**Chapter 7**

**Problem 7: Cyclist Detection**

As one of the most interesting applications of machine learning, computer vision consists of three main parts. The first one is classification, where a model is used to determine the likelihood of the existence of an object in an image. Secondly, the model can detect an object's position in addition to its probability of existing. In the object detection part, the model can detect multiple objects in the images along with their locations and probability of existence. In this problem, we want to avoid collision with cyclists by detecting their position in the image.

**About Dataset**

This problem consists of two groups of data: images and labels. The images have a height of 1024 pixels and a width of 2048 pixels. They may consist of one cyclist or more. There is one label file for each individual image indicating the position of the cyclists' bounding boxes. The first 2 numbers in label file indicate the position of the center of the bounding box relative to the image's height and width, and the last 2 numbers indicate the height and width of the bounding box.

****

**­­**

**Introduction**

Unlike typical computer programs, Machine Learning techniques will literally learn from data. Machine Learning algorithms can actually find insights and data even if they are specifically instructed on what to look for in that data, and that's what separates a Machine Learning algorithm from a typical computer program. You're just giving the Machine Learning algorithm a set of rules to follow. Instead of actually telling it what to look for, it will find the insights on its own.

**Why do we use Machine Learning to solve mechanical problems?**

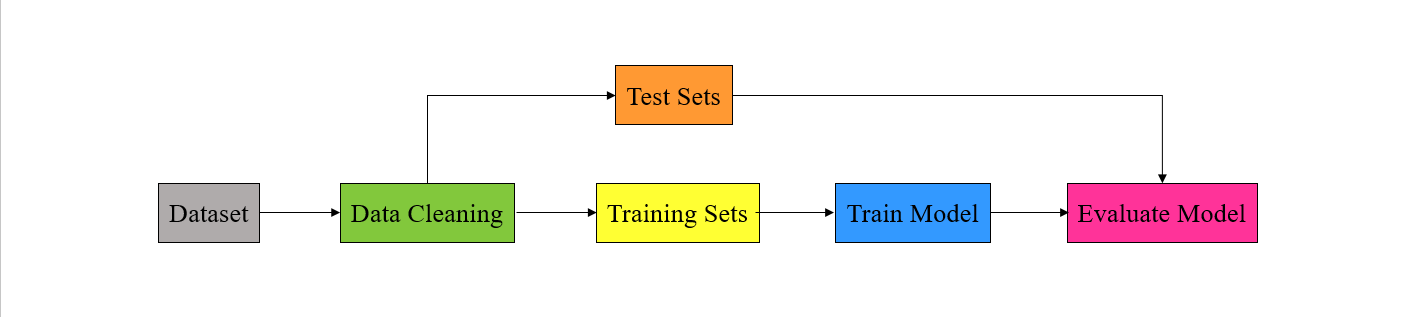
Machine learning is a method for predicting behavior or classifying data sets that, unlike common methods in mechanics, instead of being based on an intuitive model, uses a mathematical model and arbitrary functions to describe and predict the behavior of systems. In other words, machine learning is a search in the space of algorithms and parameters in such a way that it infers a model from the data (data-driven model) and based on that, predicts or categorizes the studied system. For example, using neural networks as one of the methods of traditional machines, I can perform a set of inputs based on an arbitrary number of intermediate hidden layers to the output image results. In the input and output data that are entered quantities, the middle layers do not necessarily have meanings and other adverbial expressions on them. For this reason, I can choose the number of intermediate layers and the number of nodes in each layer at will, and this approach is completely acceptable in the input to the output image. In particular, the relationship between the data is so complex that the models created with a limited number of adjustable settings express this relationship with sufficient accuracy, the efficiency of the methods using machines can be very important.

**Object Detection Problems**

As part of machine learning at a high level, supervised and unsupervised learning methods are used. Supervised learning refers to labeling historical data and using it to inform our models. This label or something we wish to predict is called the target. In supervised learning, there is a specific goal (target) for the past information, whereas in unsupervised learning, there is no specific goal. We use classification and regression in supervised learning. In classification problems, the goal is to determine which category it belongs to. It is usually True or False, but there may be multiple categories as well. In regression problems, a numerical value is our target. An object detection problem is a classification problem, in addition to predict the object’s position in the image.

**What are we going to do in this Chapter?**

We have a dataset from the Kaggle website and then clean that data. After that, we split our data into two groups (train and test). Then we train our model on the training set and after that, we evaluate our model with the test set we have.



***Figure 3-3.***

**TensorFlow**

This open-source library uses tensors to support machine learning and numerical computation. TensorFlow can be developed even by people without any programming experience and can be used in a variety of programming languages, such as Python, JavaScript, and C++.

A major application of TensorFlow is the construction of neural networks such as CNNs and RNNs. TensorFlow, since it is based on graphs, can be executed on multiple processors such as GPUs much more efficiently.



**TensorFlow Model**

The Keras main library provides two basic classes, Model and Input, which are required for every Keras model:   
  
**Input:** The input layer, as its name suggests, should be the first layer to receive input. There is only one parameter, input shape, which indicates the shape of input tensors except the batch size.  
   
**Model:** This class is the final step in the creation of a Keras model. It defines the inputs and outputs required for the final model to be created.



**TensorFlow Layers**

The Keras models, whether sequential or functional, require layers to be defined. Each layer has various parameters that need to be defined. The following layers are used in this notebook:  
  
**Flatten:** Using this layer, every multidimensional tensor is converted to a one-dimensional tensor.  
   
**MaxPool2D:** Some features in image are not necessary to process. In order to reduce the calculations, after convolution layer, max pool layer pool out important features of the image. For this layer, only the filter size needs to be defined.  
  
**Dense**: The dense layer indicates a simple neural network with neurons fully connected to one another. Two parameters must be set for this layer, including the number of neurons and the activation function.  
  
**Reshape**: It is possible to change the input shape to a custom shape by using this layer.  
  
**Rescale**: The layer multiplies each value of the input tensor by a number.  
  
**Conv2D:** By using this layer, hidden features in images can be extracted by performing convolution operations on 2D tensors. Each convolution layer has its own set of parameters, which include the number of filters, filter size, padding, etc.  
  
**Batch Normalization:** It scales input values between 0 and 1 in order to reduce computation load.



**Keras Loss**

We use two of Keras' loss functions to calculate the loss between ground truth labels and predictions:

**Binary Cross Entropy:** This method calculates the logarithmic loss between labels and predictions, which is explained in more details below.  
   
**Mean Squared Error:** By summing over squares the error between predictions and labels, this function computes loss.



**Matplotlib**

The Matplotlib package provides static and dynamic visualization in Python. Pyplot is a Matplotlib module that creates a separate figure and makes changes to this figure to generate a plot.



**NumPy**

The NumPy library contains functions that deal with arrays, which have the advantages of being processed faster than lists, consuming less memory, and reserving a static portion of memory.



**OS**

The Python OS module contains libraries and functions that can be accessed from the underlying operating system to manipulate directories, create and change directories, access files in directories, etc. It is essential when dealing with files, such as reading or writing them.



**Glob**

In Python, the glob module returns a list of all files in a directory with a specific pattern in their name. It is frequently used to access local files.

****

**Keras Backend**

In TensorFlow, all operations are performed on graphs, and the model must understand any relationships between variables and operations. Keras's backend is a useful tool for defining custom loss functions, metrics, and custom arithmetic operations.



**H5PY**

This package is useful for storing or manipulating the HDF5 binary format files.

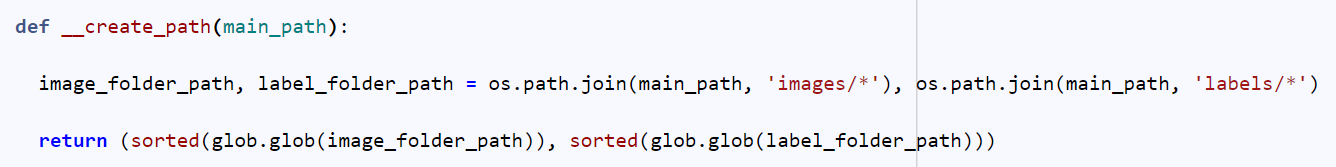
****

**Step 1. Preprocessing**

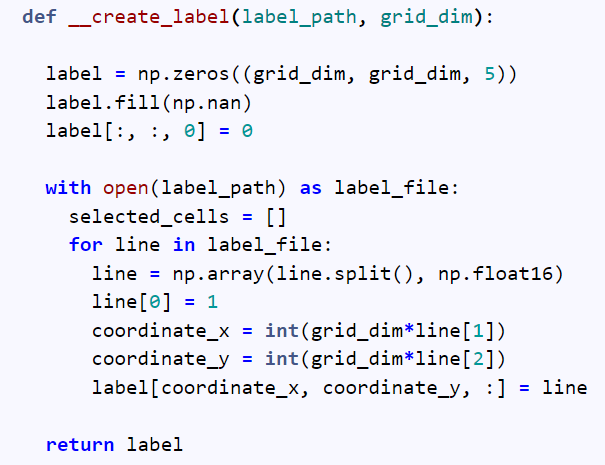
This class contains nine methods for preparing data for training. The first method is the initialization of class instances, as with any other class in Python. Four instances of this class are available, including the custom height and width of the input image, dimensions of the grid of the model, and dataset path.



