Expanding the Receptive Field for MNIST through Deformable Convolution

Dwiref Oza $\{dso2119\}$
Columbia University

***Abstract*—CNNs are inherently at a disadvantage when tackling generalizability over unseen data, especially when example images have geometric variations such as scaling and translation. This project implements deformable convolution in Keras/TensorFlow to mitigate this shortfall, with experiments testing model performance of a CNN and miniature ResNet with and without deformable convolution for the digit classification task on the MNIST dataset. The results indicate that for the CNN, introducing deformable convolution improves test accuracy for scaled and translated images of digits, while results from the miniature ResNet remain inconclusive, mostly due to the over-simplified nature of the ResNet architecture and introduction of an unoptimized implementation of deformable convolution.**

I. Introduction

Convolutional Neural Networks (CNNs) are inherently blind to large and unknown geometric transformations, affecting generalizability over unseen data. The most common strategy to mitigate this is data augmentation to learn geometric transformations. For image classification, segmentation and object detection tasks that use fully convolutional networks, being able to dynamically adjust receptive field is a very desirable property. Dai et. al [1] introduce deformable convolution, which can be swapped for standard convolution into existing networks easily. This project will aim to compare the performance of CNNs using deformable convolution in image classification. Since this is an investigative study, the MNIST dataset is a good candidate for this task.

Some of the key objectives of this project are:

- To write a TensorFlow module for deformable convolution that can easily replace or work with TensorFlow's 2D convolution function call.
- To test generalizability over geometric transformations through a simple CNN for MNIST digit classification trained with and without data augmentation.
- To compare test performance of either case against the performance of a CNN model using deformable convolution.

The first objective is met by writing a forward pass that ensures offsetting of matrix elements prior to convolution, such that an offset matrix can be passed to the Conv2D method. The second objective aims to establish a baseline model of performance in image classification when the unseen data has scaling or translation. The final objective aims to establish the improvement in digit classification over scaled and translated test images when utilizing deformable convolution in the model architecture.

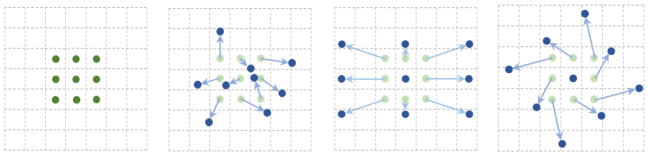


Fig. 1: (a) The green points represent the regular sampling grid for convolution, (b) shows deformed sampling locations with augmented offsets (blue arrows), a(c) and (d) are special cases of (b) with anisotropic scaling and rotation respectively.

II. Summary of the Original Paper

Dai et. al introduce two modules that augment the ability of a CNN to model transformations like scaling or translation. These are deformable convolution and deformable Region-of-Interest (RoI) pooling. Both concepts focus on augmenting the spatial sampling locations through offsets which are learned from the target tasks, without any supervisory overhead.

A. Methodology of the original paper

The original paper defines deformable convolution along with pointers on how to integrate it with popular model architectures such as ResNet, Inception, RPN and Faster R-CNN. The authors provide a fairly detailed comparison against earlier efforts and contemporary methods that aim to make models learn from spatial variance. Finally, the authors also compare the performance of the aforementioned ConvNets when implemented with and without deformable convolution on the COCO dataset. Evaluation of performance against atrous convolution, which is a closely related method, is also provided for the ResNet-101 architecture. A runtime comparison is also provided for the forward pass to illustrate the relatively low computational time cost of implementing deformable convolution.

B. Key results of the original paper

The authors demonstrate that deformable convolution performs measurably better than atrous convolution, and that a conmbination of RoI pooling and deformable convolution noticably improve the accuracy seen in the performance of the Faster R-CNN and R-FCN.

