Convolutional Neural Networks

Presented By: Don Kim - 2022.12.11

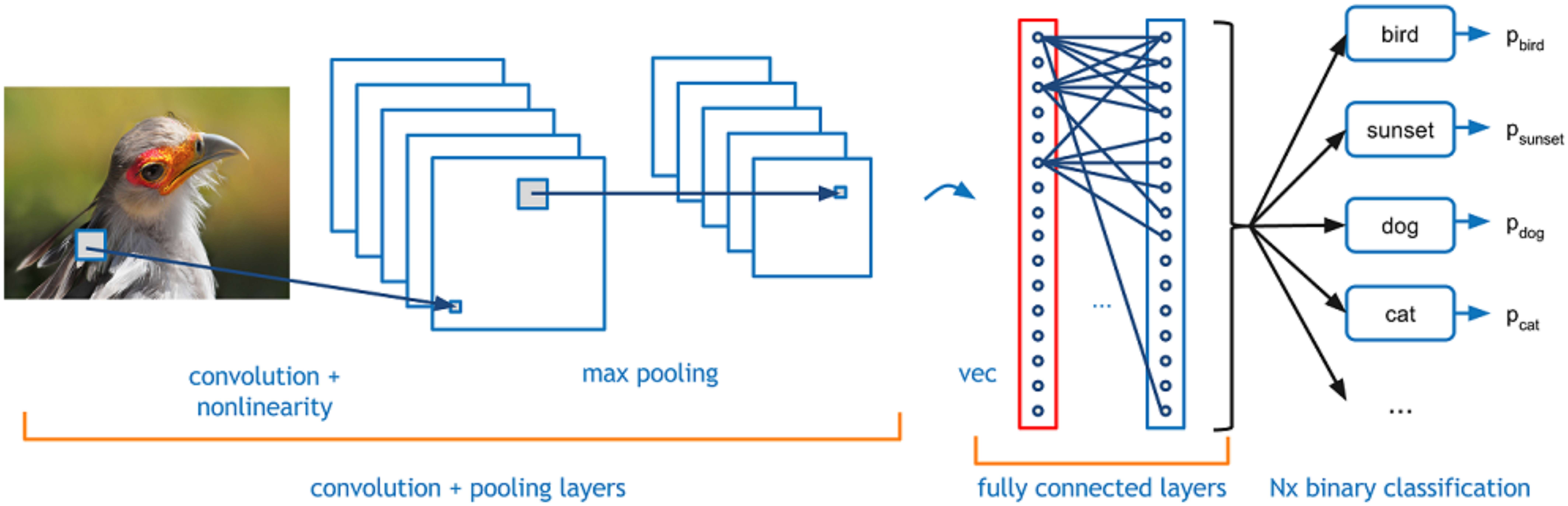
Project Description

1. Implement CNN in the GWU Neural Network Library
2. Determine the effectiveness of the implementation with a test case
3. Cannot use any existing CNN libraries (E.g. Tensorflow)

Convolutional Neural Network (CNN)

What is a CNN?

Deep learning algorithm that takes in an image as an input, assigns importance (weight) to certain features in the image, and is able to classify the image input



bird
sunset
dog
cat
...
p_{bird}
p_{sunset}
p_{dog}
p_{cat}
...
convolution +
nonlinearity
max pooling
vec
convolution + pooling layers
fully connected layers
Nx binary classification

CNNs are great for image classification compared to other models, since the convolution layer reduces images without losing its information

Parts of the CNN

1. Input Layer
2. Convolutional Layer
3. Max Pooling Layer
4. Flattening Layer
5. Dense Layer
6. Activation / Loss Layer
7. Output

For the project, the Convolutional, Max Pooling, and Flattening layer needed to be implemented

```
network = GWUNetwork()
network.add(Conv2D(...))
network.add(MaxPooling2D(...))
network.add(Flatten(...))
network.add(Dense(...))
network.add(Dense(...))
```