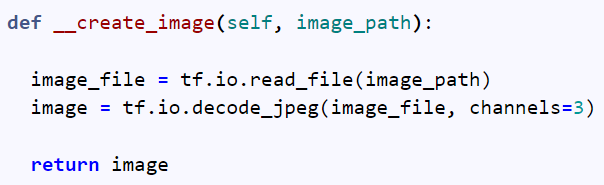
The second method simply takes the dataset path as input and searches for all images and labels in the separate labels and images folder. After finding all of the files, sort them in order to avoid further errors.



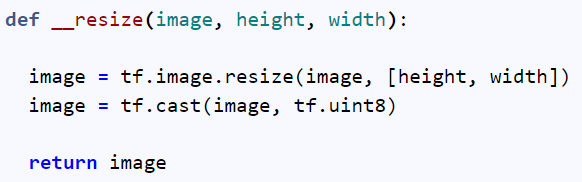
By creating a 3D array of nan with a size of grid dimension by grid dimension by five, and with a last layer assigned to zero, the next method creates a label based on the path and grid dimension. The following step is to fill the column of each cell of the grid containing the cyclist with label values. (The reasons for this type of labeling will be discussed in the model section).



The fourth method loads the image by obtaining the image path, reading the image file, and decoding it to a tensor.

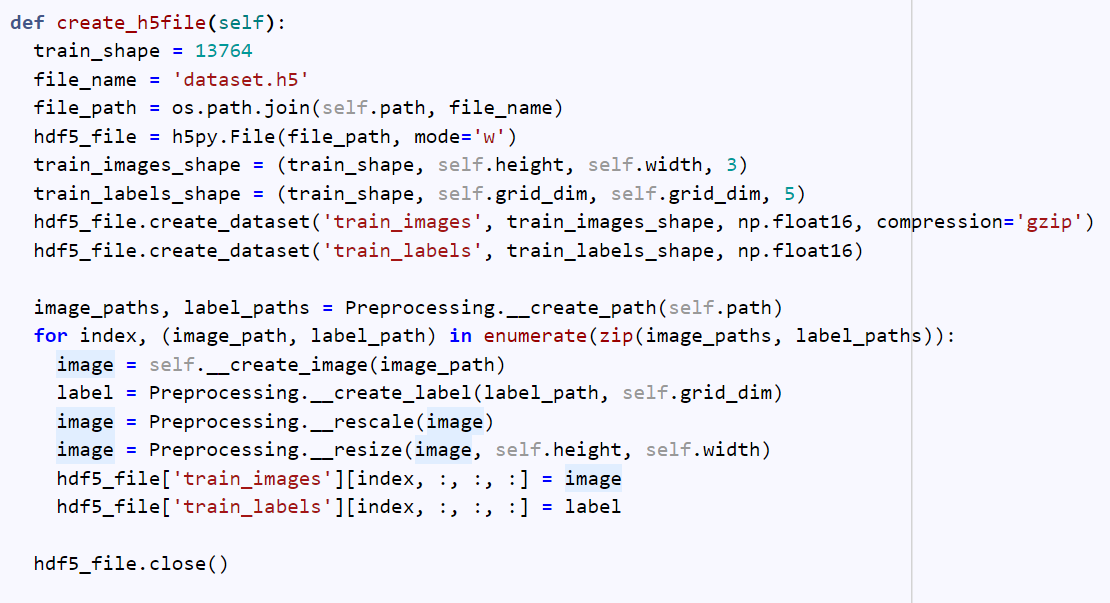


In the next method, the image is resized to a custom size. However, some pixel values might be floats. In order to avoid further errors, the type of pixels must be changed to integer.



We need to store the dataset in more compact formats to consume less memory. HDF5 and TFR are two file formats that consume less memory when read from the disk. Additionally, we can use TensorFlow dataset generators to simultaneously read these two types of files during training and save a lot of memory. These two methods create updated datasets in the following formats:

1. Create HDF5 dataset: It is necessary to iterate through each and every example of files in order to save the existing dataset to HDF5 format. We must first identify the number of files and the name of the dataset, then define the path to the desired file, then open this path by using the writing mode in order to write it into the desired directory. To save data into a predefined HDF5 file, the number of data, each part of the data, in this example an image and label, the type of data, and compression type must be determined. The final step is to iterate through all of the examples and load each image and label separately into their predefined positions.



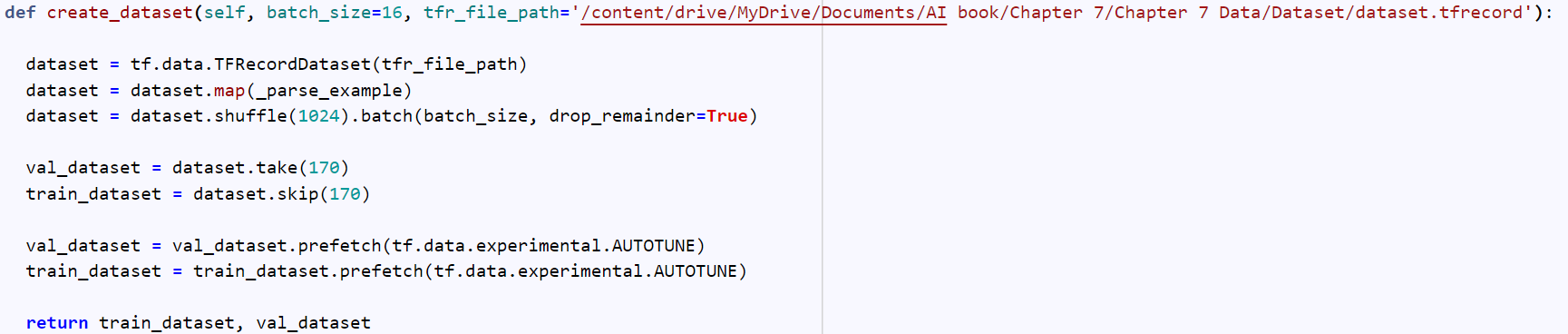
1. Create TFR dataset: It is a binary format for storing data optimized for TensorFlow. As before, we must iterate through every single example of data to save it to a separate TFR file. The dataset file paths are first loaded into lists, then a TFR file is created and ready to be written. This is followed by looping over all data paths and loading images and labels. To prepare the data for storage in TFR format, it is necessary to serialize it to strings. As soon as we have defined two features (label and image) and serialized each example to a string, the data is ready to be stored in a TFR file.

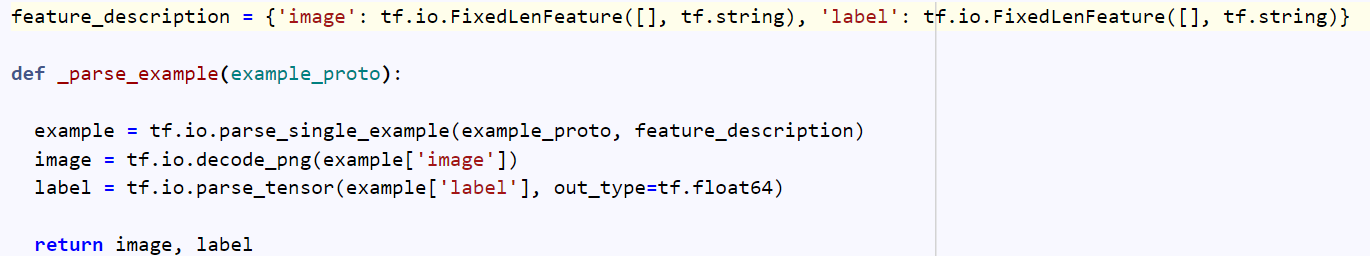


A TFR file format has some advantages over HDF5. First, it is optimized for use with TensorFlow. Second, it consumes less memory and time, which makes it more appropriate for this application.

