The George Washington University

Crowd Counting

Machine Learning II- Deep Learning

Anwesha Tomar, Jingya Gao, Wenyu Zeng
12-1-2020

Introduction:

When there is an influx of a large number of people it can cause accidents such as a stampede. During sports events, large stadiums are full of people, this project can be used in order to analyze and predict the number of people in each area and prevent any accidents from taking place. It can also be used to analyze the number of people in a mall in order to prevent overcrowding and avoid mishaps from taking place. The relevance of this project can be seen in 2020, when there is an outbreak of a virus, it is mandatory to remain indoors, this project can be used to alert authorities when the number of people goes above a certain threshold. The goal of this project is to be able to predict the number of people in each image.

In this report, we describe the dataset used, the architecture of the two models, explanations of our results, and finally evaluation of our models.

Dataset:

The dataset consists of a collection of 2000 images, each image has a size of 480x640 pixels and has 3 channels. The images were collected from a still webcam that is located inside a mall. Each image has a different number of people. We are provided with a CSV file containing labels that give the number of people present in each image.

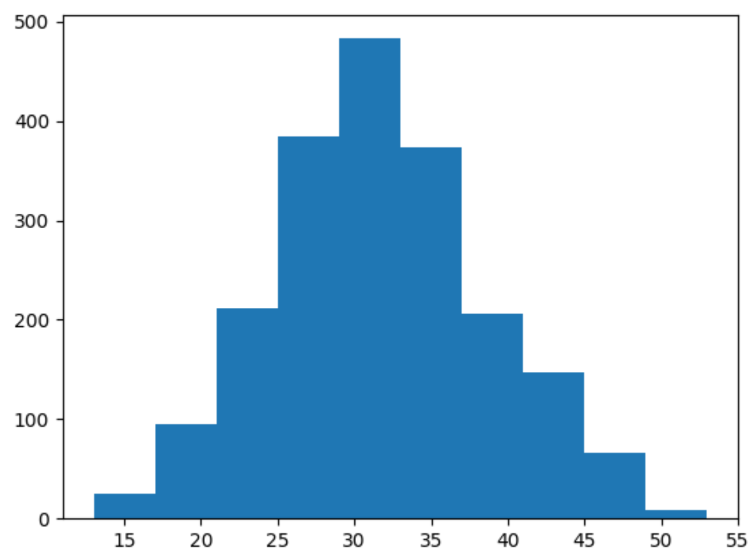

Fig 1. Distribution of labels

The image data is displayed below.

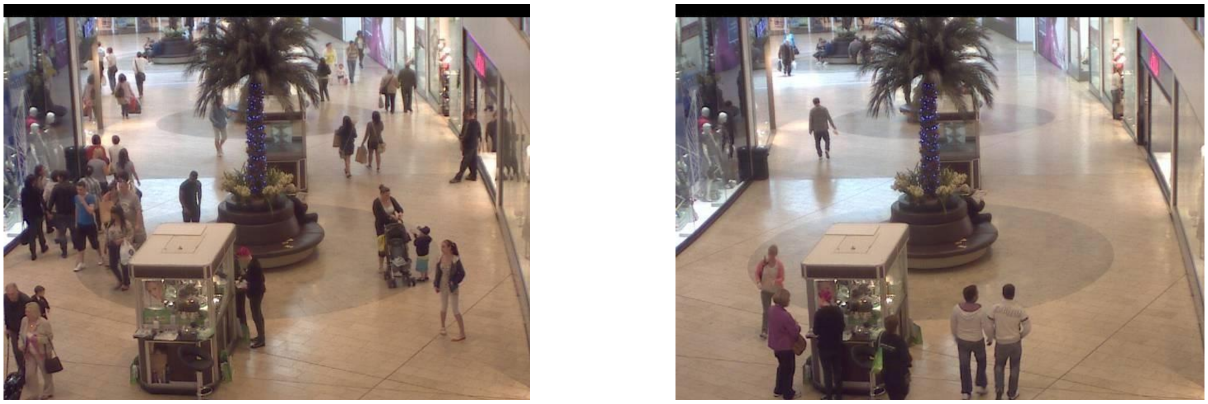

Fig 2. Image data that is input to the training models

Data Cleaning and Preprocessing:
The data is preprocessed and resized to 128x96 in order to store and train the data easily.

```
x = []
for path in [f for f in os.listdir(DATA_DIR) if f[-4:] == ".jpg"]:
    x.append(cv2.resize(cv2.imread(DATA_DIR + path), (RESIZE_TO, RESIZE_TO)))
x = np.array(x)
print("x.shape:\n", x.shape)
```

