University of Tehran
College of Engineering
School of Electrical and Computer Engineering

Intelligent Systems

Dr.Hosseini

Final Project

Navid Dehban
810198390

Nika Emami
810198356

Soroush Mesforush
810198472

Reza Jahani
810198377

Bahman 01

Contents

Abstract

In this project we shall study machine vision in two sections, first we shall perform image reconstruction, we do this using the **AlexNet** model.

In the second section we shall perform object detection using the **YOLOv3** model.

1 Question 1 : Image reconstruction from feature maps

1.1 Part A

Here we want to design neural networks using pretrained **AlexNet** weights to recognize images taken from different layers of the **AlexNet** network.

First of all we take a look at the **AlexNet** architecture.

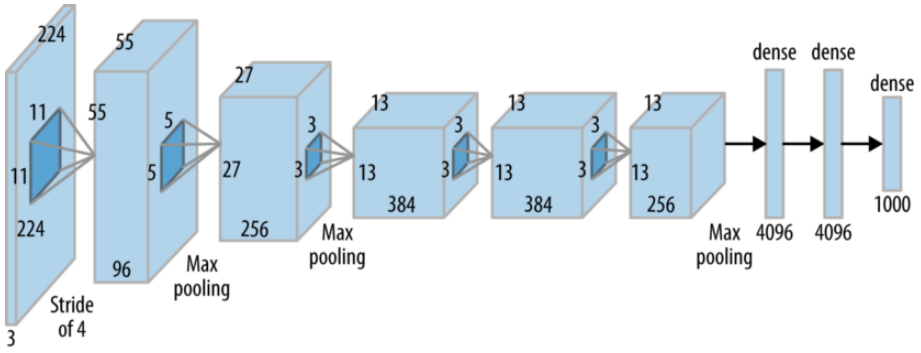


Figure 1: Alexnet architecture

This diagram matches the description of the **AlexNet** paper as follows.

layer	CONV1		CONV2		CONV3	CONV4	CONV5		FC6		FC7		FC8
processing steps	conv1 relu1	mpool1 norm1	conv2 relu2	mpool2 norm2	conv3 relu3	conv4 relu4	conv5 relu5	mpool5	fc6 relu6	drop6	fc7 relu7	drop7	fc8
out size	55	27	27	13	13	13	13	6	1	1	1	1	1
out channels	96	96	256	256	384	384	256	256	4096	4096	4096	4096	1000

Figure 2: Alexnet architecture

Now we go on to get the output of each layer and design the proper neural networks for each one accordingly.

```python
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

def show_batch(dataloader):
    dataiter = iter(dataloader)
    images, labels = dataiter.next()
    imshow(make_grid(images))

def show_image(dataloader):
    dataiter = iter(dataloader)
    images, labels = dataiter.next()
    random_num = randint(0, len(images)-1)
    imshow(images[random_num])
    label = labels[random_num]
    print(f'Label: {label}, Shape: {images[random_num].shape}')

```

Figure 3: Helper functions

```python
def generate_dataloader(data, name, transform):
    if data is None:
        return None
    if transform is None:
        dataset = datasets.ImageFolder(data, transform=T.ToTensor())
    else:
        dataset = datasets.ImageFolder(data, transform=transform)
    # idx = (dataset.targets==1) | (dataset.targets==2) | (dataset.targets==3) | (dataset.targets==4)
    # dataset.targets = dataset.targets[idx]
    # dataset.data = dataset.data[idx]/
    if use_cuda:
        kwargs = {"pin_memory": True, "num_workers": 1}
    else:
        kwargs = {}
    dataloader = DataLoader(dataset, batch_size=batch_size,
    shuffle=(name=="train"),
    **kwargs)
    return dataloader

```

Figure 4: Dataloader

```
preprocess_transform_pretrain = T.Compose([
  T.Resize(256),
  T.CenterCrop(224),
  T.RandomHorizontalFlip(),
  T.ToTensor(),
  T.Normalize(mean=[0.485, 0.456, 0.406],
  std=[0.229, 0.224, 0.225])])

