电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

实验报告

EXPERIMENT REPORT

STUDENT NAME:	JAHID SHAHIDUL ISLAM
STUDENT ID:	202324090107
COURSE NAME:	PYTHON PRACTICAL PROGRAMMING
TEACHER NAME:	PROF. RAO YUNBO
EXPERIMENT NO:	FOUR
DATE:	27th MAY 2024

1. Experiment title：Install Python Platform

2. Experiment hours：4h Experiment location: Software Building 400

3. Objectives

   At the end of this experiment, you will be able to:

   - How to install Jupyter software in your devices.
   - How to use Jupyter for Yolov1.

4. Experimental contents & step

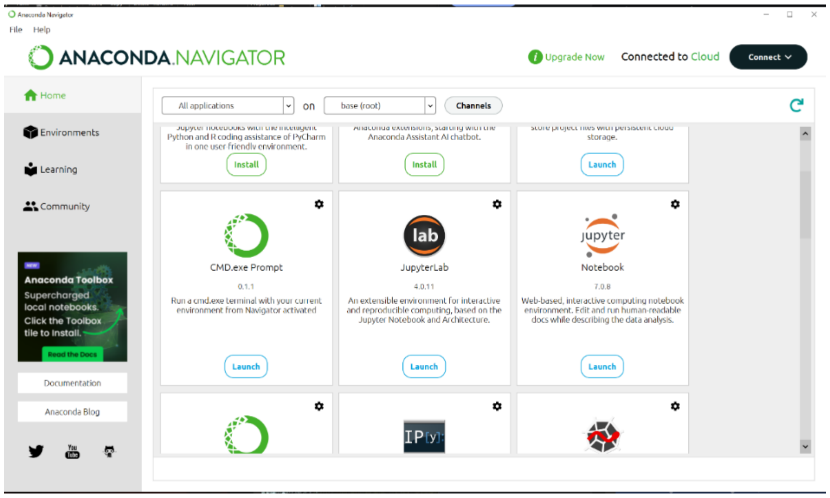
   1) Installing the Jupyter software for Windows
   2) using Jupyter for Yolov1.
   3) using Jupyter for Faster R-CNN
   4) What is CNN?
   5) What is Yolo Network?
   6) What is Faster-RCNN Network?

5. Experimental analysis

1. Installing the Jupyter software for Windows

Step 1: Launch Anaconda Navigator

Open the Anaconda Navigator application, which comes bundled with Anaconda.



Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user friendly environment.

Anaconda extensions, starting with the Anaconda Assistant AI chatbot.

store project files with persistent cloud storage.

Run a cmd.exe terminal with your current environment from Navigator activated

An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

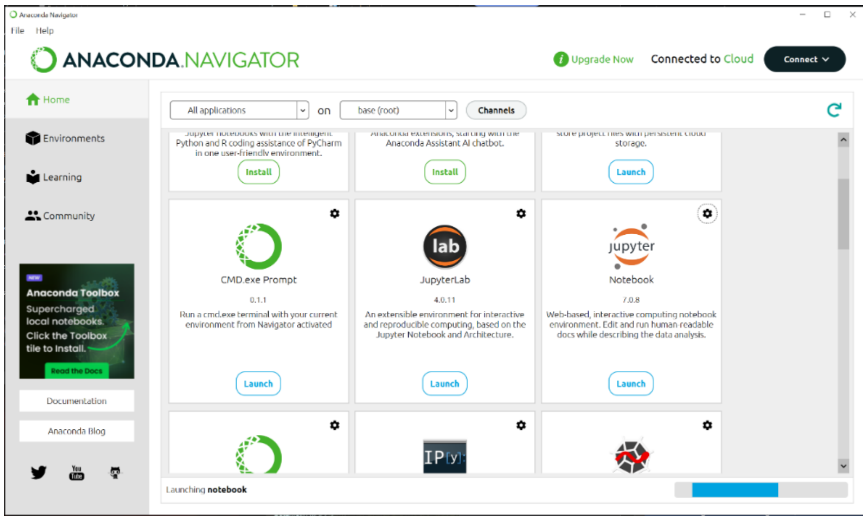
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Step 2: Install Jupyter Notebook

Within the Anaconda Navigator, locate the "Jupyter Notebook" application under the "Applications on" tab. Click the "Install" button to install Jupyter Notebook.

Step 3: Launch Jupyter Notebook

Once the installation is complete, you can launch Jupyter Notebook directly from Anaconda Navigator by clicking the "Launch" button.



Jupyter notebooks with the intelligent Python and R coding assistance of PyCharm in one user-friendly environment.

Anaconda extensions, starting with the Anaconda Assistant AI chatbot.

store project files with persistent cloud storage.

Run a cmd.exe terminal with your current environment from Navigator activated

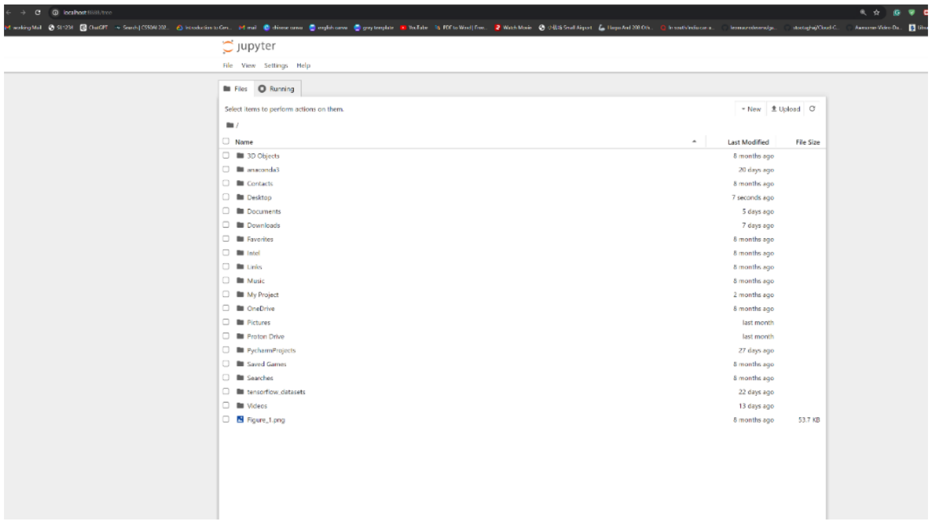
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

Launching **notebook**

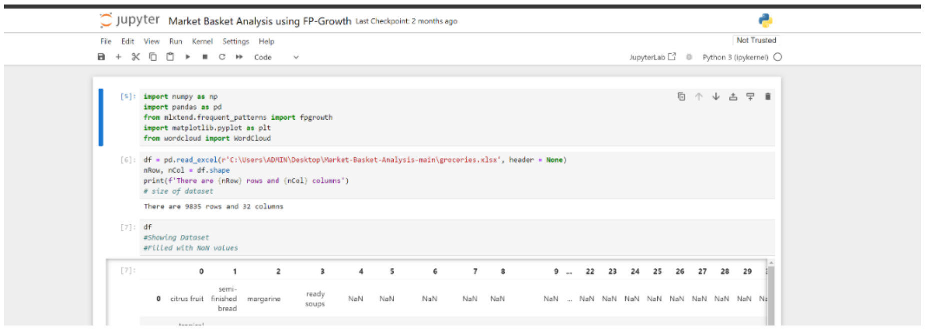
Step 4: Open a Notebook

After launching Jupyter Notebook, you'll see a web interface. Navigate to the directory where you want to create or open your notebook file, click on the desired notebook file to open it, or use the "New" button to create a new notebook.



Select items to perform actions on them.

