Assignment 4 report

Chethan Kothwal

CS512 Fall 18

A20423243

The CNN model is implemented using TensorFlow estimators.

Deliverable 1:

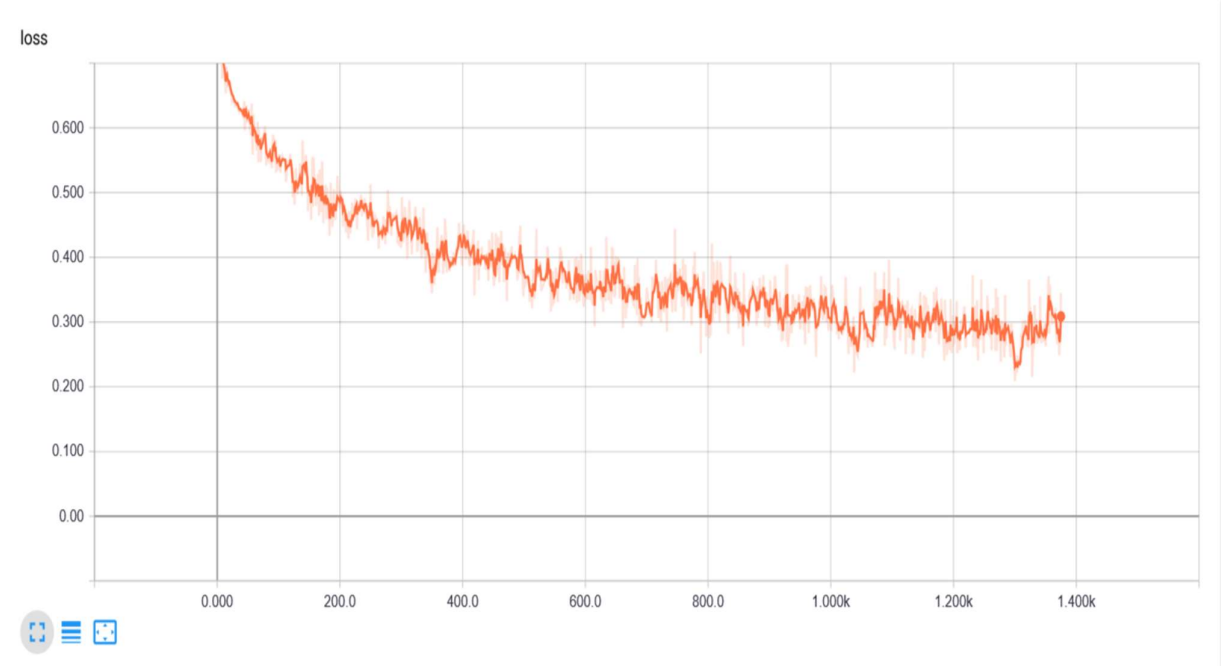
1. To train a custom CNN using TensorFlow estimator.

   The following layers are implemented in the CNN configuration:
   Convolutional layer 1: 32 filters of kernel size 5x5
   Convolutional layer 2: 64 filters of kernel size 5x5
   Pooling in both layers should down sample by a factor of 2
   Dropout rate: 40%
   Use gradient descent optimization and a learning rate of 0.001
   Compute cross entropy loss
   Number of train epochs = 5
   To execute and run the model:
   Python cnn.py
   (The model is already trained and saved as 'cnn_model' in the 'models' folder)
   To visualize the training and evaluation results using tensorboard execute:

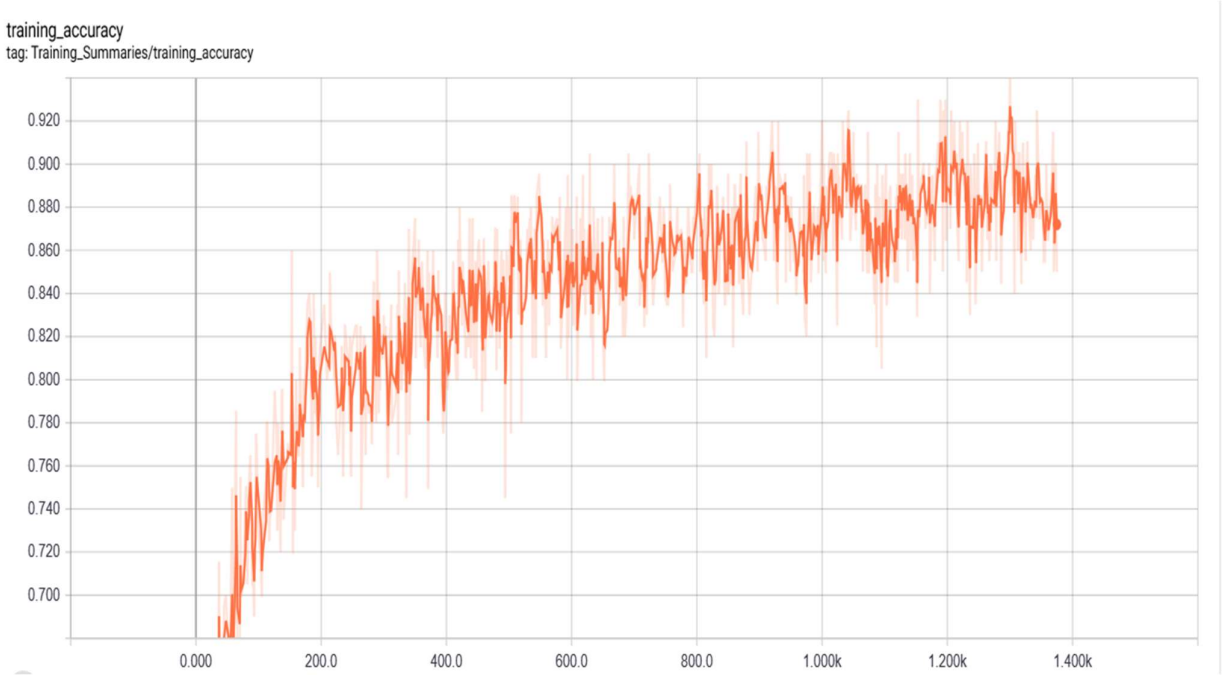
   Tensorboard - -logdir=../models/cnn_model

   The CNN is implemented using the TensorFlow estimators and the results are visualized below.

2. To report the training loss and accuracy



Graph-1 Training loss reported for each step



Graph 2 - Training accuracy for each step

Values of training loss and accuracy from the graph at the last step:

Training loss: 0.3453

Training accuracy: 0.8500

3. Evaluation report:

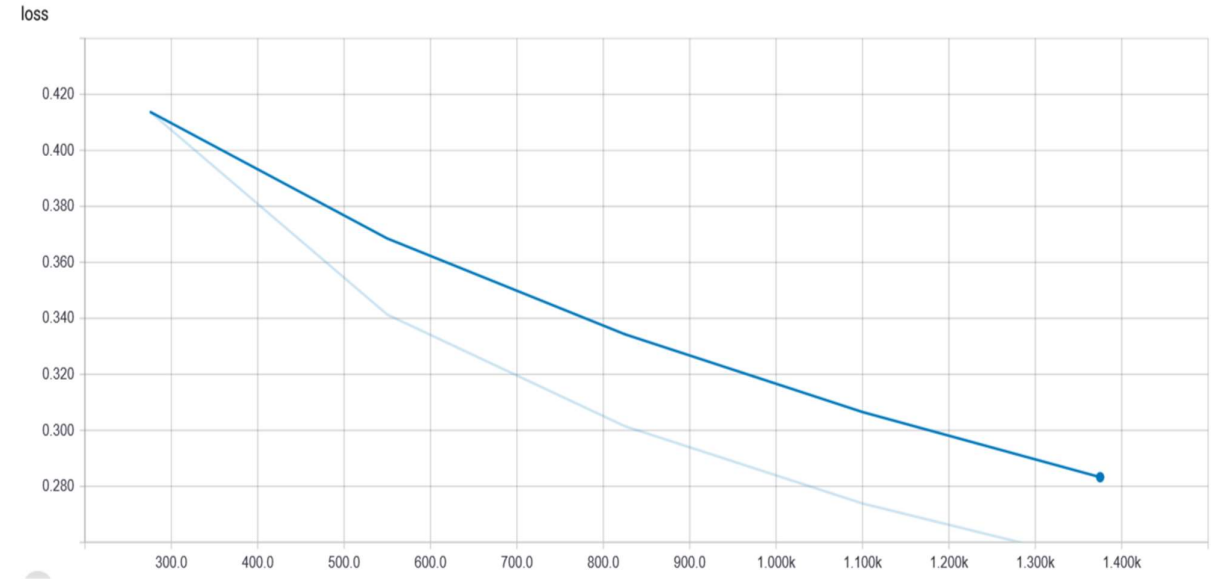
   Train Set Image Shape: (55000, 784)
   Train Set Label Shape: (55000,)
   Test Set Image Shape: (10000, 784)
   Test Set Label Shape: (10000,)
   Approx. number of Steps per Epoch: 275.0
   Checkpoint (save model) done every: 550 steps
   Save summary done every: 1 steps