```

Figure 5: Preprocessing

```
class AlexNetConv2(nn.Module):
  def init(self):
    super(AlexNetConv2, self).init()
    self.features = nn.Sequential(*list(model_alexnet.features.children())
  [:-7])
  def forward(self, x):
    x = self.features(x)
  return x

class AlexNetConv5(nn.Module):
  def __init__(self):
    super(AlexNetConv5, self).__init__()
    self.featuresdfd = nn.Sequential(*list(model_alexnet.features.children())
  [:-4])
  def forward(self, x):
    x = self.featuresdfd(x)
  return x

class AlexNetFc8(nn.Module):
  def init(self):
    super(AlexNetFc8, self).init()
    self.features = nn.Sequential(*list(model_alexnet.features.children())
  [:-1])
  def forward(self, x):
    x = self.features(x)
  return x

```

Figure 6: Layer Outputs

```
class MyModel_conv5(nn.Module):
  def __init__(self, model):
    super(MyModel_conv5, self).__init__()
    self.pre_trained = model
    self.seq = nn.Sequential(
    nn.Conv2d(256, 256, 3, 1),
    nn.Conv2d(256, 256, 3, 1),
    nn.Conv2d(256, 256, 3, 1),
    nn.ConvTranspose2d(256, 256, 5, 2),
    nn.ConvTranspose2d(256, 128, 5, 2),
    nn.ConvTranspose2d(128, 64, 5, 2),
    nn.ConvTranspose2d(64, 32, 5, 2),
    nn.ConvTranspose2d(32, 3, 5, 2),
    nn.MaxPool2d((94, 94), stride=(1, 1)),
    )

  def forward(self, x):
    x = self.pre_trained(x)
    x = self.seq(x)
  return x

  pre_trained = AlexNetConv5()
  model = MyModel_conv5(pre_trained)

```

Figure 7: Neural network

```python
def train(dataloader, model, loss_fn, optimizer):
  size = len(dataloader.dataset)
  loss_values = []
  model.train()
  running_loss = 0
  for batch, (X, y) in enumerate(dataloader):
    X, y = X.to(device), y.to(device)
    pred = model(X)
    loss = loss_fn(pred, X)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    loss_values.append(running_loss)
    if batch % 100 == 0:
      loss, current = loss.item(), batch * len(X)
      print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
  plt.plot(np.array(loss_values), 'r')

```

Figure 8: Train

```python
def test(dataloader, model, loss_fn):
  loss_values = []
  size = len(dataloader.dataset)
  num_batches = len(dataloader)
  model.eval()
  test_loss, correct = 0, 0
  with torch.no_grad():
  for X, y in dataloader:
  X, y = X.to(device), y.to(device)
  pred = model(X)
  test_loss += loss_fn(pred, X).item()
  correct += (pred.argmax(1) == X).type(torch.float).sum().item()
  test_loss /= num_batches
  correct /= size
  print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss
  :>8f} \n")

```

Figure 9: Test

Part B&C

In this part we shall run the following code.

```
epochs = 5
for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  train(train_dataloader, model, loss_func, optimizer)
  test(test_dataloader, model, loss_func)
print("Done!")

```

Figure 10: Test

The obtained results are as follows.

```
loss: 1.257729  [    0/100000]
loss: 6.333055  [ 6400/100000]
loss: 1.987759  [12800/100000]
loss: 1.413734  [19200/100000]
loss: 1.384520  [25600/100000]
loss: 1.442773  [32000/100000]
loss: 1.484911  [38400/100000]
loss: 1.264908  [44800/100000]
loss: 1.304636  [51200/100000]
loss: 1.251667  [57600/100000]
loss: 1.091855  [64000/100000]
loss: 1.137109  [70400/100000]
loss: 1.184907  [76800/100000]
loss: 1.285305  [83200/100000]
loss: 1.299947  [89600/100000]
loss: 1.248004  [96000/100000]

```

Figure 11: Test

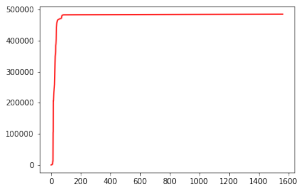


Figure 12: IoU

Part D

We can deduce that neural network layers are responsible for extracting and recognizing a feature of the input data, for example convolutional layers are responsible for finding feature for error minimization, the first edges attempt to find simple and low level features such as the edges, by deepening our network we can extract and recognize more high level networks that are a bit abstract. Other than this, it is trivial to assume that these layers discard redundant features.