**Step 2. Load Data**

A TFR file is used to load as a dataset, as we discussed earlier. The last method of the Preprocessing class loads the TRF file. TFRecordDataset is used to open the stored TFR file in a dataset variable. The parse example function is mapped to the dataset file after the dataset variable has been defined. According to its name, the Parse Example function iterates through each example and extracts its image and label from each. This is done by defining descriptions of existing features. Having parsed the dataset through examples, we shuffled and batched the dataset to prevent further bias. The dataset must then be divided into two separate datasets, training and validation, and be prefetched in order to consume less memory during reading from disk.





**Step 6. Split Training and Testing Datasets**

Data is the key to training all machine learning models, and parameters are changed based on the available data. Therefore, more data results in better results. Using gradient descent, the model calculates training data loss and attempts to decrease the loss value. However, does a decreasing training data loss mean better results on real-world data? Not necessarily. Real-world data often include data that the model has never encountered previously. Thus, for evaluating results, loss values on training data are not sufficient, and some additional data, such as validation, must be collected.

It is very critical to determine the number and size of validation data, and this selection is highly dependent upon the number and size of data. Despite the fact that it is not a principle, it is acceptable to split 20 percent of the data between validation and training. The distribution of validation data must be the same as the distribution of actual data in order to achieve better results.

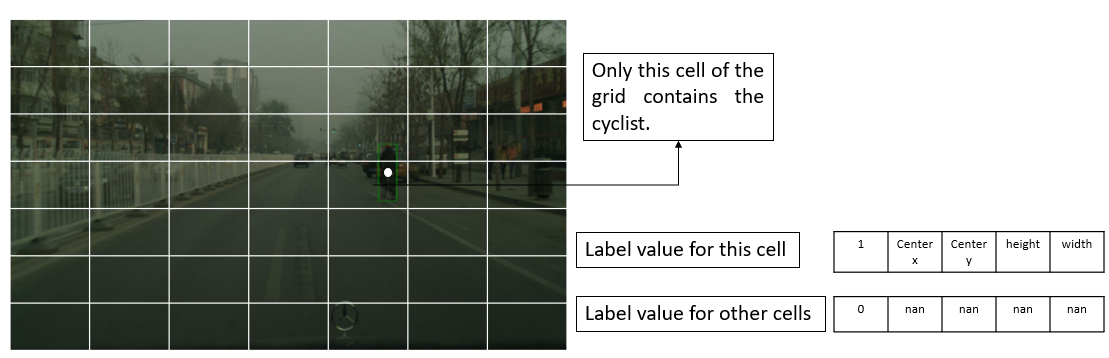
To divide a dataset into training and validation, I use the take and skip methods. In this example, 80 percent of the data is set aside for training and 20 percent for validation. The Take method accepts a number that determines the number of batches to split for data. Skip method takes the same number in order to ignore the first batches as much as you specify.

**Batch**

It is much more computationally efficient to compute loss on a small portion of data and calculate gradient descent based on that rather than computing loss on the whole data. A batch is a set of data that is used to calculate loss once a time and a batch size is the number of data in each batch. Batch size is the first element of the shape in Tensorflow and it is saved as a None object since it is independent of the model architecture.

**Step 7. Training**

In terms of object detection, YOLO is one of the most popular models because of its simplicity and speed. In essence, this model states that a human will look at an image and process it once. In this model, images are divided into grid cells and labeled according to whether the cells contain the object or not. For example, let us examine an image that only contains one cyclist. It is apparent that if this image is placed in a grid of 7 by 7, the cyclist is surrounded by two cells, but only one cell contains the cyclist's center. Therefore, any cell containing the cyclist center must be set to 1, but what about the cyclist's position in this particular cell? To assign the 4 numbers of determining the position of the bounding box, we must assign an array of length 5 for each cell that determines the probability of the existence of the cyclist, the center of the box coordinates, and the height and width of the box. If a cell contains a cyclist, the first number is set to 1 and the other four are filled with respective label values. If a cell does not contain a cyclist, the first value is set to 0 and the other four are filled with nan.

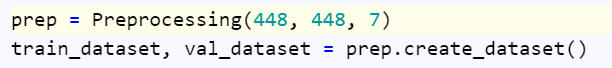


It is necessary to describe the model architecture of the YOLO model after explaining the method of labeling in it. The first layer of any model is the input layer, which is color images with 448 pixels in height and width. The YOLOv3 model has six convolutional sections, each with a variety of filters to detect numerous objects.

In order to train this huge model, a large dataset is required. There are many classes within this complex architecture. In the original YOLO models, many different types of objects are detected simultaneously, however, in this problem, we are only interested in detecting cyclists, so the original model is not needed. It is possible to benefit from the YOLO model in two ways.

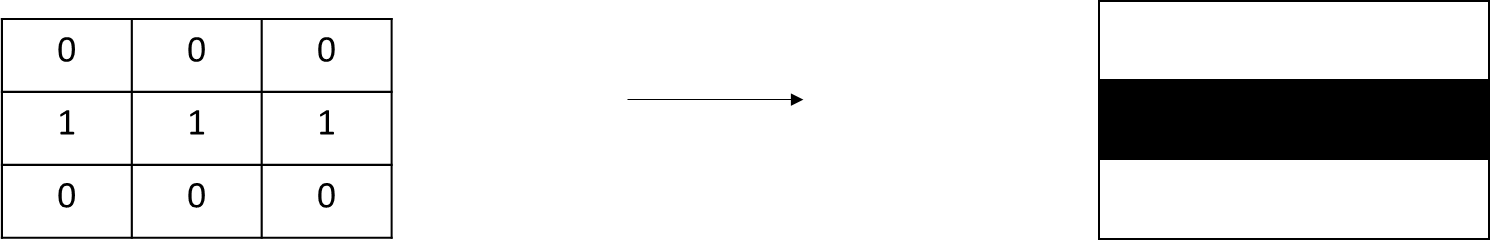
The first is by using the YOLO model as the base architecture and freezing some areas of the model, then fine-tuning the model's weights in order to improve detection.

Additionally, it is recommended that the YOLO model be used in a more compact version instead of the original model. Using the same general architecture with different hyperparameters, such as the number of filters, is helpful if you have sufficient data and wish to build the model from scratch. We proceed with the second method.