Fig 3. Code to pre-process data

As the data is not evenly distributed, we introduce an image generator.

```
# add ImageDataGenerator
datagen = ImageDataGenerator(
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    rotation_range=30,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    vertical_flip=False,
    shear_range=0.5)
```

Fig 4. Code for image augmentation

Architecture of the models:

Background:

For this project, we are using CNN and a pre-trained ResNet 50 model. We are using the CNN model mainly because it's a fully connected feed-forward neural network. CNN is very effective in reducing the number of parameters and maintain the image quality at the same time. By adding a regression layer at the end of the network, we are able to fit the model with this regression problem.

A similar reason is used for the pre-trained ResNet 50 model. ResNet 50 is a 50 layers Residual Network. We can also change the output layer to make the model predict the number of people in the image. The ResNet 50 model differs from the ordinary model in that it's using skip or shortcut connections. With the shortcut connections, the model measures the residuals calculate by each layer, and make sure the higher layer performs at least as good as the lower layer, and not worse.

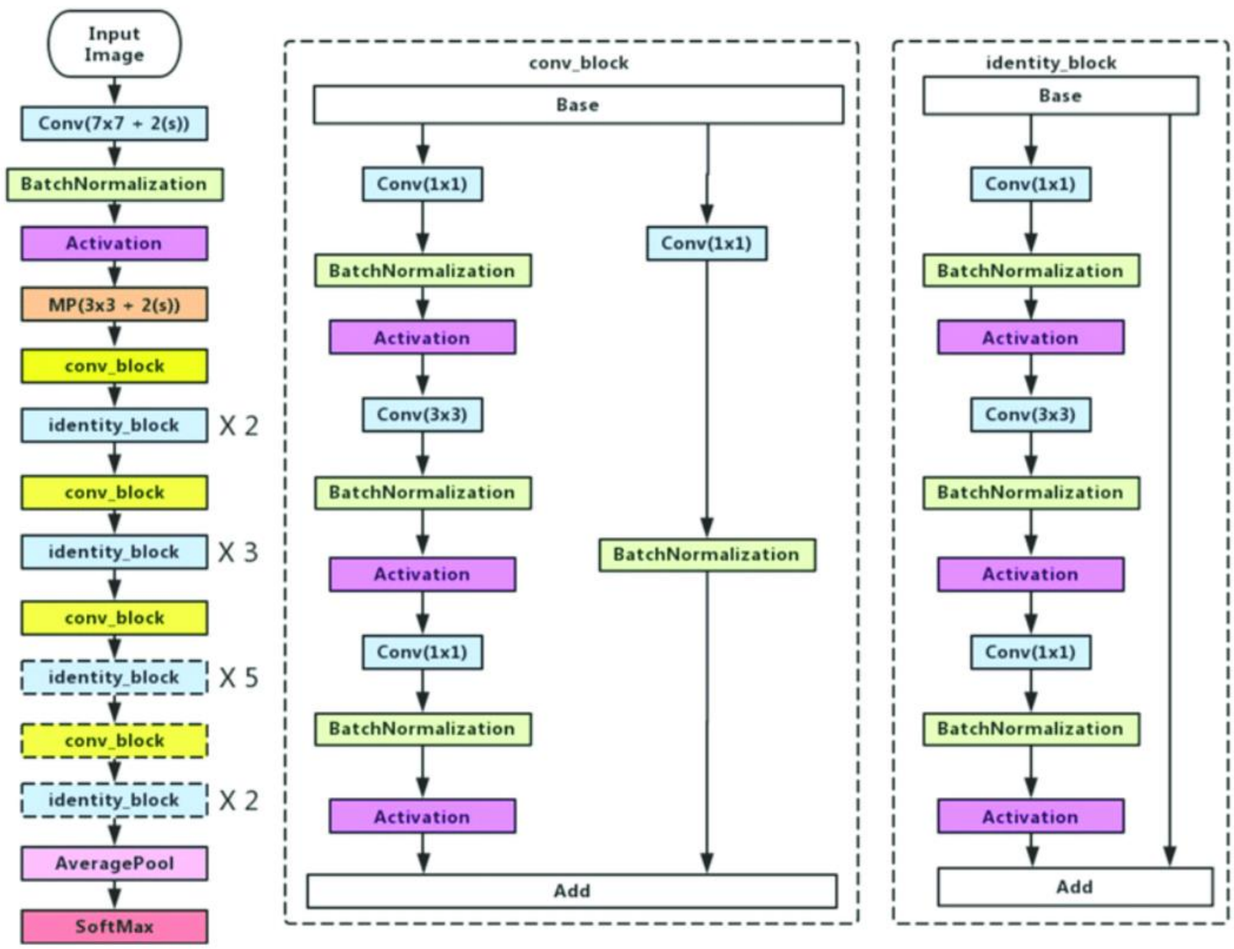

Fig 5. Sample ResNet-50 Architecture

Hyper-parameter selection:

We use various techniques to find the appropriate learning rate, batch size, and numbers of epochs. In order to fine-tune the model, we use a Learning Rate scheduler to find the best learning rate for the model.

```
lr_monitor = tf.keras.callbacks.LearningRateScheduler(