2 Question 2 : Object detection and recognition

2.1 History

2.1.1 Part 1

Here we shall give some preliminary explanations about the idea behind each algorithm and model.

Hog Algorithm

The Histogram of oriented gradients algorithm also known as the **HOG** algorithm is a popular feature descriptor widely used in image processing and computer vision.

This algorithm is a simple and powerful feature descriptor employing linear **SVM** machine learning algorithm. By feature descriptors we mean the representation of an image that simply extracts the useful information and disregards the unnecessary information from the image. HOG is not only used for face detection but also it is widely used for object detection like cars, pets, and fruits. **HOG** is very robust for object detection because object shapes are characterized using the local intensity gradient distribution and edge direction.

Now we shall explain the functionality of the algorithm. The algorithm consists of three steps.

- **Step 1**

  Dividing the photo into smaller cells

- **Step 2**

  Calculation of the histogram for each cell. This histogram is formed by the calculated gradients of each cell. For this approach two kernels are employed to calculate the gradient. Then the magnitude and the direction of the gradients are calculated via the following formulas.

$$g = \sqrt{g_x^2 + g_y^2}, \quad \theta = \arctan\left(\frac{g_y}{g_x}\right)$$

  One of the key points in this section is that the magnitude of the gradient increases when sharp changes in intensity occurs, this is handy in figuring out the edges.

Figure 13: The Kernels

- **Step 3**

  After completing step two, we merge all the histograms to form one histogram which is used as a feature vector.

To understand the concept behind **HOG**, we must pay attention to the fact that the role of feature descriptor is used to define an image by its color intensities of pixels. The feature descriptor tries to capture all edges or curves within an image in order to know outline information. Using the first order directive kernels, you can calculate gradient magnitude and then obtain the orientation. Each orientation is assigned to different bins to obtain a histogram. In each bin, gradient magnitudes are stored according to their orientation angles.

Now we shall take a look at the pros and cons of the **HOG** algorithm.

- **Pros**
  - Very useful for representing structural objects with low form variation, this means things that have the same common form such as buildings,people etc...
  - Powerful descriptor
  - Accurate for object classification

  Cons
  - May result in large feature vectors
  - Not a very fast descriptor
  - Accurate for object classification
  - If we have objects with structural variation such as rotation, the algorithm shall be faulty.

Viola-Jones

The **Viola-Jones** algorithm, is a framework to perform facial detection. This algorithm is of a cascaded object form, it uses **Haar**-like features and **Adaboost** learning to detect faces in photos. The algorithm starts with a set of simple rectangular features, which are evaluated on a scale-invariant image pyramid to produce a series of feature responses. These responses are then passed through a cascaded set of simple classifiers to quickly reject non-face regions, while retaining face regions for further analysis. The final output is a set of face regions in the input image.

It is important to note that the algorithm works with grayscale images, given an image, the algorithm looks at many smaller subregions and tries to find a face by looking for specific features in each subregion. It needs to check many different positions and scales because an image can contain many faces of various sizes. Viola and Jones used **Haar**-like features to detect faces in this algorithm.

Now we shall take a look at the steps of this algorithm.

- **Selecting Haar-like features**

- **Creating an integral image**
- **Running AdaBoost training**
- **Creating classifier cascades**

Haar-Like functions

Haar-like features are digital image features utilized in object recognition. We know that faces share some properties that are the same for everybody, of the human face like the eyes region is darker than its neighbor pixels, and the nose region is brighter than the eye region.

A simple way to find out which region is lighter or darker is to sum up the pixel values of both regions and compare them. The sum of pixel values in the darker region will be smaller than the sum of pixels in the lighter region. If one side is lighter than the other, it may be an edge of an eyebrow or sometimes the middle portion may be shinier than the surrounding boxes, which can be interpreted as a nose This can be accomplished using **Haar**-like features and with the help of them, we can interpret the different parts of a face.

There are 3 types of Haar-like features that Viola and Jones used in their research.

