PROJECT REPORT

ON

"VIRTUAL RECOGNITION"

Submitted to

International Institute of Information Technology Bangalore

BY

VIVEK CHAUDHARY	MT2019136
VIVEK GUPTA	MT2019137

Contents

Chapter 1

Dress Recognition

1.1 DEFINITION

1.1.1 PROBLEM STATEMENT

A camera is placed at the entrance of a retail shop. Your company supplies the video analytics solution to this shop. You are given a snap i.e. a image frame sampled every 5 second. Assuming people enter one-by-one at the entrance, you have to find if there is a person in every snap, and if so what are they wearing - "Formal Shirt/ T-shirt/ Saree/Kurti/None of the above.

1.1.2 DATA DESCRIPTION

The data set used here is downloaded using web scraping. We have scraped data using Beautiful-Soup python library. The final dataset consist of 7500 images, with 7000 images used for training and remaining 500 images are used for testing purpose. The size of each image so scraped vary between 520x420 to 600x500. Each image also has 3 color channels.

`https://drive.google.com/open?id=1-0FTQ_Y3SNrPn5mdCgHSp5ZZ4p1FuBJu`

1.2 CONCEPTS

1.2.1 YOLO : You Only Look Once

Object detection is one of the classical problems in computer vision: Recognize what the objects are inside a given image and also where they are in the image. Detection is a more complex problem than classification, which can also recognize objects but doesn't tell you exactly where the object is located in the image. YOLO is a clever neural network for doing object detection in real-time.

1.2.2 HOW YOLO WORKS

YOLO actually looks at the image just once (hence its name: You Only Look Once) but in a clever way. YOLO divides up the image into a grid of 13 by 13 cells:

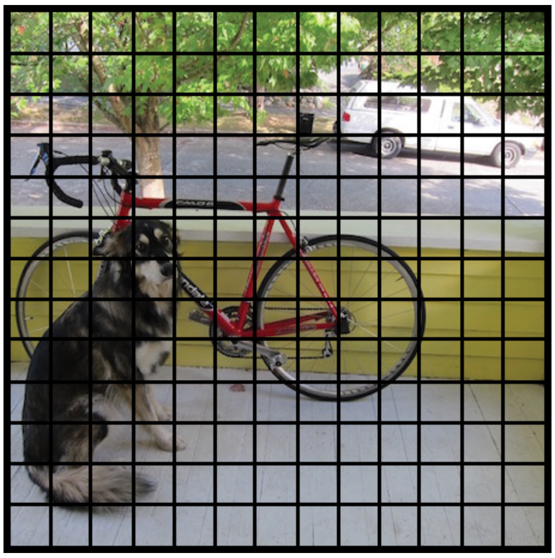

Figure 1.1

Each of these cells is responsible for predicting 5 bounding boxes. A bounding box describes the rectangle that encloses an object. YOLO also outputs a confidence score that tells us how certain it is that the predicted bounding box actually encloses some object. This score doesn't say anything about what kind of object is in the box, just if the shape of the box is any good. The predicted bounding boxes may look something like the following (the higher the confidence score, the fatter the box is drawn):

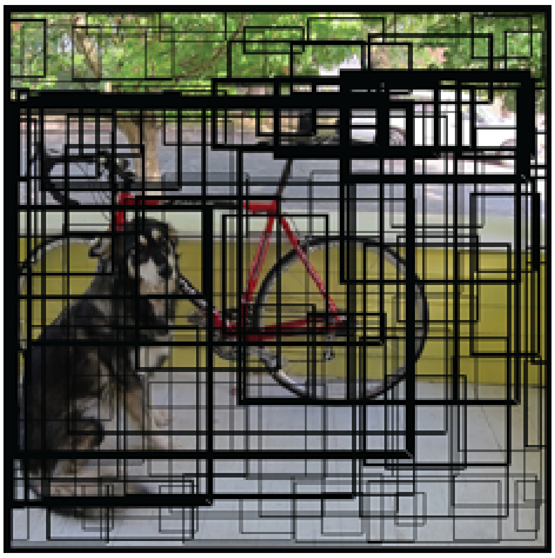

Figure 1.2

For each bounding box, the cell also predicts a class. This works just like a classifier: it gives a probability distribution over all the possible classes.

