

cs512 Assignment 3: Report

Yash Patel

Department of Computer Science

Illinois Institute of Technology

November 09, 2020

Abstract

This is a report of programming assignment 3 which covers construct and train convolution neural network, hyper-parameter tuning and inference for classifying MNIST dataset in even and odd classes.

1. Problem statement

You will train and evaluate a convolution neural network (CNN) to classify images of numbers in the MNIST dataset as either even or odd. Images in MNIST dataset are labeled for 10-digit classes and so you will have to create labels of odd/even and discard the original labels. You may not use the original digit labels for training. Make sure that your program can save and load weights so that training can proceed with previous results. You will be required to log and plot some metrics as a function of training and evaluation iteration.

2. Proposed solution

- **Required packages:**

- 1) numpy
- 2) pandas
- 3) matplotlib
- 4) os
- 5) opencv
- 6) keras/tensorflow

- **model creation:**

- 1) Function named `get_model` creates a sequential model of layers.
- 2) For creating a model, it takes 9 arguments for its 9 parameters which are listed below:

- **number_of_layers:** default value is 2. This parameter is for how many layers of conv2d, batchnormalization, maxpooling2d, dropout and dense should be created in the model.
 - **dilation_rate:** default value is 1. This parameter is for controlling receptive field of kernel in conv2d layer.
 - **stride:** default value is 1. This parameter is for controlling shift of the kernel in conv2d layer.
 - **dropout_p:** default value is 0.5. This parameter is for controlling how many inputs should be droppedout for next layer.
 - **optimizer:** default value is adam. This parameter is for controlling optimization method during compilation of the model.
 - **learning_rate:** default value is 0.001. This parameter is for controlling how fast model should learn the weights.
 - **loss_function:** default value is binary_crossentropy. This parameter is for controlling loss function.
 - **initializer:** default value is glorot_uniform. This parameter is for controlling weight initialization of layers.
 - **normalization:** default value is None. This parameter is for controlling addition of batchnormalization layer in model.
- 3) Using these passed arguments to this function it creates a model by adding layers in it.
 - 4) After creating a model, it compiles this model, print summary of this model and then return this created model.
- **Showing statistics:**
 - 1) Function named show_plot shows two plots.
 - One is for showing training and validation loss during model training.
 - Second is for showing training and validation accuracy during model training.
 - 2) It required 1 argument for its 1 parameter which is listed below:
 - history: this history is from model fitting which has statistics of loss and accuracy in each iteration.
 - 3) Using this history, it shows two plots list above as a function of epochs.
 - **Process MNIST dataset:**
 - 1) Load MNIST dataset in train and test parts.
 - 2) Create single list of data using train and test part.
 - 3) Create labels of even or odd using given 10-digit labels.
 - 4) Create data frame of this data and then split this data frame in the ration of 55:10:5 for train, validation and test data.
 - 5) Create numpy array of these images and labels.

- 6) Reshape these numpy images, convert them in float data type and normalize numpy images in range of 0 to 1 by dividing with 255.

- **Construct and train CNN:**

- 1) Here, I first check if saved model is exist or nor.
- 2) If model is already existing then load that model.
- 3) If model is not existing then create a model using get_model function.

Here, I am passing arguments which are listed below:

- **number_of_layers** = 2
 - **dilation_rate** = 1
 - **stride** = 1
 - **dropout_p** = 0.5
 - **optimizer** = adam
 - **learning_rate** = 0.001
 - **loss_function** = binary_crossentropy
 - **initializer** = he_uniform
 - **normalization** = None
- 4) after getting a model, fit it using training data and validation data for batch size of 64 and epochs of 10. Here is choose to use verbose for seeing the training process.
 - 5) After training of the model, I will get history and using this history I will show training and validation loss and accuracy using show_plot function.
 - 6) After training of the model, I will save this trained model in models directory.
 - 7) After loading saved model or fitting new model, I will evaluate this model on train, validation and test data. And also print these evaluation results.

- **Hyper-parameter tuning:**

- 1) here I evaluate different variations of the basic network as described below and measure performance.
 - **Changing the network architecture:** for this I change number_of_layers from 2 to 3 during model creation.
 - **Changing the receptive field:** for this I change dilation_rate from 1 to 2 in convolution layers.
 - **Changing the stride:** for this I change stride from 1 to 2 in convolution layers.
 - **Changing optimizer:** for this I change optimizer from adam to sgd and rmsprop during compilation of the model.
 - **Changing loss function:** for this I change loss_function from binary_crossentropy to mean_squared_error during compilation of the model.

- **Changing dropout:** for this I change dropout_p from 0.5 to 0.25 in dropout layer.
 - **Changing learning rate:** for this I change learning_rate from 0.001 to 0.01 and 0.1 during compilation of the model.
 - **Adding batch normalization:** for this I change normalization from None to batch during model creation.
 - **Using different weight initializers:** for this I change initializer from he_uniform to glorot_uniform during addition of conv2d, batchnormalization and dense layers.
- 2) So, here I make 12 different models.
- 3) If some of these models are already exist then load those models.
- 4) If some of these models are not exist then create those models using get_model function, fit them using training and validation data for batch size of 64 and epochs of 10 and choose verbose to see the training process.
- 5) After loading all the models or fitting all the models, I will evaluate these models on training, validation and test data. And also print those results.
- 6) For evaluation the best model, I evaluate all these models on the testing subset of the data and save their loss and accuracy in losses and accuracies lists and using argmax on accuracies I find the best model from all of these 12 models.
- **Inference:**
 - 1) In this part, I use best validation model on testing dataset.
 - 2) For inference, I made images with 400x400 dimensions of handwritten digits from 0 to 9 and each image contains only one digit.
 - 3) Using opencv, I do some basic image processing to prepare the image for my CNN. Like:
 - Resize image in 28x28 dimensions.
 - Convert image from rgb to grayscale.
 - For smoothing, apply gaussian blur with 3x3 window size and 3/5 standard deviation.
 - For converting in inverse binary image, I use two methods which are listed below:
 - adaptive gaussian thresholding
 - otsu thresholding
 - 4) Using matplotlib, I show original image and inverse binary image
 - 5) Convert inverse binary images in numpy arrays. Reshape it in 28x28x1 dimensions. Convert them in float data type. Normalize them in the range of 0 to 1 by dividing with 255.
 - 6) By using best model, predict labels for these handwritten digit images, classify them in even or odd classes and print them.

