IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

PREDICT HANDWRITTEN DIGIT WITH CNN AND COMPARE TYPES OF POOLING LAYERS

[1]Ms. Nupur Dongariya, [2]Dr. Ankush Verma, [3]Dr. Manoj Ramaiya
[1]Assistant Professor, [2]Associate Professor, [3]Associate Professor
[1, 2, 3] Institute of Advance Computing
[1, 2, 3] Sage University, Indore, India

Abstract: A common practice to gain invariant features in object recognition models is to aggregate multiple low-level features over a small neighborhood. However, the differences between those models make a comparison of the properties of different aggregation functions hard. Our aim is to predict the Handwritten Digit by training MNIST handwritten digit dataset by Convolutional Neural Network and Comparing the Accuracy, Loss, Validation Accuracy, Validation Loss, Time is taken, Test data Accuracy with Confusion Matrix of MaxPooling2D, GlobalAveragePooling2D, AveragePooling2D layers in our CNN model. Empirical results show that a maximum pooling operation significantly outperforms subsampling operations. We achieve 0.5-0.7% error in the maximum pooling layer and 1-0.7 % error in the Average Pooling layer while 8-9% error in Global Average Pooling Layer. While entire models have been extensively compared, there has been no research evaluating the choice of the aggregation function so far. The aim of our work is therefore to empirically determine which of the established aggregation functions is more suitable for vision tasks. Additionally, we investigate if ideas from signal processing, such as overlapping receptive fields and window functions can improve recognition performance.

Abbreviations. tf – Tensorflow, cnn – Convolutional Neural Network, nn – Neural Network, mnist – Modified National Institute of Standards and Technology database.

I. Introduction

The MNIST database of handwritten digits, use in this project, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analysing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, image classification, medical image analysis. The main structural feature of Regular Nets is that all the neurons are connected. For example, when we have images with 28 by 28 pixels in greyscale, we will end up having 784 (28 x 28 x 1) neurons in a layer that seems manageable. However, most images have way more pixels and they are not grey-scaled. Therefore, assuming that we have a set of color images in 4K Ultra HD, we will have 26,542,080 (4096 x 2160 x 3) different neurons connected in the first layer which is not manageable. Therefore, we can say that Regular Nets are not scalable for image classification. However, especially when it comes to images, there seems to be little correlation or relation between two individual pixels unless they are close to each other. This leads to the idea of Convolutional Layers and Pooling Layers [2]. Many recent object recognition architectures are based on the model of the mammal visual cortex proposed by Hubel and Wiesel [3]. According to their findings, the visual area V1 consists of simple cells and complex cells. While simple cells perform feature extraction, complex cells combine several such local features from a small spatial neighborhood. It is assumed that spatial pooling is crucial to obtain translation-invariant features. Deep Neural Networks now excel at image classification, detection, and segmentation. When used to scan images using a sliding window, however, their high computational complexity can bring even the most powerful hardware to its knees. We show how dynamic programming can speed up the process by orders of magnitude, even when max-pooling layers are present. On two-dimensional feature maps, pooling is typically applied in 2×2 patches of the feature map with a stride of (2, 2). Average pooling involves calculating the average for each patch of the feature map. This means that each 2×2 square of the feature map is downsampled to the average value in the square. In a nutshell, the reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence, the term feature map), and it's more informative to look at the maximal presence of different features than at their average presence. These models can be broadly distinguished by the operation that summarizes over a spatial neighborhood. Earlier models perform a subsampling operation, where the average overall input values are propagated to the next layer. Such architectures include the Neocognitron, CNNs, and the Neural Abstraction Pyramid. A different approach is to compute the maximum value in a neighborhood. This direction is taken by the HMAX model and some variants of CNNs.

II. Solution Development:

The MNIST dataset is one of the most common datasets used for image classification and accessible from many different sources. Even Tensorflow and Keras allow us to import and download the MNIST dataset directly from their API. The MNIST database contains 60,000 training images and 10,000 testing images taken from American Census Bureau employees and American high school students [1]. I have separated these two groups as train and test and also separated the labels and the images. x_train and x_test parts contain greyscale RGB codes (from 0 to 255) while y_train and y_test parts contain labels from 0 to 9 which represents which number they are. Each image is of size 28x28, to visualize these numbers, we can get help from matplotlib.