The confidence score for the bounding box and the class prediction are combined into one final score that tells us the probability that this bounding box contains a specific type of object. For example, the big fat yellow box on the left is 85% sure it contains the object "dog":

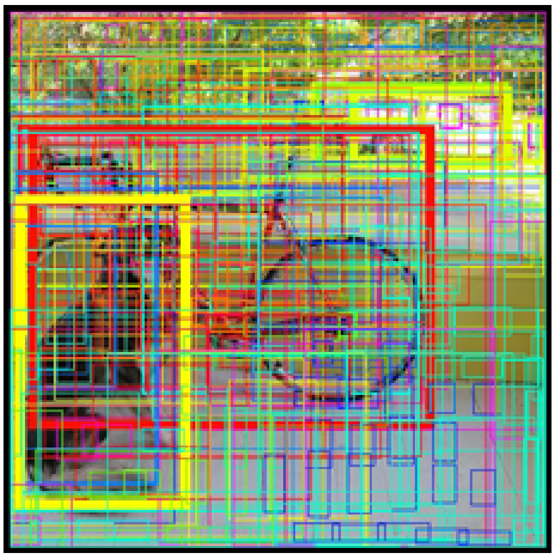

Figure 1.3

Since there are 13×13 = 169 grid cells and each cell predicts 5 bounding boxes, we end up with 845 bounding boxes in total. It turns out that most of these boxes will have very low confidence scores, so we only keep the boxes whose final score is 30% or more (you can change this threshold depending on how accurate you want the detector to be). The final prediction is then:

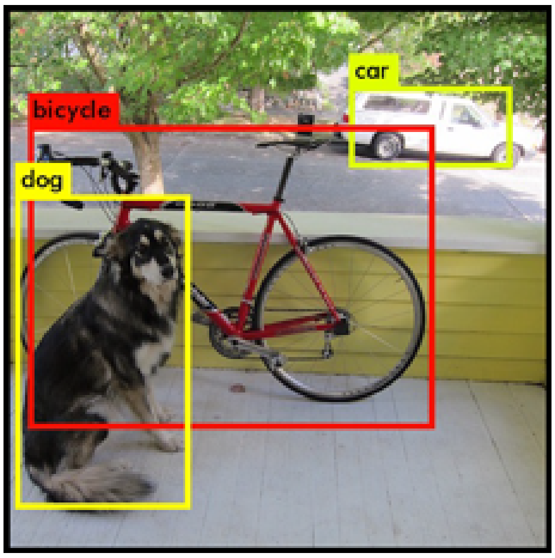

Figure 1.4

From the 845 total bounding boxes we only kept these three because they gave the best results. But note that even though there were 845 separate predictions, they were all made at the same time — the neural network just ran once. And that's why YOLO is so powerful and fast.

1.2.3 ARCHITECTURE of YOLO

The architecture of YOLO is simple, it's just a convolutional neural network:

```
Layer         kernel  stride  output shape
---------------------------------------------
Input                         (416, 416, 3)
```

```
Convolution    3×3      1      (416, 416, 16)
MaxPooling     2×2      2      (208, 208, 16)
Convolution    3×3      1      (208, 208, 32)
MaxPooling     2×2      2      (104, 104, 32)
Convolution    3×3      1      (104, 104, 64)
MaxPooling     2×2      2      (52, 52, 64)
Convolution    3×3      1      (52, 52, 128)
MaxPooling     2×2      2      (26, 26, 128)
Convolution    3×3      1      (26, 26, 256)
MaxPooling     2×2      2      (13, 13, 256)
Convolution    3×3      1      (13, 13, 512)
MaxPooling     2×2      1      (13, 13, 512)
Convolution    3×3      1      (13, 13, 1024)
Convolution    3×3      1      (13, 13, 1024)
Convolution    1×1      1      (13, 13, 125)
---------------------------------------------
```

This neural network only uses standard layer types: convolution with a 3×3 kernel and max-pooling with a 2×2 kernel.