   Hyper Parameters
   Number of CNN Layers: 2
   Batch size used: 200
   Total number of epochs trained: 5
   Dropout Rate: 0.4
   Learning Rate: 0.001

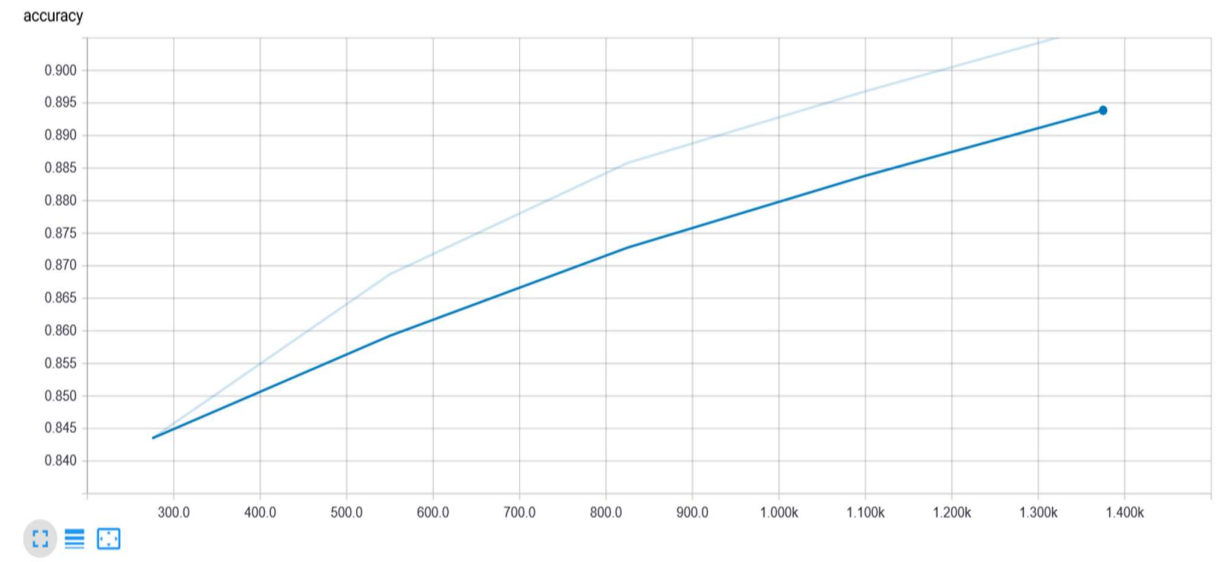
   Epoch: 1
   {'precision': 0.84408706, 'global_step': 275, 'accuracy': 0.8435, 'loss': 0.41379663, 'recall': 0.848246}
   Epoch: 2
   {'precision': 0.8720079, 'global_step': 550, 'accuracy': 0.8687, 'loss': 0.34131134, 'recall': 0.8687426}

Epoch: 3
{'precision': 0.8893069, 'global_step': 825, 'accuracy': 0.8858, 'loss': 0.30146685, 'recall': 0.8851005}
Epoch: 4
{'precision': 0.89893407, 'global_step': 1100, 'accuracy': 0.8968, 'loss': 0.27388746, 'recall': 0.8975167}
Epoch: 5
{'precision': 0.913738, 'global_step': 1375, 'accuracy': 0.907, 'loss': 0.2529326, 'recall': 0.9018526}