Name	Last Modified	File Size
3D Objects	8 months ago	
anaconda3	20 days ago	
Contacts	8 months ago	
Desktop	7 seconds ago	
Documents	5 days ago	
Downloads	7 days ago	
Favorites	8 months ago	
Intel	8 months ago	
Links	8 months ago	
Music	8 months ago	
My Project	2 months ago	
OneDrive	8 months ago	
Pictures	last month	
Proton Drive	last month	
PycharmProjects	27 days ago	
Saved Games	8 months ago	
Searches	8 months ago	
tensorflow_datasets	22 days ago	
Videos	13 days ago	
Figure_1.png	8 months ago	53.7 KB

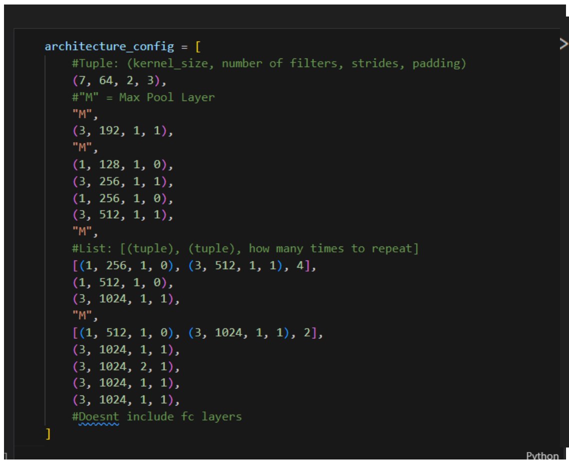

jupyter Market Basket Analysis using FP-Growth Last Checkpoint: 2 months ago
File Edit View Run Kernel Settings Help
Not Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
[5]: import numpy as np
     import pandas as pd
     from mlxtend.frequent_patterns import fpgrowth
     import matplotlib.pyplot as plt
     from wordcloud import WordCloud

[6]: df = pd.read_excel(r'C:\Users\ADMIN\Desktop\Market-Basket-Analysis-main\groceries.xlsx', header = None)
     nRow, nCol = df.shape
     print(f'There are {nRow} rows and {nCol} columns')
     # size of dataset

     There are 9835 rows and 32 columns

[7]: df
     #Showing Dataset
     #Filled with NaN values

[7]:      0            1                      2          3            4    5    6    7    8    9   ...  22   23   24   25   26   27   28   29
     0    citrus fruit semi-finished bread   margarine  ready soups  NaN  NaN  NaN  NaN  NaN  NaN ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
```

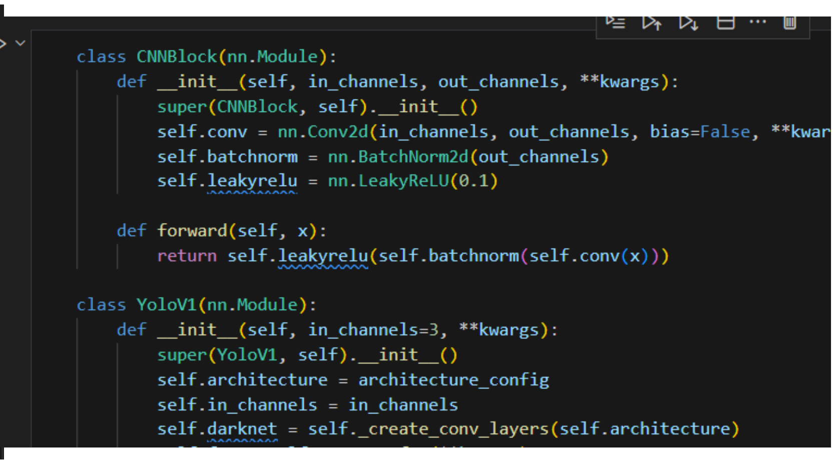
2. using Jupyter for Yolov1.

Step 1: Model Definition

- **Model Architecture:** We use PyTorch to define the YOLOv1 model architecture using a tuple structure to store kernel size, strides, padding, and layer information.
- **YOLOv1 Class:** We create a class named YOLOv1 to encapsulate the model structure and forward pass logic.



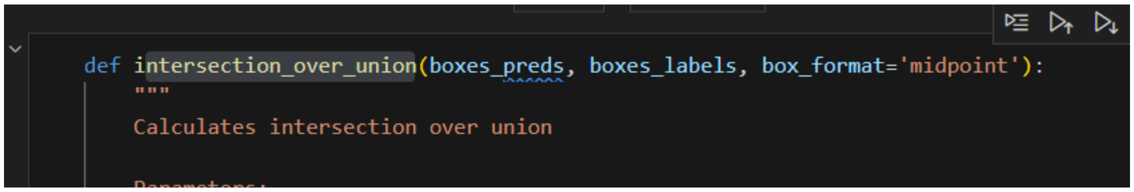
```python
architecture_config = [
    #Tuple: (kernel_size, number of filters, strides, padding)
    (7, 64, 2, 3),
    #"M" = Max Pool Layer
    "M",
    (3, 192, 1, 1),
    "M",
    (1, 128, 1, 0),
    (3, 256, 1, 1),
    (1, 256, 1, 0),
    (3, 512, 1, 1),
    "M",
    #List: [(tuple), (tuple), how many times to repeat]
    [(1, 256, 1, 0), (3, 512, 1, 1), 4],
    (1, 512, 1, 0),
    (3, 1024, 1, 1),
    "M",
    [(1, 512, 1, 0), (3, 1024, 1, 1), 2],
    (3, 1024, 1, 1),
    (3, 1024, 2, 1),
    (3, 1024, 1, 1),
    (3, 1024, 1, 1),
    #Doesnt include fc layers
]
```



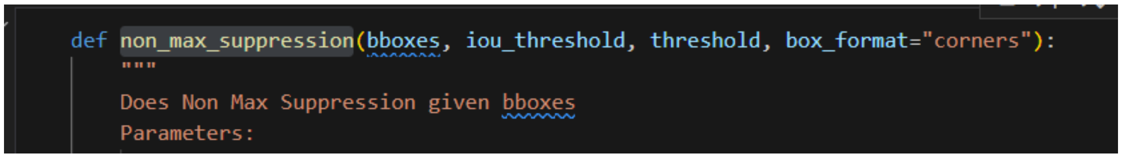
```python
class CNNBlock(nn.Module):
    def __init__(self, in_channels, out_channels, **kwargs):
        super(CNNBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, bias=False, **kwar
        self.batchnorm = nn.BatchNorm2d(out_channels)
        self.leakyrelu = nn.LeakyReLU(0.1)

    def forward(self, x):
        return self.leakyrelu(self.batchnorm(self.conv(x)))

class YoloV1(nn.Module):
    def __init__(self, in_channels=3, **kwargs):
        super(YoloV1, self).__init__()
        self.architecture = architecture_config
        self.in_channels = in_channels
        self.darknet = self._create_conv_layers(self.architecture)
```

Step 2: Utility Functions

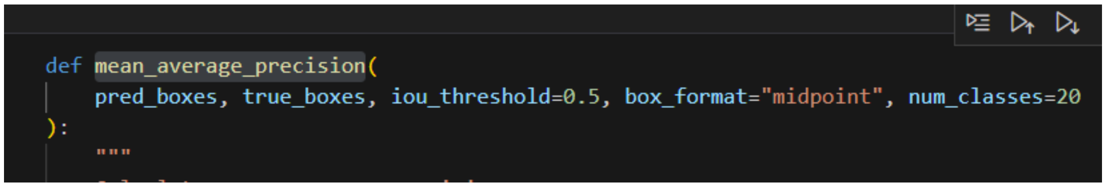
- **intersection_over_union:** Calculates the Intersection over Union (IoU) metric, a common measure of object detection accuracy.
- **non_max_suppression:** Implements non-maximum suppression (NMS), a technique to filter out redundant detections of the same object.
- **mean_average_precision:** Calculates the mean average precision (mAP), a key metric for evaluating object detection models.



