Convolutional Neural Networks

Moisey Alaev, Naoya Kumagai

Abstract— **Convolutional Neural Networks (CNNs) are a type of deep Artificial Neural Networks (ANN/NN) applied to data with a grid-like topology. CNNs are primarily used in computer vision tasks such as image recognition and analysis. We will illustrate the mathematical foundations of CNNs and demonstrate the application of the model computationally to the famous MNIST data set of images.**

I. INTRODUCTION

The concept of Convolutional Neural Networks (CNNs) had its groundings in biological research conducted as early as the 1950s. In their critical study of mammalian visual cortexes that won David Hubel and Torsten Wiesel a Nobel Prize, they identified that neurons individually correspond to small regions of the visual field leading to the description of a "receptive field" [3]. By the 1990s the first modern CNN was created by Yann LeCun and the trained on the aforementioned MNIST data set [3]. Most recently, in 2012 a CNN model called AlexNet achieved incredible accuracy labeling pictures, demonstrating almost half the error percentage of the human eye in visually identification. This breakthrough led to the current resurgence of CNNs and their wide applications in not only image classification, but also medical imaging, recommender systems, and natural language processing [2]. However, in this investigation, we will primarily focus on applying CNNs to classification of images.

A. Overview of CNN Procedure

The fundamental operation of CNNs that sets these structures apart from regular NNs is the feature learning aspect. In the feature learning section of the network, the driving force is the convolution step. Convolution is a linear operation which takes the place of general matrix multiplication in one of the processing layers. In contrast to a traditional NN, CNNs are able to develop an effective computation of features–characteristics of images such as edges–that are aggregated into feature maps or convolutional layers. These convolutional layers are then activated using a non-linear activation function, usually Rectified Linear Unit (ReLU). With there being several of such layers in the network, the deeper we travel from the input, the higher level these features become and the closer we get to classifying the image. However, before we consider a deep network we must first pass our feature maps to the next major step of the process: pooling.

Pooling in essence is the grouping together of nearby information. The initial incentivization of pooling the convolutional layer is to reduce dimensionality for greater computational efficiency. However, a potentially greater outcome of this operation is that it allows our trained models to properly detect images in spite of any small initial distorting transformations. The process of pooling, much like feature mapping, analyzes a section of the input at a time, but unlike feature mapping it reduces the block into one data element of the pooled layers. The method by which this is done varies, but often times uses the maximum element of the block as the aforementioned single data element, this commonly implemented method is called max pooling.

Now that we have described the two major processes of the feature learning phase, convolution with activation and pooling, we can repeat these methods over several layers. Therefore, making the feature learning phase "deep". Then after this feature learning iterative process, we flatten the final result of pooled information and pass it to the classification phase.

This classification phase reflects the structure of a tradition NN, containing several hidden layers each with fully connected layers of neurons. At the end of this classification layer we apply a final non-linear activation, usually the softmax function. In the end we are left with a vector of probabilities of what the image is to be classified as and the network chooses the most probable option as the result.

B. Deliverables

We will take our mathematical and conceptual understanding of neural networks and expand them to these two major concepts. Our research will investigate the mathematical backbone of these individual components and how they work together to perform the ultimate task of image detection. After which, we will implement our own CNN model specifically designed to tackle reading the MNIST dataset of handwritten digits using libraries such as Keras in an IPython Notebook.

II. MATHEMATICAL FOUNDATION

In this section, we present the mathematical foundations of convolutional neural networks. First, we will investigate how the convolution operation works. Then, we shall discuss the use of padding the input image data. Afterwards, our group will link the convolutional step to the pooling step. Finally, we will describe how the weights for the convolution filters are optimized through backpropagation.