3. Implementation details

- **Design issues:**
 - One issue I found was making a common `get_model` function that returns every type of models with different number of conv2d, dropout, batchnormalization, dense layers.
 - Second issue I found was making different models with different parameters. So, how many model should I make and what should be the parameters for them was an issue.
 - Third issue I found was adding layernormalization. I am using keras instead of tensorflow.keras. keras doesn't have layernormalization but tensorflow.keras has.
- **Problems faced and solutions:**
 - For the first design issue, I considered two super layers with group of layers. First super layer was made out of (conv2d, optional batchnormalization, maxpooling2d, optional batchnormalization, dropout, optional batchnormalization). Second super layer was made out of (dense, optional batchnormalization). In `get_model` function, `number_of_layer` parameter is for repeating these super layers. In conv2d layer, number of units are double in every next super layer. In dense layer, number of units are half in every next super layer.
 - For the second design issue, I made 12 different models by changing one of the parameters in `get_model`.
 - Another problem I faced was split MNIST data into three train, validation, and test subsets. For that I made single dataset out of train and test data. Store it in data frame and then split it.
- **Instructions for using program:**
 - Put all source code and directories (MODELS and handwritten-digit) together.
 - Install all required python packages.
 - Execute Homework-3.ipynb file in Jupyter notebook.

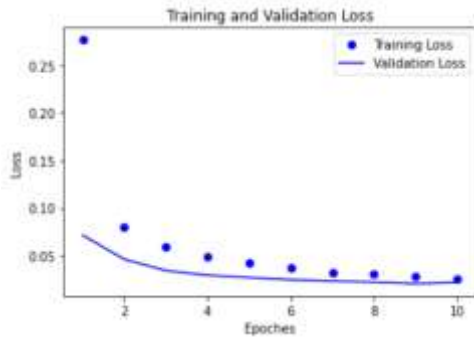
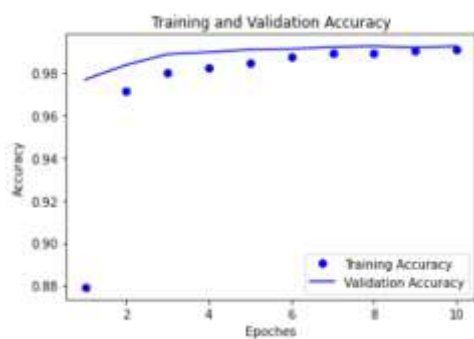
4. Results and Discussion

I. Construct and train CNN:

In this part model's parameters, train and validation loss and accuracy, loss and accuracy of final step and loss and accuracy of the model on train, validation and test dataset are shown below.

```
number_of_layers : 2
dilation_rate : 1
stride : 1
dropout_p : 0.5
```

optimizer : adam
learning_rate : 0.001
loss_function : binary_crossentropy
initializer : he_uniform
normalization : None

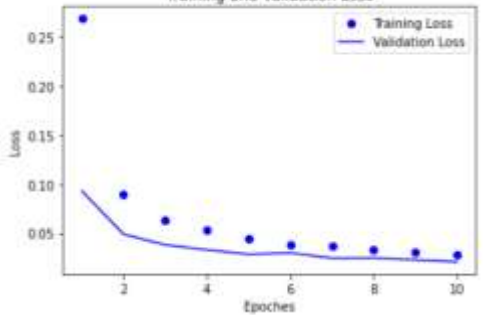
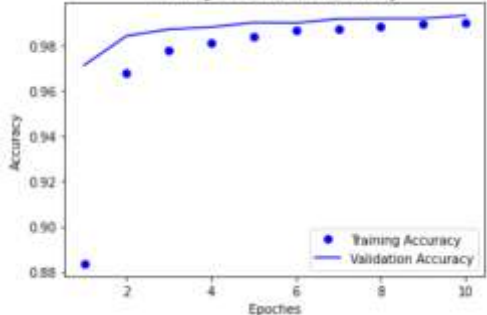
Train and validation loss	Train and validation accuracy
	
Loss and accuracy of final training step	loss: 0.0261 - accuracy: 0.9909 - val_loss: 0.0217 - val_accuracy: 0.9925
Training data - (loss, accuracy) : (0.010357376720768993 , 0.9969454407691956) Validation data - (loss, accuracy) : (0.021694011484645306 , 0.9925000071525574) Testing data - (loss, accuracy) : (0.019028786597773434 , 0.9937999844551086)	

Here, I can say that this model works better on testing data than validation data with 99.37% accuracy.

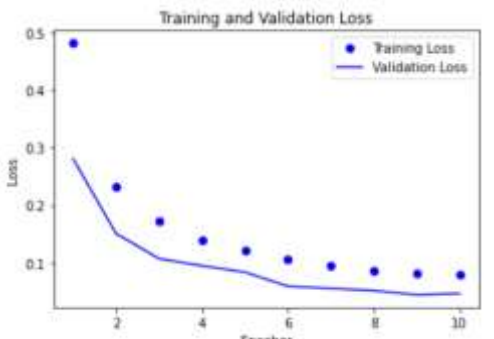
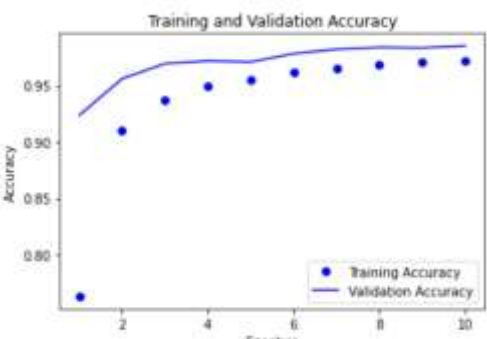
II. Hyper-parameter tuning:

In this part 12 different models’ different parameters, train and validation loss and accuracy, loss and accuracy of final step and loss and accuracy of the models on train, validation and test dataset are shown below.

Model-1	
number_of_layers : 2 dilation_rate : 1 stride : 1 dropout_p : 0.5 optimizer : adam learning_rate : 0.001 loss_function : binary_crossentropy initializer : he_uniform normalization : None	
Train and validation loss	Train and validation accuracy

	
Loss and accuracy of final training step	loss: 0.0280 - accuracy: 0.9904 - val_loss: 0.0208 - val_accuracy: 0.9933
Training data - (loss, accuracy) : (0.010276111710507591 , 0.9969454407691956) Validation data - (loss, accuracy) : (0.020842380492109805 , 0.9933000206947327) Testing data - (loss, accuracy) : (0.018032204249314964 , 0.944000244140625)	

This is the same model as part one but height accuracy of 99.44% accuracy. This model is the best model from all of these 12 models.