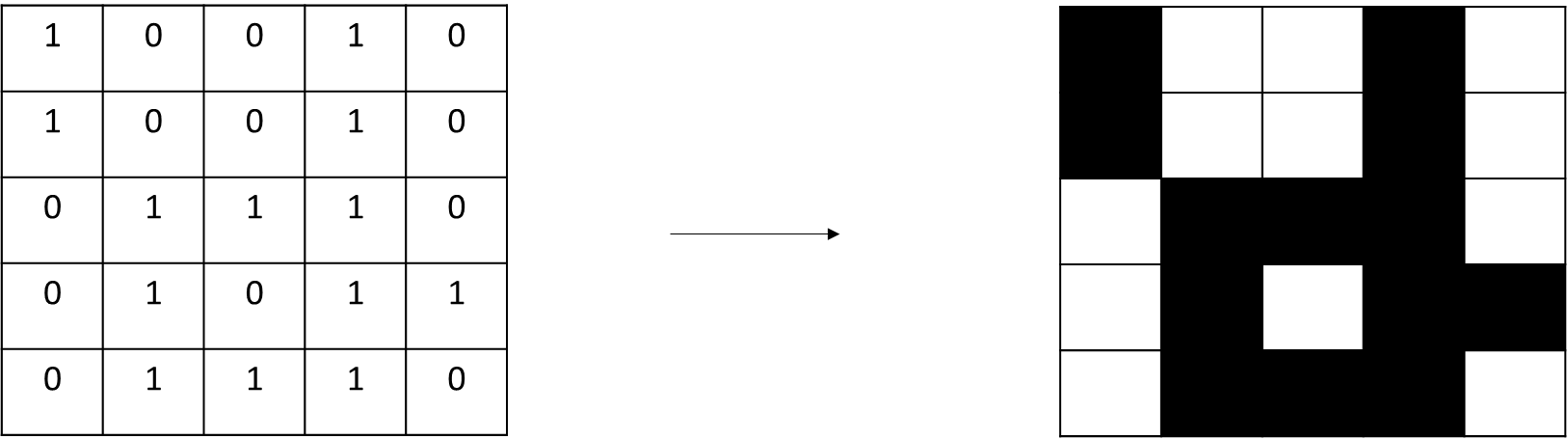


**Convolution**

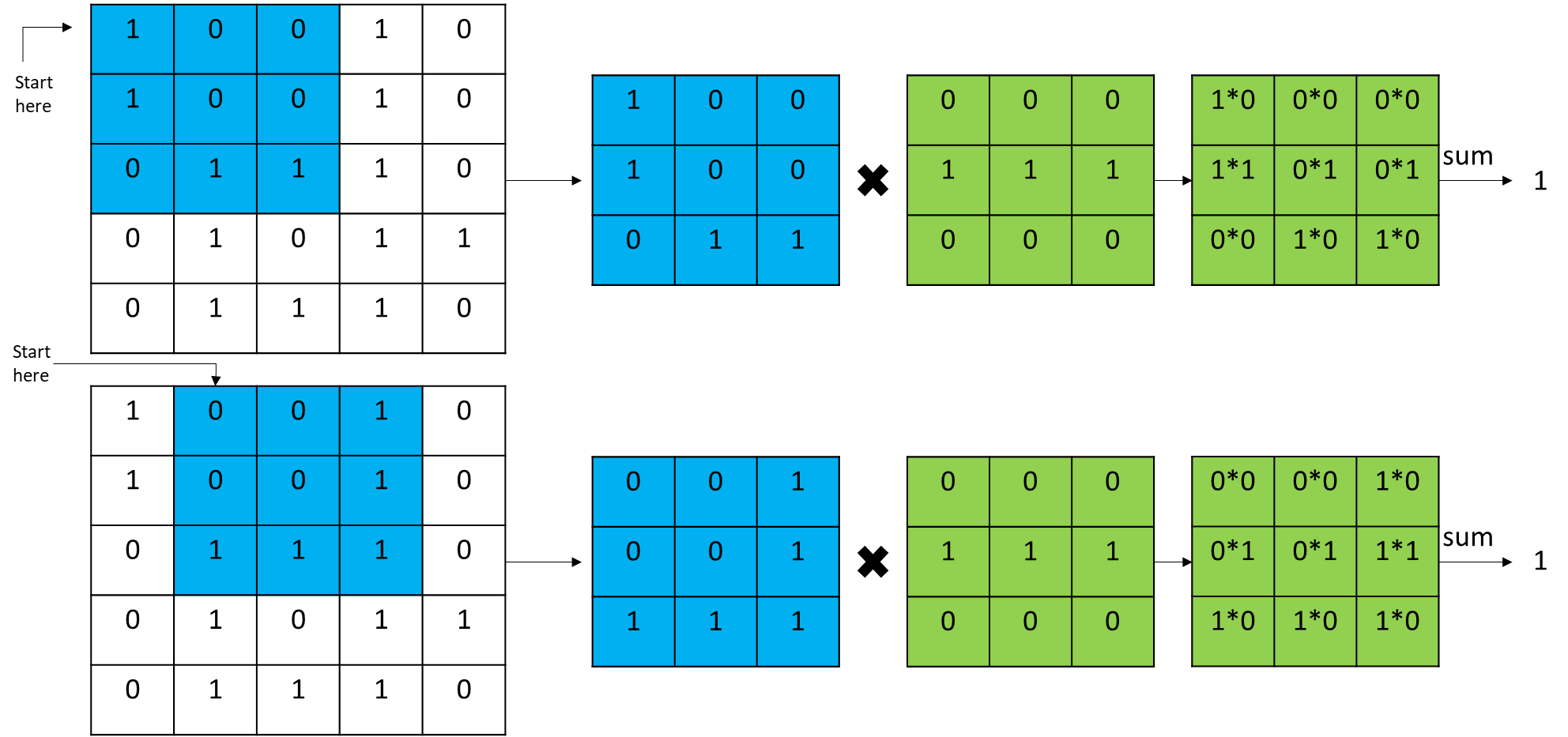
Convolution is an operation that is specifically designed to extract useful information from images. Essentially, this operation is based on human vision and how the brain detects specific patterns in images. In order to extract hidden features from an image, a special filter must be applied to the image. For example, this filter can be used to detect horizontal lines.



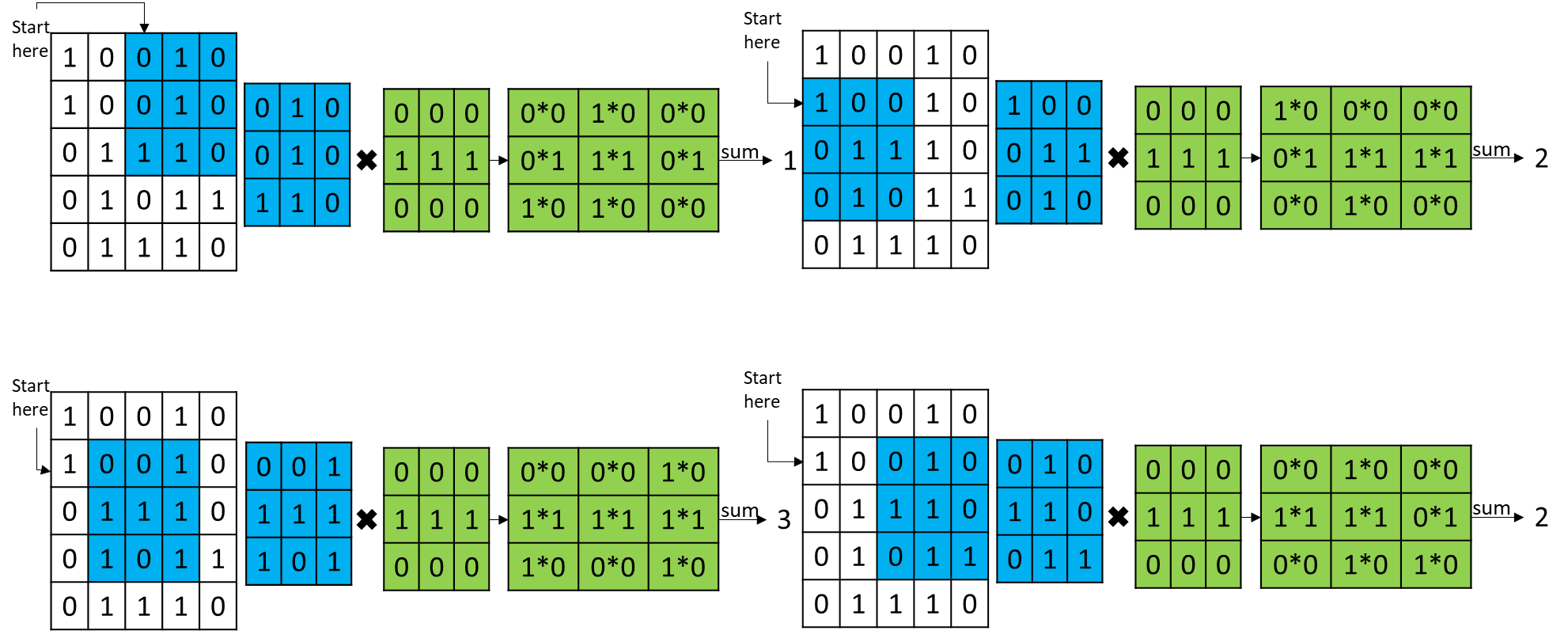
As can be seen, this filter has a horizontal line within itself. In order to clarify the convolution operation, let's mention an example and compute convolution on it.