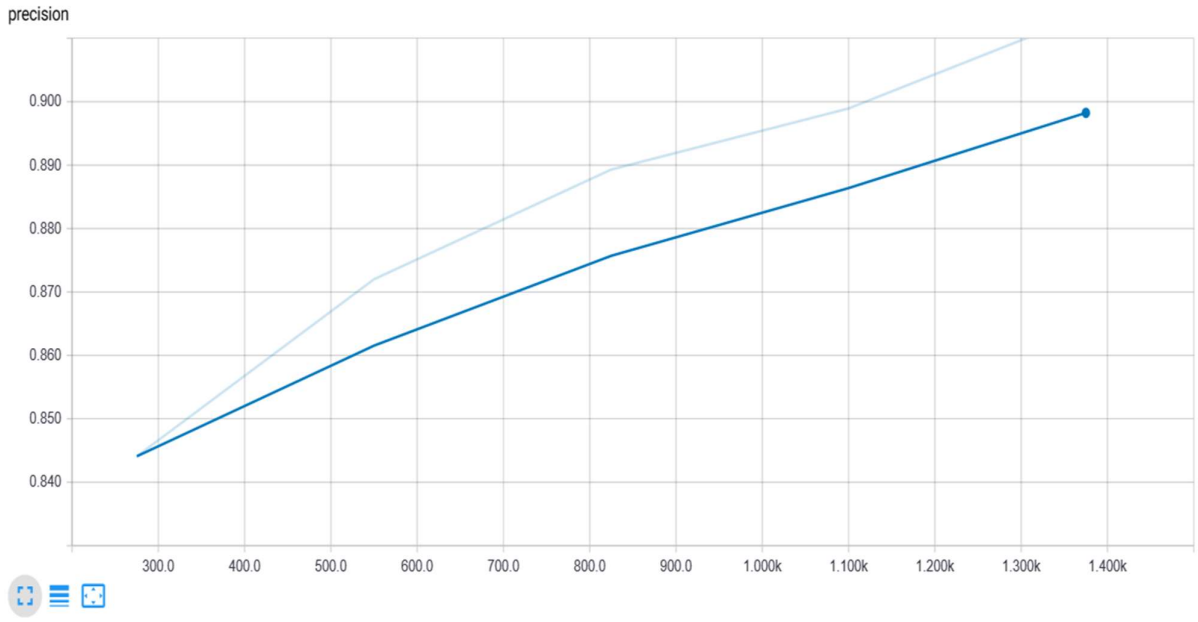
Runtime: 1 minutes



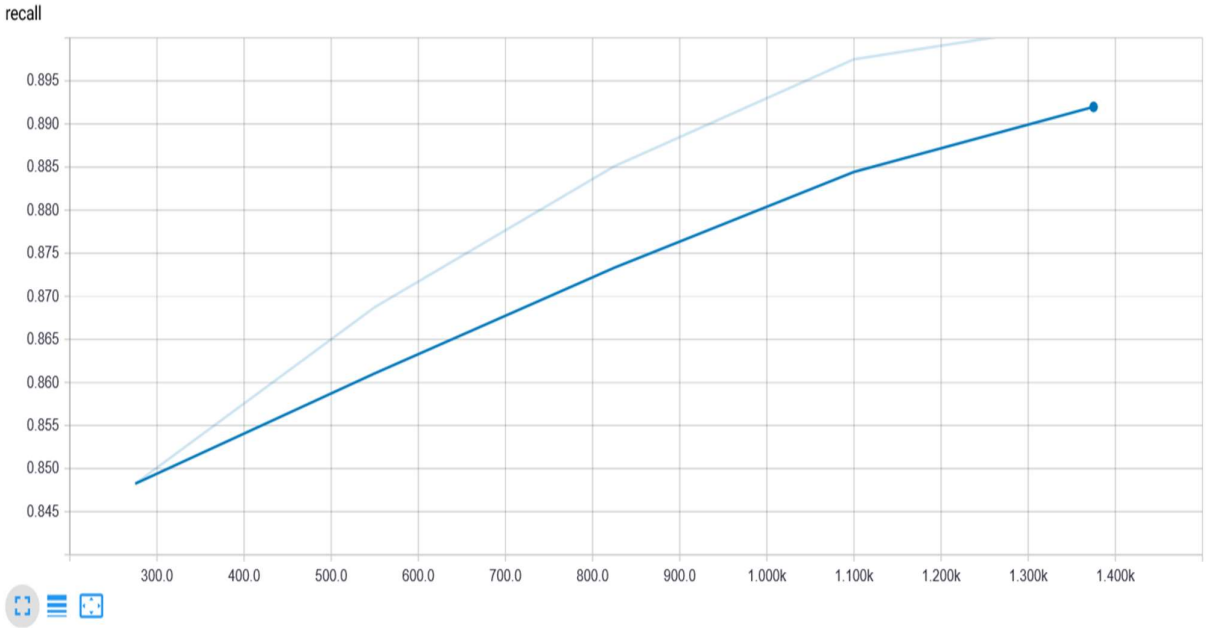
Graph-3: Plot of evaluation loss for each epoch



