4/8/2021

HeadCount

Report for AI2004 Final Project

Team:

Amulya Pedapudi

Tejas Vyas

Facilitator:

Dr. Sikder Kamruzzaman

In this report:

Problem Definition and Project Pitch

Problem Definition

Crowd Counting has a variety of uses, including counting the number of people who attend political gatherings, social and sporting activities, and so on.

In today's time of social distancing, this problem becomes even more important, as counting crowds allow public health agencies and governments to assess rule enforcement as well as to prepare analysis on spread of the pandemic.

In terms of implementation using AI, crowd counting is an extremely hard issue, particularly in dense crowds. This is primarily due to:

- Clutter, overlap, and occlusions being a common occurrence in places of crowd.
- In a single perspective view, it's difficult to consider the shape and scale of the object in relation to the context.

Project Pitch

In order to solve the problem described above, we can employ a convolutional neural network in combination with density map estimation, which predicts a density map over the input image before summing to obtain the object count.

In order to do this, our idea is to create a web application called **HeadCount** an existing algorithm ResNet50, and apply Transfer Learning to it to allow the model to use our new dataset and get trained to understand how to count crowds in a given set of images.

Exploratory Data Analysis

Dataset

In order to prepare the web application, we use the data obtained from a Kaggle Dataset – Crowd Counting.

This dataset is accessible at:

https://www.kaggle.com/fmena14/crowd-counting

The dataset is made up of RGB images of video frames (as inputs) and an object counting on each frame, which is the number of pedestrians (objects) in the image. The photos are 480x640 pixels at three channels of the same spot taken by a webcam in a mall, but each frame has a different number of people, indicating a crowd counting problem. It uses data from an earlier research paper – Mall Dataset, which can be accessed here:

http://www.bmva.org/bmvc/2012/BMVC/paper021/paper021.pdf

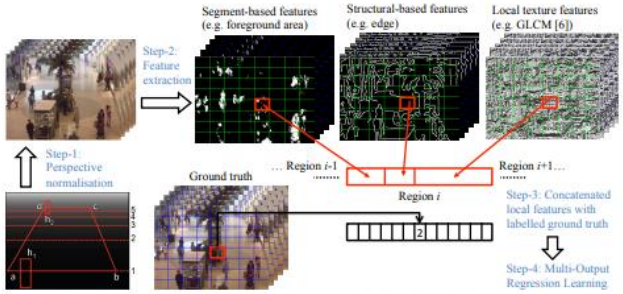


Figure 2: A multi-output regression framework for localised crowd counting by feature mining.

We include the image data in the form of a NumPy binary file (.npy format) that you can load with the numpy load feature. If you want to load the JPG images one by one, we include them in a folder.

Each image has one label/target that represents the number of people on the frame JPG image. These are counting integer numbers; for example, there are 29 objects in the image below (people).

Exploratory Analysis

Because this is a simple image dataset, our EDA is going to be limited to checking the dataset and reviewing the images. Let us see how much memory our dataset is consuming without the images pre-loaded. We have 2000 records with each corresponding to the individual image.

```
In [3]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   id          2000 non-null   int64 
 1   count       2000 non-null   int64 
 2   image_name  2000 non-null   object
dtypes: int64(2), object(1)
memory usage: 47.0+ KB
```

Let us see some stats about the head count:

```
In [4]: df.describe()
```

Out[4]:

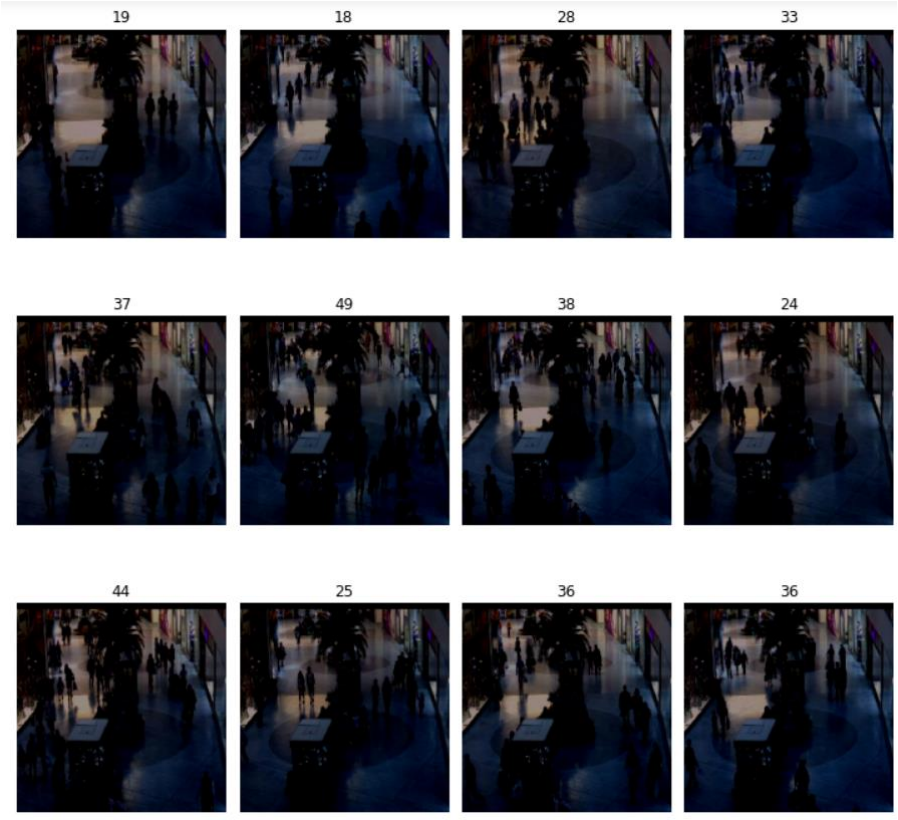
	id	count
count	2000.000000	2000.000000
mean	1000.500000	31.157500
std	577.494589	6.945417
min	1.000000	13.000000
25%	500.750000	26.000000
50%	1000.500000	31.000000
75%	1500.250000	36.000000
max	2000.000000	53.000000

In order to review the images, we'd be using the ImageDataGenerator module. Let's create an instance of it and bind the necessary data:

In [5]:
```
# We can use the size 224 and a batch_size of 32
size = 224
batch_size = 32

img_datagen = ImageDataGenerator(
    rescale = 1.0/255.0,  # We rescale the pixels to be between 0 and 1.
    featurewise_center = False, samplewise_center = False,  # set centering mean values to 0
    featurewise_std_normalization = False,  samplewise_std_normalization = False,  # no normalization needed
    zca_whitening = False,
    horizontal_flip = False, vertical_flip = False, # no image manipulations needed
    validation_split = 0.2,  # dividing data into 80-20 split
    preprocessing_function = resnet50.preprocess_input, #Setting resnet as preprocessing function
)
```

In [7]:
```
params = dict(
    dataframe=df,
    directory='frames/frames',
    x_col="image_name",
    y_col="count",
    weight_col=None,
    target_size=(size, size),
    color_mode='rgb',
    class_mode="raw",
    batch_size=batch_size,
)

# Splitting Dataset
training_datagen = img_datagen.flow_from_dataframe(subset = 'training', **params)
testing_datagen = img_datagen.flow_from_dataframe(subset = 'validation', **params)

#Let's review a sample of the images
batch = next(training_datagen)
fig, axes = plt.subplots(3, 4, figsize=(10, 10))
axes = axes.flatten()
for i in range(12):
    ax = axes[i]
    ax.imshow(batch[0][i])
    ax.axis('off')
    ax.set_title(batch[1][i])
plt.tight_layout()
plt.show()
```

With the logic above we can see the count and images:



Solution Development

Core Architecture

Our web application has 2 main components:

- Deep Learning Model using Transfer Learning.
- Flask Web Application for user image uploading and results.

Deep Learning Model

To begin with our model we begin with loading an instance of ResNet50 model with pretrained weights using the ImageNet dataset and average pooling applied to it. We can then use this model and transfer learning to train a model which can manage our updated dataset:

```
In [8]: resnet_model = resnet50.ResNet50(weights = 'imagenet', include_top = False,  input_shape = (size, size, 3),  pooling = 'avg')
```

```
In [9]: # Output is pooling layer
        pooling_x = resnet_model.output

        # Adding Single Connected Layer
        pooling_x = Dense(1024, activation = 'relu')(pooling_x)

        # Output layer - returns number of people in the image
        output_y = Dense(1, activation = 'linear')(pooling_x)

        # Setting up the model
        model = Model(inputs = resnet_model.input, outputs = output_y)
```