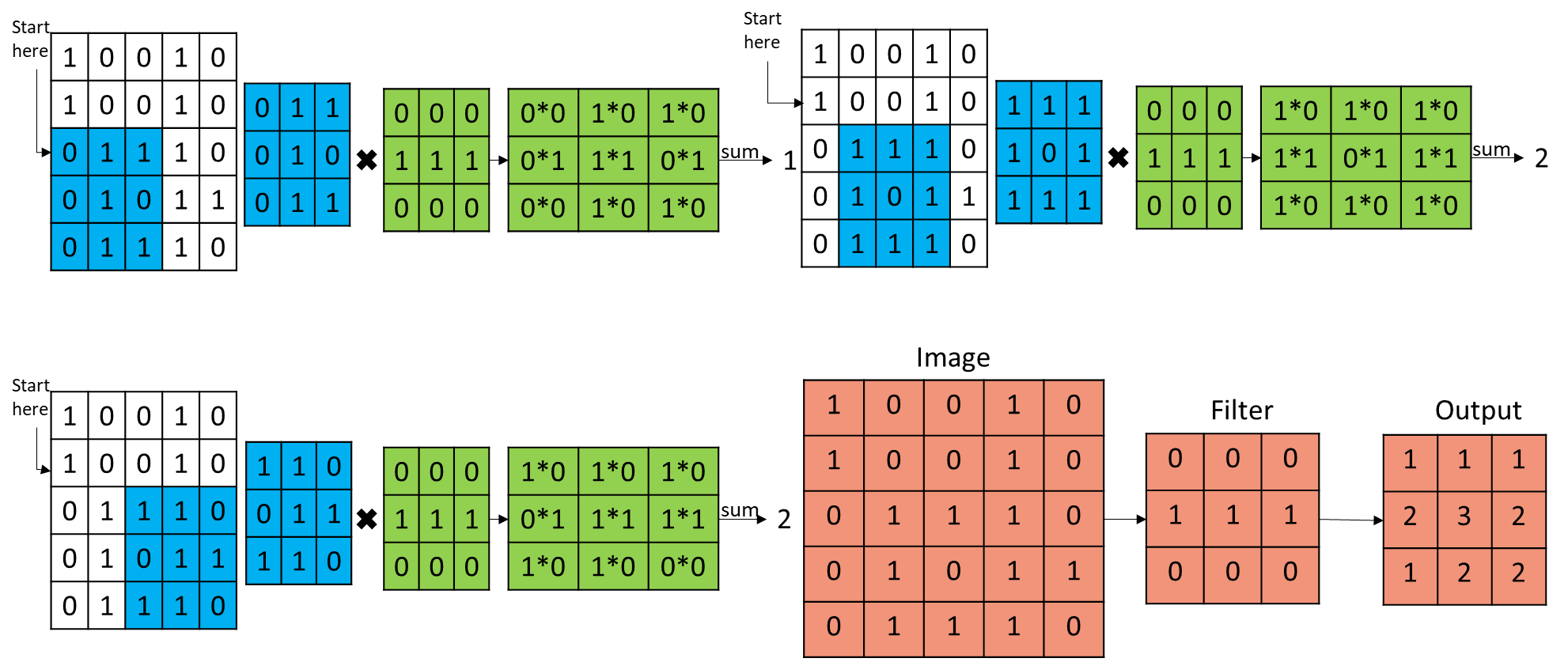
To perform a convolution operation, the elementwise product of the most upper left part of the image and filter is computed, then the next part of the image is computed, and so on.



The process continues until the end is reached.

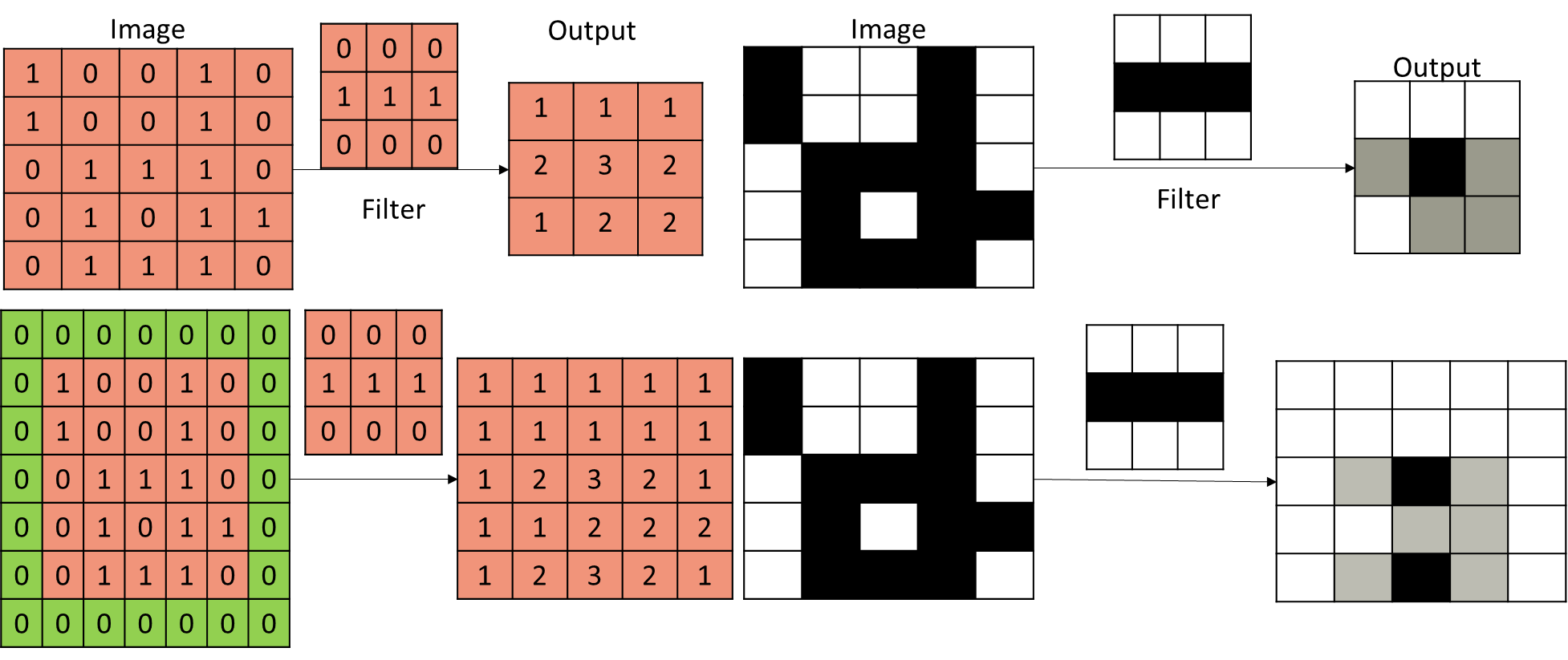
****

In the end, all nine computed elements are positioned in their respective positions.

****

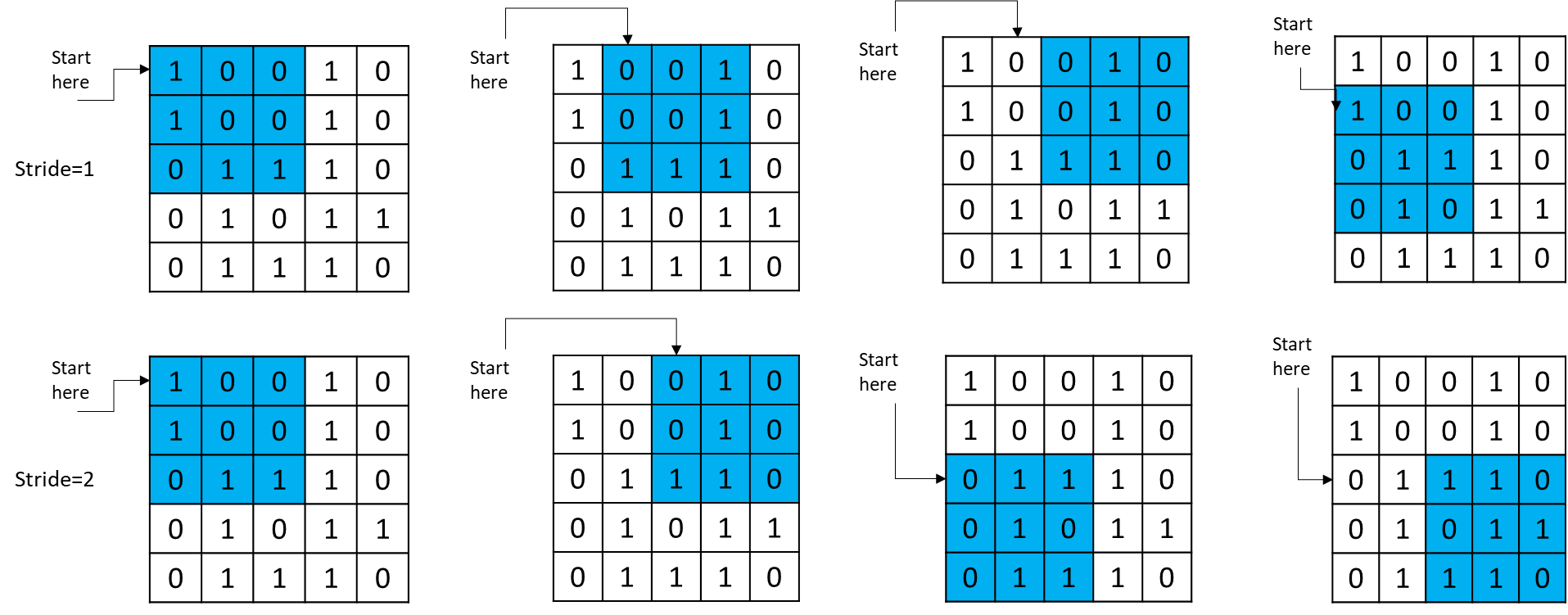
**Padding**

Clearly, the filter detects only the middle horizontal line (the middle element of the output is black). The horizontal line on the edge is not detected by the filter. The dimensions of the image are increased by two and empty elements are replaced by zero in order to detect features on the edges.