Model-2 number_of_layers : 3 dilation_rate : 1 stride : 1 dropout_p : 0.5 optimizer : adam learning_rate : 0.001 loss_function : binary_crossentropy initializer : he_uniform normalization : None	
Train and validation loss 	Train and validation accuracy 
Loss and accuracy of final training step	loss: 0.0794 - accuracy: 0.9723 - val_loss: 0.0467 - val_accuracy: 0.9852

```

Training data - (loss, accuracy) : (0.03988109018396248 , 0.9889272451400757)
Validation data - (loss, accuracy) : (0.04674166419655085 , 0.9851999878883362)
Testing data - (loss, accuracy) : (0.04024459846019745 , 0.9887999892234802)

```

In this model, when I increase number_of_layers and train with same number of epochs then it shows less accuracy. I think this model requires more epochs to train weights of higher number of trainable parameters.

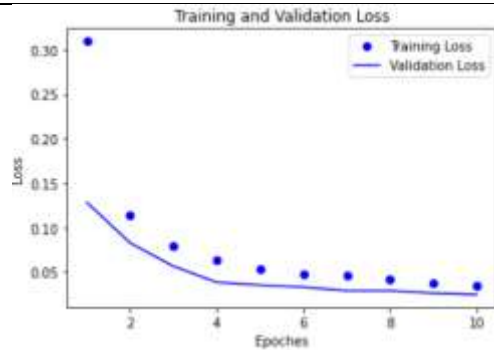
Model-3

```

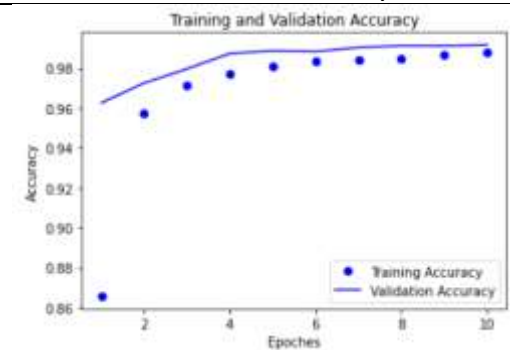
number_of_layers : 2
dilation_rate : 2
stride : 1
dropout_p : 0.5
optimizer : adam
learning_rate : 0.001
loss_function : binary_crossentropy
initializer : he_uniform
normalization : None

```

Train and validation loss



Train and validation accuracy



Loss and accuracy of final training step

```

loss: 0.0352 - accuracy: 0.9878 - val_loss: 0.0247 - val_accuracy: 0.9917

```

```

Training data - (loss, accuracy) : (0.014623942727307705 , 0.9956545233726501)
Validation data - (loss, accuracy) : (0.024715870844386516 , 0.9916999936103821)
Testing data - (loss, accuracy) : (0.02057221309095621 , 0.993399977684021)

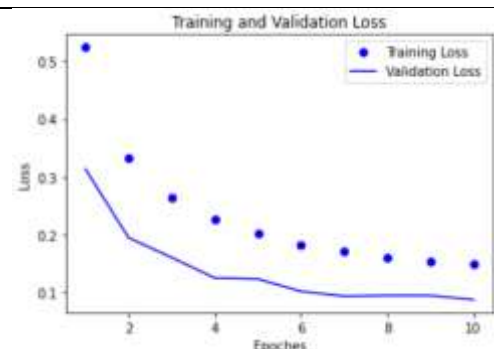
```

In this model, when I increase dilation_rate of the conv2d layers, it doesn't show that much difference in accuracy.

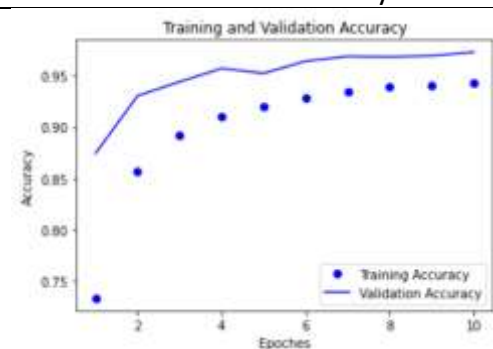
Model-4

number_of_layers : 2
 dilation_rate : 1
 stride : 2
 dropout_p : 0.5
 optimizer : adam
 learning_rate : 0.001
 loss_function : binary_crossentropy
 initializer : he_uniform
 normalization : None

Train and validation loss



Train and validation accuracy



Loss and accuracy of final training step

loss: 0.1485 - accuracy: 0.94
 31 - val_loss: 0.0871 - val_a
 ccuracy: 0.9731

Training data - (loss, accuracy) : (0.07915368268164721 , 0.9
 772909283638)
 Validation data - (loss, accuracy) : (0.08706249529123307 , 0
 .9731000065803528)
 Testing data - (loss, accuracy) : (0.08648525658845901 , 0.97
 46000170707703)

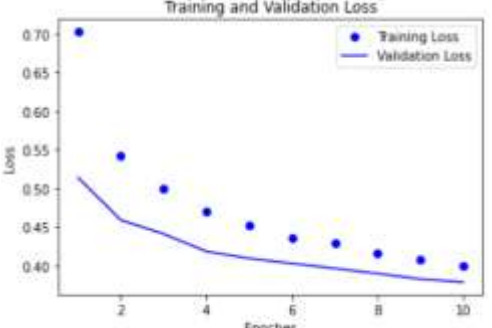
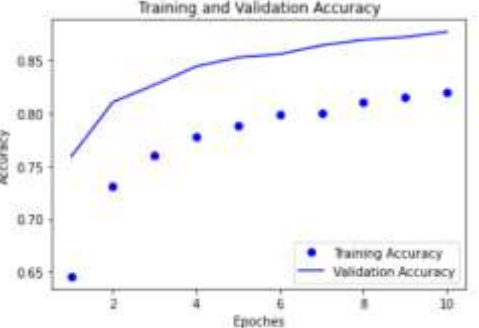
In this model, when I increase stride in conv2d layers, it shows less accuracy then first model. I think this is because of when I use higher stride it reduces the dimensions of the inputs. So, less number of inputs doesn't help model to train better.