1.3 METHODOLOGY

1.3.1 APPROACH

1. Download yolov3 weights, and build the pretrained Yolo model
2. Optimize the YOLO helper funtions
3. Preparing the data
4. Building and compiling of the classification model
5. Training and evaluating the classification model
6. Saving the classification model to disk for reuse
7. Using the YOLOv3 model and classification model to predict the category of each frame

1.3.2 YOLOv3 pretrained model

YOLO (You Only Look Once) is a clever neural network for doing object detection in real-time. Object detection is a computer vision task that involves both localizing one or more objects within an image and classifying each object in the image. As the first step we load the model weights to build the model. Later we save the model into model.h5 file.

```
# define the model
model = make_yolov3_model()
# load the model weights
weight_reader = WeightReader('yolov3.weights')
# set the model weights into the model
weight_reader.load_weights(model)
# save the model to file
model.save('model.h5')
```

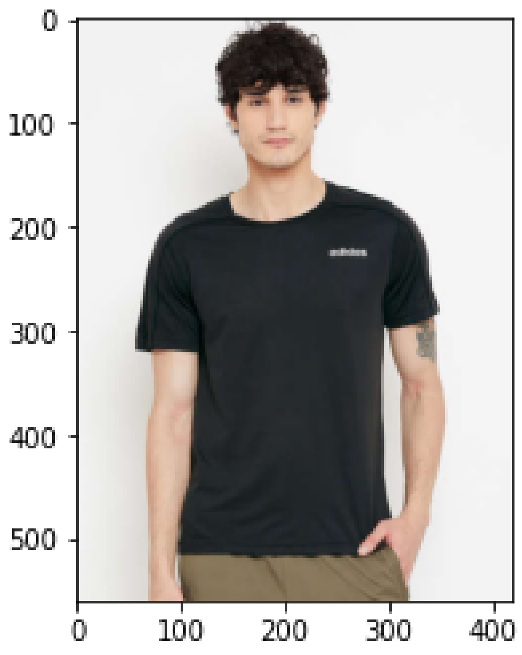

Figure 1.5: Given Input

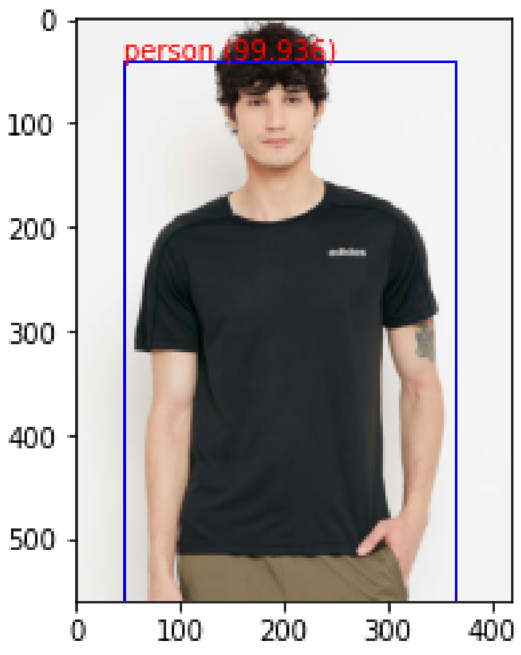

Figure 1.6: Output from yolo

1.3.3 Optimizing YOLO helper funtions

The helper functions provided by the experiencor are very slow when it comes to decoding the yolo model's output. It takes 2 seconds to process every frame. So we extensively optimize the experiencor script itself to give processing of 10 FPS.

```python
# load and prepare an image
def load_image_pixels(filename, shape):
    image = cv2.imread(filename)
    width = image.shape[1]
    height = image.shape[0]
    # load the image with the required size
    image = cv2.resize(image,shape)
    # scale pixel values to [0, 1]
    image = image.astype('float32')
    image /= 255.0
    # add a dimension so that we have one sample
    image = expand_dims(image, 0)
    return image, width, height
```

```python
def do_nms(boxes, nms_thresh):
  if len(boxes) > 0:
    # nb_class = len(boxes[0].classes)
    nb_class = 1
  else:
    return
  for c in range(nb_class):
    sorted_indices = np.argsort([-box.classes[c] for box in boxes])
    for i in range(len(sorted_indices)):
      index_i = sorted_indices[i]
      if boxes[index_i].classes[c] == 0: continue
      for j in range(i+1, len(sorted_indices)):
        index_j = sorted_indices[j]
        if bbox_iou(boxes[index_i], boxes[index_j]) >= nms_thresh:
          boxes[index_j].classes[c] = 0
```

```python
def decode_netout(netout, anchors, obj_thresh, net_h, net_w):
  grid_h, grid_w = netout.shape[:2]
```

```
nb_box = 3
netout = netout.reshape((grid_h, grid_w, nb_box, -1))
nb_class = netout.shape[-1] - 5
boxes = []
netout[..., :2]  = _sigmoid(netout[..., :2])
netout[..., 4:]  = _sigmoid(netout[..., 4:])
netout[..., 5:]  = netout[..., 4][..., np.newaxis] * netout[..., 5:]