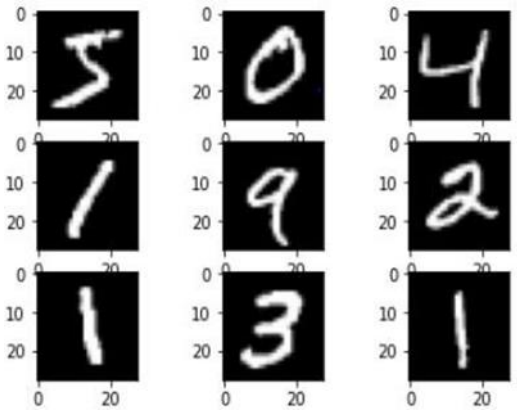

Fig. 2.1: Image Data set

The size of the data is (60000, 28, 28). Where 60000 represents the number of images in the training dataset and (28, 28) represents the size of the image: 28 x 28 pixel. To be able to use the dataset in Keras API, we need 4- dims NumPy arrays. However, as we see above, our array is 3-dims. In addition, we must normalize our data as it is always required in neural network models. We can achieve this by dividing the RGB codes to 255 (which is the maximum RGB code minus the minimum RGB code). Convolutional layers in a convolutional neural network systematically apply learned filters to input images to create feature maps that summarize the presence of those features in the input. Convolutional layers prove very effective, and stacking convolutional layers in deep models allows layers close to the input to learn low-level features (e.g. lines) and layers deeper in the model to learn high-order or more abstract features, like shapes or specific objects. A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image [5]. A common approach to addressing this problem from signal processing is called downsampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.

A pooling layer is a new layer added after the convolutional layer. Specifically, after a nonlinearity (e.g. ReLU) has been applied to the feature maps output by a convolutional layer; for example, the layers in a model may look as follows: Input Image, Convolutional Layer, Nonlinearity, and Pooling Layer. The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model. The pooling layer operates upon each feature map separately to create a new set of the same number of pooled feature maps [5].

Max Pooling

If we want to downsample it, we can use a pooling operation that is known as “max pooling” (more specifically, this is two- dimensional max pooling). In this pooling operation, a $H \times W$ “block” slides over the input data, where H is the height and W the width of the block. The stride (i.e. how much it steps during the sliding operation) is often equal to the pool size so that its effect equals a reduction in height and width. For each block, or “pool”, the operation simply involves computing the max value, like this [4]:

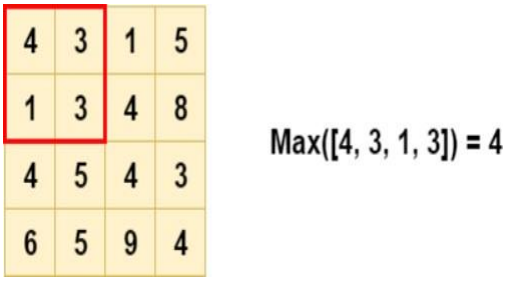

Fig. 2.2: 2-D Max Pooling

Global Average Pooling

When applying Global Average Pooling, the pool size is still set to the size of the layer input, but rather than the maximum, the average of the pool is taken [5] :

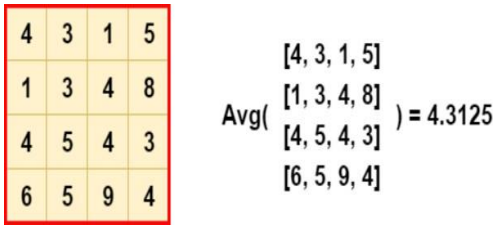

Fig. 2.3: Global Average Pooling

II. Implementation

We will build our model by using high-level Keras API which uses either Tensorflow or Thea no on the backend. I would like to mention that there are several high-level TensorFlow APIs such as Layers, Keras, and Estimators which help us create neural networks with high-level knowledge. However, this may lead to confusion since they all vary in their implementation structure. Therefore, if you see completely different codes for the same neural network although they all use TensorFlow, this is why. I will use the most straightforward API which is Keras. Therefore, I will import the Sequential Model from Keras and add Conv2D, MaxPooling, Flatten, Dropout, and Dense layers. I have already talked about Conv2D, Maxpooling, and Dense layers. In addition, Dropout layers fight with the overfitting by disregarding some of the neurons while training while Flatten layers flatten 2D arrays to 1D arrays before building the fully connected layers. The result of using a pooling layer and creating downsampled or pooled feature maps is a summarized version of the features detected in the input [6]. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation.

We created 2 different models for comparing different types of pooling layers in it.