```python
def intersection_over_union(boxes_preds, boxes_labels, box_format='midpoint'):
    """
    Calculates intersection over union
```

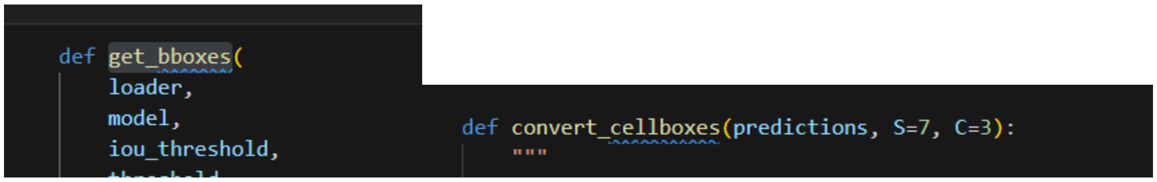


```python
def non_max_suppression(bboxes, iou_threshold, threshold, box_format="corners"):
    """
    Does Non Max Suppression given bboxes
    Parameters:
```



```python
def mean_average_precision(
    pred_boxes, true_boxes, iou_threshold=0.5, box_format="midpoint", num_classes=20
):
    """
```

- **get_bboxes:** Extracts bounding boxes and confidence scores from the model output.
- **convert_cellboxes:** Transforms the model's output into the desired bounding box format.

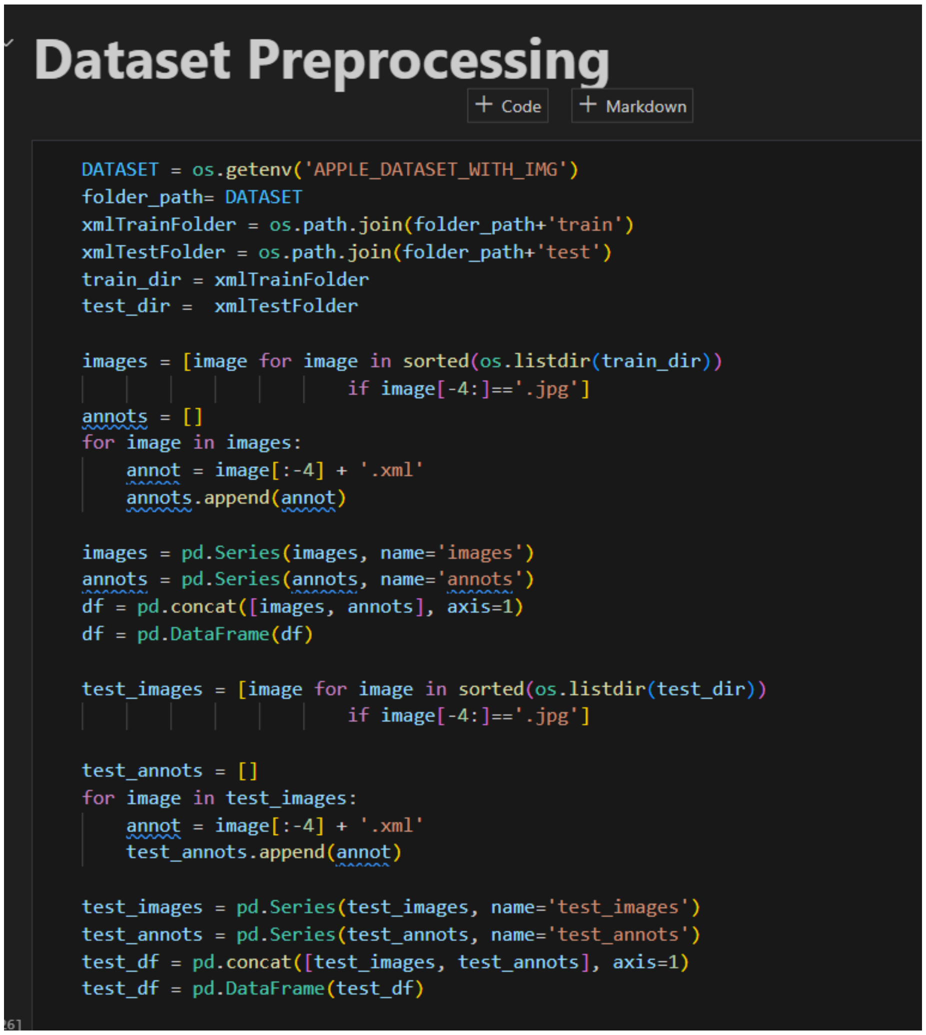