netout[..., 5:] *= netout[..., 5:] > obj_thresh

for i in range(grid_h*grid_w):
  row = i / grid_w
  col = i % grid_w
  for b in range(nb_box):
    # 4th element is objectness score
    objectness = netout[int(row)][int(col)][b][4]
    if(objectness.all() <= obj_thresh): continue
    # first 4 elements are x, y, w, and h
    x, y, w, h = netout[int(row)][int(col)][b][:4]
    x = (col + x) / grid_w # center position, unit: image width
    y = (row + y) / grid_h # center position, unit: image height
    w = anchors[2 * b + 0] * np.exp(w) / net_w # unit: image width
    h = anchors[2 * b + 1] * np.exp(h) / net_h # unit: image height
    # last elements are class probabilities
    classes = netout[int(row)][col][b][5:]
    box = BoundBox(x-w/2, y-h/2, x+w/2, y+h/2, objectness, classes)
    boxes.append(box)
return boxes
```

Result of Optimization

```
start_time = time.time()
img_in_frame,v_boxes, v_labels, v_scores = detect_person('Dress_Recognition/Test-Dataset/T-Shirt.80.
    jpg')
print("--- %s seconds ---" % (time.time() - start_time))
```

Output:

```
--- 0.10506391525268555 seconds ---
```

1.3.4 Preparing the data

The images in the dataset are generated using webscraping and they vary in dimensions. Since the CNN model cannot accept data of varying dimensions, we have resized the images into (416x416x3) dimensions. This resizing is done with the help of cv2.resize() method.

```
x_train_all = []
y_train_all = []

shape = (416,416)
for dirname, _, filenames in os.walk('Dress_Recognition/Dataset'):
  for filename in filenames:
    img_in_frame,v_boxes, v_labels, v_scores = detect_person(os.path.join(dirname, filename))
    for i in range(len(img_in_frame)):
      color = (255,255,255)
      img = np.full((416,416,3), color, dtype=np.uint8)
      y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
      img[y1:y2,x1:x2] = cropped_img[y1:y2,x1:x2]*255

      x_train_all.append(img)
      y_train_all.append(np.where(categories==(filename.split('.')[:-2]))[0][0])

x_test = []
y_test = []
for dirname, _, filenames in os.walk('Dress_Recognition/Test-Dataset'):
  for filename in filenames:
    img_in_frame,v_boxes, v_labels, v_scores = detect_person(os.path.join(dirname, filename))
    for i in range(len(img_in_frame)):
      color = (255,255,255)
      img = np.full((416,416,3), color, dtype=np.uint8)
      y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
      img[y1:y2,x1:x2] = cropped_img[y1:y2,x1:x2]*255

      x_test.append(img)
      y_test.append(np.where(categories==(filename.split('.')[:-2]))[0][0])

x_train_all = np.array(x_train_all)
x_test = np.array(x_test)

y_train_all = np.array(y_train_all)
y_test = np.array(y_test)
```

```
x_train_all: (7000, 416, 416, 3)
y_train_all: (7000,)
x_test: (500, 416, 416, 3)
y_test: (500,)
```