Max Pooling

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 64)        640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 14, 128)       73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 7, 7, 256)         295168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 128)               295040
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 665,994
Trainable params: 665,994
Non-trainable params: 0
```

Fig. 3.1: Tensorflow-Keras model with MaxPooling2D

Global Average Pooling

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 28, 28, 64)        640
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 128)       73856
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 28, 28, 256)       295168
_________________________________________________________________
global_average_pooling2d (Gl (None, 256)               0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
=================================================================
Total params: 403,850
Trainable params: 403,850
Non-trainable params: 0
_________________________________________________________________
```

Fig. 3.2: Tensorflow-Keras model with GlobalAveragePooling2D

We may experiment with any number for the first Dense layer; however, the final Dense layer must have 10 neurons since we have 10 number classes (0, 1, 2… 9). You may always experiment with kernel size, pool size, activation functions, dropout rate, and several neurons in the first dense layer to get a better result. With the above code, we created a non-optimized empty CNN. Now it is time to set an optimizer with a given loss function that uses a metric. Then, we can fit the model by using our train data. You can experiment with the optimizer, loss function, metrics, and epochs. However, I can say that the Adam optimizer is usually out-performs the other optimizers. I am not sure if you can change the loss function for multi-class classification. Feel free to experiment and comment below. The epoch number might seem a bit small. Since the MNIST dataset does not require heavy computing power, we may easily experiment with the epoch number as well.

V. Result Analysis

Max Pooling

```
Epoch 1/10
844/844 [==============================] - 11s 13ms/step - loss: 0.1421 - accuracy: 0.9567 - val_loss: 0.0389
Epoch 2/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0403 - accuracy: 0.9876 - val_loss: 0.0407
Epoch 3/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0264 - accuracy: 0.9919 - val_loss: 0.0403
Epoch 4/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0179 - accuracy: 0.9943 - val_loss: 0.0467
Epoch 5/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0147 - accuracy: 0.9952 - val_loss: 0.0335
Epoch 6/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0130 - accuracy: 0.9958 - val_loss: 0.0303
Epoch 7/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0112 - accuracy: 0.9964 - val_loss: 0.0384
Epoch 8/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0089 - accuracy: 0.9971 - val_loss: 0.0541
Epoch 9/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0091 - accuracy: 0.9970 - val_loss: 0.0457
Epoch 10/10
844/844 [==============================] - 10s 12ms/step - loss: 0.0056 - accuracy: 0.9980 - val_loss: 0.0337
```

Fig. 4.1: The final result for Accuracy, Loss, Validation Accuracy, and Validation Loss by Max Pooling

Graphical Representation

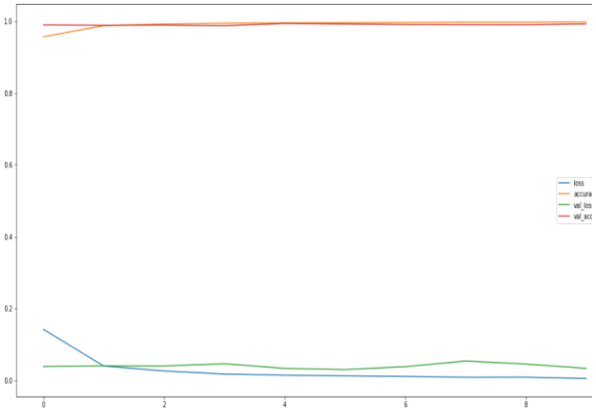

Fig. 4.2: Graphical Representation of final result by Max Pooling

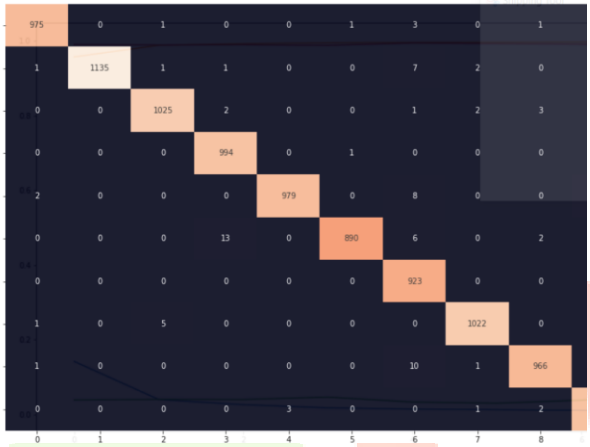

Fig. 4.3: Confusion Matrix in form of the heat map (Testing data Accuracy = 98.8%)