```
def get_bboxes(
    loader,
    model,
    iou_threshold,
```

```
def convert_cellboxes(predictions, S=7, C=3):
    """
```

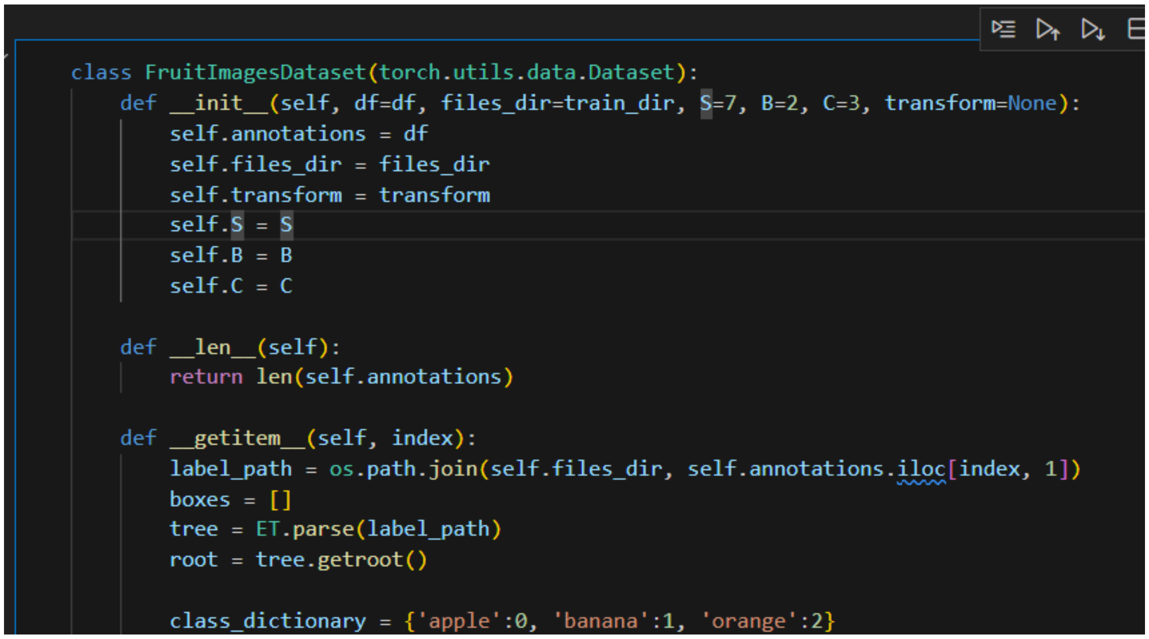
Step 3: Data Preprocessing

- **Dataset Loading:** We load the chosen dataset, preprocess the images (e.g., resizing, normalization), and organize it into training and validation sets.



Dataset Preprocessing

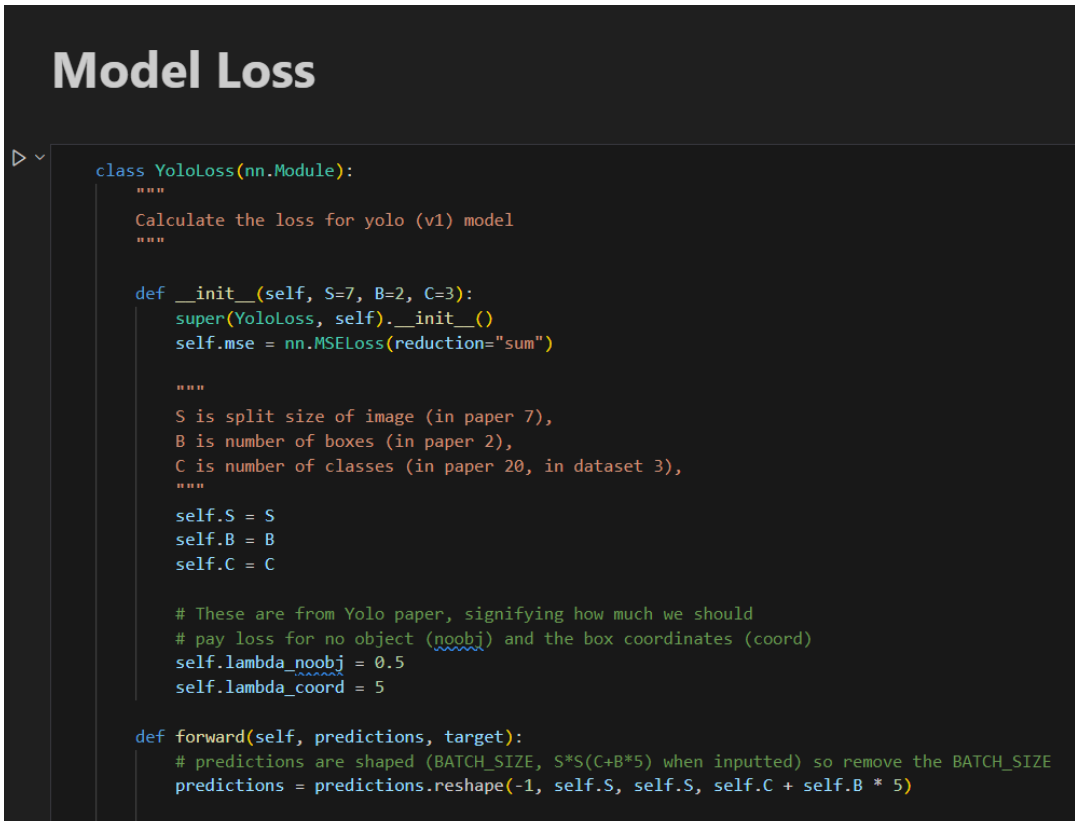
```
DATASET = os.getenv('APPLE_DATASET_WITH_IMG')
folder_path= DATASET
xmlTrainFolder = os.path.join(folder_path+'train')
xmlTestFolder = os.path.join(folder_path+'test')
train_dir = xmlTrainFolder
test_dir =  xmlTestFolder

images = [image for image in sorted(os.listdir(train_dir))
                        if image[-4:]=='.jpg']
annots = []
for image in images:
    annot = image[:-4] + '.xml'
    annots.append(annot)

images = pd.Series(images, name='images')
annots = pd.Series(annots, name='annots')
df = pd.concat([images, annots], axis=1)
df = pd.DataFrame(df)

test_images = [image for image in sorted(os.listdir(test_dir))
                        if image[-4:]=='.jpg']

test_annots = []
for image in test_images:
    annot = image[:-4] + '.xml'
    test_annots.append(annot)

test_images = pd.Series(test_images, name='test_images')
test_annots = pd.Series(test_annots, name='test_annots')
test_df = pd.concat([test_images, test_annots], axis=1)
test_df = pd.DataFrame(test_df)
```



```
class FruitImagesDataset(torch.utils.data.Dataset):
    def __init__(self, df=df, files_dir=train_dir, S=7, B=2, C=3, transform=None):
        self.annotations = df
        self.files_dir = files_dir
        self.transform = transform
        self.S = S
        self.B = B
        self.C = C

    def __len__(self):
        return len(self.annotations)

    def __getitem__(self, index):
        label_path = os.path.join(self.files_dir, self.annotations.iloc[index, 1])
        boxes = []
        tree = ET.parse(label_path)
        root = tree.getroot()

        class_dictionary = {'apple':0, 'banana':1, 'orange':2}
```

Step 4: Model Training

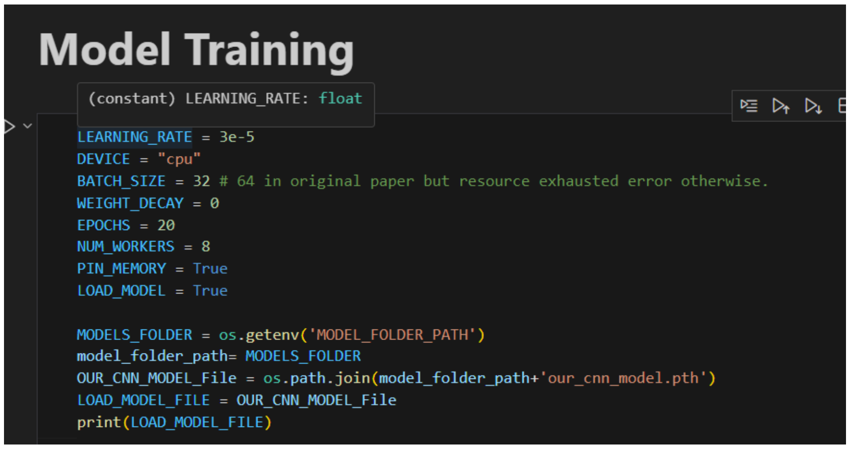
➢ **YoloLoss Class:** We define a custom loss function class (e.g., YoloLoss) specific to the YOLOv1 architecture.



Model Loss

```
class YoloLoss(nn.Module):
    """
    Calculate the loss for yolo (v1) model
    """

    def __init__(self, S=7, B=2, C=3):
        super(YoloLoss, self).__init__()
        self.mse = nn.MSELoss(reduction="sum")

        """
        S is split size of image (in paper 7),
        B is number of boxes (in paper 2),
        C is number of classes (in paper 20, in dataset 3),
        """
        self.S = S
        self.B = B
        self.C = C

        # These are from Yolo paper, signifying how much we should
        # pay loss for no object (noobj) and the box coordinates (coord)
        self.lambda_noobj = 0.5
        self.lambda_coord = 5