1.3.5 Building and compiling of the model

The architecture we have followed here is 3 convolution layers followed by pooling and batch normalization at each layer, then 4 fully connected layers and a softmax layer respectively. Multiple filters are used at each convolution layer, for different types of feature extraction. One intuitive explanation can be if first filter helps in detecting the straight lines in the image, second filter will help in detecting circles and so on.

```
cfmodel = Sequential()

cfmodel.add(Conv2D(10,kernel_size=3,activation='relu',input_shape=input_shape, padding='same'))
cfmodel.add(BatchNormalization())

cfmodel.add(Conv2D(20,kernel_size=3,activation='relu'))
cfmodel.add(MaxPool2D(2))
cfmodel.add(BatchNormalization())

cfmodel.add(Conv2D(30,kernel_size=3,activation='relu'))
cfmodel.add(MaxPool2D(2))
cfmodel.add(BatchNormalization())

cfmodel.add(Flatten())

cfmodel.add(Dense(30,activation='relu'))
cfmodel.add(Dropout(0.25))
cfmodel.add(BatchNormalization())

cfmodel.add(Dense(50,activation='relu'))
cfmodel.add(Dropout(0.25))
cfmodel.add(BatchNormalization())

cfmodel.add(Dense(80,activation='relu'))
cfmodel.add(Dropout(0.25))
cfmodel.add(BatchNormalization())

cfmodel.add(Dense(100,activation='relu'))
cfmodel.add(Dropout(0.25))
cfmodel.add(BatchNormalization())

cfmodel.add(Dense(5,activation='softmax'))
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

```
=================================================================
conv2d_7 (Conv2D)            (None, 416, 416, 10)      280
_________________________________________________________________
batch_normalization_10 (Batc (None, 416, 416, 10)      40
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 414, 414, 20)      1820
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 207, 207, 20)      0
_________________________________________________________________
batch_normalization_11 (Batc (None, 207, 207, 20)      80
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 205, 205, 30)      5430
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 102, 102, 30)      0
_________________________________________________________________
batch_normalization_12 (Batc (None, 102, 102, 30)      120
_________________________________________________________________
flatten_2 (Flatten)          (None, 312120)            0
_________________________________________________________________
dense_6 (Dense)              (None, 30)                9363630
_________________________________________________________________
dropout_5 (Dropout)          (None, 30)                0
_________________________________________________________________
batch_normalization_13 (Batc (None, 30)                120
_________________________________________________________________
dense_7 (Dense)              (None, 50)                1550
_________________________________________________________________
dropout_6 (Dropout)          (None, 50)                0
_________________________________________________________________
batch_normalization_14 (Batc (None, 50)                200
_________________________________________________________________
dense_8 (Dense)              (None, 80)                4080
_________________________________________________________________
dropout_7 (Dropout)          (None, 80)                0
_________________________________________________________________
batch_normalization_15 (Batc (None, 80)                320
_________________________________________________________________
dense_9 (Dense)              (None, 100)               8100
_________________________________________________________________
dropout_8 (Dropout)          (None, 100)               0
_________________________________________________________________
batch_normalization_16 (Batc (None, 100)               400
_________________________________________________________________
dense_10 (Dense)             (None, 5)                 505
=================================================================
Total params: 9,386,675
Trainable params: 9,386,035
Non-trainable params: 640
_________________________________________________________________
```

1.3.6 Training and evaluating the model

After the model architecture is defined, we need to compile it. During compilation step we are using categorical_crossentropy as this is a classification problem and adam optimizer is used. Then model needs to be trained with training data to be able to classify the categories : 'Formal_Shirt', 'Kurti', 'Saree', 'T-Shirt', 'Unknown'. The model is trained for 50 epochs and at the end of each epoch we save the model, we are using two callback functions checkpoint and early_monitor, to save and load the best weights found during training. The best weights are computed using the validation score computed at the end of each epoch.