A. Mathematical Formulation

1) Convolution: Convolution is the process of taking an input image and performing an element wise matrix multiplication with a filter of a smaller dimension in length and height of an image, but always of the same depth as the input image to produce one or most likely several feature maps. To start this process, we preface how the input image can be defined computationally: let the pixel dimensionality define the height and width. Therefore, let any arbitrary image have its dimensions be $H \times W \times D$ for height, width, and depth, respectively. Here, depth takes on one of two possible values depending on the image type: gray-scale or red, green, and blue (RGB). If our image is gray-scaled then $D = 1$ and if it is colored $D = 3$, for each red, green, and blue channel. Hence, each pixel holds a value, or values for a colored image. For instance, the values of a gray-scaled image could lie in the range $[x, y]$ where x would be zero illumination and y would be full illumination. Usually, $(x, y) = (0, 1)$ or $(x, y) = (0, 255)$. While, for colored images, each color channel would have values in $[0, 255]$.

In order to understand the mathematical principals, suppose for simplicity, that we take a 2D input of size $(H \times W)$ and subsequently a 2D kernel K of size $k_1, k2$. To perform convolution we must apply the filter K onto the input image I. This is equivalent to performing the convolution of the I and K.

$$S_{i,j} = (I * K)_{i,j} \qquad i \in H, j \in W \tag{1}$$

where S is a the output matrix of the convolution, known as the feature map. To expand the RHS of (1) note that the (i, j) element of S is determined by performing element wise multiplication: summing the product of (m, n) element of I with the $(i - m, j - n)$ element of K over all elements of the kernel, i.e. over all instances of m and n. Therefore, we can expand (1) as

$$\begin{aligned} S_{i,j} &= (I * K)_{i,j} \\ &= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i-m, j-n)K(m,n) \end{aligned} \tag{2}$$

Now further extrapolate (2) by negating both m and n:

$$S_{i,j} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n)K(-m,-n)$$

At this step we can flip the kernel both vertically and horizontally by allowing $K(-m, -n) = K(m, n)$ to obtain the cross correlation operation

$$(I \otimes K)_{i,j} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n)K(m,n) \tag{3}$$

[6]

As can be seen by comparing the convolution equation (2) with the cross-correlation equation in (3), the only difference is the horizontal and vertical flipping of the kernel. Hence, the cross-correlation operation is fundamentally equivalent to the convolution function with the exception of the elements of the kernel being learned in a "flipped" manner. This is often the preferred form of convolution as there is less variation in the range of valid values of m by n [4]. In fact, cross-correlation is virtually always implemented in place of convolution in machine learning libraries.

Note that there is generally a bias term added to the convolutional expression. Using the notations defined above, the (i, j) element in a feature map can derived by

$$(I * K)_{i,j} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \sum_{c=1}^{D} K_{m,n,c} \cdot I_{i+m,j+n,c} + b.$$

Since convolution, by nature, will not detect the features of the pixels that are closer to the image boundary as well as those depicted by pixels closer towards the center of the image, padding becomes a powerful technique to circumvent this problem. Let the input image size be $H \times W$, kernel size be $k \times k$, and padding size p—referring to the number of single layer frames containing zeros placed around the image. Now define the three common types of padding: "same padding", in which $p = 1$. Consequently, the input size will be modified to $(H+2) \times (W+2)$. Next, "full padding", refers to padding size $p = k - 1$. Thus, the new input size becomes $(H + 2(k - 1)) \times (W + 2(k - 1))$. Lastly, "valid padding" performs no modifications to the image, serving as the basic input.

Now, let us define how the filter is transversed over the input image by means of "strides". A stride refers to a unit "movement" of the kernel as it takes the element wise product along the input image. Suppose s defines the stride; We then move the kernel s pixels in each direction as we take the element wise product. Making $s > 1$ has the benefit of producing a smaller output size, therefore reducing computational cost. However, if s is too large it will fail to capture the critical characteristics of the feature maps. Hence, in practice, a stride bigger than 3 is rare.

In general, convolution between an input matrix and kernel produces an output feature map of size

$$(\frac{H + 2p - k}{s} + 1) \times (\frac{W + 2p - k}{s} + 1)$$

.

2) Pooling: Now that we have obtained the necessary convolution operation, the next step is to describe pooling. After passing our filters through the input image we are left with our feature maps. Since we apply convolution between every layer and often several times per layer, the computational intensity of maintaining the parameters from the feature maps progressively grows. Therefore, to counter dealing with large spatial sizes and to overall reduce the computation time between layers, we apply the pooling operation[6].