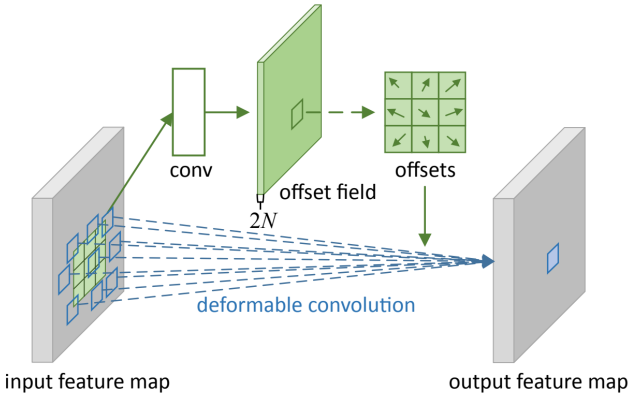


Fig. 2: An illustration of 3×3 deformable convolution

III. Deformable Convolution

2D convolution involves first sampling through a regular grid R over the input feature map, followed by summation of sampled values that are weighted by a factor w. The grid R defines the receptive field, its size and dilation. Therefore,

$$R = \{(-1,-1),(-1,0), \quad . \quad . \quad .,(0,1),(1,1)\}$$

defines a 3×3 kernel with dilation 1. For every location p_0 on the output feature map y,

$$y(p_0) = \sum_{p_n \in R} w(p_n) \cdot x(p_0 + p_n),$$

where p_n indicates the locaions in R.

On the other hand in deformable convolution, R is augmented with offset values as defined by $\{\Delta p_n | n = 1, ...N\}$ where $N = |R|$. Thus, the original expression for the output feature map now becomes:

$$y(p_0) = \sum_{p_n \in R} w(p_n) \cdot x(p_0 + p_n + \Delta p_n)$$

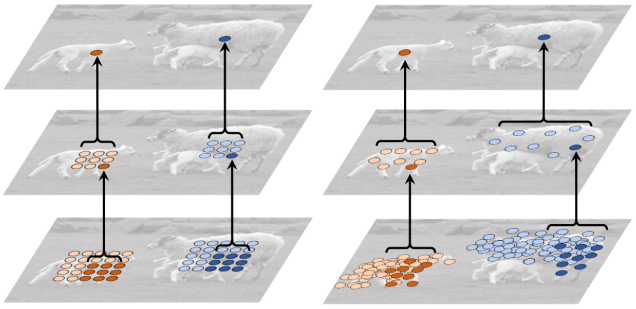
The sampling is thus irregular and at the indices given by $p_n + \Delta p_n$. The offset Δp_n is typiically a fraction and thus bilinear interpolation is required to define a discrete offset, resulting in:

$$x(p) = \sum_q G(q,p) \cdot x(q),$$

where p denotes the arbitrary location $p = (P_0 + p_n + \Delta p_n)$. q enumerates the integral spatial points in the feature map x, while G is simply the kernel for bilinear interpolation. Simply put, the offsets are obtained by applying a convolutional layer over the original input. The convolution kernel has identical spatial resolution and dilation compared to the current convolution layer. This is succinctly represented in Fig. 2.

IV. Methology

For this project, the forward pass involving the offsets had to be coded in TensorFlow/Keras by creating a child class of the in-built Conv2D class. This made parameter inheritance a trivial obstacle which would otherwise have proved too complex if an attempt was made to define deformable convolution behaviour from the ground up.



(a) standard convolution (b) deformable convolution

Fig. 3: A visual representation of how deformable convolution is carried out on the same image, compared to traditional convolution. Notice the wider spread created by the offsets, which effectively increase the receptive field for the operation.

The MNIST dataset was normalized, divided into test, train and validation groups. After this an ImageDataGenerator object was created for scaling and translating the test group to obtain another test group with geometric transformation. The training, validation and testing performance of a standard 4 layer CNN with and without deformable convolution was measured. A similar test was conducted on a miniature version of the ResNet architecture. The results of these experiments follow.

A. 4 Layer CNN

The first model tested for MNIST classification is a standard 4-layer CNN with a fully connected layer and softmax activation for 10 classes at the output. The training performance is visualized in Fig. 4.