```
cfmodel.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
training = cfmodel.fit(x_train_all,target, validation_split=0.3, epochs=50, callbacks=[checkpoint,
    early_monitor])
```

```
Train on 4900 samples, validate on 2100 samples
Epoch 1/50
4900/4900 [==============================] - 58s 12ms/step - loss: 1.4142
- accuracy: 0.4488 - val_loss: 1.5247 - val_accuracy: 0.3843
Epoch 2/50
4900/4900 [==============================] - 57s 12ms/step - loss: 0.9058
- accuracy: 0.6535 - val_loss: 1.0828 - val_accuracy: 0.5314
Epoch 3/50
4900/4900 [==============================] - 57s 12ms/step - loss: 0.6695
- accuracy: 0.7494 - val_loss: 0.4891 - val_accuracy: 0.8500
Epoch 4/50
4900/4900 [==============================] - 57s 12ms/step - loss: 0.5705
- accuracy: 0.7980 - val_loss: 0.4995 - val_accuracy: 0.8357
Epoch 5/50
4900/4900 [==============================] - 57s 12ms/step - loss: 0.4213
- accuracy: 0.8588 - val_loss: 0.3174 - val_accuracy: 0.9010
.......
.......
.......
```

Accuracy Result

After a lot of experiments we have reached the stage where our model has 96.34 % accuracy. If we visualize the whole training log, then with more number of epochs the loss and accuracy of the model on training and testing data converged thus making the model a stable one.

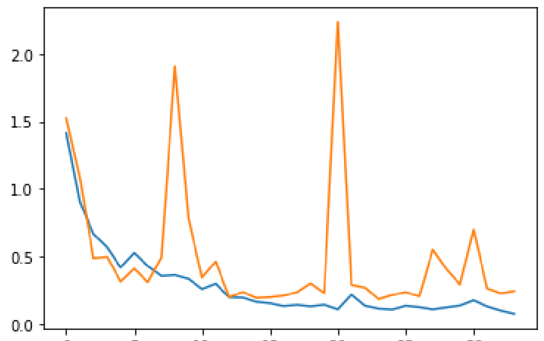

Figure 1.7: train_loss: blue , val_loss: yellow

```
cfmodel.evaluate(x_test,test_target)[1]*100
```

Output :

```
500/500 [==============================] - 2s 3ms/step
96.39999866485596
```

1.3.7 Saving the classification model to disk and reuse

Once we trained the model, we need to use it for various experiments. The training of classification model contains 9.6 million parameters to be tuned. This took around 4 hours of computing using gpu in google colab notebook. So it's better to save the model once computed to disk, and resuse it later.

```
# saving the model to /content/cfmodel96.34.h5
cfmodel.save('/content/cfmodel96.34.h5')

# loading the model from /content/cfmodel96.34.h5
cfmodel = load_model('/content/cfmodel96.34.h5')
```

1.3.8 Predicting the category of each image

Now that we have yolov3 model and classification model, we are are now ready to make predictions using these. Due to the optimizations done in step 2, we can now predict the presence of a person in an image in about 0.1 seconds only. By using the output of yolo model we now have the region of interest where our classification model will do it's job. So for this purpose we have cropped the image according to box generated by the yolo model. Now we add white padding to the cropped image. After this step we have final (416x416x3) sized image. This can be used as an input to our classification model which finally assigns the predicted class to the image.

```
# function which uses the cfmodel model to make prediction on output of yolov3 model.
def make_prediction(cropped_img, box):
  color = (255,255,255)
  padded_image = np.full((416,416,3), color, dtype=np.uint8)
  y1, x1, y2, x2 = box.ymin, box.xmin, box.ymax, box.xmax
  padded_image[y1:y2,x1:x2] = cropped_img[y1:y2,x1:x2]*255
  pad = []
  pad.append(padded_image)
  pad = np.array(pad)
  prediction = cfmodel.predict_classes(pad)
  categories[prediction]
  prediction = cfmodel.predict_classes(pad)
  return categories[prediction]
```