Model-5

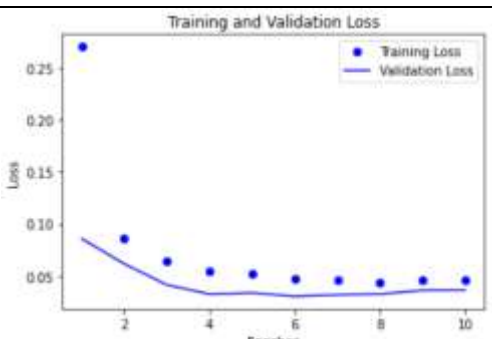
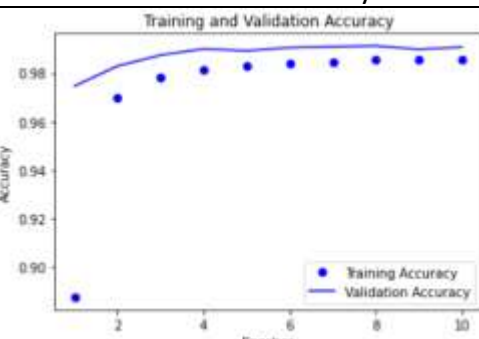
number_of_layers : 2
 dilation_rate : 1
 stride : 1
 dropout_p : 0.5
 optimizer : sg
 learning_rate : 0.001
 loss_function : binary_crossentropy
 initializer : he_uniform
 normalization : None

Train and validation loss

Train and validation accuracy

	
Loss and accuracy of final training step	loss: 0.3989 - accuracy: 0.8202 - val_loss: 0.3785 - val_accuracy: 0.8769
Training data - (loss, accuracy) : (0.3733444333856756 , 0.881781816482544) Validation data - (loss, accuracy) : (0.3785229285240173 , 0.8769000172615051) Testing data - (loss, accuracy) : (0.3830253333091736 , 0.8697999715805054)	

In this model, when I use sgd for optimization, it shows less accuracy than first model. I think sgd required higher epochs than adam to reach to the optimam solution.

Model-6 number_of_layers : 2 dilation_rate : 1 stride : 1 dropout_p : 0.5 optimizer : rmsprop learning_rate : 0.001 loss_function : binary_crossentropy initializer : he_uniform normalization : None	
Train and validation loss 	Train and validation accuracy 
Loss and accuracy of final training step	loss: 0.0458 - accuracy: 0.9857 - val_loss: 0.0363 - val_accuracy: 0.9907

```

Training data - (loss, accuracy) : (0.03164118455919353 , 0.9
941999912261963)
Validation data - (loss, accuracy) : (0.036325702239573 , 0.9
907000064849854)
Testing data - (loss, accuracy) : (0.0343943447291851 , 0.993
399977684021)

```

In this model, when I use rmsprop for optimization, it shows good results on testing dataset. I think respop and adam optimizers shows better results on small number of epochs. That means they helps model to learn quickly.

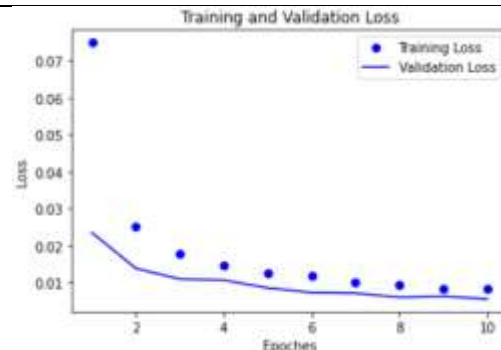
Model-7

```

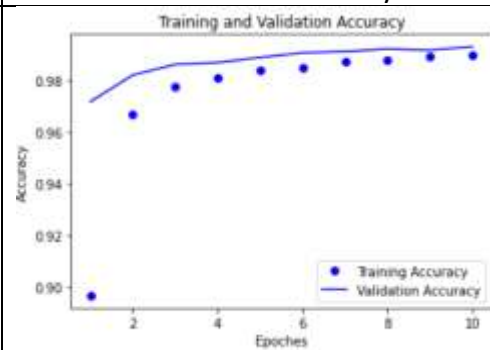
number_of_layers : 2
dilation_rate : 1
stride : 1
dropout_p : 0.5
optimizer : adam
learning_rate : 0.001
loss_function : mean_squared_error
initializer : he_uniform
normalization : None

```

Train and validation loss



Train and validation accuracy



Loss and accuracy of final training step

```

loss: 0.0082 - accuracy: 0.98
95 - val_loss: 0.0056 - val_a
ccuracy: 0.9930

```

```

Training data - (loss, accuracy) : (0.0031143723383156943 , 0
.9961636066436768)
Validation data - (loss, accuracy) : (0.005623456428517966 ,
0.9929999709129333)
Testing data - (loss, accuracy) : (0.005239149547155103 , 0.9
929999709129333)

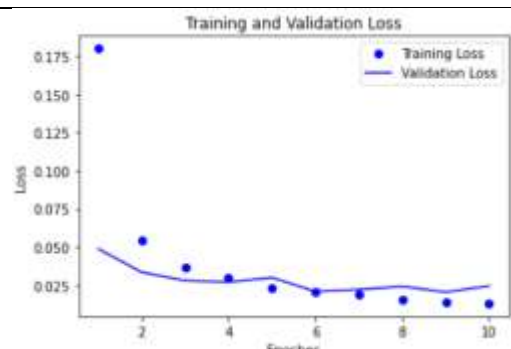
```

In this model, when I use mean_squared_error for loss_function, it shows good result on testing dataset. But, not as good as binary_crossentropy.