Pooling is applied during forward propagation by taking an input block of larger dimensional size corresponding to the "receptive field" and transforming it into a 1×1 output block in the pooled matrix [6][5]. The receptive field refers to the region of the input that contributes to the output value [5]. Further, there are different ways to choose how the output block is determined from the receptive field. This sprouts several different types of pooling, but we will describe the two most commonly implemented: average-pooling and max-pooling. Max-pooling chooses the largest value in the receptive field as the output block. Average-pooling takes the sample average of the values in the receptive field as the output block.

3) Backpropagation: The network improves its accuracy through minimizing its loss function using backpropagation as in a fully-connected neural network. A characteristic of CNNs is the application of convolution during its backpropagation process. Accordingly, let E be the loss function and $x^l_{i,j}$ be the input vector at layer l plus the bias, i.e.

$$x^l_{i,j} = \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l$$

where

$$o^{l-1}_{i+m,j+n} = f(x^{l-1}_{i+m,j+n})$$

and f is the activation function. Now we aim to minimize $\frac{\partial E}{\partial w^l_{m',n'}}$ in layer l, where $\partial w^l_{m',n'}$ is the change in the m', n' entry in the kernel. For simplicity, let the output feature map size be $(H - k_1 + 1) \times (W - k_2 + 1)$: set stride $= 1$ and padding size $= 0$. Apply the chain rule and define $\delta^l_{i,j}$ as follows,

$$\begin{aligned}\frac{\partial E}{\partial w^l_{m',n'}} &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x^l_{i,j}} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} \\ &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} \qquad (4)\end{aligned}$$

Substituting in the definition of $x^l_{i,j}$ into (4), we can write

$$\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} = \frac{\partial}{\partial w^l_{m',n'}} \left(\sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l \right)$$

Now expand the summation and then take the partial derivative. Except for when $m = m'$ and $n = n'$, the derivatives become 0. Thus,

$$\begin{aligned}\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} &= \frac{\partial}{\partial w^l_{m',n'}} \left(w^l_{m',n'} o^{l-1}_{i+m',j+n'} \right) \\ &= o^{l-1}_{i+m',j+n'} \qquad (5)\end{aligned}$$

Substituting (5) into (4),

$$\begin{aligned}\frac{\partial E}{\partial w^l_{m',n'}} &= \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2} \delta^l_{i,j} o^{l-1}_{i+m',j+n'} \\ &= \text{rot}_{180^\circ}\left\{\delta^l_{i,j}\right\} * o^{l-1}_{m',n'} \qquad (6)\end{aligned}$$

rot_{180° indicates that the delta matrix—a matrix with the partial derivatives of the entries of the input matrix as its entries—is flipped vertically and horizontally. Next, we find

$$\delta^l_{i,j} = \frac{\partial E}{\partial x^l_{i,j}}.$$

Recall that in equation (2) the (i, j) entry of the output is a function of the entries of the input matrix in the horizontal range $[i, i + k_1 - 1]$ and the vertical range $[j, j + k_2 - 1]$. Here, we consider the opposite: which entries of the output matrix are effected by the (i', j') entry of the input matrix. Let that region be denoted as Q.

By the chain rule,

$$\frac{\partial E}{\partial x^l_{i',j'}} = \sum_{i,j \in Q} \frac{\partial E}{\partial x^{l+1}_Q} \frac{\partial x^{l+1}_Q}{\partial x^l_{i',j'}}$$

From observation, Q is in the horizontal range $[i', i' - (k_1 - 1)]$ and vertical range $[j', j' - (k_2 - 1)]$. Therefore, we can rewrite the above equation and define δ similar to how we did earlier in equation 4.

$$\begin{aligned}\frac{\partial E}{\partial x^l_{i',j'}} &= \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \frac{\partial E}{\partial x^{l+1}_{i'-m,j'-n}} \frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^l_{i',j'}} \\ &= \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} \frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^l_{i',j'}}\end{aligned}$$

