ph.D. degree in artificial intelligence under the supervision of Dr. Xin Zhong with the University of Nebraska at Omaha. His research interests include computer vision, deep learning, and adversarial examples.



XIN ZHONG (Member, IEEE) received the Ph.D. degree from the New Jersey Institute of Technology, Newark, NJ, USA, in 2018. He is currently an Assistant Professor with the Department of Computer Science, University of Nebraska Omaha. His research interests include digital image processing and analysis, computer vision, pattern recognition, computational intelligence, machine learning, deep learning, and image watermarking.



PEI-CHI HUANG (Member, IEEE) received the Ph.D. degree from The University of Texas at Austin, TX, USA, in 2017. She is currently an Assistant Professor with the Robotics, Networking, Artificial Intelligence (R.N.A.) Laboratory, Department of Computer Science, University of Nebraska Omaha. Her research interests include cyber-physical systems, machine learning, robotics, real-time computing, and scheduling algorithms.

...