Convolutional Layer

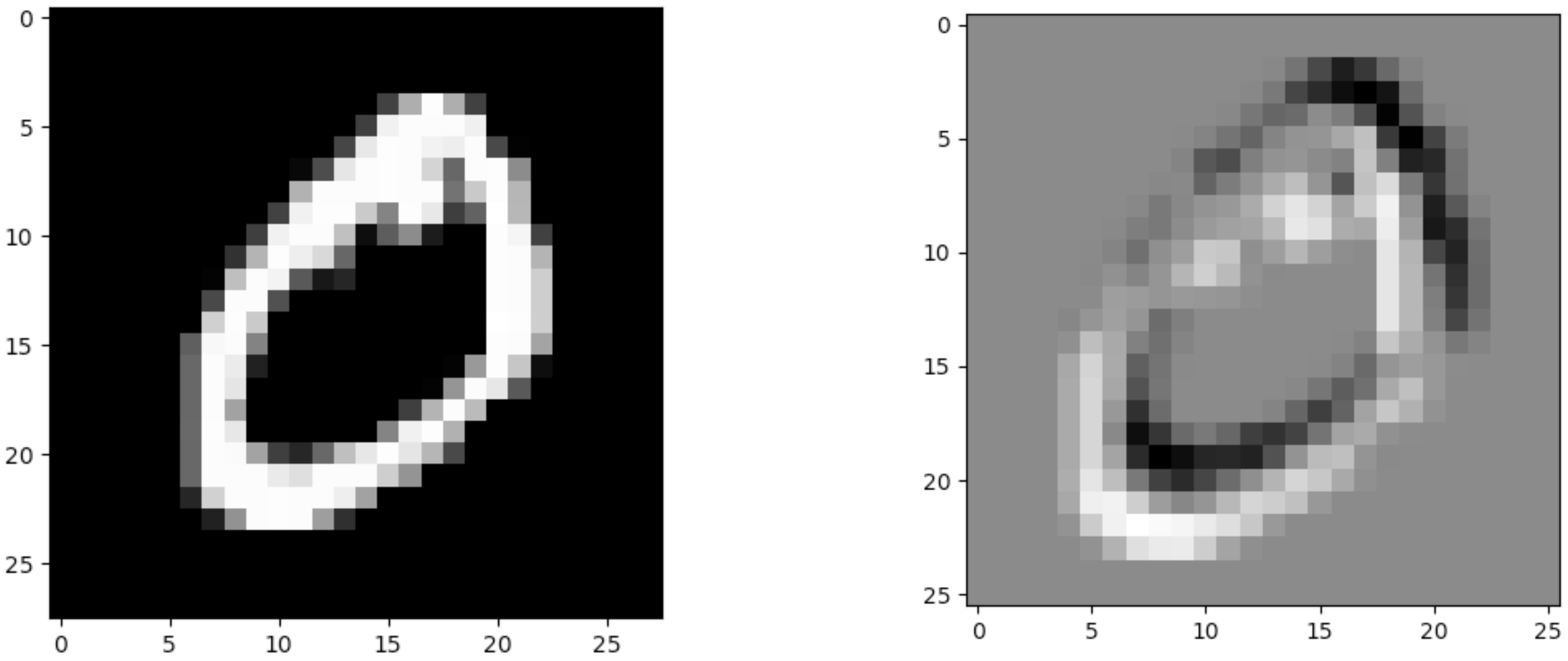
"Sliding a filter over an image"

Purpose: Extracts the high-level features to a feature map, such as edges, from the input image by applying a kernel / filter

Kernel = Matrix of size NxN that moves over the input image and performs dot product with the input image sub-regions (weights set randomly)

$$\begin{bmatrix} 7 & 2 & 3 & 3 & 8 \\ 4 & 5 & 3 & 8 & 4 \\ 3 & 3 & 2 & 8 & 4 \\ 2 & 8 & 7 & 2 & 7 \\ 5 & 4 & 4 & 5 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 6 & & \\ & & \\ & & \end{bmatrix}$$

7x1+4x1+3x1+
2x0+5x0+3x0+
3x-1+3x-1+2x-1
= 6



0
5
10
15
20
25

After convolution, of an image size of M x M and a kernel of size N x N, the resulting dimension is (M - N) + 1

E.g. Convolution of 28x28 Image, 3x3 kernel, and “y” filters results in (28 - 3) + 1 => (y , 24 , 24) feature map

Forward Propagation

```
output = np.zeros((self.num_filters, convRow, convColumn))

for i in range (self.num_filters):
    for x in range(convRow):
        for y in range (convColumn):
            for z in range(self.kernel_size):
                for v in range (self.kernel_size):
                    output[i, x, y] += input[x + z, y + v] * currentFilters[i, z, v]
```

Input = (1 , 28, 28)
Kernel Size = (3 , 3)
Output = (num_filters, (28 - 3) +1, (28 - 3)+1)

Back Propagation

```
    for x in range (0, self.kernel_size):
        for y in range (0, self.kernel_size):
            for z in range (0, self.convolve_size):
                for v in range (0, self.convolve_size):
                    kernel_gradient[i, x, y] += self.input[x + z, y + v] * output_error[i, z, v]

# Update Filters with Learning Rate
self.kernel -= np.array(kernel_gradient) * learning_rate
```