Graph-4: Plot of evaluation accuracy for each epoch



Graph-5: Plot for evaluation precision at each epoch



Graph-6: Plot for evaluation recall at each epoch

Values for last epoch of evaluation:

Epoch: 5

{'precision': 0.913738, 'accuracy': 0.907, 'loss': 0.2529326, 'recall': 0.9018526}

Deliverable 2:

Few parameters were changed in the CNN model to observe the changes and to select the parameters that will be useful for better performance of the model.

Adam optimizer:

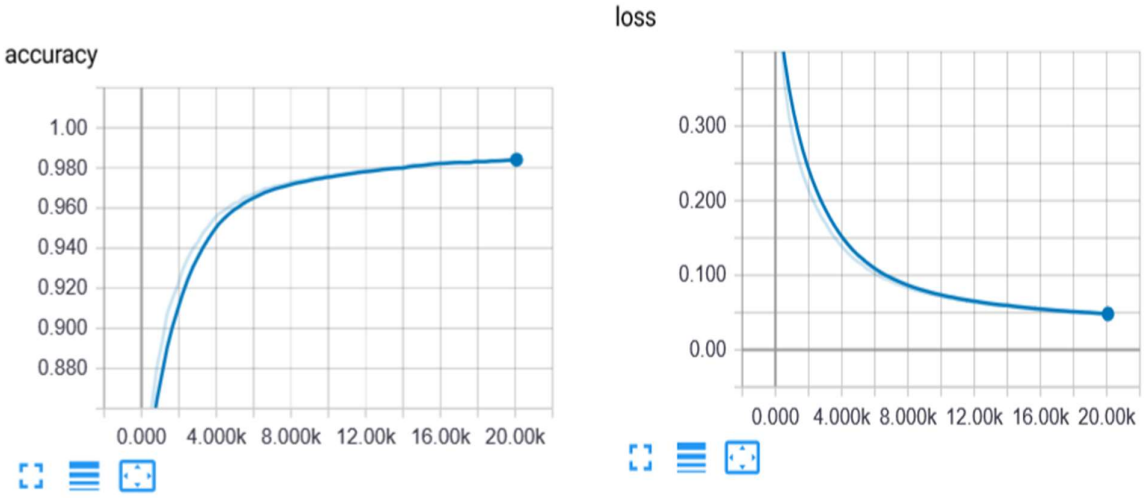
Changing the default gradient descent optimizer to Adam optimizer increased the performance of the model the most. Evaluation accuracy reached 98.87% for training up to only 5 epochs. The evaluation loss is also significantly reduced compared to the gradient descent optimizer.

Gradient descent optimizer (5 epochs):
{'loss': 0.2529326, 'accuracy': 0.907, 'precision': 0.913738, 'recall': 0.9018526}
Adam optimizer (5 epochs):
{'loss': 0.036992088, 'accuracy': 0.9887, 'precision': 0.994222, 'recall': 0.983445}

Number of epochs:

The evaluation accuracy of the model increases with increase in the number of epoch the model is trained for. The evaluation loss decreases with increase in the number of epochs the model is trained for. The model was trained up to a maximum of 73 epochs and compared with the default model trained for 5 epochs. Comparing the loss, accuracy of the last epoch in each model shows that the model trained for 73 epochs performs better than the models trained for lesser number of epochs.