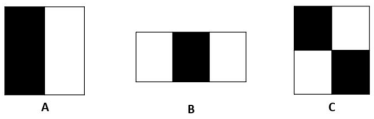
- **Edge features**
- **Four-sided features**
- **Line-features**

Edge features and Line features are useful for detecting edges and lines respectively. The four-sided features are used for finding diagonal features.

The value of the feature is calculated as a single number: the sum of pixel values in the black area minus the sum of pixel values in the white area. The value is zero for a plain surface in which all the pixels have the same value, and thus, provide no useful information.

Since our faces are of complex shapes with darker and brighter spots, a **Haar**-like feature gives you a large number when the areas in the black and white rectangles are very different. Using this value, we get a piece of valid information out of the image.

To be useful, a **Haar**-like feature needs to give you a large number, meaning that the areas in the black and white rectangles are very different.



Integral Images

Calculations to perform different mathematical computations on the pixels of photos are time consuming and have a great cost, so we must use integral images also known as summed-area tables, this allows us to perform calculations quickly so we can understand whether a feature fits the criteria or not.

AdaBoost

There are many features used in face identification, but all of them aren't optimal and very useful in performing this task, so we use the **AdaBoost** algorithm which is a machine learning algorithm to extract the best features in facial recognition.

AdaBoost checks the performance of all classifiers with the features we support to it, after this it chooses the best feature (type and size of the feature) based on the performance.

Now we shall give some complementary explanation about how we can calculate and evaluate the performance of a classifier. We must evaluate each classifier on the subregions used for training, some shall produce strong responses and some shall not, those that are classified as positive means that it has successfully been classified as a human face, so all in all classifiers that performed well are better and should be given a higher weight, we class this classifier a **boosted classifer**.

All in all, he algorithm is setting a minimum threshold to determine whether something can be classified as a useful feature or not.

2.1.2 Part 2

Here we have implemented the **Violo-Jones** algorithm on a photo of David Beckham. The codes and output are as follows.

```
def detect_faces(image_path, cascade_Path):

  faceCascade = cv2.CascadeClassifier(cascade_Path)

  frame = cv2.imread(image_path)
  frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  faces = faceCascade.detectMultiScale(gray, flags=cv2.CASCADE_SCALE_IMAGE)

  return faces, frame

```

Figure 14: Face detection function

```
def show_detected_faces(faces, frame):
  for (x, y, w, h) in faces:
    cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 3)
  plt.imshow(frame)
  plt.show()

```

Figure 15: Face showing function

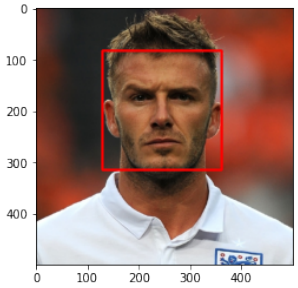


Figure 16: Face detection output

2.2 Implementing YOLO

2.2.1 Familiarity with the model

The **YOLOv3** model has been trained on the **COCO** dataset which has more than 30000 photos and 80 different classes.

In this project we have used the **YOLOv3-320** model, this model has 320 pixels in width and height.

2.2.2 Understanding the theory of the problem

Part 2.1

Here we shall firstly explain what a grid cell is in general.

Grid Cells

Grid cells have a biologic interpretation which is interesting, it is implied that grid cells are neurons which become active at multiple locations in an environment.

The idea behind **YOLO** is to output a number for the dimension which is large enough for all the objects to fit in it. so the original picture is cropped and divided into an $n \times n$ grid, each cell in this grid is interpreted as a grid cell.

As implied in the original **YOLO** paper, each grid cell predicts two bounding boxes and one class, this is a limitation of the **YOLO** model.

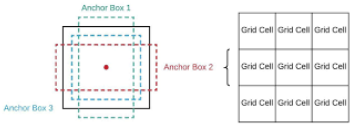


Figure 17: Grid Cell

Part 2.2

Here we shall explain the concept of a bounding box, obviously bounding boxes are used in object detection and are generally boxes in the shape of rectangles that surround an object which is to be detected, essentially it is the border's of the

coordinates that enclose the object, these boxes are used to identify the position and type of the object, a bounding box in an image is shown below.