1. Filter Update

Perform convolution between the original input into Convolutional layer and loss gradient from previous layer to get “update_values”

Then, multiply “update_values” with the learning rate and update the current filter values

$$\begin{bmatrix} \frac{\partial L}{\partial F_{11}} & \frac{\partial L}{\partial F_{12}} \\ \frac{\partial L}{\partial F_{21}} & \frac{\partial L}{\partial F_{22}} \end{bmatrix} = \text{Convolution}\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} \right)$$

where

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \text{Input } X \qquad \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} = \frac{\partial L}{\partial O}$$

Loss gradient from previous layer

2. Gradient

Perform full convolution between the passed loss gradient and current filter to get "new_gradient"

Then, pass "new_gradient" into the next layer in back propagation

Backpropagation in a Convolutional Layer of a CNN

Finding the gradients:

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input } X \text{ , Loss gradient } \frac{\partial L}{\partial O}\right)$$

$$\frac{\partial L}{\partial X} = \begin{array}{c}\text{Full}\\\text{Convolution}\end{array}\left(\begin{array}{c}180^{\circ}\text{rotated}\\\text{Filter } F\end{array} , \begin{array}{c}\text{Loss}\\\text{Gradient}\end{array} \frac{\partial L}{\partial O}\right)$$

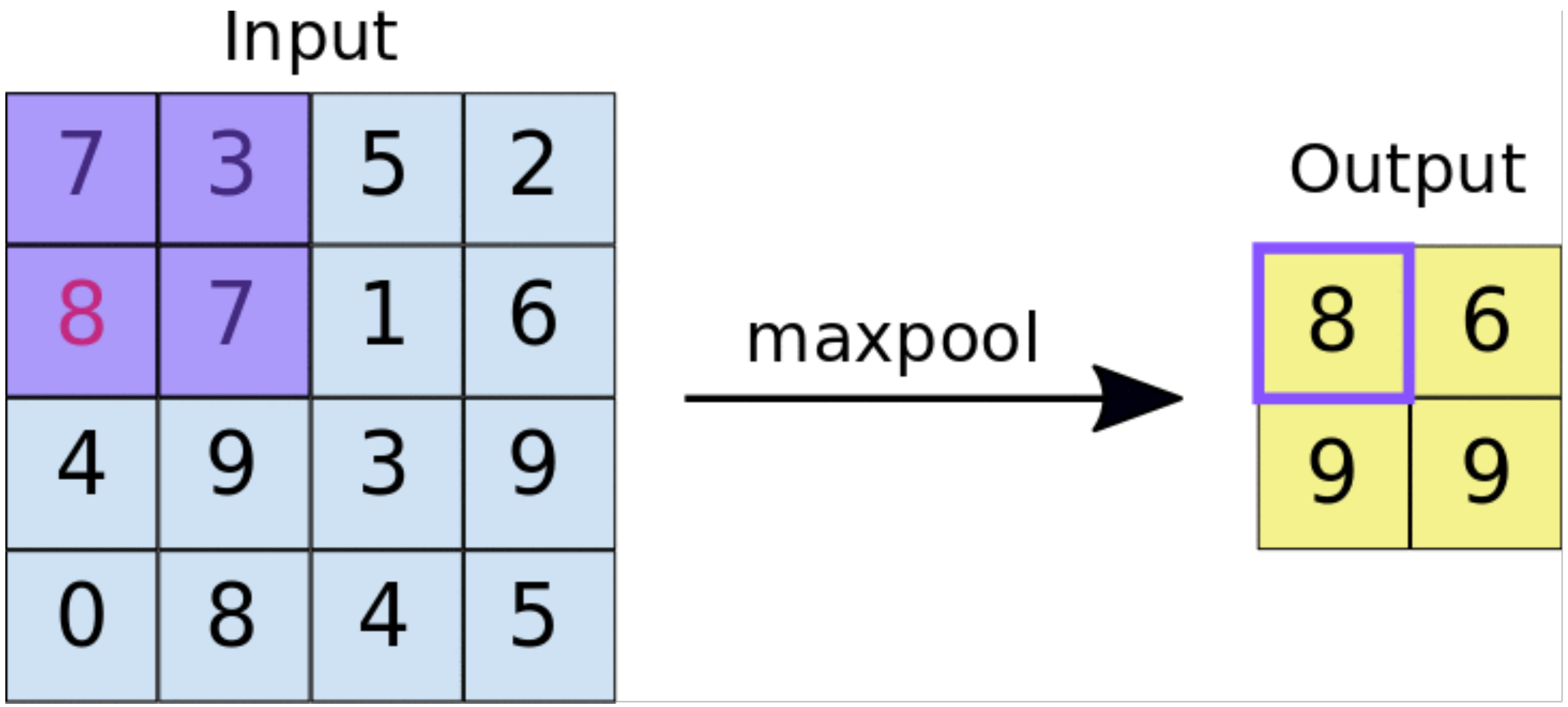
Max Pooling Layer

Purpose: Used to reduce the dimensions of feature maps and reduce the amount of computation required by the network