    def forward(self, predictions, target):
        # predictions are shaped (BATCH_SIZE, S*S(C+B*5) when inputted) so remove the BATCH_SIZE
        predictions = predictions.reshape(-1, self.S, self.S, self.C + self.B * 5)
```

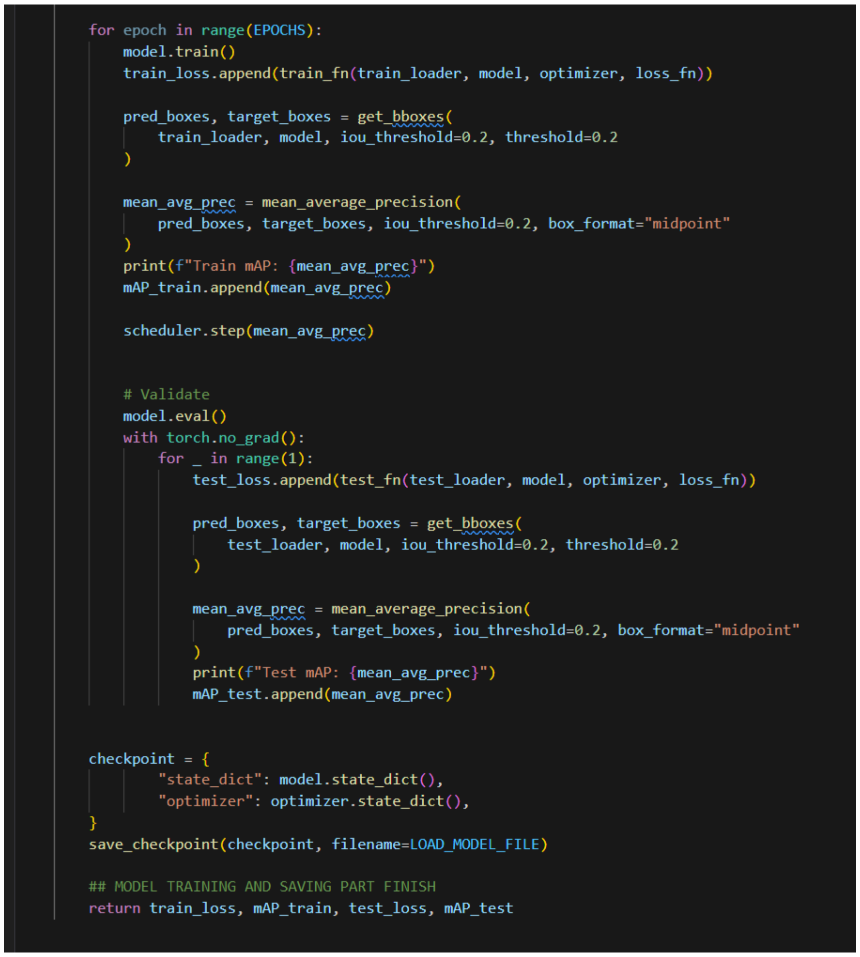
➢ **Training Parameters:** We set training hyperparameters like learning rate, batch size, number of epochs, and optimizer.



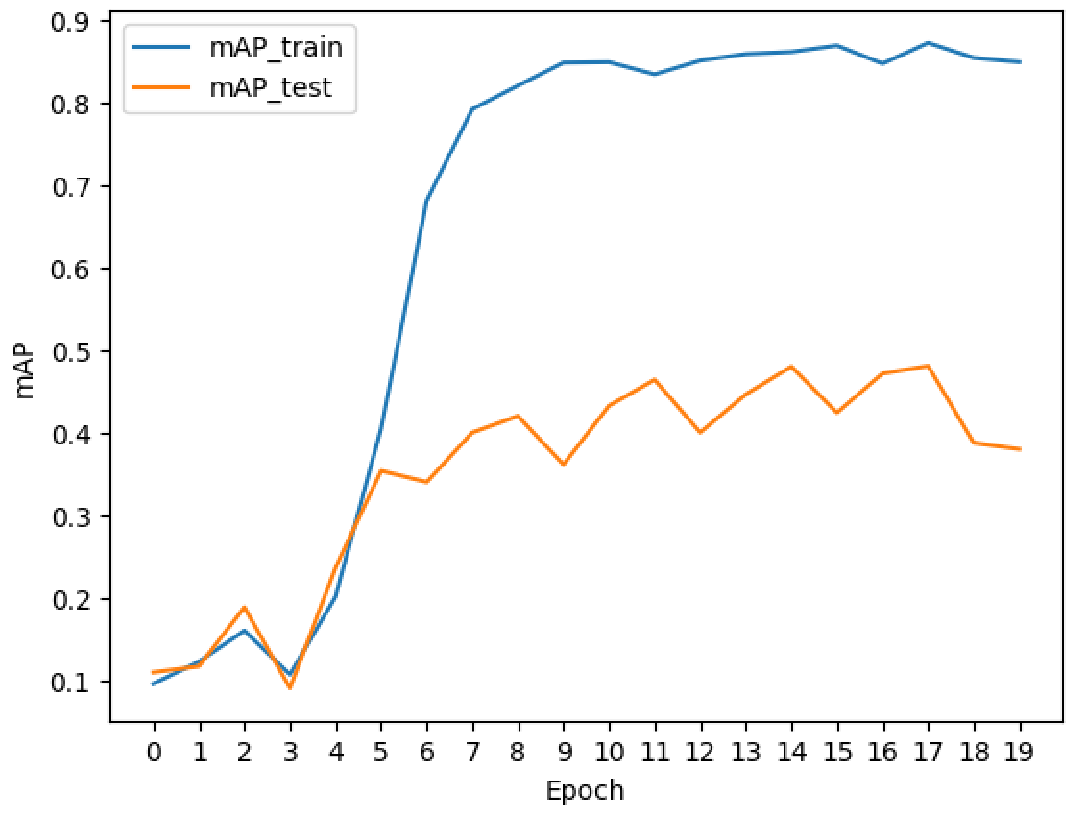
Model Training

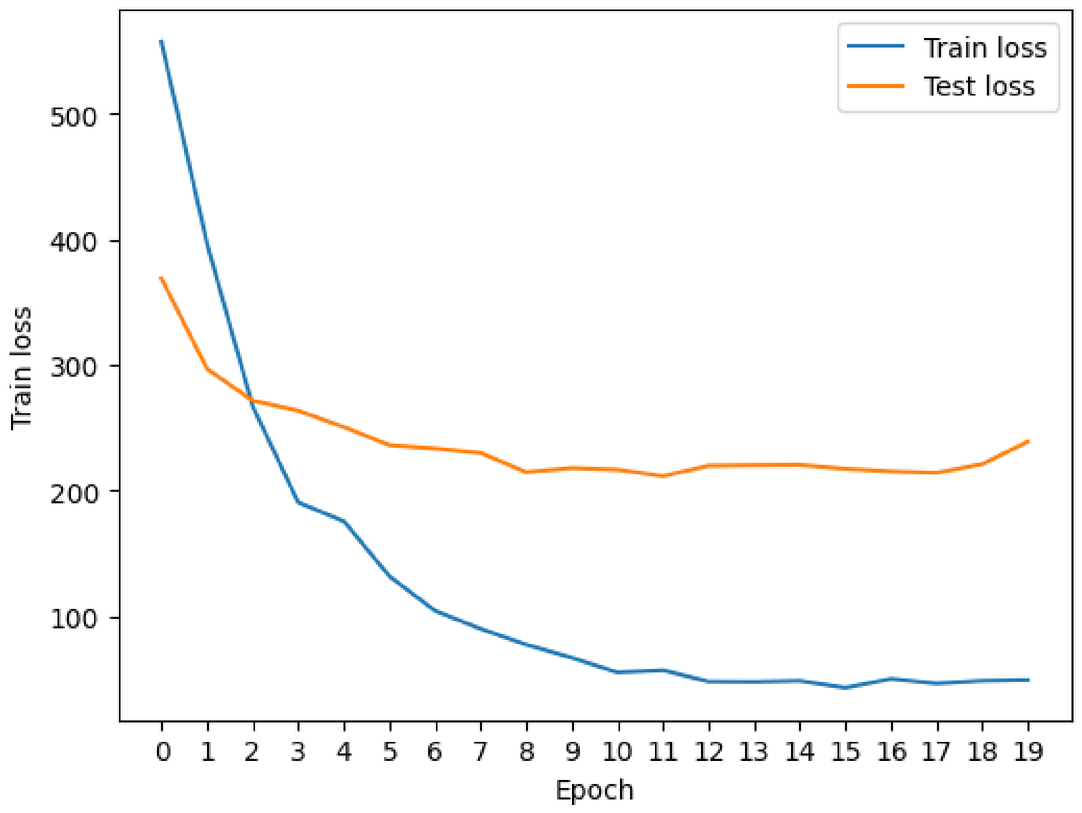
```
(constant) LEARNING_RATE: float

LEARNING_RATE = 3e-5
DEVICE = "cpu"
BATCH_SIZE = 32 # 64 in original paper but resource exhausted error otherwise.
WEIGHT_DECAY = 0
EPOCHS = 20
NUM_WORKERS = 8
PIN_MEMORY = True
LOAD_MODEL = True

MODELS_FOLDER = os.getenv('MODEL_FOLDER_PATH')
model_folder_path= MODELS_FOLDER
OUR_CNN_MODEL_File = os.path.join(model_folder_path+'our_cnn_model.pth')
LOAD_MODEL_FILE = OUR_CNN_MODEL_File
print(LOAD_MODEL_FILE)
```

➢ **Training Loop:** We implement a training loop using Jupyter to iterate through the training data, calculate loss, perform backpropagation, and update model weights.