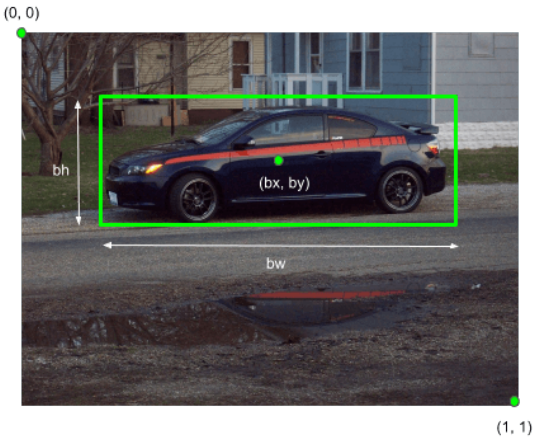


Figure 18: Bounding box

To procure a bounding box we need the coordinates of its edges and the coordinates of the middle is also appreciated. It is also important to imply that the class of the object inside the bounding box is important. The parameters needed to define a bounding box are listed below.

- **Class**
- **Coordinates of edges**
- **Coordinates of center**
- **Height and Width**
- **Confidence**

  This represents the possibility of the object being in the box.

Part 2.3

The tensor output of the YOLO model contains elements described as below: $(Pc, x, y, h, w, c_1, c_2, c_3, \ldots, c80)$

- **Pc**:Probability of existence of an object within the cell
- **x**:x-axis center of the bounding box
- **y**:y-axis center of the bounding box
- **h**:height of the bounding box
- **w**:width of the bounding box
- **c__i**:1 if class i is in the box, 0 if class i is not the box

$$Outputsize = 5 + Num(classes)$$

Part 2.4

Here we shall explain how the confidence parameter is obtained. First of all it is important to know what the confidence score indicates, this score shows the certainty that the box contains an object and how accurate the box performs its prediction.

We know the the YOLO divides the image into an $n \times n$ grid, the confidence score shall be calculated using the following formula.

$$Confidence = Pr\left[Obj\right] \times IoU$$

Where $Pr\left[Obj\right]$ is the probability of the object existing in the bounding box and **IoU** is the intersection over union between the predicted box and the real thing.

IoU(intersection over union)

This metric is used to evaluate the performance of object detection, this is done by comparing the ground truth bounding box to the predicted bounding box. A graphical description is shown below.

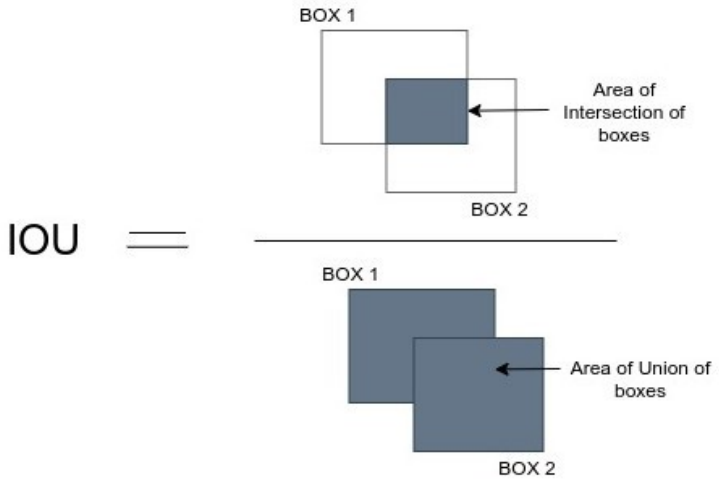


Figure 19: IoU

$$\textbf{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Part 2.5

As pointed out in the **YOLO** paper, the sum-squared error due to ease in optimization, but this doesn't work out as we want it to, we aim to maximize the average precision, many grid cells may not contain objects, so the confidence decreases considerably, hence the model might become unstable. This is remedied by using an appropriate loss function.

The **YOLO** loss function is broken into three parts as follows.

- **Bounding box coordinates**
- **Bounding box score prediction**
- **Class score prediction**

YOLO predicts multiple bounding boxes per grid cell. At training time we only want one bounding box predictor to be responsible for each object. We assign

one predictor to be "responsible" for predicting an object based on which prediction has the highest current IOU with the ground truth. This leads to specialization between the bounding box predictors. Each predictor gets better at predicting certain sizes, aspect ratios, or classes of object, improving overall recall.