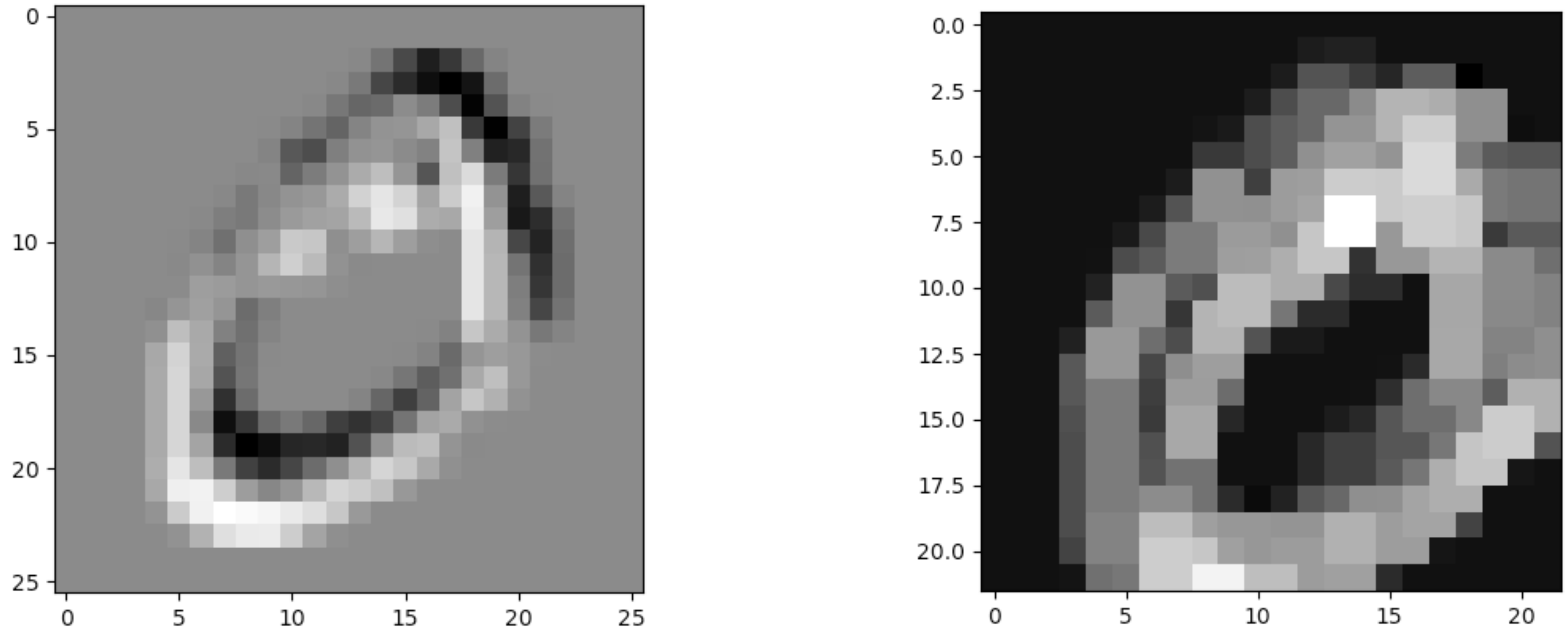
Moves over the feature maps and selects the maximum element from the NxN filter (pooling region)

Pooling Size = NxN window that moves across the subregions of the feature maps

Strides = Integer that specifies how far the pooling window moves on each step



Pooling Size = 2 x 2
Strides = 2


0
5
10
15
20
25
0
5
10
15
20
25
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20

Forward Propagation

```
output = np.zeros((num_filters, self.output_size, self.output_size))

for i in range (0, num_filters):
    for x in range (0, tempOutputSize):
        for y in range (0, tempOutputSize):
            tempArray = input[i, x*self.strides:(x*self.strides)+self.pool_size, y*self.
            output[i, x, y] = np.max(tempArray)
```

x*self.strides : (x*self.strides)+self.pool_size
y*self.strides : (y*self.strides)+self.pool_size