```
    for epoch in range(EPOCHS):
        model.train()
        train_loss.append(train_fn(train_loader, model, optimizer, loss_fn))

        pred_boxes, target_boxes = get_bboxes(
            train_loader, model, iou_threshold=0.2, threshold=0.2
        )

        mean_avg_prec = mean_average_precision(
            pred_boxes, target_boxes, iou_threshold=0.2, box_format="midpoint"
        )
        print(f"Train mAP: {mean_avg_prec}")
        mAP_train.append(mean_avg_prec)

        scheduler.step(mean_avg_prec)


        # Validate
        model.eval()
        with torch.no_grad():
            for _ in range(1):
                test_loss.append(test_fn(test_loader, model, optimizer, loss_fn))

                pred_boxes, target_boxes = get_bboxes(
                    test_loader, model, iou_threshold=0.2, threshold=0.2
                )

                mean_avg_prec = mean_average_precision(
                    pred_boxes, target_boxes, iou_threshold=0.2, box_format="midpoint"
                )
                print(f"Test mAP: {mean_avg_prec}")
                mAP_test.append(mean_avg_prec)


    checkpoint = {
            "state_dict": model.state_dict(),
            "optimizer": optimizer.state_dict(),
    }
    save_checkpoint(checkpoint, filename=LOAD_MODEL_FILE)

    ## MODEL TRAINING AND SAVING PART FINISH
    return train_loss, mAP_train, test_loss, mAP_test
```


mAP_train
mAP_test
mAP
Epoch
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


Train loss
Test loss
Train loss
Epoch
100
200
300
400
500
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

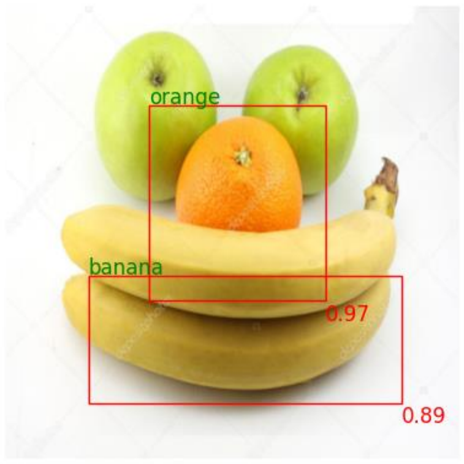
Step 5: Model Testing

- **Sample Image:** We select a sample image from the test set to demonstrate model performance.

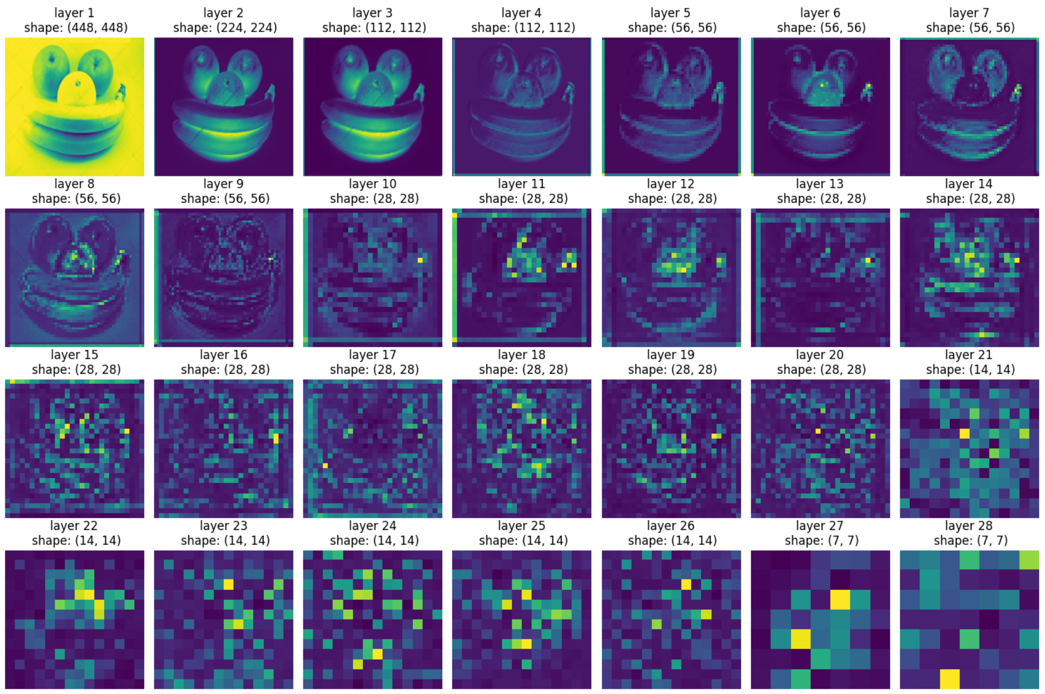
Visualization: We use Jupyter to visualize the following:

- **The original input image.**

- **The model's predictions (bounding boxes and class labels) overlaid on the image.**



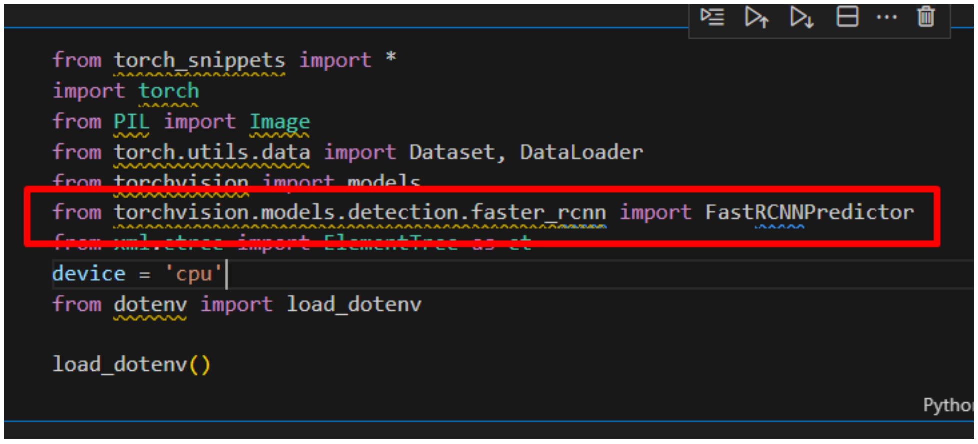
➢ **The intermediate convolution steps to visualize feature extraction.**



3. using Jupyter for Faster R-cnn

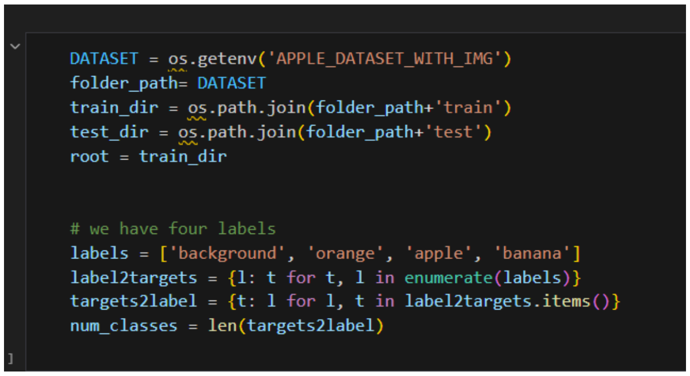
Step 1: Model Initialization

➢ **Import FastRCNNPredictor:** We begin by importing the FastRCNNPredictor class from PyTorch, which allows us to define the final classification and bounding box regression layers of the Faster R-CNN model.



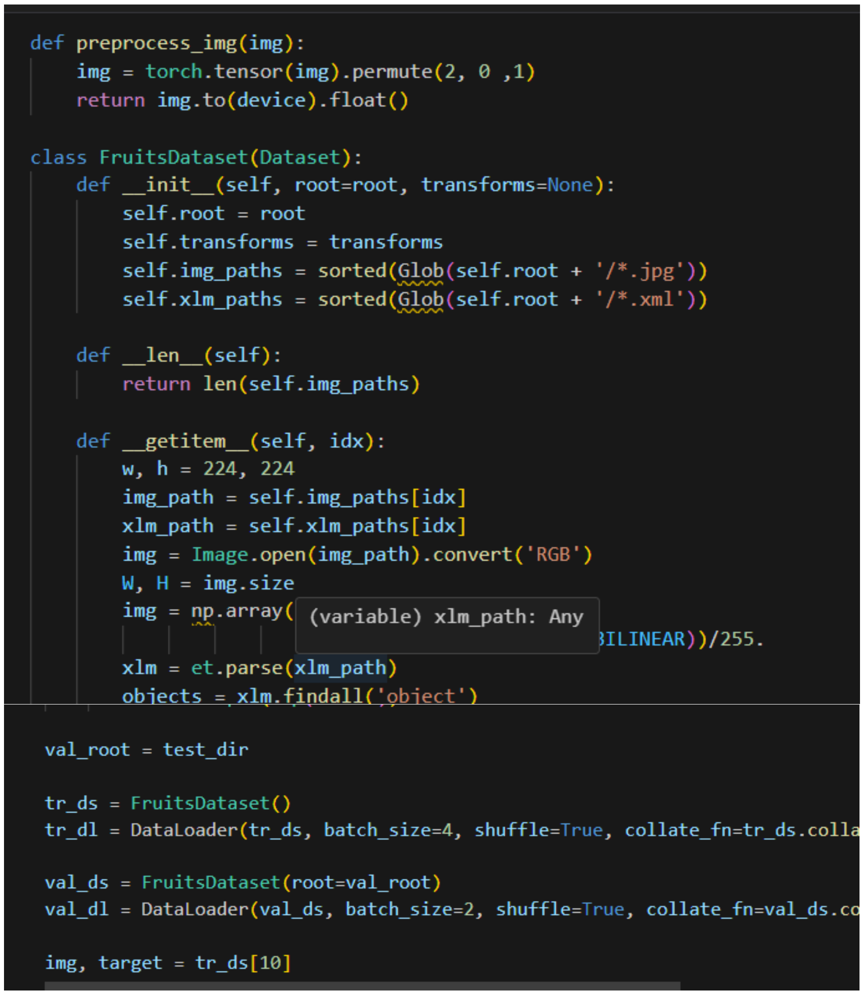
```
from torch_snippets import *
import torch
from PIL import Image
from torch.utils.data import Dataset, DataLoader
from torchvision import models
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from xml.etree import ElementTree as et
device = 'cpu'
from dotenv import load_dotenv