Recall the definition of $x^l_{i,j}$ to say

$$x^{l+1}_{i'-m,j'-n} = \sum_{m'}\sum_{n'} w^{l+1}_{m',n'} o^l_{i'-m+m',j'-n+n'} + b^{l+1}$$

Using this equation and take the partial derivatives over all of the respective components in a similar manner as (5). We get

$$\begin{aligned}\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^l_{i',j'}} &= \frac{\partial}{\partial x^l_{i',j'}} \left(w^{l+1}_{m,n} o^l_{i',j'} \right) \\ &= \frac{\partial}{\partial x^l_{i',j'}} \left(w^{l+1}_{m,n} f\left(x^l_{i',j'}\right)\right) \\ &= w^{l+1}_{m,n} \frac{\partial}{\partial x^l_{i',j'}} \left(f\left(x^l_{i',j'}\right)\right) \\ &= w^{l+1}_{m,n} f'\left(x^l_{i',j'}\right).\end{aligned}$$

Again by substitution,

$$\begin{aligned}\frac{\partial E}{\partial x^l_{i',j'}} &= \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} w^{l+1}_{m,n} f'\left(x^l_{i',j'}\right) \\ &= \text{rot}_{180^\circ}\left\{ \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'+m,j'+n} w^{l+1}_{m,n} \right\} f'\left(x^l_{i',j'}\right) \\ &= \delta^{l+1}_{i',j'} * \text{rot}_{180^\circ}\left\{w^{l+1}_{m,n}\right\} f'\left(x^l_{i',j'}\right) \\ &= \frac{\partial E}{\partial x^{l+1}_{i',j'}} * \text{rot}_{180^\circ}\left\{w^{l+1}_{m,n}\right\} f'\left(x^l_{i',j'}\right) \qquad \blacksquare\end{aligned}$$

Hence, we have formulated the necessary equation to develop a recursive algorithm for back propagation. [6]

III. Experiment

In this section, we apply CNN to the MNIST image dataset [7]. This dataset consists of 60,000 images of handwritten digits from 0-9 that are each 28 by 28 pixels. The correct label is correspondingly attached to each digit. We used 10 percent of the training data as validation data. The dataset also contains 10,000 more images for testing. Figure 1 shows examples of some of the images.

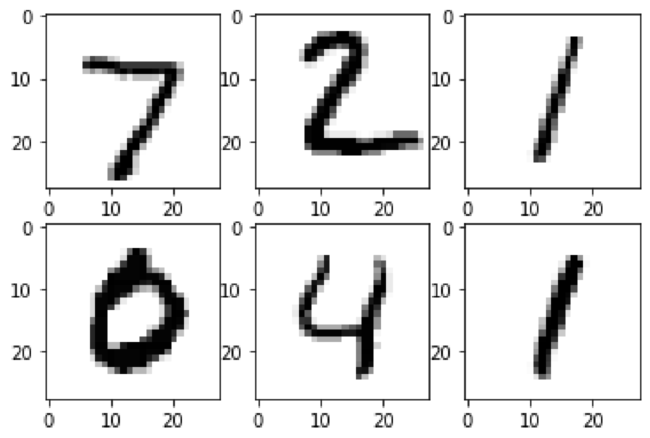

Fig. 1. Six sample images from the MNIST database.

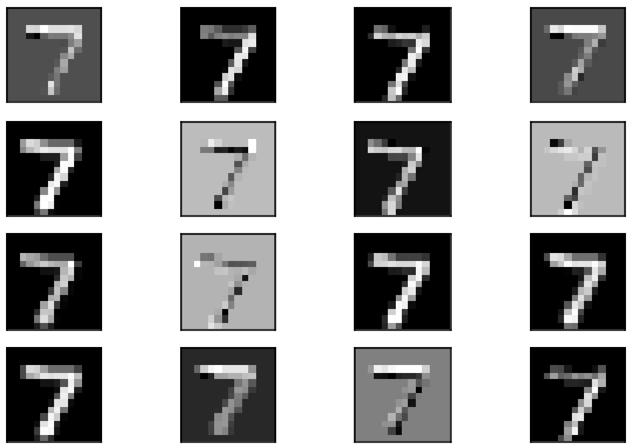

Fig. 2. Feature maps of the top-left image in Figure 1 extracted from the first convolutional layer.