5 epochs (default model):
{'loss': 0.2529326, 'accuracy': 0.907, 'precision': 0.913738, 'recall': 0.9018526}
30 epochs:
{'loss': 0.08116169, 'accuracy': 0.9734, 'precision': 0.9726701, 'recall': 0.97497046}
50 epochs:
{'loss': 0.058949225, 'accuracy': 0.9802, 'precision': 0.9844992, 'recall': 0.97635}
73 epochs:
{'loss': 0.047550872, 'accuracy': 0.9844, 'precision': 0.98596835, 'recall': 0.98324794}



Graph plots of evaluation accuracy and loss for the model trained for 73 epochs

Learning rate:

The learning rate was increased to 0.01 from 0.001 and the model was trained for 5 epochs. There was increase in the evaluation accuracy and decrease in evaluation loss of the model with the learning rate of 0.01

Learning rate - 0.01:

{'precision': 0.9755764, 'accuracy': 0.9794,'loss': 0.063331366, 'recall': 0.98403627}

Learning rate - 0.001(default):

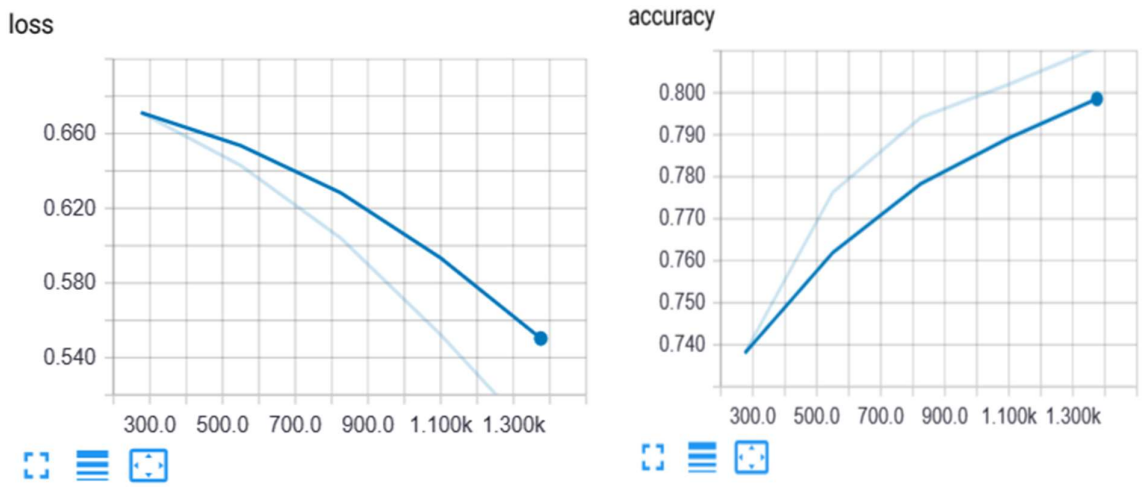
{'precision': 0.913738, 'accuracy': 0.907, 'loss': 0.2529326, 'recall': 0.9018526}

Using different weight initializers:

Xavier:

Using the Xavier weight initializer reduces the performance of the model against the default model, both being trained for 5 epochs. The model has lower evaluation accuracy and higher evaluation loss compared to the default model with fast_conv weight initializer.

Fast_conv(default):
{'loss': 0.2529326, 'precision': 0.913738, 'recall': 0.9018526 'accuracy': 0.907}
Xavier:
{'loss': 0.44857702, 'precision': 0.827553, 'recall': 0.8464722, 'accuracy': 0.8326}



Graph plots of evaluation accuracy and evaluation loss of the model trained with Xavier weight initializer.

He weights initializer:

The evaluation accuracy increases, and the evaluation loss decreases with the model using He weight initializer trained for 5 epochs. The model gives a better performance compared to the default model trained for 5 epochs.

Fast_conv(default):
{'loss': 0.2529326, 'precision': 0.913738, 'recall': 0.9018526 'accuracy': 0.907}
He_rec:
{'loss': 0.19666673, 'precision': 0.92719483, 'recall': 0.9387071, 'accuracy': 0.9315}

Graph plots of evaluation loss and accuracy for the model with He weight initializer

Deliverable 3:

Problem description:

To develop an application:
1. Accept as input an image of a handwritten digit. Assume each image contains one digit
2. Using OpenCV do some basic image preprocessing to prepare the image for your CNN
a. Resize the image to fit your model's image size requirement
b. Transform grayscale image to a binary image. Consider using the GaussianBlur() and adaptiveThreshold(), or any other type of binary thresholding that performs well.
c. Display the original image and binary image in two separate windows.
3. Using your CNN classify the binary image.
4. Your program should continuously request the path to an image file, process the image, and output the class (even/odd) of the image. The output could be displayed to the console or printed on the displayed image.
5. Program should terminate when 'q' or ESC key is entered.