MaxPooling layer basically **halves** the feature maps

Back Propagation

```
input_gradient = np.zeros(self.input_shape)

for i in range (0, self.num_filters):
    y_coord = 0
    for x in range (0, self.output_size):
        x_coord = 0
        for y in range (0, self.output_size):
            input_sub = self.input[i, x*self.strides:(x*self.strides)+self.pool_size, y*self.strides:(y*self.strides)+self.pool_size]
            max = np.max(input_sub)
            result = unravel_index(input_sub.argmax(), input_sub.shape)
            max_x = result[0]
            max_y = result[1]
            input_gradient[i, x*self.strides:(x*self.strides)+self.pool_size, y*self.strides:(y*self.strides)+self.pool_size][max_x, max_y] = max
return input_gradient
```

Finds the maximum element from the array within the pooling window and returns an array filled with zeros, except the maximum elements within each window

1	2	4	5
7	9	20	3

1	2
7	9

4	5
20	3

Pooling Size = 2

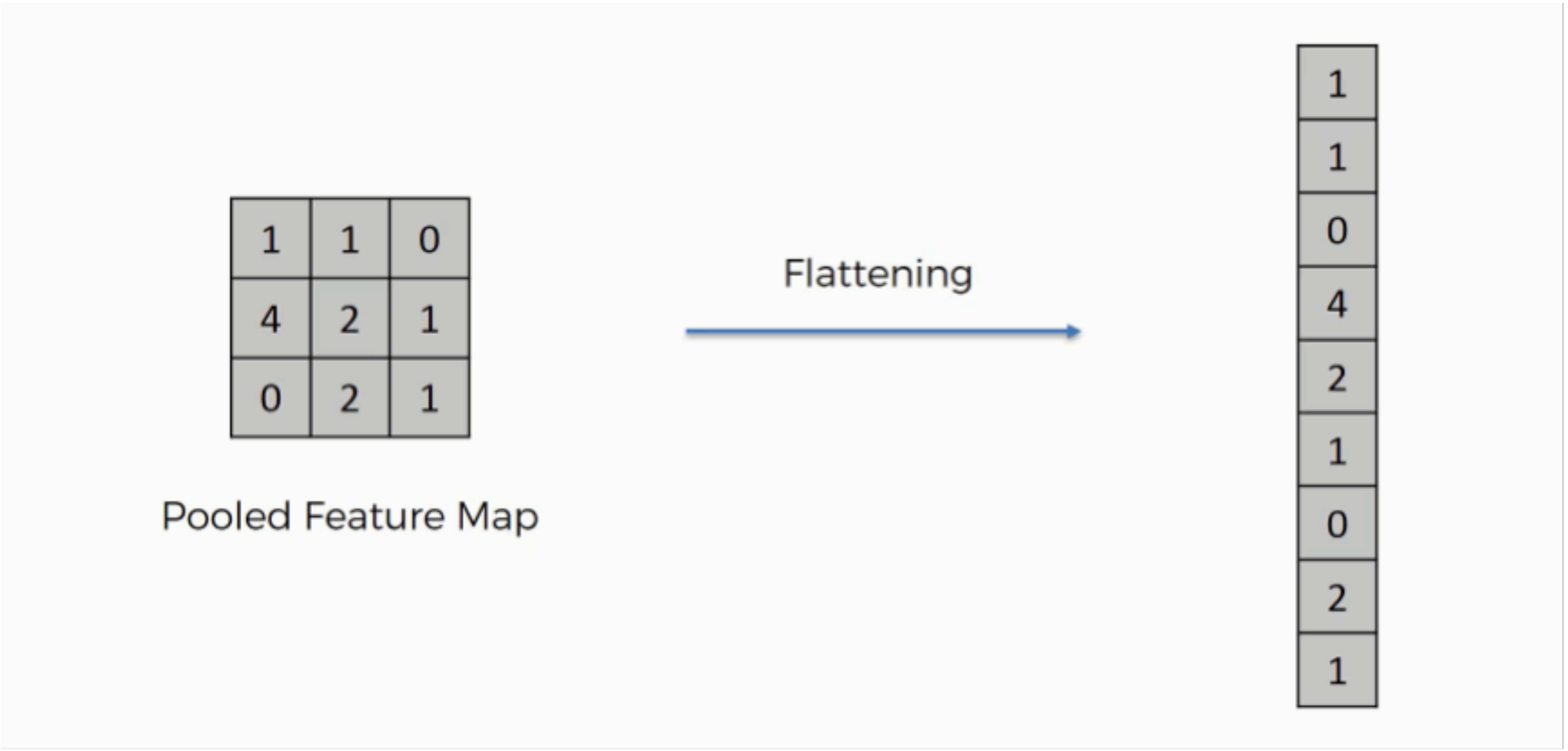
Return:

0	0	0	0
0	9	20	0

Flattening Layer

Purpose: Properly formats the pooled feature maps to be inserted in the dense layer

Takes dimension (X, Y, Z) pooled feature maps, and flattens them to an (1, (X*Y*Z)) flattened layer



Input Size = (1, 3, 3)

Flattened = (1 , (1*3*3)) => (1, 9)

Forward Propagation

```
self.before_flattened_shape = input.shape
output = np.array([input.flatten()])
```

Back Propagation

```
before_flattened = input.reshape(self.before_flattened_shape)
return before_flattened
```

Returns the flattened array back to its original shape

E.g. (1, 18) => (2, 3, 3)

Activation & Loss Functions

Binary Classification CNN:

"Classifying inputs into two categories"

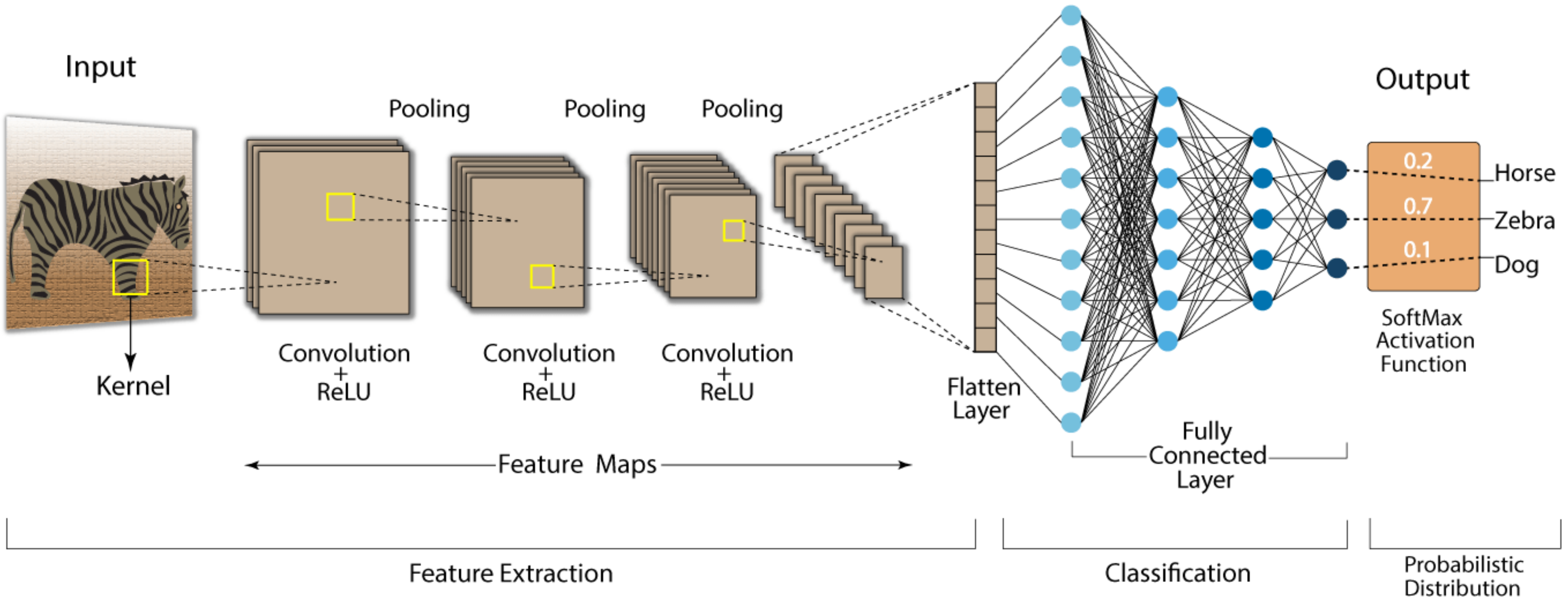
1. Activation = Sigmoid
2. Loss = Log Loss

Multi-Class Classification CNN:

"Classifying inputs into multiple categories"