load_dotenv()
```

➢ **Define Dataset and Labels:** We define our dataset and specify the labels (object classes) present in the data.



```
DATASET = os.getenv('APPLE_DATASET_WITH_IMG')
folder_path= DATASET
train_dir = os.path.join(folder_path+'train')
test_dir = os.path.join(folder_path+'test')
root = train_dir


# we have four labels
labels = ['background', 'orange', 'apple', 'banana']
label2targets = {l: t for t, l in enumerate(labels)}
targets2label = {t: l for l, t in label2targets.items()}
num_classes = len(targets2label)
```

➢ **Data Preprocessing:** We perform necessary data preprocessing steps like image resizing, normalization, and augmentation to prepare the data for the model.

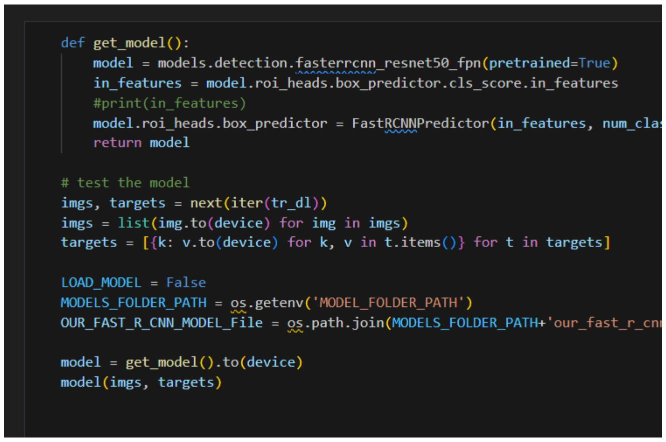


```
def preprocess_img(img):
    img = torch.tensor(img).permute(2, 0 ,1)
    return img.to(device).float()

class FruitsDataset(Dataset):
    def __init__(self, root=root, transforms=None):
        self.root = root
        self.transforms = transforms
        self.img_paths = sorted(Glob(self.root + '/*.jpg'))
        self.xlm_paths = sorted(Glob(self.root + '/*.xml'))

    def __len__(self):
        return len(self.img_paths)

    def __getitem__(self, idx):
        w, h = 224, 224
        img_path = self.img_paths[idx]
        xlm_path = self.xlm_paths[idx]
        img = Image.open(img_path).convert('RGB')
        W, H = img.size
        img = np.array( (variable) xlm_path: Any
                                              BILINEAR))/255.
        xlm = et.parse(xlm_path)
        objects = xlm.findall('object')
```

```
val_root = test_dir

tr_ds = FruitsDataset()
tr_dl = DataLoader(tr_ds, batch_size=4, shuffle=True, collate_fn=tr_ds.colla

val_ds = FruitsDataset(root=val_root)
val_dl = DataLoader(val_ds, batch_size=2, shuffle=True, collate_fn=val_ds.co

img, target = tr_ds[10]
```

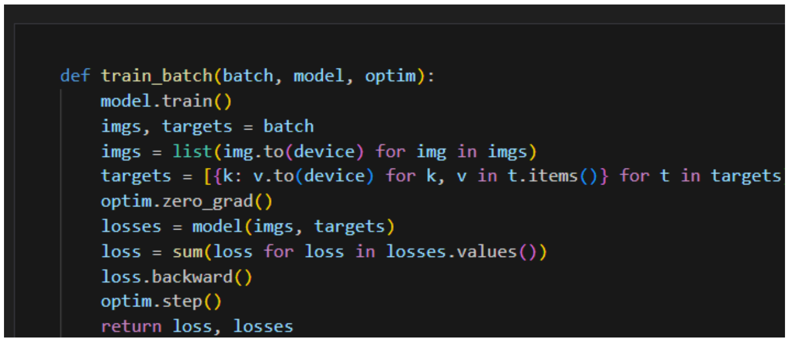
Step 2: Model Definition and Training Setup

- **get_model Function:** This function takes the desired backbone (e.g., ResNet, VGG) and defines the full Faster R-CNN model architecture. It utilizes FastRCNNPredictor to create the final classification and regression layers.



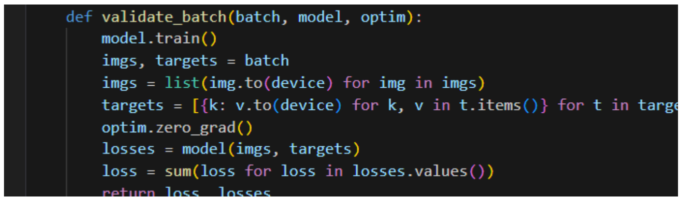
```
def get_model():
    model = models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    #print(in_features)
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_clas
    return model