The loss function is as follows.

$$\begin{aligned}
&\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{\omega_i} - \sqrt{\hat{\omega}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}^2 \right) \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2, \quad \lambda_{coord} = 5, \lambda_{noobj} = 0.5
\end{aligned}$$

In the following notation, $\mathbb{1}_{i}^{obj}$means that if the object appears in cell i and $\mathbb{1}_{ij}^{obj}$ means that the jth bounding box predictor in cell ii is responsible for it.

To show which part of the loss function belongs to which we have :

$$\begin{aligned}
&\text{Bbox coords : } \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[(x - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left[\left(\sqrt{\omega_i} - \sqrt{\hat{\omega}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}^2 \right) \right] \\
&\text{Bbox score : } \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\
&\text{Class score : } \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2
\end{aligned}$$

The parameters effective on the loss function are shown above.

Part 2.6

Here we shall explain the concept of **Non Max Suppression**.

To make it simple **Non Max Suppression** (NMS) is a technique used in computer vision, it is a series of algorithms to select a bounding box between overlapping boxes, this is somewhat a form of whittling down the boxes to a few which are needed for object recognition.

This algorithm considers two things as shown below.

- **IoU**
- **Confidence**

Considering these two parameters the best boxes are chosen.

Part 2.7

To solve this, we use multiple bounding boxes. First, we predefine different shapes of bounding boxes. So we can have multiple predictions with multiple bounding boxes. Knowing that output of each cell would obey the format of $(P_c, x, y, h, w, c_1, c_2, c_3)$, when more than 1 object exists in a grid cell, the output would be the concatenated version of this type of output. For example, in the case of two bounding boxes and three classes the output will be vector of size $(16, 1)$and the vector contains, $(Pc_1, x_1, y_1, h_1, w_1, c_1, c_2, c_3, Pc_2, x_2, y_2, h_2, w_2, c_1, c_2, c_3)$.

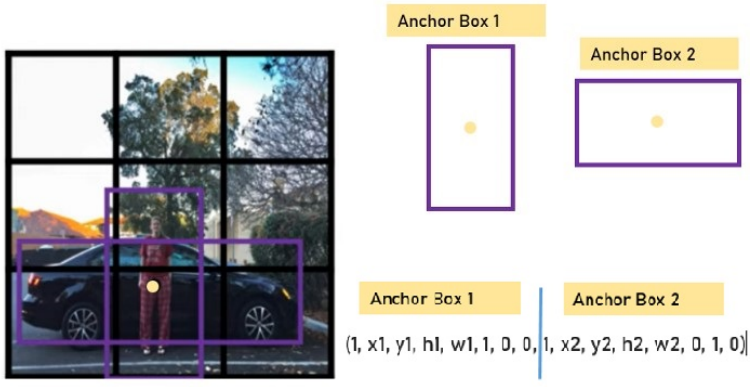


Figure 20: IoU

2.2.3 Implementation steps

Here we want to detect three objects, **Car**,**Bus** and **Person** from the wanted video.

Part 3.a & 3.b

As explained in part 3.a, to be able to perform object detection in a video, we must split the video into consecutive frames and then perform the desired task on each frame, we do this with the help of **open cv**. We know that in each frame we must extract the predicted outputs, so we design a **find_objects** function as follows.

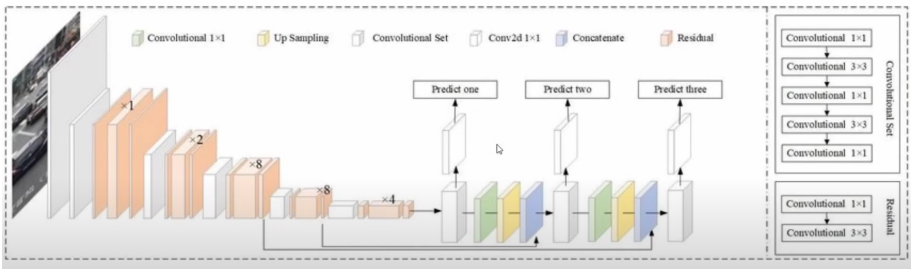


Figure 21: YOLO architecture

As observed in **Figure 16** final layers consist of three output layers. Each layer's output returns a tensor including different detections. Each of these detections is a tensor of size 1×85, below we present a short description of this tensor:

$$d = (P_c, x, , y, h, w, c_1, c_2, \ldots, c_{80})$$

- **Pc**:Probability of existence of an object within the cell
- **x**:x-axis center of the bounding box
- **y**:y-axis center of the bounding box
- **h**:height of the bounding box