Now we can review the Model:

```
In [10]: # Let's see how our model looks
         model.summary()
```

```
__________________________________________________________________________________________________
res5c_branch2c (Conv2D)         (None, 7, 7, 2048)   1050624     activation_47[0][0]
__________________________________________________________________________________________________
bn5c_branch2c (BatchNormalizati (None, 7, 7, 2048)   8192        res5c_branch2c[0][0]
__________________________________________________________________________________________________
add_15 (Add)                    (None, 7, 7, 2048)   0           bn5c_branch2c[0][0]
                                                                 activation_45[0][0]
__________________________________________________________________________________________________
activation_48 (Activation)      (None, 7, 7, 2048)   0           add_15[0][0]
__________________________________________________________________________________________________
global_average_pooling2d (Globa (None, 2048)         0           activation_48[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 1024)         2098176     global_average_pooling2d[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 1)            1025        dense[0][0]
==================================================================================================
Total params: 25,686,913
Trainable params: 25,633,793
Non-trainable params: 53,120
__________________________________________________________________________________________________
```

With this is model is ready for training. We train the model for 40 epochs using Adam optimizer and Mean Squared Error as the loss function.

In addition, we add a Learning Rate annealer to make sure the rate is adjusted dynamically when a plateau is hit.

Finally, we save the model as hdf5 file so it can be loaded later in the web app.

In [13]:
```
# Model Compilation
model.compile( optimizer = Adam(lr = 0.001), loss = "mean_squared_error",
              metrics = ['mean_squared_error', 'mean_absolute_error'])

# Setting up a Learning Rate Annealer to dynamically decrease learning rate on Plateus
learning_rate_reduction = ReduceLROnPlateau(monitor = 'val_mean_squared_error', patience = 3, verbose = 1, factor = 0.25, min_lr

# Model Fitting
history = model.fit_generator(generator = training_datagen, epochs = 40,
                              validation_data = testing_datagen, verbose = 2,
                              callbacks=[learning_rate_reduction])
```

```
 - 23s - loss: 0.6375 - mean_squared_error: 0.6375 - mean_absolute_error: 0.6544 - val_loss: 3.5335 - val_mean_squared_erro
r: 3.5335 - val_mean_absolute_error: 1.4651
Epoch 35/40
 - 23s - loss: 1.2684 - mean_squared_error: 1.2684 - mean_absolute_error: 0.8823 - val_loss: 3.5939 - val_mean_squared_erro
r: 3.5939 - val_mean_absolute_error: 1.4756
Epoch 36/40
 - 23s - loss: 0.8683 - mean_squared_error: 0.8683 - mean_absolute_error: 0.6559 - val_loss: 3.5499 - val_mean_squared_erro
r: 3.5499 - val_mean_absolute_error: 1.4675
Epoch 37/40
 - 23s - loss: 0.8385 - mean_squared_error: 0.8385 - mean_absolute_error: 0.7356 - val_loss: 3.5062 - val_mean_squared_erro
r: 3.5062 - val_mean_absolute_error: 1.4605
Epoch 38/40
 - 23s - loss: 0.7533 - mean_squared_error: 0.7533 - mean_absolute_error: 0.6999 - val_loss: 3.5387 - val_mean_squared_erro
r: 3.5387 - val_mean_absolute_error: 1.4655
Epoch 39/40
 - 23s - loss: 0.7832 - mean_squared_error: 0.7832 - mean_absolute_error: 0.6864 - val_loss: 3.5237 - val_mean_squared_erro
r: 3.5237 - val_mean_absolute_error: 1.4633
Epoch 40/40
 - 24s - loss: 1.0077 - mean_squared_error: 1.0077 - mean_absolute_error: 0.7884 - val_loss: 3.5619 - val_mean_squared_erro
r: 3.5619 - val_mean_absolute_error: 1.4698
```

Let's save the model

In [14]:
```
model.save("headcount_model.hdf5")
```

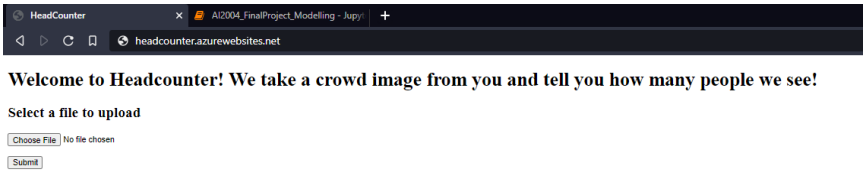
Web Application

In order to consume our model, we create a web application using Flask framework where we add a simple index.html which supports uploads from a user, loads the h5 model and performs a prediction and returns the results back to the user.