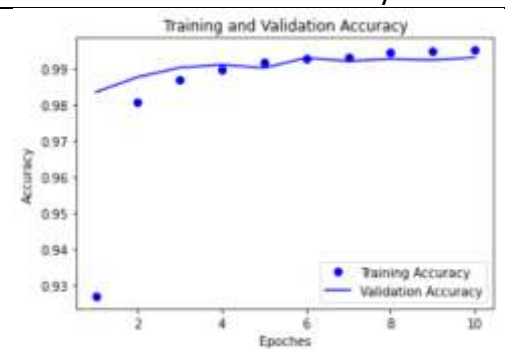
Model-8

number_of_layers : 2
 dilation_rate : 1
 stride : 1
 dropout_p : 0.25
 optimizer : adam
 learning_rate : 0.001
 loss_function : binary_crossentropy
 initializer : he_uniform
 normalization : None

Train and validation loss



Train and validation accuracy



Loss and accuracy of final training step

loss: 0.0132 - accuracy: 0.99
 51 - val_loss: 0.0245 - val_a
 ccuracy: 0.9931

Training data - (loss, accuracy) : (0.005853351866197772 , 0.9980363845825195)
 Validation data - (loss, accuracy) : (0.02453343954986194 , 0.9930999875068665)
 Testing data - (loss, accuracy) : (0.01887139388122014 , 0.9936000108718872)

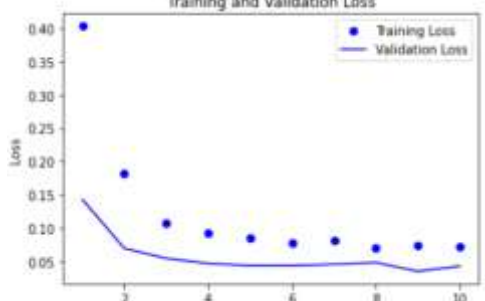
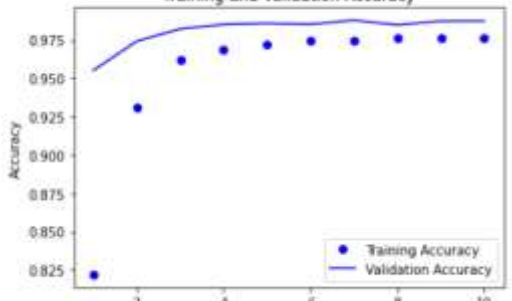
In this model, when I decrease dropout_p, it shows good result on testing data, but not as good as first model. I think because it increases number of parameters in input. So, higher number of input parameters model requires higher number of epochs or model is overfitting with higher number of input parameters.

Model-9

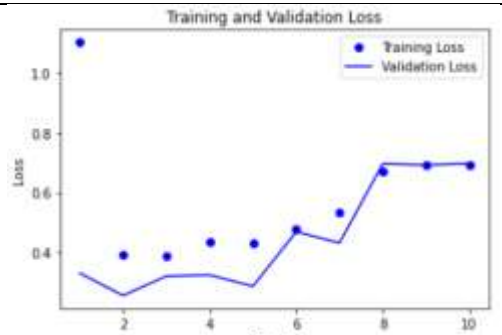
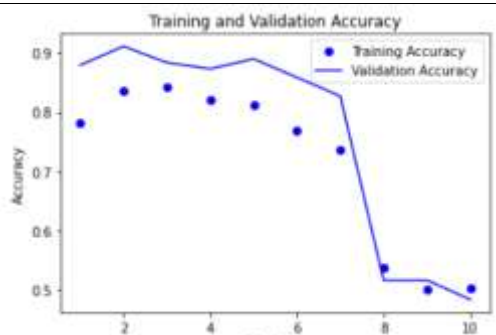
number_of_layers : 2
 dilation_rate : 1
 stride : 1
 dropout_p : 0.5
 optimizer : adam
 learning_rate : 0.01
 loss_function : binary_crossentropy
 initializer : he_uniform
 normalization : None

Train and validation loss

Train and validation accuracy

	
<p>Loss and accuracy of final training step</p>	<p>loss: 0.0717 - accuracy: 0.976 1 - val_loss: 0.0427 - val_accuracy: 0.9873</p>
<p>Training data - (loss, accuracy) : (0.03673034160597758 , 0.989690899848938) Validation data - (loss, accuracy) : (0.04273662411645055 , 0.9872999787330627) Testing data - (loss, accuracy) : (0.03814719068929553 , 0.9894000291824341)</p>	

In this model, when I increase learning_rate with small value, it shows good result. But, not as good as first model. I think this model requires more epochs to train or it cannot reach to optimum solution.

<p>Model-10</p> <p>number_of_layers : 2 dilation_rate : 1 stride : 1 dropout_p : 0.5 optimizer : adam learning_rate : 0.1 loss_function : binary_crossentropy initializer : he_uniform normalization : None</p>	
<p>Train and validation loss</p> 	<p>Train and validation accuracy</p> 
<p>Loss and accuracy of final training step</p>	<p>loss: 0.6945 - accuracy: 0.5023 - val_loss: 0.6972 - val_accuracy: 0.4836</p>

```

Training data - (loss, accuracy) : (0.6960061482169412 , 0.49
27818179130554)
Validation data - (loss, accuracy) : (0.6971550449371338 , 0.
483599990606308)
Testing data - (loss, accuracy) : (0.695628493976593 , 0.4957
999885082245)

```

In this model, when I increase learning_rate too much, it shows bad result on all the datasets. I think this is because large learning rate takes large steps and doesn't help to reach to the optimum solution.

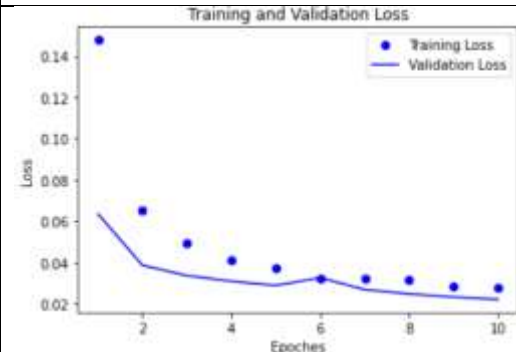
Model-11

```

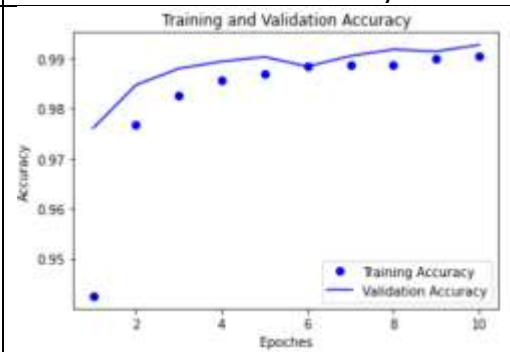
number_of_layers : 2
dilation_rate : 1
stride : 1
dropout_p : 0.5
optimizer : adam
learning_rate : 0.001
loss_function : binary_crossentropy
initializer : he_uniform
normalization : batch