1. Activation = Softmax
2. Loss = Categorical Cross Entropy

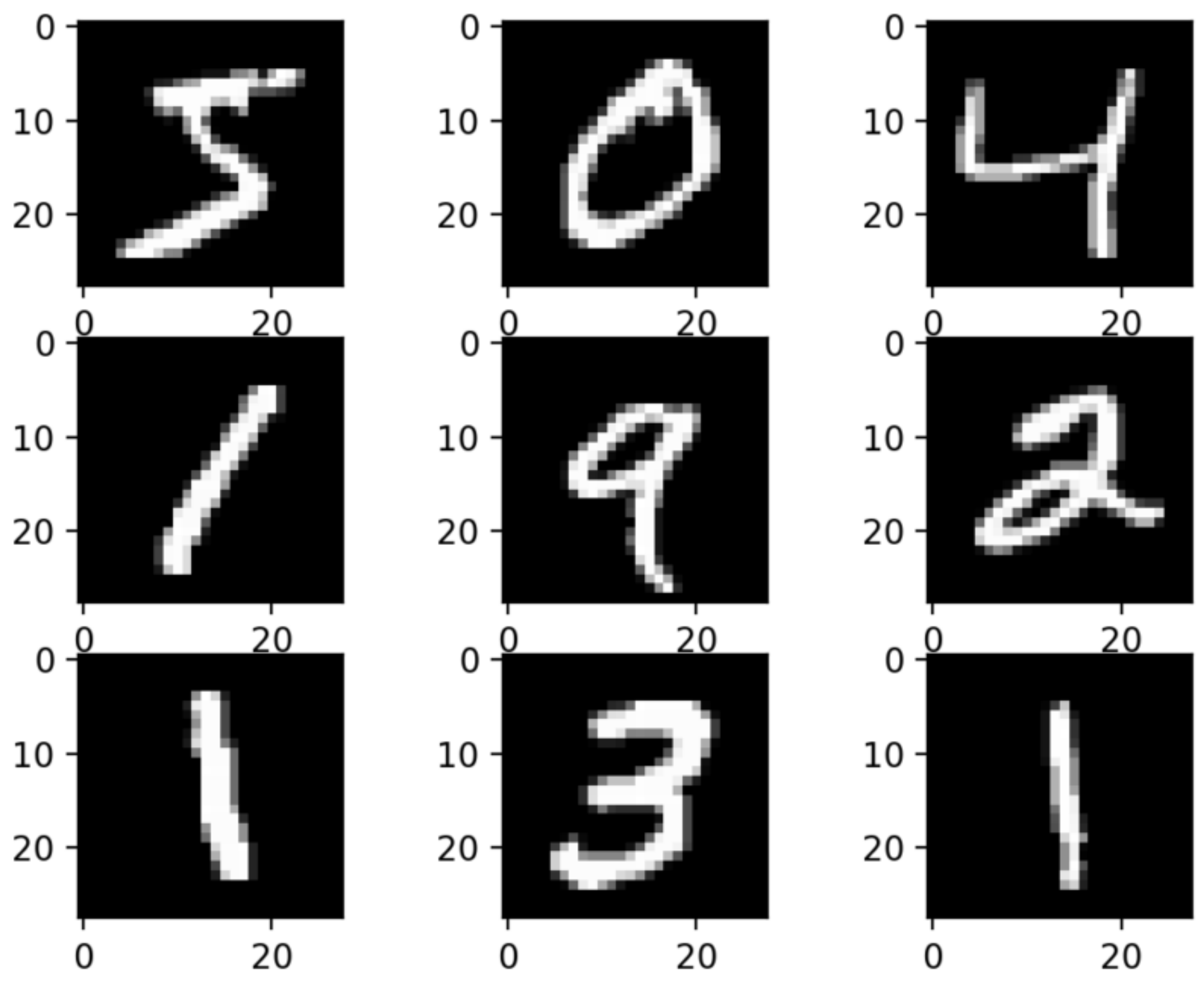
Convolution Neural Network (CNN)



Example

Data: MNIST Handwritten Digit Dataset

Implemented a Binary Classification CNN that identifies images (28 x 28) of 0 and 1 from the dataset



CNN Model (Binary Classification)

```
network = GWUNetwork()
network.add(Conv2D(input_size=28, kernel_size=3))
network.add(MaxPooling2D(pool_size=2, strides=2, input_size=23))
network.add(Flatten(input_size=(11,11)))
network.add(Dense(100, add_bias=False, activation='relu'))
network.add(Dense(2, add_bias=False, activation=‘sigmoid'))
network.compile(loss=‘log_loss’, lr=0.001)
network.fit(x_train_subset, y_train_subset, epochs=2)
results = network.predict(x_test_subset)
```

```
# Import MNIST data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train_subset = []
y_train_subset = []
x_test_subset = []
y_test_subset = []


# Separate images into two classes
for i in range (0, 1500):
    if (y_train[i] == 0 or y_train[i] == 1):
        x_train_subset.append(x_train[i])
        y_train_subset.append(y_train[i])

for i in range (0, 150):
    if (y_test[i] == 0 or y_test[i] == 1):
        x_test_subset.append(x_test[i])
        y_test_subset.append(y_test[I])
```

```
x_train_subset = np.array(x_train_subset[:200])
x_test_subset = np.array(x_test_subset[:10])
y_train_subset = np.array(y_train_subset[:200])
# Use keras to_categorical for binary classification problem
y_train_subset = np.array(tf.keras.utils.to_categorical(y_train_subset, num_classes=2))
y_test_subset = np.array(y_test_subset[:10])

# Normalize data
x_train_subset = x_train_subset.reshape(x_train_subset.shape[0], 28,
28).astype('float32')
x_test_subset = x_test_subset.reshape(x_test_subset.shape[0], 28, 28).astype('float32')
x_train_subset /= 255.0
x_test_subset /= 255.0
```