Global Average Pooling

```
Epoch 1/10
844/844 [==============================] - 34s 41ms/step - loss: 1.1797 - accuracy: 0.5882 - val_loss: 0.4919 - val_accuracy: 0.8570
Epoch 2/10
844/844 [==============================] - 34s 40ms/step - loss: 0.4486 - accuracy: 0.8566 - val_loss: 0.3140 - val_accuracy: 0.9015
Epoch 3/10
844/844 [==============================] - 34s 40ms/step - loss: 0.2902 - accuracy: 0.9104 - val_loss: 0.1966 - val_accuracy: 0.9373
Epoch 4/10
844/844 [==============================] - 34s 40ms/step - loss: 0.2136 - accuracy: 0.9349 - val_loss: 0.1411 - val_accuracy: 0.9553
Epoch 5/10
844/844 [==============================] - 34s 40ms/step - loss: 0.1690 - accuracy: 0.9498 - val_loss: 0.1185 - val_accuracy: 0.9617
Epoch 6/10
844/844 [==============================] - 34s 40ms/step - loss: 0.1380 - accuracy: 0.9586 - val_loss: 0.1118 - val_accuracy: 0.9640
Epoch 7/10
844/844 [==============================] - 34s 40ms/step - loss: 0.1189 - accuracy: 0.9636 - val_loss: 0.1004 - val_accuracy: 0.9680
Epoch 8/10
844/844 [==============================] - 34s 40ms/step - loss: 0.1054 - accuracy: 0.9680 - val_loss: 0.0817 - val_accuracy: 0.9745
Epoch 9/10
844/844 [==============================] - 34s 40ms/step - loss: 0.0932 - accuracy: 0.9714 - val_loss: 0.0721 - val_accuracy: 0.9777
Epoch 10/10
844/844 [==============================] - 34s 40ms/step - loss: 0.0842 - accuracy: 0.9745 - val_loss: 0.0652 - val_accuracy: 0.9805
```

Fig. 4.4: Final result for Accuracy, Loss, Validation Accuracy, and Validation Loss by Global Avg. Pooling

Graphical Representation

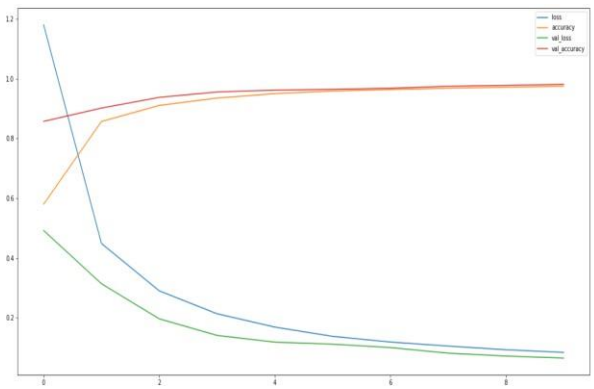

Fig. 4.5: Graphical Representation of final result by Global Average Pooling

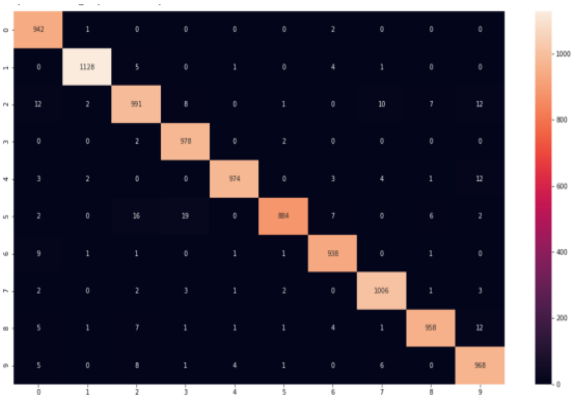

Fig. 4.5 Confusion Matrix in form of the heat map (Testing data Accuracy = 97.6%)

So, the Final Conclusion is for Image classification Max Pooling layer is better than using the Global Average Pooling layer.

V. References

1. Yann LeCun, Courant Institute, NYU, Corinna Cortes, Google Labs, New York, Christopher J.C. Burges, Microsoft Research, Redmond: THE MNIST DATABASE of handwritten digits.
2. a Claudiu Ciresan; Ueli Meier; Luca Maria Gambardella; Jurgen Schmidhuber: Convolutional Neural Network Committees for Handwritten Character Classification
3. Ahmed, A., Yu, K., Xu, W., Gong, Y., Xing, E.: Training hierarchical feed-forward visual recognition models using transfer learning from pseudo- tasks. In: Forsyth, D., Torr, P., Zisserman, A. (eds.) ECCV 2008, Part III. LNCS, vol. 5304, pp. 69–82. Springer, Heidelberg (2008)
4. Alessandro Giusti; Dan C. Cireşan; Jonathan Masci; Luca M. Gambardella; Jürgen Schmidhuber: Fast image scanning with deep max-pooling convolutional neural networks
5. Jason Brownlee Article: A Gentle Introduction to Pooling Layers for Convolutional Neural Networks
6. Behnke, S.: Hierarchical Neural Networks for Image Interpretation. LNCS, vol. 2766. Springer, Heidelberg (2003)