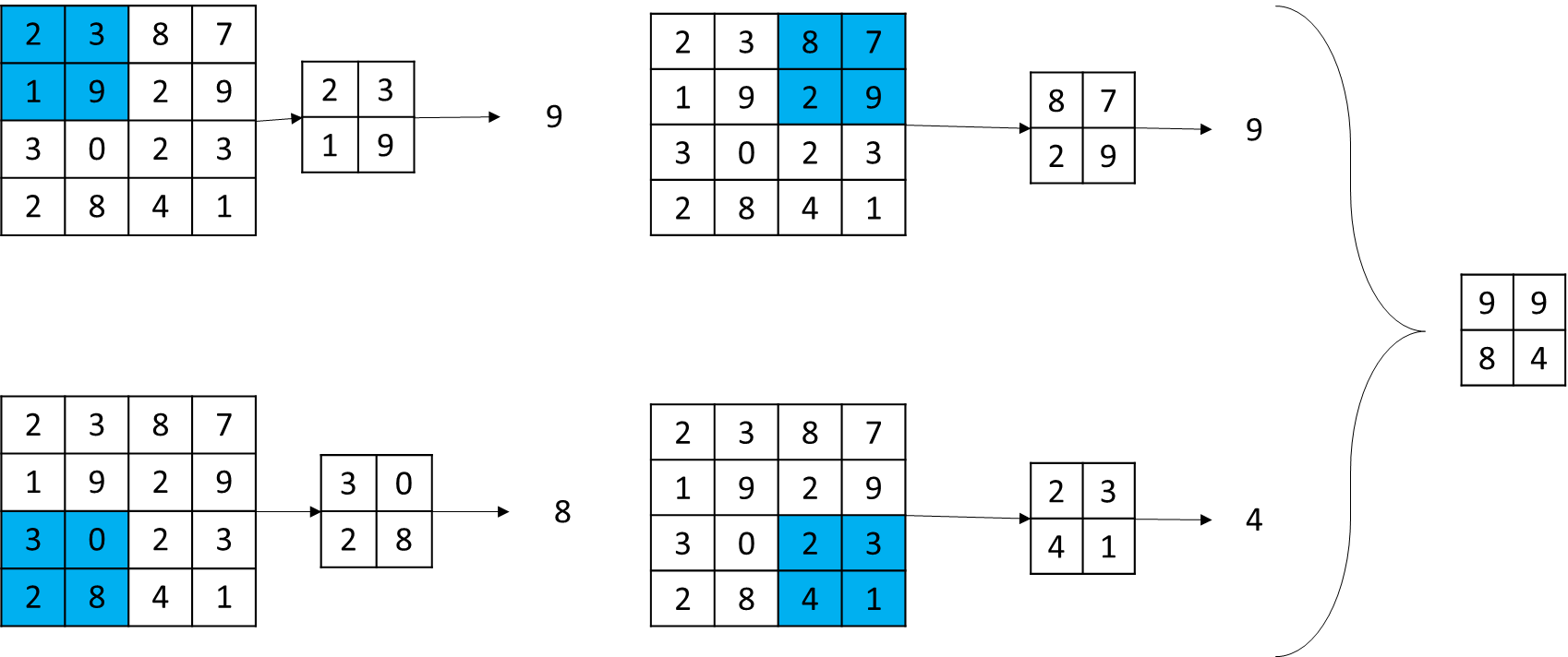
**Stride**

A stride is the step taken by the filter in each convolution operation. In the first example, stride is 1. In the second example, stride is 2.



**Max Pooling**

The max pooling function extracts the largest value from the kernel, similar to convolution filters, by importing kernels in each step. The example below shows a 2 by 2 max pooling operation.



**Batch Normalization**

In deep neural networks, batch normalization is used to improve training speed and stability. It involves subtracting the mean and dividing by the standard deviation of each batch of inputs.

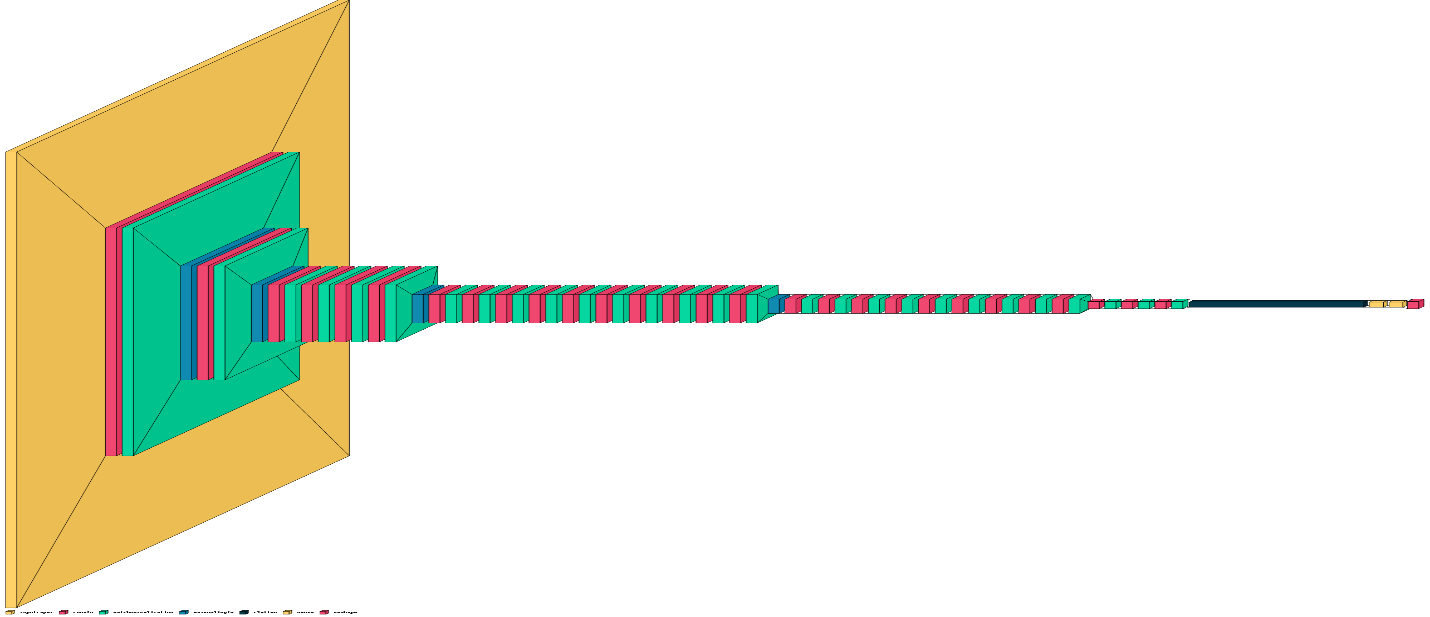
Based on this method, the mean of each feature map is close to zero and the standard deviation is close to 1. This can help improve the performance of the network by reducing the effects of covariate shift, which occurs when the distribution of inputs to a layer change during training.

The batch normalization process is usually applied after each layer's activation function has been executed, and before the weights are updated during backpropagation. Several studies have demonstrated that it enhances the training speed and stability of deep neural networks, enabling them to converge more quickly and perform better.

**Create Model**

In this function, a neural network architecture is created. The first layer is the input layer. The input shape of (448, 448, 3) refers to a color image of 448 by 448 pixels.

In architecture, there are two main components. The first is the encoder, which extracts useful information from the image by decreasing its dimension. The second part is the classifier, which attempts to classify encoder output.