```

Train and validation loss



Train and validation accuracy



Loss and accuracy of final training step

```

loss: 0.0276 - accuracy: 0.99
05 - val_loss: 0.0221 - val_a
ccuracy: 0.9927

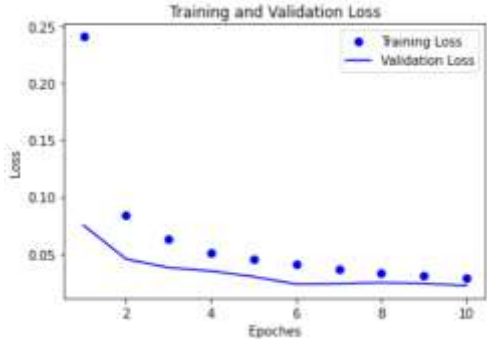
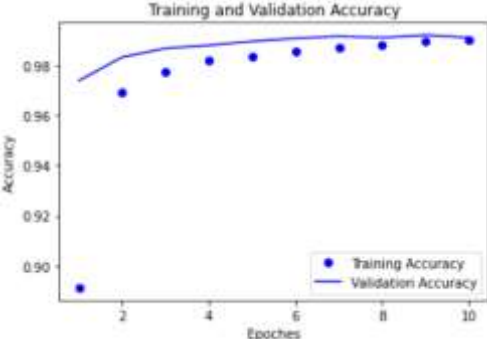
```

```

Training data - (loss, accuracy) : (0.01009475883035709 , 0.9
968909025192261)
Validation data - (loss, accuracy) : (0.022058307805191725 ,
0.9926999807357788)
Testing data - (loss, accuracy) : (0.018907563838083296 , 0.9
940000176429749)

```

In this model, when I use batch normalization, it shows good result on testing dataset. But, not as good as first model. The difference of accuracy is too small. I think this model required more epochs to train.

Model-12 number_of_layers : 2 dilation_rate : 1 stride : 1 dropout_p : 0.5 optimizer : adam learning_rate : 0.001 loss_function : binary_crossentropy initializer : glorot_uniform normalization : None	
Train and validation loss 	Train and validation accuracy 
Loss and accuracy of final training step	loss: 0.0292 - accuracy: 0.9900 - val_loss: 0.0224 - val_accuracy: 0.9912
Training data - (loss, accuracy) : (0.012220898719843139 , 0.9959999918937683) Validation data - (loss, accuracy) : (0.022370123519469053 , 0.9911999702453613) Testing data - (loss, accuracy) : (0.017333031244948505 , 0.937999844551086)	

In this model, when I use glorot_uniform for weight initialization, it shows good result on testing dataset. But, not as good as first model. I think this is because of randomness and multiple attempts with same parameters of the model or higher number of epochs can show good result.

III. Inference:

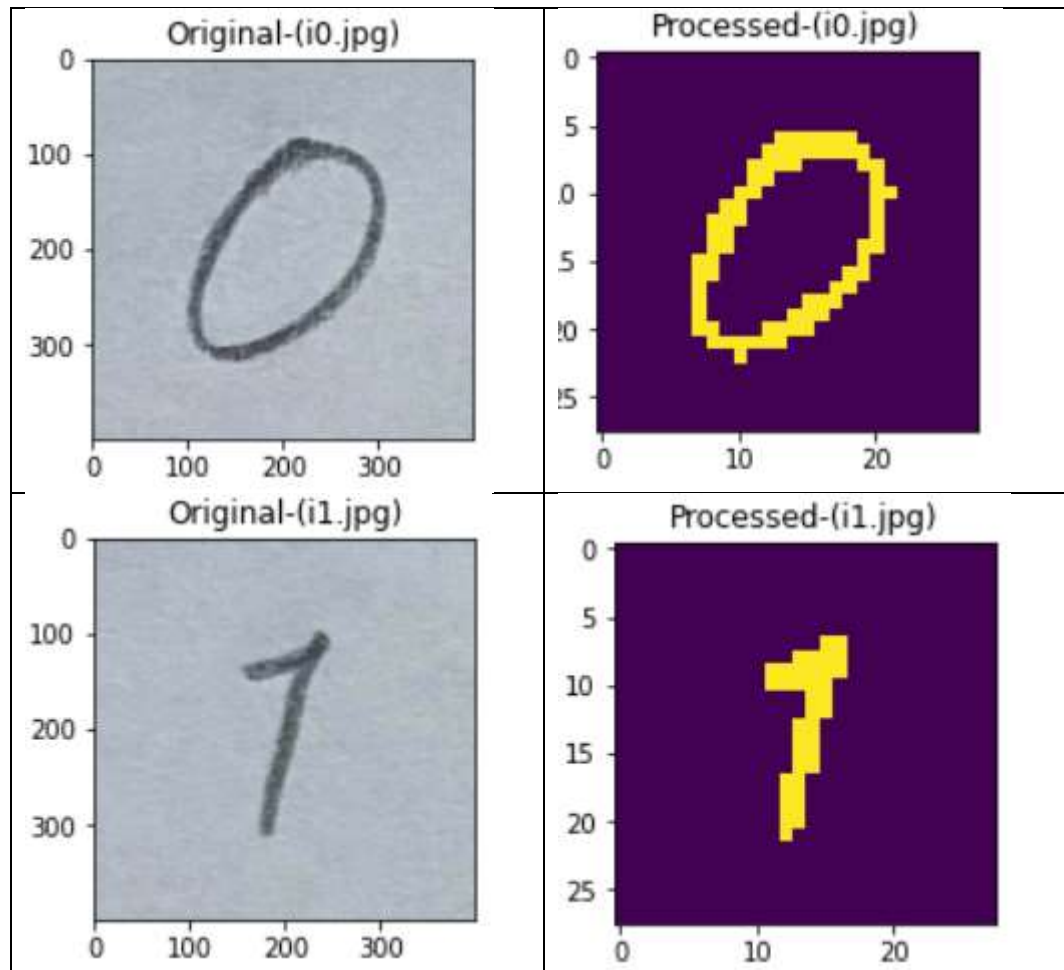
In this part evaluation of the best model on handwritten digits, their actual labels, their predicted probability and labels are shown below.