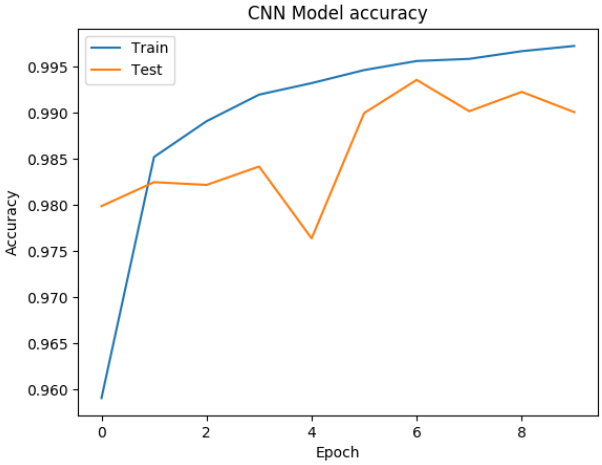


Fig. 4: Training vs. validation accuracy for the 4-layer CNN.

B. 4 Layer CNN with ConvOffset2D

The second model involved augmenting the convolutional receptive field using the ConvOffset2D class created for this project. This effectively introduced deformable convolution in all but the input and fully connected layers of the model. The training and validation performance for this network is shown in Fig. 5

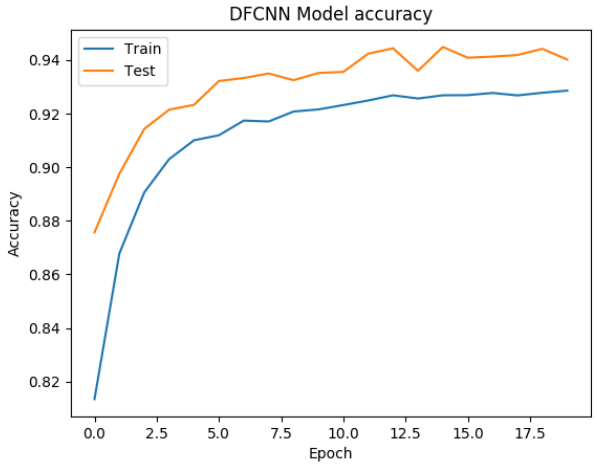


Fig. 5: Training vs. validation accuracy for the 4-layer CNN with deformable convolution.

C. Miniature ResNet

The third model was a simple, miniaturized implementation of the ResNet architecture with 2 ResNet blocks depth. A basic ResNet block is illustrated in Fig. 6.

The training and validation performance for this model are shown in Fig. 7

D. Miniature ResNet with Deformable Convolution

The fourth model implemented offsets to both convolutions within the ResNet block. This model did not perform as expected. The training and validation performance are plotted in Fig. 8

V. Results

The results for the CNN were as expected, with the test accuracy for scaled and transformed MNIST digits was better for the model with deformable convolution.

The 4-layer CNN achieved 99.01% accuracy when tested on the plain MNIST digits, but performed poorly with an accuracy of 60.67% when introduced to the scaled and transformed test set. In contrast, the CNN with deformable convolution performed much better towards unseen geometrical transformations, achieving a respectable 94.1% accuracy on scaled and transformed data, while also managing to learn robustly enough to achieve 96.11% on the plain digit classification.

The ResNet experiment did not yield results are expected. The regular mini-ResNet managing 92.35% accuracy on the test set, and an accuracy of 49.57% on unseen scaled images.