Implementation details:

The application uses the model trained in the deliverable 1. The model function is imported to the application and the new image is predicted as odd or even using the TensorFlow estimator prediction.

Loading the trained model function to the cnn_test application:

model_fun = cnn.cnn_model_fn

image_classifier = tf.estimator.Estimator(model_fn=model_fun, model_dir=model_dir, params=parameters)

To predict the new image input:

predictions = image_classifier.predict(predict_input_fn)

Operations performed on the image before prediction include:

- Converting the image to grayscale by default in case of a RGB input image
- Resizing the given image to the size supported by the CNN model that is 28*28
- Applying the gaussian blur function in OpenCV to reduce noise

- Adaptive thresholding the image to convert it into a binary image using cv2.adaptiveThreshold() from OpenCV
- The image values are then flattened and reshaped into an array of size 1x748 to send as input to the estimator predictor

Instructions to execute:

- To train the model first execute the file cnn.py. (The model is trained and is present in the models folder)

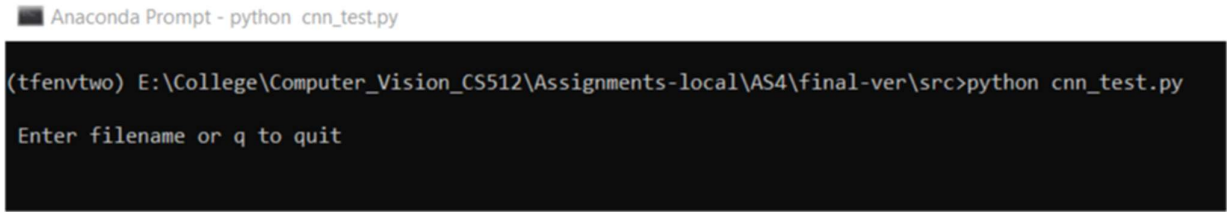
  Python cnn.py

- To visualize the training and evaluation results using tensorboard execute:

  Tensorboard - -logdir=../models/cnn_model

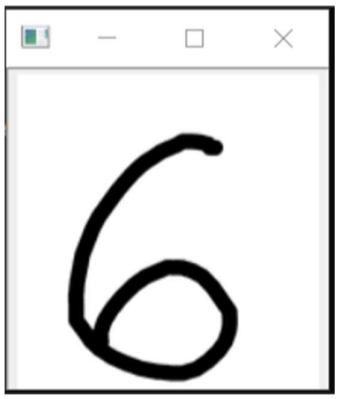
- After the model is trained execute the cnn_test.py to input an image of choice:
  Python cnn_test.py
- There will be a prompt to enter the filename of the image to predict using the CNN. Input the filename. The application displays both the original image and binary image. Press any key keeping the image window in the foreground to close the image windows and see the predicted value of the application if the given number is odd or even.
- Few images on which the application was tested is provided in the data folder.

Results and discussion:

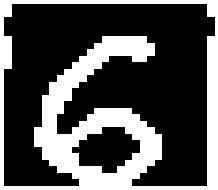


Prompt to enter the filename

EVEN NUMBER:



Original Image



Binary Image

```
 Enter filename or q to quit
../data/6.png
Processing..


##############################################

 It is an even number

##############################################
```

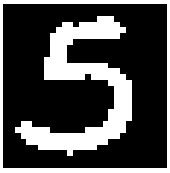
Prediction output in the console

ODD NUMBER:

```
 Enter filename or q to quit
../data/5.png
Processing..

########################################

It is an odd number

########################################
```

Console output

Original Image



Binary Image

```
(tfenvtwo) E:\College\Computer_Vision_CS512\Assignments-local\AS4\final-ver\src>python cnn_test.py

 Enter filename or q to quit
../data/5.png
Processing..

##########################################

It is an odd number

##########################################


 Enter filename or q to quit
```

The application asks for another filename after the prediction of one image. The application quits when the user inputs ‘q’ to the console.

References:

- TensorFlow estimator: https://www.tensorflow.org/tutorials/estimators/cnn

- OpenCV: https://docs.opencv.org/3.0-beta/index.html