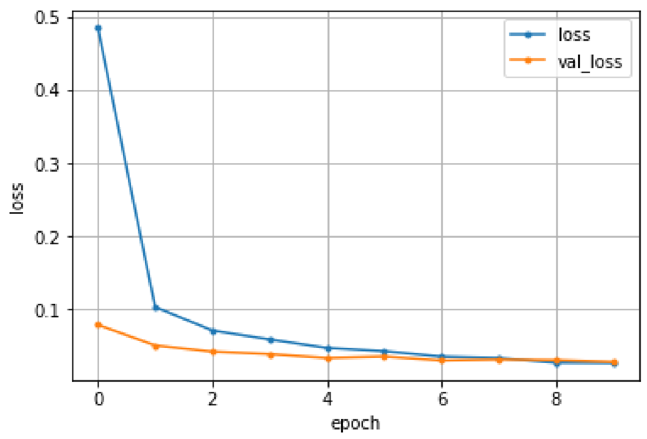

Fig. 3. Loss calculated by cross-entropy between prediction and actual label in the training and validation data.

A. Steps

We implement a CNN model with a stride 1 and with the following ordered layers:

1) Convolution layer (16 feature maps, 2×2 filter, same padding)
2) Max pooling layer (Pooling size 2×2)
3) Convolution layer (128 feature maps. 3×3 filter, valid padding, ReLU activation
4) Convolution layer (256 feature maps, 3×3 filter, valid padding, ReLu activation)
5) Max Pooling (Pooling size 2×2)
6) Dropout layer (Drop rate 50 percent)
7) Flatten layer
8) Fully connected layer (Output size 128, ReLU activation)
9) Dropout layer (Drop rate 25 percent)
10) Fully Connected layer (Output size 10)
11) Softmax activation layer

By applying these stacked layers we develop an intelligent model that is able to extract several different features as demonstrated in Figure 2. Furthermore, Figures 3 and 4 illustrate the model's loss and accuracy, respectively, for the training and validation sets. In the final step of the process we must use the softmax activation function. Softmax is defined by

$$y_i = \frac{e^{x_i}}{\sum_{k=1}^{N} e^{x_k}}, \text{ for i = 1,2,...,N}$$

Applying the softmax function on a vector of size 10, we get an output vector of identical size with non-negative floats that sum up to 1. The final prediction is made by choosing the digit that corresponds to the maximum argument of the output vector.

Recall the loss is calculated using the cross-entropy function:

$$H(p, q) = -\sum_{x} p(x) \ln(q(x)).$$

In this MNIST example, the output vector of the softmax function is p in the equation above and the correct label is q. (If the correct digit is 2, $p = (0, 0, 1, 0, ..., 0)$)

B. Results

Implementing the model on the 10,000 testing images, we achieved an accuracy of about 99.2 percent. As seen in the Figure 4, after 10 epochs—iterations through the entire dataset—learning plateaued. There is no apparent over-fitting seen in Figures 3 and 4, which can be attributed to the two dropout layers we implemented. Moreover, Figure 5 shows the prediction our model made from the six images in Figure 1. For each image, the model distributed probabilities almost equal to 1 for the correct digit, i.e. the model had "no doubt" in choosing the correct digit.

IV. CONCLUSIONS

Convolutional Neural Networks are powerful architectures geared towards meeting modern day visual learning tasks without requiring very large amounts of computational time.

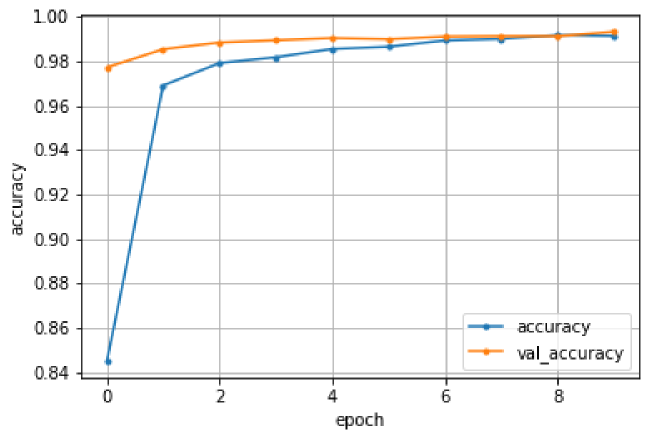

Fig. 4. Accuracy of final predictions after taking the max argument of the softmax (activation) layer output of the training and validation data.