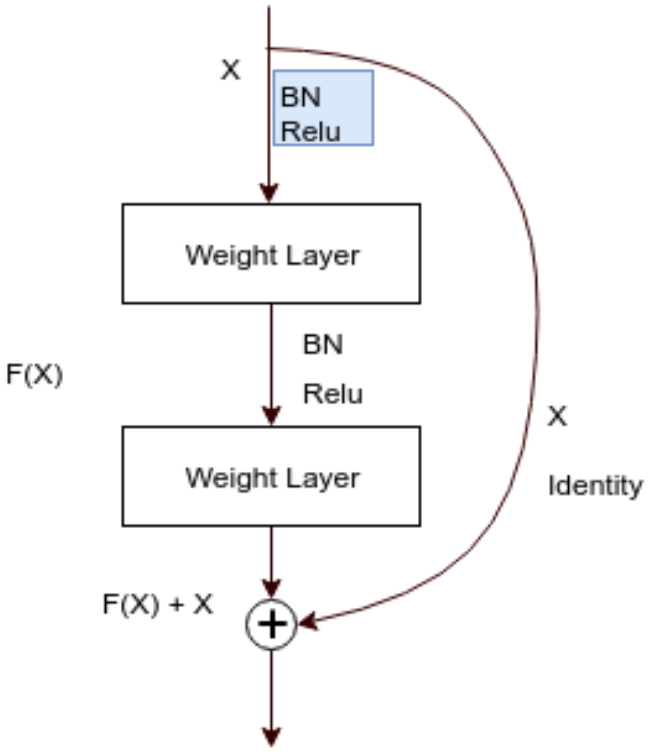


Fig. 6: A basic ResNet block.

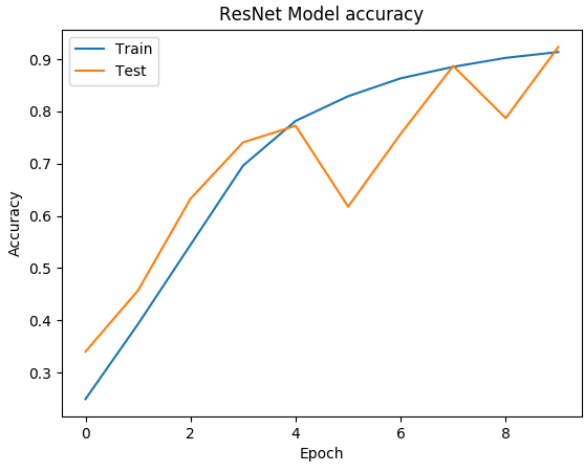


Fig. 7: Training vs. validation accuracy for the 2-block mini-ResNet.

In sharp contrast to the CNN experiment, the mini-ResNet with deformable convolution managed 42.42% accuracy in exactly the same number of training epochs as above, i.e. 10, and performing poorly when tested on scaled images, managing only 13.1%. It should be noted that 10 epochs is not sufficient to train this model as the loss and validation accuracy had not stabilized in just 10 epochs.

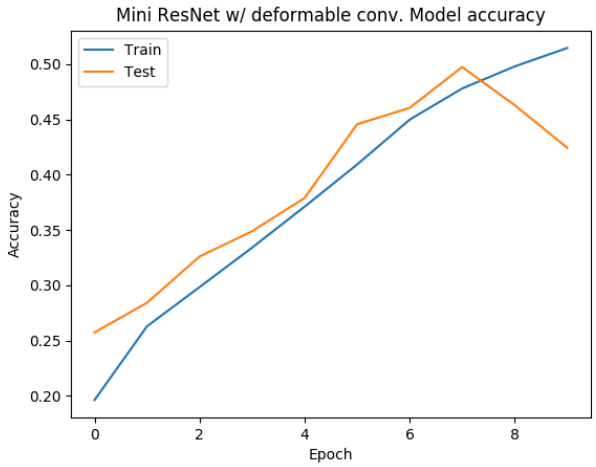


Fig. 8: Training vs. validation accuracy for the 2-block mini-ResNet with deformable convolution.

VI. Conclusion and Observations

The forward pass takes close to 300 ms while the same for regular convolution is just 10 ms. The implementation in this project is extremely inefficient, but given the scope of the project, a fully optimized implementation was untenable. A key limitation is also the fact that this implementation can only be used with the Keras backend. A true TensorFlow implementation would make the module more generalizable.