    lambda epochs: 1e-8 * 10 ** (epochs / 20))

model_check = ModelCheckpoint("1205-CNN.hdf5", monitor="val_loss", verbose=1, save_best_only=True

learning_rate_reduction = ReduceLROnPlateau(
    monitor='val_mean_absolute_error',
    patience=3,
    verbose=1,
    factor=0.2,
    min_lr=0.000001
)

epochs = 100
datagen.fit(x_train)
# fits the model on batches with real-time data augmentation:
history = model.fit(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_monitor, model_check],
                    shuffle=True)
```

Fig 6. Code for finding appropriate learning rate

From this we select the learning rate to be 3e-4, as it gives us the least loss value. We make use of the Learning Rate Scheduler that reduces the learning rate in order to best fit the model.


```
lr_monitor = tf.keras.callbacks.LearningRateScheduler(
    lambda epochs: 1e-8 * 10 ** (epochs / 20))

model_check = ModelCheckpoint("1205-CNN.hdf5", monitor="val_loss", verbose=1, save_best_only=True

learning_rate_reduction = ReduceLROnPlateau(
    monitor='val_mean_absolute_error',
    patience=3,
    verbose=1,
    factor=0.2,
    min_lr=0.000001
)

epochs = 100
datagen.fit(x_train)
# fits the model on batches with real-time data augmentation:
history = model.fit(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_monitor, model_check],
                    shuffle=True)
```


Fig 7. Code for the hyper-parameters

The batch size is selected as 32. As you can see in the figures below the loss value drops quickly and considerably for batch size 32, while for batch size 64 and 128 it takes longer for the loss values to drop.

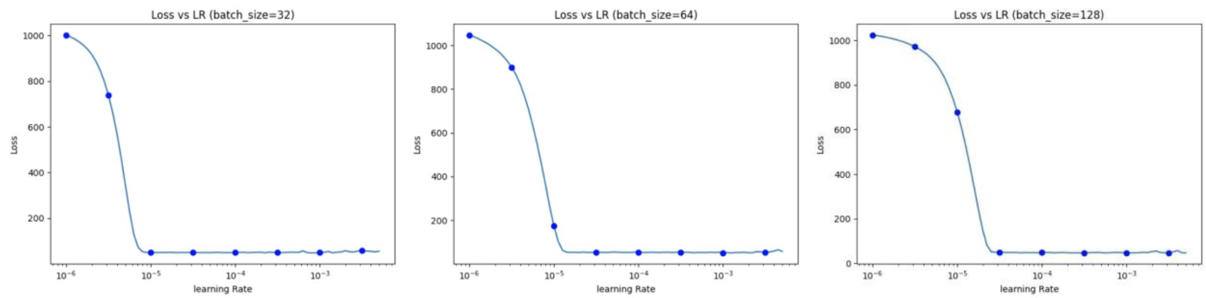

Fig 8. Learning rate vs Loss

Training CNN model:

While training the CNN model we make use of relu as the activation function and Adam as the optimizer function.

```
# %% ------------------------------------- Model Prep -----------------------------------------

# create CNN model
model = tf.keras.Sequential([

    tf.keras.layers.Conv2D(16, (5, 5), input_shape=(128, 96, 3), activation=tf.keras.activations.
    tf.keras.layers.MaxPool2D(2, 2),
    tf.keras.layers.Conv2D(32, (3, 3), activation=tf.keras.activations.relu),
    tf.keras.layers.MaxPool2D(2, 2),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(),
    # tf.keras.layers.Dense(400, activation=tf.keras.activations.relu),
    # tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(1)

])

model.compile(loss="mean_squared_error",  # This is a classic regression score - the lower the be
              metrics=['mean_absolute_error'],
              optimizer=tf.keras.optimizers.Adam(lr=3e-4))
```

Fig 9. Code to compile model

For fitting the model we make use of 100 epochs, increasing the number of epochs cause the model to overfit the data, therefore we chose 100 epochs. The final model was trained on a batch size of 32.

```

epochs = 100
datagen.fit(x_train)
# fits the model on batches with real-time data augmentation:
history = model.fit(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_monitor, model_check],
                    shuffle=True)

```

Fig 10. Code to fit the model

Training ResNet 50 model:

The data is set up by using the “flow from dataframe” function. The data is split into training and validation.

```
datagen = ImageDataGenerator(
    rescale=1. / 255,
    featurewise_center=False,
    samplewise_center=False,
    featurewise_std_normalization=False,
    samplewise_std_normalization=False,
    zca_whitening=False,
    horizontal_flip=True,
    vertical_flip=True,
    validation_split=0.2,

    preprocessing_function=resnet50.preprocess_input
)

flow_params = dict(
    dataframe=label,
    directory='/home/ubuntu/Deep-Learning/Final_Project/frames/frames',
    x_col="image_name",
    y_col="count",
    weight_col=None,
    target_size=(128, 96),
    color_mode='rgb',
    class_mode="raw",
    batch_size=batch,
    shuffle=True,
    seed=0
)

# The dataset is split to training and validation sets
train_generator = datagen.flow_from_dataframe(
    subset='training',
    **flow_params
)
valid_generator = datagen.flow_from_dataframe(
    subset='validation',
    **flow_params
)
```

Fig 11. Code snippet for data splitting

Training the model, the top seven layers, the last parts of the pre-trained model, are made untrainable to avoid overfitting and also to fit this regression problem. We add one dense layer and one final output layer that gives out one parameter, the prediction of the number of people in the image.

```
base_model = resnet50.ResNet50(
    weights='imagenet',
    include_top=False,
    input_shape=(128, 96, 3),
    pooling='avg'
)

# Change the top (the last parts) of the network.
x = base_model.output
x = Dense(1024, activation='relu')(x)
predictions = Dense(1, activation='linear')(x)
model = Model(inputs=base_model.input, outputs=predictions)
k = -7
for layer in model.layers[:k]:
    layer.trainable = False
print('Trainable:')
for layer in model.layers[k:]:
    print(layer.name)
    layer.trainable = True
```

Fig 12. Compiling ResNet 50 model

Finally to compile the model we make use of the Adam optimizer. For training the model, we apply a learning rate monitor that focuses on the validation mean squared error. For every 3 epochs, if the mean squared error doesn’t go down, the learning rate will drop by 20%, the lowest learning rate will not go beyond 1e-6.

```

optimizer = Adam(learning_rate=3e-4)

model.compile(
    optimizer=optimizer,
    loss="mean_squared_error",  # This is a classic regression score - the lower the better
    metrics=['mean_absolute_error', 'mean_squared_error']
)

learning_rate_reduction = ReduceLROnPlateau(
    monitor='val_mean_squared_error',
    patience=3,
    verbose=1,
    factor=0.2,
    min_lr=0.000001
)

model_check = ModelCheckpoint('resnet50_cc.hdf5', monitor='val_loss', verbose=2, save_best_only=T

# Fit the model
history = model.fit(
    train_generator,
    epochs=105,
    validation_data=valid_generator,
    verbose=2,
    callbacks=[learning_rate_reduction, model_check]
)
```