**def** create\_model**(**self, batch\_size**=**16, input\_shape**=**(448, 448, 3)**)**:

    input **=** Input(shape**=**input\_shape, batch\_size**=**batch\_size)

    input **=** Rescaling(1**/**255)(input)

    x **=** Conv2D(8, (7, 7), 2, activation**=**'relu', padding**=**'same')(input)

    x **=** BatchNormalization()(x)

    x **=** MaxPool2D((2,2), 2)(x)

    x **=** Conv2D(12, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** MaxPool2D((2,2), 2)(x)

    x **=** Conv2D(8, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(16, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(16, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** MaxPool2D((2, 2), 2)(x)

    x **=** Conv2D(16, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(16, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(16, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** MaxPool2D((2, 2), 2)(x)

    x **=** Conv2D(32, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(32, (1, 1), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), 2, activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Conv2D(64, (3, 3), activation**=**'relu', padding**=**'same')(x)

    x **=** BatchNormalization()(x)

    x **=** Flatten()(x)

    x **=** Dense(256, activation**=**'relu')(x)

    x **=** Dense(245, activation**=**'sigmoid')(x)

    x **=** Reshape((7, 7, 5))(x)

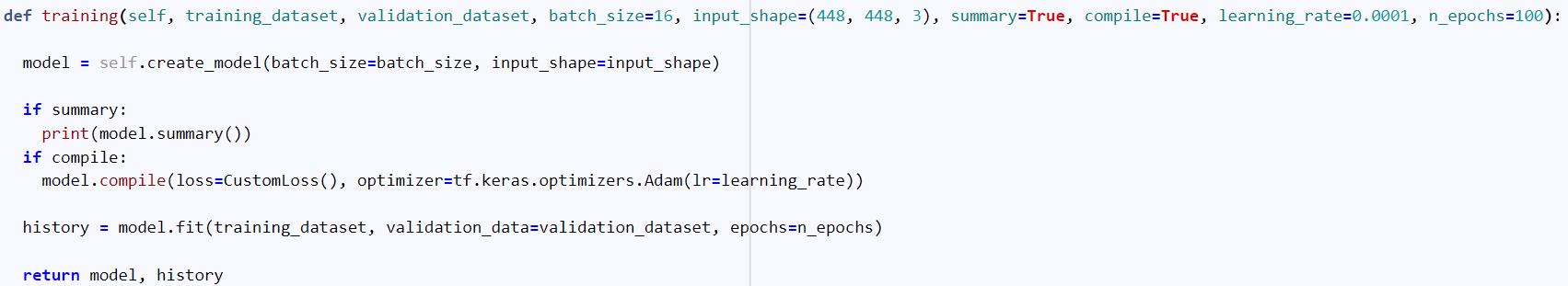
**return** Model(inputs**=**input, outputs**=**x)

**Sigmoid**

The Sigmoid function converts a number into a probability distribution. Neural networks deal with numbers on various scales. In image segmentation, the model outputs a matrix of numbers with dimensions equal to the original image. All pixel values must be between 0 and 1 in order for this matrix to be represented as an image. This can be done using the Sigmoid function as follows:

**Compile Model**

Compile model means to define 3 sets of parameters, loss function, optimizer and metrics. Loss function is a function that compute loss. In this example has been set to custom the Dice loss that defined previously.

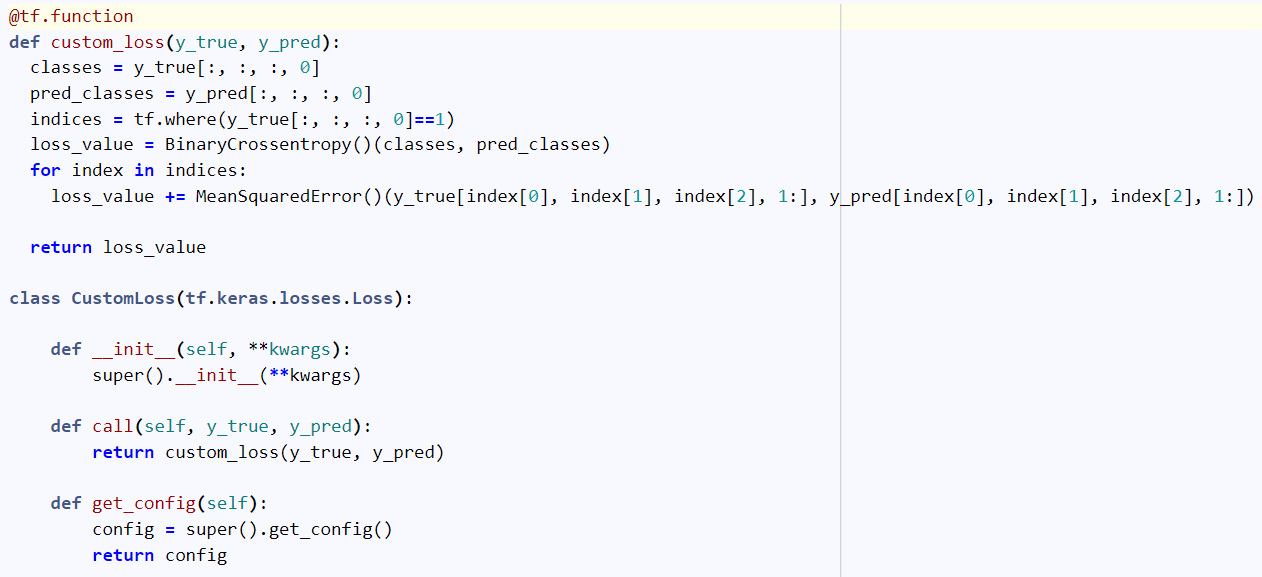
****

**Adam**

Optimizer is a function that can be used to compute gradient descent. It is set to Adam in this example. The SGD algorithm is very noisy, and it does not descend well on curves. To decrease noise in steps moving averages, a new parameter called Momentum is defined. The SGD algorithm with Momentum performs better on curves and requires fewer steps to converge. This method employs two momentum variables, a first-order momentum, and a second-order momentum, as well as an epsilon value that prevents division by zero. Adam is extremely efficient and useful for the convergence of local minima.

**Custom Loss**

To compute loss over labels and predictions in this particular problem, first we need to find the cells that contain the cyclist in the label and then extract the same cell position from prediction. After completing this step, we have some arrays of length 5 which the first number indicates the probability of the cyclist existence and the other 4 is positions of the bounding box. To compute loss, we need to compute cross entropy loss between probability values and add to mean squared error of the other 4.



**Binary Cross Entropy**

The binary cross entropy is a loss function that calculates the logarithmic loss of data:

In binary classification, there are only 2 classes: 0 and 1. This loss function computes a large loss every time there is a difference between the true label (y) and the prediction (p(y)), Whenever y equals 0, the first term of loss becomes zero, while the second part depends on the prediction. When prediction is near zero, logarithmic terms become near zero, and total loss is near zero, whereas if prediction is near one, they become near –infinity and total loss reaches infinity, and vice versa if y equals one. Finally, Binary cross entropy is calculated by averaging all losses.

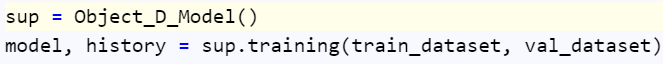
**Mean Squared Error**

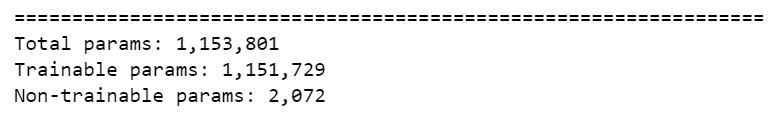
MSE loss is calculated by dividing the squared difference between label and prediction values by the numbers of data.

**Summary**

In the model summary, we can see the output and number of parameters for each layer, which provides a better understanding of the model. Approximately all parameters are trainable. More than 1 and a half million parameters indicate the significant decrease in the number of parameters that has taken place.