VII. References

[1] J. Dai et. al, *"Deformable Convolutional Networks,"*, 2017

[2] $github.com/mythrandire/E4040Project\_2019\_DFCN$

VIII. Appendix: Code

A. Definition of ConvOffset2D

```
class ConvOffset2D(Conv2D):
    """ConvOffset2D"""

    def __init__(self, filters, init_normal_stddev=0.01, **kwargs):
        """Init"""

        self.filters = filters
        super(ConvOffset2D, self).__init__(
            self.filters * 2, (3, 3), padding='same', use_bias=False,
            # TODO gradients are near zero if init is zeros
            kernel_initializer='zeros',
            # kernel_initializer=RandomNormal(0, init_normal_stddev),
            **kwargs
        )

    def call(self, x):
        # TODO offsets probably have no nonlinearity?
        x_shape = x.get_shape()
        offsets = super(ConvOffset2D, self).call(x)

        offsets = self._to_bc_h_w_2(offsets, x_shape)
        x = self._to_bc_h_w(x, x_shape)
        x_offset = tf_batch_map_offsets(x, offsets)
        x_offset = self._to_b_h_w_c(x_offset, x_shape)
        return x_offset

    def compute_output_shape(self, input_shape):
        return input_shape

    @staticmethod
    def _to_bc_h_w_2(x, x_shape):
        """(b, h, w, 2c) -> (b*c, h, w, 2)"""
        x = tf.transpose(x, [0, 3, 1, 2])
        x = tf.reshape(x, (-1, int(x_shape[1]), int(x_shape[2]), 2))
        return x

    @staticmethod
    def _to_bc_h_w(x, x_shape):
        """(b, h, w, c) -> (b*c, h, w)"""
        x = tf.transpose(x, [0, 3, 1, 2])
        x = tf.reshape(x, (-1, int(x_shape[1]), int(x_shape[2])))
        return x

    @staticmethod
    def _to_b_h_w_c(x, x_shape):
        """(b*c, h, w) -> (b, h, w, c)"""
        x = tf.reshape(
            x, (-1, int(x_shape[3]), int(x_shape[1]), int(x_shape[2]))
        )
        x = tf.transpose(x, [0, 2, 3, 1])
        return x
```

B.

C. CNN

```
def get_cnn():
    inputs = l = Input((28, 28, 1), name='input')

    # conv11
    l = Conv2D(32, (3, 3), padding='same', name='conv11')(l)
    l = Activation('relu', name='conv11_relu')(l)
    l = BatchNormalization(name='conv11_bn')(l)

    # conv12
    l = Conv2D(64, (3, 3), padding='same', strides=(2, 2), name='conv12')(l)
    l = Activation('relu', name='conv12_relu')(l)
    l = BatchNormalization(name='conv12_bn')(l)

    # conv21
    l = Conv2D(128, (3, 3), padding='same', name='conv21')(l)
    l = Activation('relu', name='conv21_relu')(l)
    l = BatchNormalization(name='conv21_bn')(l)

    # conv22
    l = Conv2D(128, (3, 3), padding='same', strides=(2, 2), name='conv22')(l)
    l = Activation('relu', name='conv22_relu')(l)
    l = BatchNormalization(name='conv22_bn')(l)

    # out
    l = GlobalAvgPool2D(name='avg_pool')(l)
    l = Dense(10, name='fc1')(l)
    outputs = l = Activation('softmax', name='out')(l)

    return inputs, outputs
```