Fig 13. Optimizer and training code

Evaluation:

Models	Mean Absolute Error	Mean Square Error
CNN	5.55	50.29
ResNet 50	4.92	36.31

Fig 14. Table with metrics evaluations

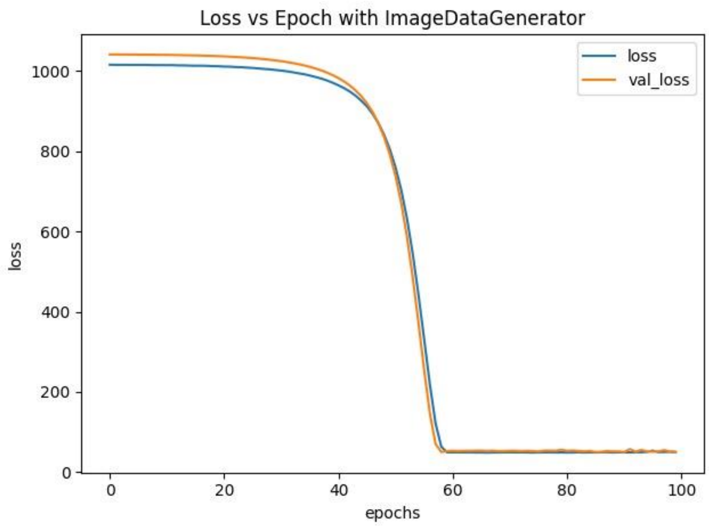

Fig 15. CNN Training loss & validation loss

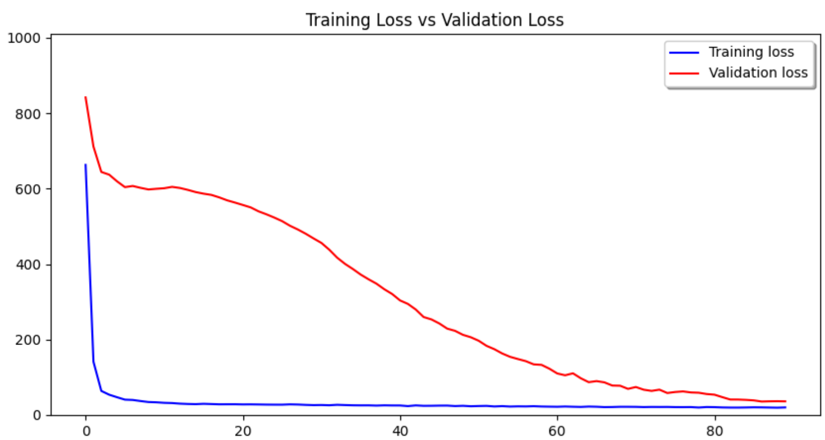

Fig 16. ResNet50 Training loss & validation loss

Prediction results:

- CNN model:

Fig 17. CNN model predict results vs actual results

- ResNet 50 model:

Fig 18. ResNet 50 model predict results vs actual results

Conclusion:

We can see that the ResNet50 model performed better than the CNN model, but there is still room for improvement. In future we would look to collect more data and improve the performance of our model. As of now we are limited because the data collected does not have any images for more than 55 people, our next step would be to find such data points.

References:

1. https://www.researchgate.net/figure/Left-ResNet50-architecture-Blocks-with-dotted-line-represents-modules-that-might-be_fig3_331364877
2. https://www.kaggle.com/shovalt/crowd-counting-keras-pretrained-resnet50-cnn
3. https://www.kaggle.com/rmishra258/counting-crowd-with-cnn-social-distancing-project
4. https://www.mathworks.com/help/deeplearning/ref/resnet50.html#:~:text=ResNet%2D50%20is%20a%20convolutional,%2C%20pencil%2C%20and%20many%20animals.