- **w**:width of the bounding box
- **c__i**:1 if class i is in the box, 0 if class i is not the box

```
def find_objects(conf_Thresh, outputs, img):
height, width, channel = img.shape
bounding_box = []
label_Id = []
bb_conf = []

for output in outputs:
  for detection in output:
    scores = detection[5:]
    label = np.argmax(scores)
    confidence = scores[label]
    if confidence > conf_Thresh:
      w,h = int(detection[2]* width), int(detection[3]*height)
      x,y = int((detection[0]*width)-w/2), int((detection[1]*height)-h/2)
      bounding_box.append([x,y,w,h])
      label_Id.append(label)
      bb_conf.append(float(confidence))
return bounding_box, label_Id, bb_conf, height, width

```

Figure 22: find_objects

Part 3.c

Here we design a function to show the detected objects and their labels on the input video, we shall also create a new video featuring labels on cars, people and buses.

```
def show_detected_object(labels, desired_labels, bounding_box, label_Id,
  bb_conf, conf_Thresh, NMS_Thresh):

indices = cv2.dnn.NMSBoxes(bounding_box, bb_conf, conf_Thresh, NMS_Thresh)
for i in indices:
  box = bounding_box[i]
  x,y,w,h = box[0], box[1], box[2], box[3]
if labels[label_Id[i]] in desired_labels:
  cv2.rectangle(img, (x,y),(x+w,y+h),(255,0,255),2)
  cv2.putText(img,f'{labels[label_Id[i]].upper()} {int(bb_conf[i]*100)}%',
    (x,y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.6,(255,0,255),2)
return img

```

Figure 23: show_detected_object

```
def display_video(video):
fig = plt.figure(figsize=(10,10))

mov = []
for i in range(len(video)):
  img =plt.imshow(video[i],animated=True)
  plt.axis('off')
  mov.append([img])

anime = animation.ArtistAnimation(fig, mov, interval=50, repeat_delay=1000)

plt.close()
return anime

```

Figure 24: display_video

```
cap = cv2.VideoCapture(video_dir)
yolo_size = 320
conf_Thresh = 0.5
NMS_Thresh = 0.3
video = []

width = int(cap.get(3))
height = int(cap.get(4))
fps = cap.get(cv2.CAP_PROP_FPS)
out = cv2.VideoWriter('output.mp4', cv2.VideoWriter_fourcc('M','J','P','G'), fps, (width, height))

while True:
  success, img = cap.read()
  if success == False:
    break
  blob = cv2.dnn.blobFromImage(img, 1/255,(yolo_size,yolo_size),[0,0,0],crop=False)
  model.setInput(blob)
  layerNames = model.getLayerNames()
  outputNames = [layerNames[i-1] for i in model.getUnconnectedOutLayers()]
  outputs = model.forward(outputNames)
  bounding_box, label_Id, bb_conf, height, width = find_objects(conf_Thresh, outputs, img)
  image = show_detected_object(labels, desired_labels, bounding_box, label_Id, bb_conf, conf_Thresh,
    NMS_Thresh)
  out.write(image)
  video.append(image)

cap.release()
out.release()
cv2.destroyAllWindows()

```

Figure 25: Video Exporter

References

[1] Reshad Hosseini, *Intelligent Systems Lecture Notes, Fall 01*

[2] Hands-On Convolutional Neural Networks with TensorFlow

[3] Intersection over Union (IoU) for object detection

[4] A Comprehensive Guide To Object Detection Using YOLO Framework

[5] Learn about bounding boxes in image processing

[6] A Gentle Introduction Into The Histogram Of Oriented Gradients

[7] The Viola-Jones Face Detection Algorithm

[8] Get middle layer in pytorch