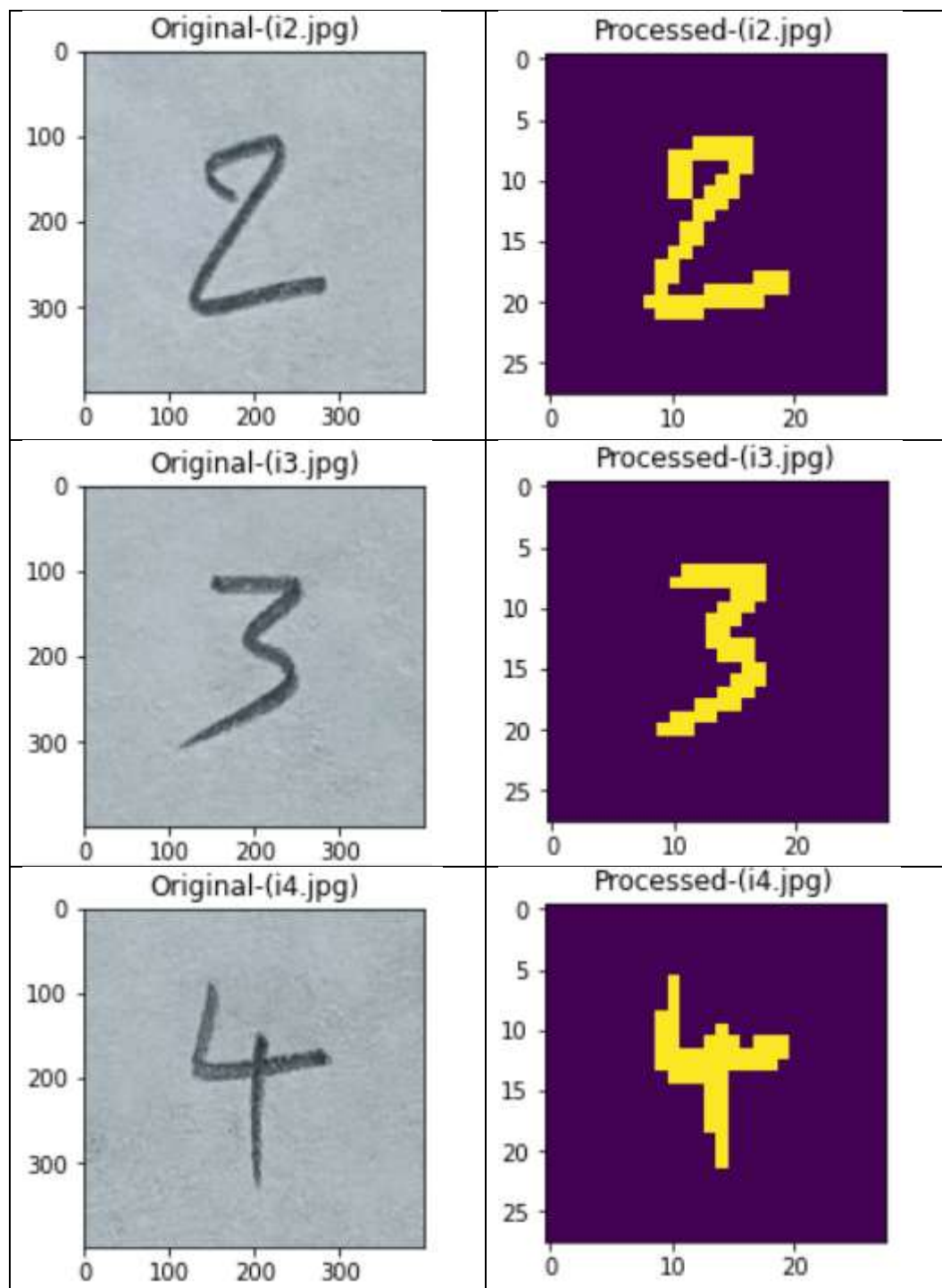
(test_image_file : i0.jpg)-(digit : 0)-(actual : EVEN)-(prediction : [0.3677864968776703 , EVEN])
(test_image_file : i1.jpg)-(digit : 1)-(actual : ODD)-(prediction : [0.99037104845047 , ODD])
(test_image_file : i2.jpg)-(digit : 2)-(actual : EVEN)-(prediction : [0.05503886938095093 , EVEN])