D. DFCNN

```
def get_df_cnn(trainable):
    inputs = l = Input((28, 28, 1), name='input')

    # conv11
    l = Conv2D(32, (3, 3), padding='same', name='conv11', trainable=trainable)(l)
    l = Activation('relu', name='conv11_relu')(l)
    l = BatchNormalization(name='conv11_bn')(l)

    # conv12
    l_offset = ConvOffset2D(32, name='conv12_offset')(l)
    l = Conv2D(64, (3, 3), padding='same', strides=(2, 2), name='conv12', trainable=trainable)(l_offs
    l = Activation('relu', name='conv12_relu')(l)
    l = BatchNormalization(name='conv12_bn')(l)

    # conv21
    l_offset = ConvOffset2D(64, name='conv21_offset')(l)
    l = Conv2D(128, (3, 3), padding='same', name='conv21', trainable=trainable)(l_offset)
    l = Activation('relu', name='conv21_relu')(l)
    l = BatchNormalization(name='conv21_bn')(l)

    # conv22
    l_offset = ConvOffset2D(128, name='conv22_offset')(l)
    l = Conv2D(128, (3, 3), padding='same', strides=(2, 2), name='conv22', trainable=trainable)(l_of
    l = Activation('relu', name='conv22_relu')(l)
    l = BatchNormalization(name='conv22_bn')(l)

    # out
    l = GlobalAvgPool2D(name='avg_pool')(l)
    l = Dense(10, name='fc1', trainable=trainable)(l)
    outputs = l = Activation('softmax', name='out')(l)

    return inputs, outputs
```

E. Mini-ResNet

```
def resblock(n_output, upscale=False):

    def f(x):

        # first pre-actvation
        h = BatchNormalization()(x)
        h = Activation(relu)(h)
        # first convolution
        h = Conv2D(kernel_size=3, filters=n_output, strides=1, padding='same')(h)

        # second pre-activation
        h = BatchNormalization()(x)
```

```
        h = Activation(relu)(h)
        # second convolution
        h = Conv2D(kernel_size=3, filters=n_output, strides=1, padding='same')(h)

        if upscale:
            # 1x1 conv2d
            f = Conv2D(kernel_size=1, filters=n_output, strides=1, padding='same')(x)
        else:
            # identity
            f = x
        return add([f, h])
    return f

def get_mini_resnet():

    inputs = Input((28, 28, 1), name='input')
    # first conv2d with post-activation to transform the input data to some reasonable form
    x = Conv2D(kernel_size=3, filters=16, strides=1, padding='same', name = 'c_in')(inputs)
    x = BatchNormalization(name = 'bn_1')(x)
    x = Activation(relu)(x)
    # F_1
    x = resblock(16)(x)
    # F_2
    x = resblock(16)(x)
    # last activation of the entire network's output
    x = BatchNormalization()(x)
    x = Activation(relu)(x)
    x = GlobalAveragePooling2D()(x)

    # dropout for more robust learning
    # last softmax layer
    x = Dense(units=10)(x)
    outputs = x = Activation(softmax, name= 'output_layer')(x)

    return inputs, outputs
```

F. Mini-DFResNet

```
def df_resblock(n_output, trainable, upscale=False):

    def f(x):

        h = BatchNormalization()(x)
        h = Activation(relu)(h)
        h_offset = ConvOffset2D(n_output)(h)
        h = Conv2D(n_output, (3, 3), strides = 1, padding = 'same', trainable = trainable)(h_offset)

        # second pre-activation
        h = BatchNormalization()(x)
        h = Activation(relu)(h)
        h_offset = ConvOffset2D(n_output)(h)
        # second convolution
        h = Conv2D(n_output, (3, 3), strides=1, padding='same', trainable = trainable)(h_offset)

        if upscale:
            # 1x1 conv2d
            f = Conv2D(kernel_size=1, filters=n_output, strides=1, padding='same')(x)
        else:
            # identity
            f = x
        return add([f, h])
    return f


def get_mini_df_resnet(trainable):

    inputs = Input((28, 28, 1), name='input')
    # first conv2d with post-activation to transform the input data to some reasonable form
    x = Conv2D(kernel_size=3, filters=16, strides=1, padding='same', name = 'c_in')(inputs)
    x = BatchNormalization(name = 'bn_1')(x)
    x = Activation(relu)(x)
    # F_1
    x = df_resblock(16, trainable)(x)
    # F_2
    x = df_resblock(16, trainable)(x)
    # last activation of the entire network's output
    x = BatchNormalization()(x)
    x = Activation(relu)(x)
    x = GlobalAveragePooling2D()(x)

    # last softmax layer
    x = Dense(units=10)(x)
    outputs = x = Activation(softmax, name= 'output_layer')(x)

    return inputs, outputs
```