We begin with loading the h5 model to the app.py:

```
# Imports
import os
from flask import Flask, flash, request, redirect, url_for, render_template
from werkzeug.utils import secure_filename
from keras.models import load_model
import PIL.Image
from keras.applications.resnet50 import preprocess_input
from keras.preprocessing import image
import numpy as np

# Load Model
model = load_model("headcount_model.h5")

# Setup Flask app with template_folder path as "pages/"
app = Flask(__name__, template_folder="pages")
app.secret_key = "8sd9fh39fgh398fh3"
UPLOAD_FOLDER = 'static/'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024

ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg'}
```

On the user end, the user can see the below landing page and upload a file.



When file is uploaded, we use the POST call to get the file, and then use the predict function to process the file into a correct dimension and then use our model to create a prediction, returning back the image and prediction of number of people.

```python
def predict(imagepath):
    """Takes a valid 64bit image file, preprocessed and returns the final prediction"""
    img = image.load_img(imagepath, target_size=(224, 224))
    img = image.img_to_array(img)
    x = preprocess_input(np.expand_dims(img.copy(), axis=0))
    prediction = model.predict(x)
    return np.concatenate(prediction)


def allowed_file(filename):
    """Take a filename and returns whether the file is of correct format"""
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS


# Default route handling
@app.route('/', methods=['GET', 'POST'])
def initialize():
    """Handles home page and post calls retrieving the image query"""
    if request.method == 'POST':
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        file = request.files['file']
        if file.filename == '':
            flash('No image selected for uploading')
            return redirect(request.url)
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            # print('upload_image filename: ' + filename)
            flash('Image successfully uploaded - see our prediction below:')
            result = predict(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return render_template('index.html', filename=filename, prediction=str(round(result[0]/100)))
        else:
            flash('Allowed image types are -> png, jpg, jpeg, gif')
            return redirect(request.url)
    return render_template('index.html')
```

Solution Deployment

Deployment Infrastructure

In order to deploy our project, we have 3 components:

- ML Model h5 file
- GitHub Repository
- Azure App Service
    - The app service containerizes the application when used with Local Git on the back end, we do not need to manually create a Docker container for this when using the Kudu deployment tool.

Model HDF5 File

We create this file during the model building and this file holds the weights we received after training our model in the Jupyter notebook.

This file is included in the Flask app as a resource and does not need to be deployed specially. In case any changes are necessary, the core Git can be updated, and the Azure app service will redeploy the app with the updated model in place.

GitHub Repository

We create a GitHub repository to store our code, dataset, and any documentation to be reviewed. The source link for the repository can be seen below:

https://github.com/tejas1794/HeadCounter

This repository can also be linked to the final host to perform CI/CD.

Azure App Service

Azure app service is our main host for the web application. In order to upload on Azure App Service, we use their Kudu deployment service.

We create a local private git repository which houses a copy of our app. We do this for couple reasons:

- Having a private git allows us to safely leave CI/CD on, allowing immediate builds as soon as code is pushed to the master branch.
- Private git allows better security as well as the ability for us to only include Flask app and model, saving storage on the VM.

Kudu also containerizes our application in the backend when building, so we can also avoid having Docker container. The live version of the app can be accessed through:

https://headcounter.azurewebsites.net/

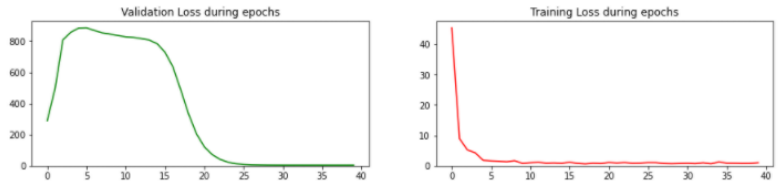
Azure does support Docker containers for deployment so in a subsequent version we can Dockerize the application for better portability (if a chance of hosts such as Google App Engine/Heroku/AWS is necessary).

Results

Deep Learning Model

After the model was built, we stored history by iteration which now allows us to review how the loss changed during training iterations:

```
In [24]: fig, ax = plt.subplots(1, 2, figsize = (15, 3))

         ax[0].set_ylim(top = np.max(history.history['val_loss'])*1.05)
         ax[1].set_ylim(top = np.max(history.history['loss'])*1.05)

         ax[0].set_title("Validation Loss during epochs")
         ax[1].set_title("Training Loss during epochs")

         ax[0].plot(history.history['val_loss'], color = 'g')
         ax[1].plot(history.history['loss'], color = 'r')

Out[24]: [<matplotlib.lines.Line2D at 0x1cf7ec2cf28>]
```