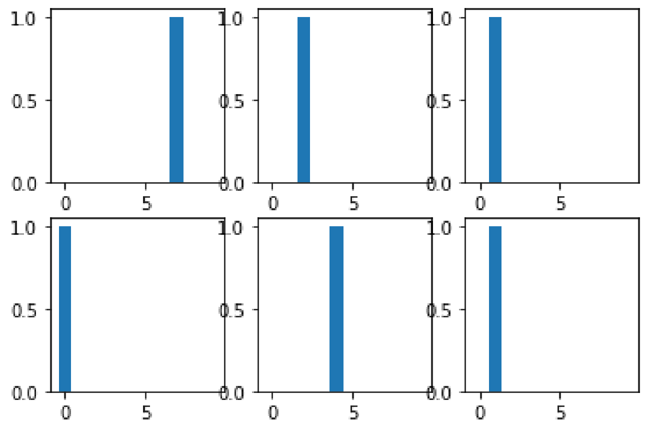

Fig. 5. Prediction made by the model after the softmax (activation) layer output. Each bar graph corresponds to the images in Figure 1.

As a result of our investigation we have have uncovered the following features that make CNNs so useful: 1. Sparse Interactions:
As defined in the mathematical foundation, filters are always of smaller size in height and width; therefore the connection between a certain pixel in the receptive field is "sparsely" connected to its feature map. This blatantly contrasts with the full connectivity of ordinary neural networks and makes sparse interactions the primary aspect of CNNs that allows them to identify low level features such as edges in an image.

2. Parameter Sharing:
Opposed to traditional Neural Networks, in the feature learning phase of CNNs we do not apply a traditional weight matrix which contains an individual weight for each desired connection in a NN. Instead, the weights in the feature learning phase—represented by the elements of the kernel—each transverse every pixel position in the input image with the exception of the boundary depending on padding implementation. In this way, parameter sharing considerably reduces the total number of parameters that are necessary to train the model.

3. Equivariant Representations:
When applying feature learning our model in essence develops an understanding of small features like edges to more abstract features like entire numbers the deeper we travel through the layers. As a result, if our image were to be distorted to a reasonable degree in illumination and position the model should still be able to identify the low level features. In turn, the model would remain capable of developing the high level features necessary to properly handle tasks of image analysis. Hence, equivariant representations describe CNNs' invariance to transformations in the input.

Going back to the experiment, our 99.2% accuracy was high; however, the highest classification models achieve about 99.7% [1]. To improve our score, possible measures we could take are tuning hyperparameters such as the optimizer, number of feature maps, and kernel size, introducing advanced techniques such as data augmentation, batch normalization, decaying learning rate.

In future experiments with CNNs we may test our model on datasets with RGB images containing various backgrounds since MNIST images have similar monotone backgrounds making learning easier. Additionally, on the implementation of the model we could challenge ourselves to use lower level libraries such as TensorFlow or even build a simple CNN from scratch in NumPy.

References

[1] How to score 97%, 98%, 99%, and 100%. `https://www.kaggle.com/c/digit-recognizer/discussion/61480` (Accessed: 2020-12-11).

[2] Ashwin Bhandare et al. Applications of convolutional networks. September 2016. `http://ijcsit.com/ijcsit-v7issue5.php`.

[3] Andrew Fogg. A history of machine learning and deep learning, Dec 2019.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] Antonino Ingargiola. Deep-dive into convolutional networks, Apr 2019. `https://towardsdatascience.com/deep-dive-into-convolutional-networks-48db75969fdf`.

[6] Jefkine Kafunah. Backpropagation in convolutional neural networks. `https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/` (Accessed: 2020-12-08).

[7] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.