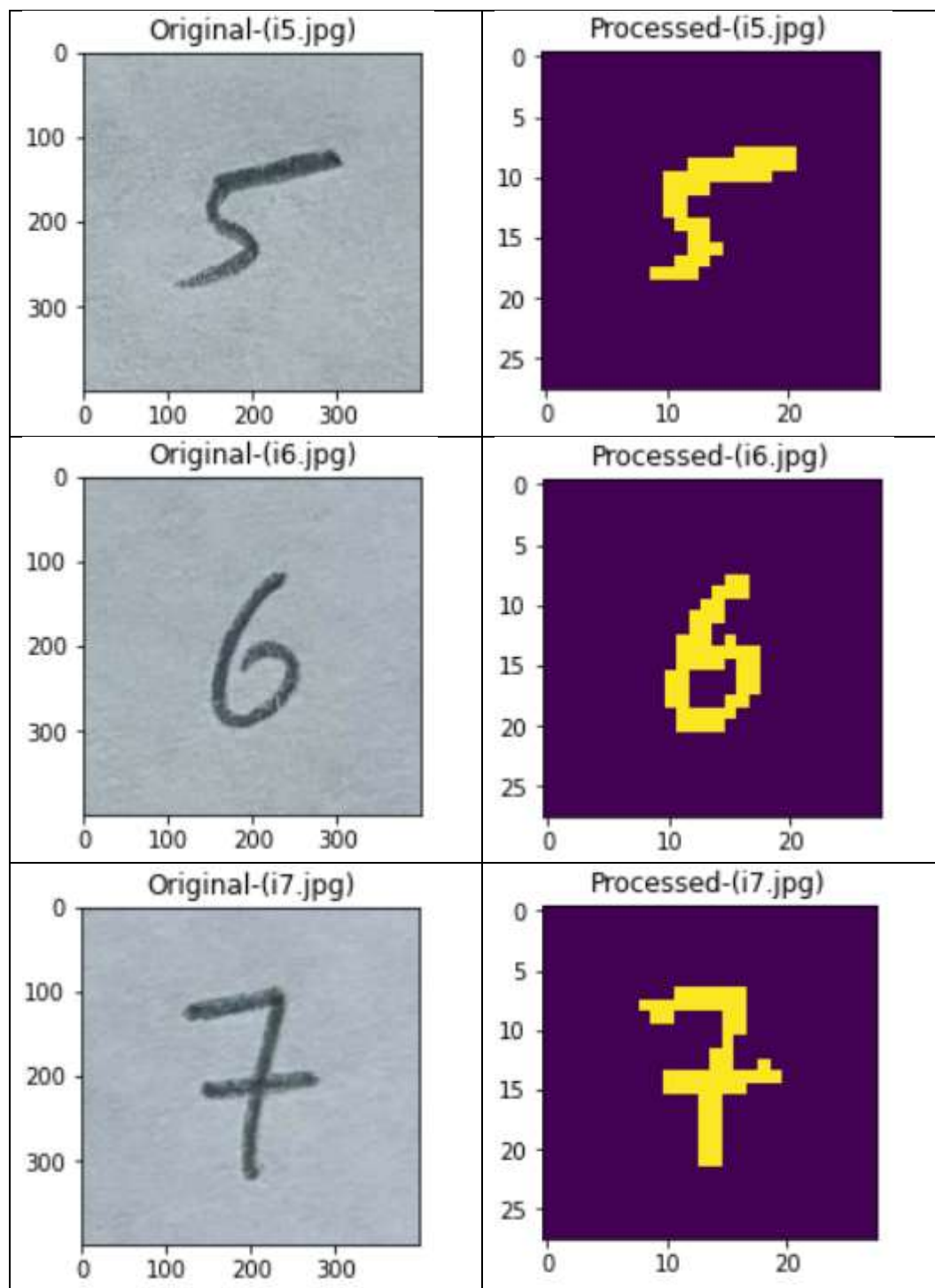
(test_image_file : i3.jpg)-(digit : 3)-(actual : ODD)-(prediction : [0.992933988571167 , ODD])
(test_image_file : i4.jpg)-(digit : 4)-(actual : EVEN)-(prediction : [0.014066331088542938 , EVEN])
(test_image_file : i5.jpg)-(digit : 5)-(actual : ODD)-(prediction : [0.9995754361152649 , ODD])
(test_image_file : i6.jpg)-(digit : 6)-(actual : EVEN)-(prediction : [0.03467428311705589 , EVEN])
(test_image_file : i7.jpg)-(digit : 7)-(actual : ODD)-(prediction : [0.9840235710144043 , ODD])
(test_image_file : i8.jpg)-(digit : 8)-(actual : EVEN)-(prediction : [0.2994788587093353 , EVEN])
(test_image_file : i9.jpg)-(digit : 9)-(actual : ODD)-(prediction : [0.6563121676445007 , ODD])

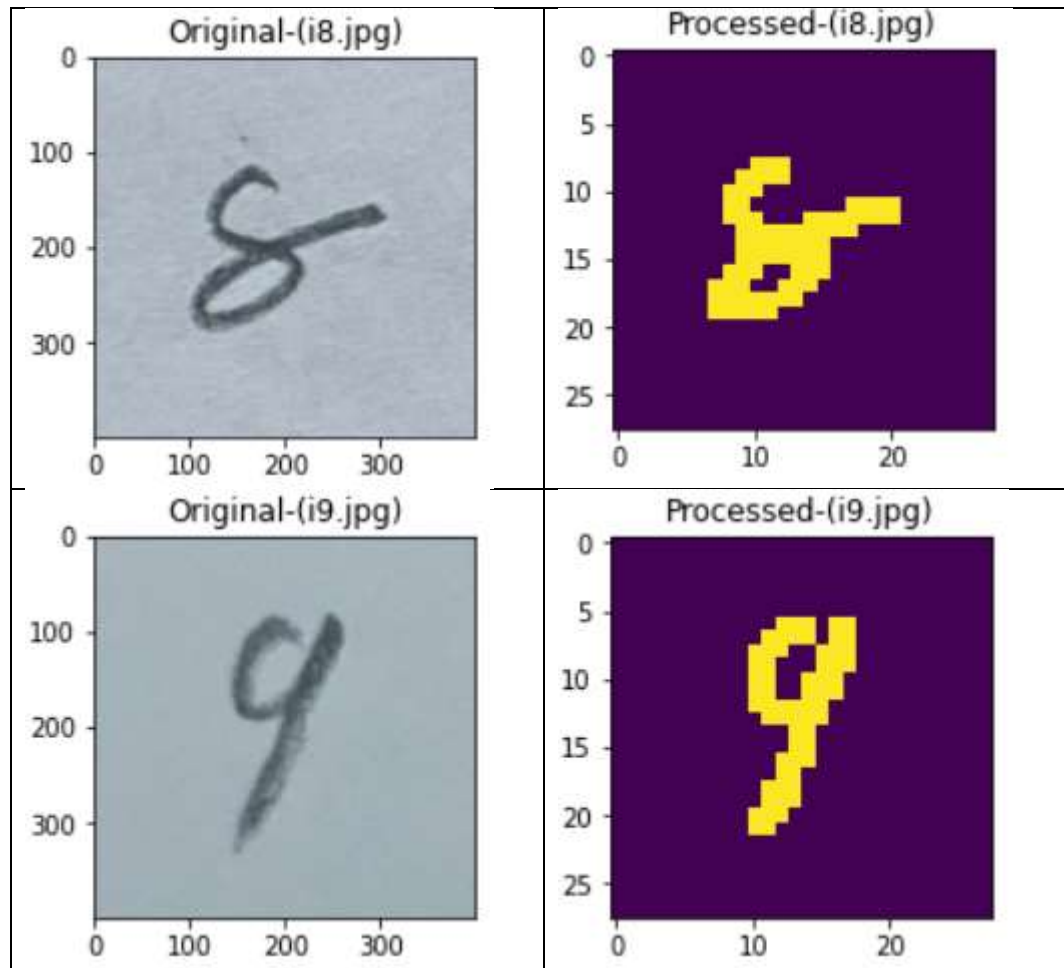
From this part I can say that first model predicted correct labels and gave 100% accuracy on my own handwritten digits testing dataset.

In this part original handwritten digits' image and their inverse binary image are shown below.









For converting image into inverse binary image otsu thresholding showed better result than adaptive gaussian thresholding.