Output: Size 2 array that outputs the probability of the image being a “0” or a “1”

Example:

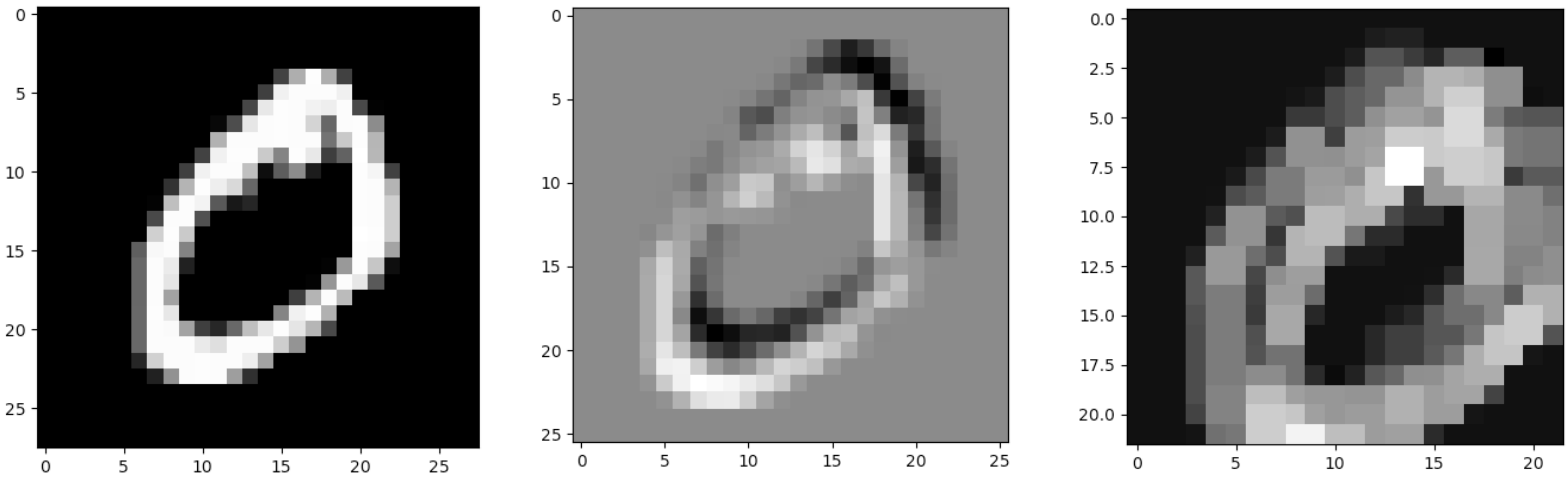
y_true [0, 1]

Output [0.2583, 0. 7417]

Output

```
[array([[0.5, 0.5]]), array([[0.52618988, 0.52502099]]), array([[0.5, 0.5]]), array([[0.5, 0.5]]),
 array([[0.48784266, 0.54074427]]), array([[0.5, 0.5]]), array([[0.57479947, 0.53996119]]), array(
[[0.51681995, 0.51206615]]), array([[0.5059675 , 0.55003086]]), array([[0.5, 0.5]])]

Actual: [1, 0, 1, 0, 0, 1, 0, 0, 1, 1]
Predic: [1, 0, 1, 1, 1, 1, 0, 0, 1, 1]
```


0
5
10
15
20
25
0
5
10
15
20
25
0
5
10
15
20
25
0
5
10
15
20
25
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
5
10
15
20

Final Thoughts:

1. Learned a lot about the details of how a CNN works

2. Most difficult part personally was the back propagation of the convolutional layer and understanding the math behind it. Current implementation needs tweaking

References:

https://www.youtube.com/watch?v=Lakz2MoHy6o

https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509

https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/

Thank You.