It seems like our annealer did help, as the validation loss decreased significantly during the course after 12th epoch. Now let us create predictions on the entire validation set to review our performance.

```
In [39]: # Going through the generators and labels and making predictions for each
testing_datagen.reset()
labels = []
preds = []
for i in range(len(testing_datagen)):
    img = next(testing_datagen)

    #Predict
    pred_i = model.predict(img[0])[:,0]
    labels_i = img[1]
    labels.append(labels_i)
    preds.append(pred_i)

# Create a concatenated array for all labels and preds
cat_labels = np.concatenate(labels)
cat_preds = np.concatenate(preds)

# Rounding the predictions to get a better idea of closeness
for i in range(len(cat_preds)):
    cat_preds[i] = round(cat_preds[i])

df_preds = pd.DataFrame({'Actual Count': cat_labels, 'Predicted Count': cat_preds})
```

```
In [41]: df_preds
```

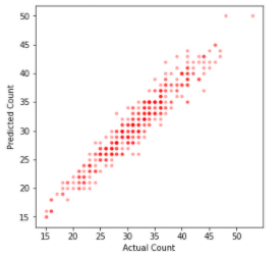
Out[41]:

	Actual Count	Predicted Count
0	25	26.0
1	35	37.0
2	34	32.0
3	28	29.0
4	28	33.0
...	...	...
395	21	20.0
396	18	20.0
397	34	31.0
398	28	26.0
399	33	29.0

400 rows × 2 columns

We can see the model is close with predictions. We can visualize better by a scatter plot:

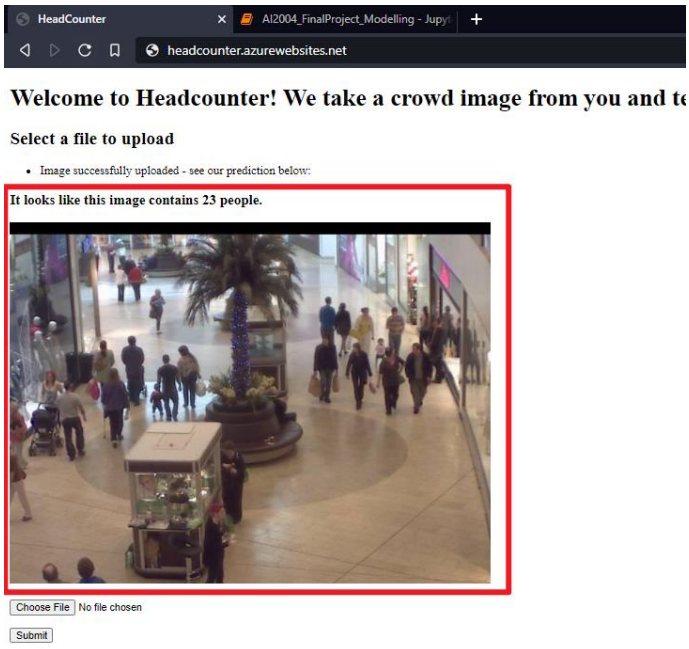
```
In [47]: ax = df_preds.plot.scatter('Actual Count', 'Predicted Count',
                                    alpha = 0.3, s = 10, figsize = (5,5), c = "r")
plt.show()
```



The scatter plot clearly shows us how the predicted and actual counts are remarkably close to a linear trendline, which shows our regression algorithm is extremely close the counts.

Flask Web Application

Below we can see how the app looks like when built, and how adding an image returns the count.



Coding and Repository Standards

In order to prepare an application which scalable and ready for enterprises, we followed best practices as applicable in the industry, such as following PEP8 guidelines in the code.

Additionally, with repositories we divided our project into logical directories, to allow easy review by development team.

Work Distribution

In order to successfully complete the project, the work was distributed equally between the team.

Task	Member	Time Taken
Initial Project Research	Amulya Pedapudi	4 hrs.
Proposal Preparation	Team	4 hrs.
Dataset Research	Amulya Pedapudi	2 hrs.
Exploratory Data Analysis	Amulya Pedapudi	4 hrs.
Data Preprocessing	Amulya Pedapudi	4 hrs.
Model Prototyping	Tejas Vyas	4 hrs.
Model Analysis	Tejas Vyas	4 hrs.
Web Application Development	Tejas Vyas	8 hrs.
Application Release and Deployment	Tejas Vyas	8 hrs.
Report Preparation	Team	4 hrs.

Additionally, we had weekly meetings to review progress from Proposal submission to Deadline day allowing seamless communication between the teammates.

Conflict Resolution

During the project, no conflicts arose between the team. We also had work divided equally between teammates to allow maximum performance.