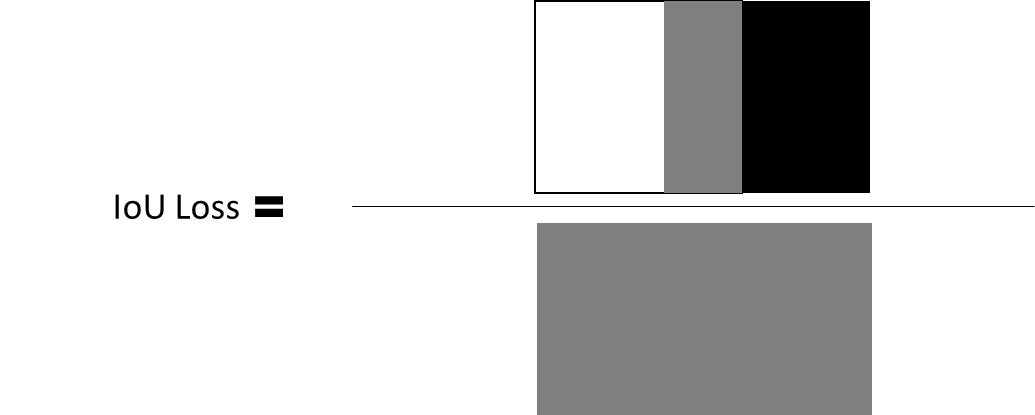
During training, all parameters are trainable. As the model continues to learn, it updates its values using gradient descent; however, non-trainable parameters remain fixed during training. The process of setting more trainable parameters requires additional time and computation; however, it completes the learning process.





**IoU Loss**

The term IoU loss refers to intersection over union loss and it is very similar to dice loss. IoU loss calculates the intersection of two images and then divides that by the union between the two images to calculate the loss.



**Fit Model**

After all previous steps were completed to prepare the model for training, all that remains is to fit the model over the data. In order for the training model to be fit over the data, training and validation data need to be determined, as well as a number of epochs, where an epoch is the number of times the model was trained.

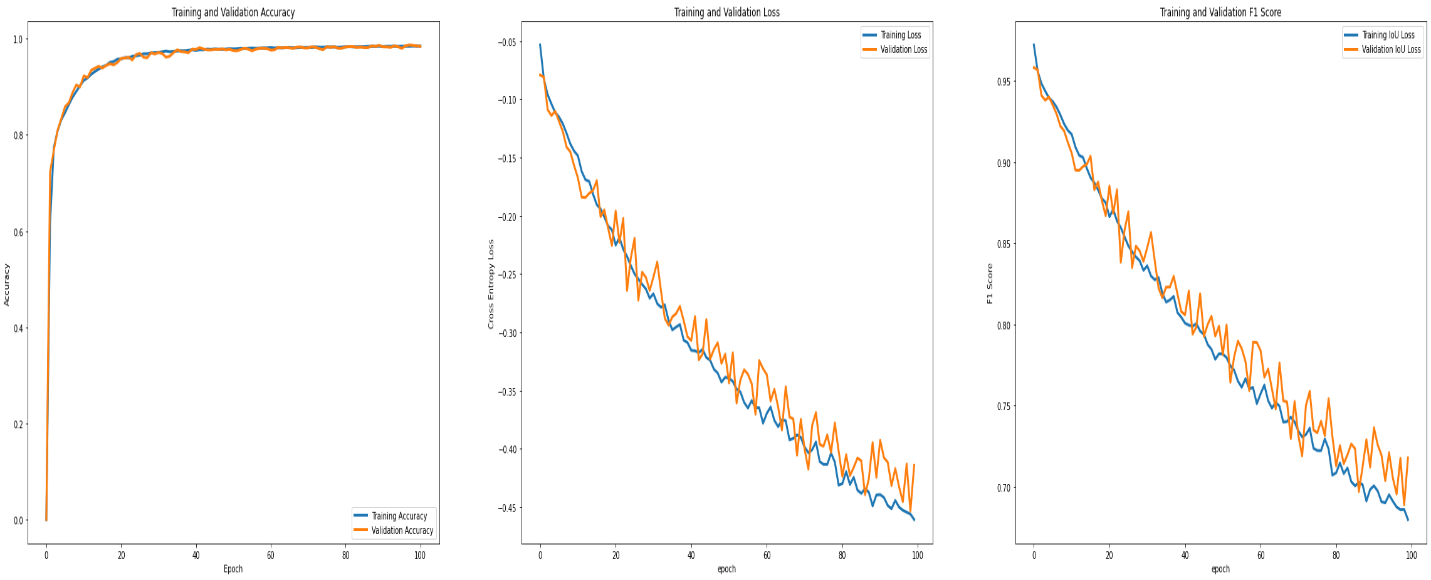
**Epoch**

In training, the training process passes through all training data, so the number of training epochs indicates how many times the training process has been passed through all training data. When dealing with batched datasets, the number of times that training algorithm has performed is not important. The number of epochs is an important parameter to control the model since it indicates how many times the loss was computed on the entire data set.

**Analysis**

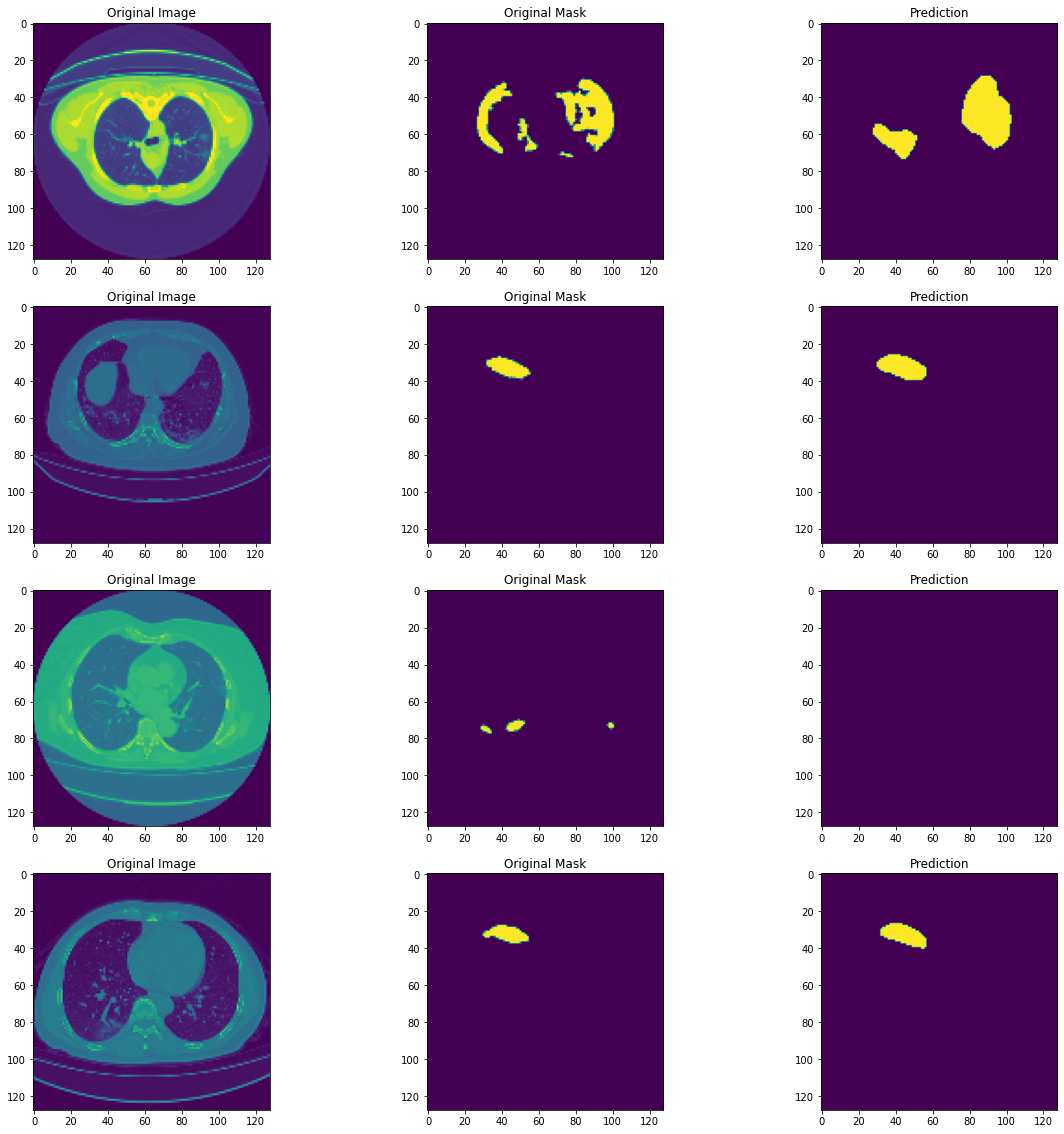
Using training as the number of epochs, it is possible to inspect the parameters of loss and accuracy. It is evident from the loss and accuracy plot that approximately 98% accuracy is acceptable for both validation and training.



****

**Evaluate**

In order to observe the performance of the model, output of the model on the 1 batch of validation data is represented.

****

**References**