```
# function which uses the yolov3 model to detect persons in an image.
def detect_person(image_loc):
  image, image_w, image_h = load_image_pixels(image_loc, (416,416))
  yhat = model.predict(image)
  boxes = list()
  boxes += decode_netout(yhat[0][0], anchors[i], class_threshold, input_h, input_w)
  correct_yolo_boxes(boxes, 416, 416, input_h, input_w)
  do_nms(boxes, 0.5)
  v_boxes, v_labels, v_scores = get_boxes(boxes, class_threshold)
  return image, v_boxes, v_labels, v_scores
```

Predictions done

Here, I'll be listing some of the predictions done using cfmodel model and yolo model.

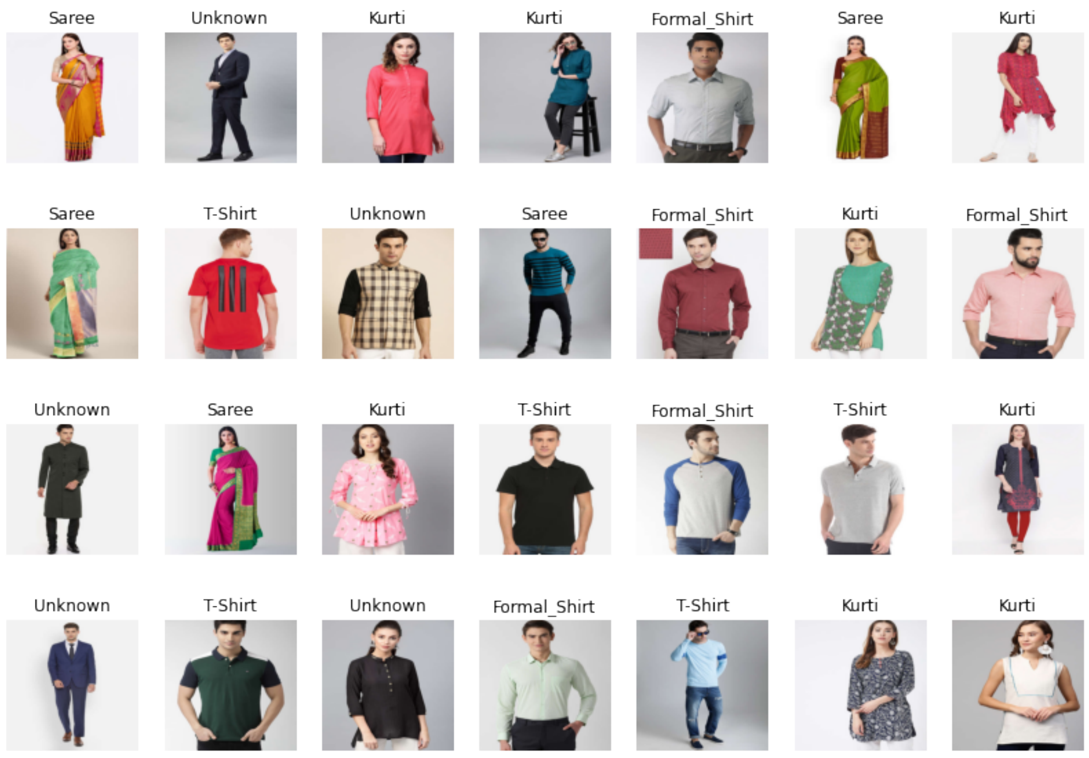

Figure 1.8: predictions

1.4 CONCLUSION

Now we can see that using yolo along with the classification model gives astounding performance, both in terms of accuracy and performance. We have used the speed of yolo to capture the image portions of specific interest and then we used classification cnn model to make required predictons.

```
start_time = time.time()

img_in_frame,v_boxes, v_labels, v_scores = detect_person('Dress_Recognition/Test-Dataset/T-Shirt.81.
    jpg')
for i in range(len(img_in_frame)):
  print(make_prediction(img_in_frame[i], v_boxes[i]))

print("--- %s seconds ---" % (time.time() - start_time))
```

Output :

```
['T-Shirt']
--- 0.12205100059509277 seconds ---
```

We were able to achieve the best accuracy of 96.34% on our dress dataset. The time it takes to process single frame is also improved to 0.12 seconds Now we are successful in making realtime system much which can even process videos at a speed of 10 FPS.