# test the model
imgs, targets = next(iter(tr_dl))
imgs = list(img.to(device) for img in imgs)
targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

LOAD_MODEL = False
MODELS_FOLDER_PATH = os.getenv('MODEL_FOLDER_PATH')
OUR_FAST_R_CNN_MODEL_File = os.path.join(MODELS_FOLDER_PATH+'our_fast_r_cnn

model = get_model().to(device)
model(imgs, targets)
```

- **train Function:** This function implements the training loop for the model. It iterates through training data batches, calculates loss, performs backpropagation, and updates model parameters.



```
def train_batch(batch, model, optim):
    model.train()
    imgs, targets = batch
    imgs = list(img.to(device) for img in imgs)
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets
    optim.zero_grad()
    losses = model(imgs, targets)
    loss = sum(loss for loss in losses.values())
    loss.backward()
    optim.step()
    return loss, losses
```

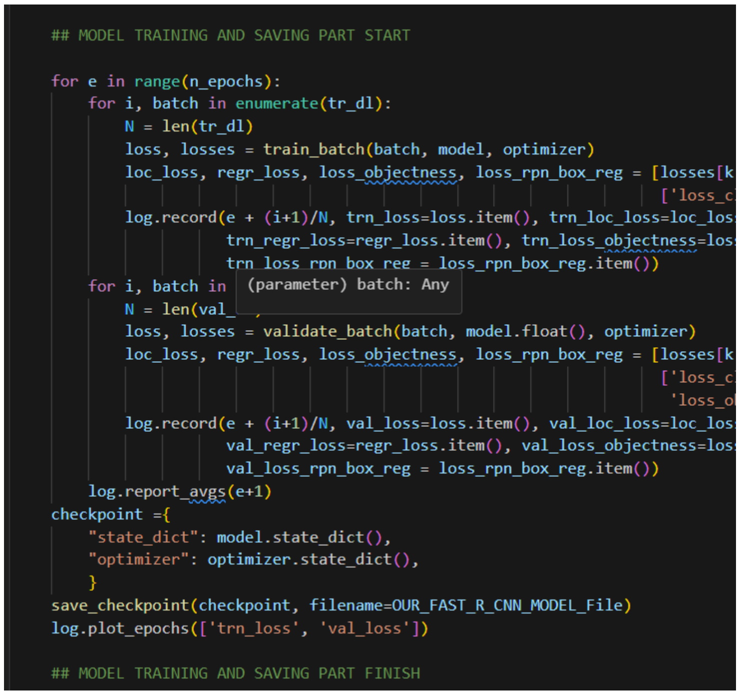
- **validate Function:** This function evaluates the model's performance on a validation dataset, calculating metrics like mean average precision (mAP) to assess its accuracy.



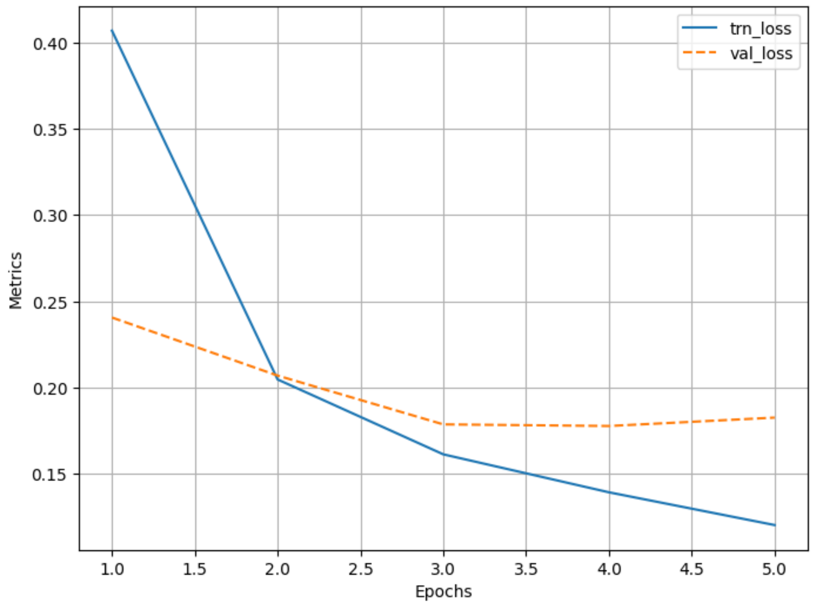
```
def validate_batch(batch, model, optim):
    model.train()
    imgs, targets = batch
    imgs = list(img.to(device) for img in imgs)
    targets = [{k: v.to(device) for k, v in t.items()} for t in targe
    optim.zero_grad()
    losses = model(imgs, targets)
    loss = sum(loss for loss in losses.values())
    return loss, losses
```

Step 3: Training and Model Saving

- **Training Loop:** We call the train function to train the model for a specified number of epochs, potentially utilizing Jupyter's interactive nature to monitor progress.
- **Model Saving:** We save the trained model's parameters to a file for later use, ensuring that we can reuse the model without retraining.

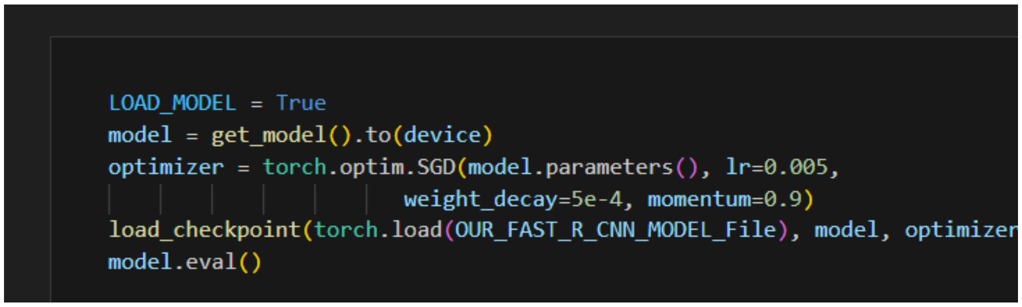


```
## MODEL TRAINING AND SAVING PART START

for e in range(n_epochs):
    for i, batch in enumerate(tr_dl):
        N = len(tr_dl)
        loss, losses = train_batch(batch, model, optimizer)
        loc_loss, regr_loss, loss_objectness, loss_rpn_box_reg = [losses[k
                                                                  ['loss_c
        log.record(e + (i+1)/N, trn_loss=loss.item(), trn_loc_loss=loc_los
                   trn_regr_loss=regr_loss.item(), trn_loss_objectness=los
                   trn_loss_rpn_box_reg = loss_rpn_box_reg.item())
    for i, batch in  (parameter) batch: Any
        N = len(val_
        loss, losses = validate_batch(batch, model.float(), optimizer)
        loc_loss, regr_loss, loss_objectness, loss_rpn_box_reg = [losses[k
                                                                  ['loss_c
                                                                   'loss_o
        log.record(e + (i+1)/N, val_loss=loss.item(), val_loc_loss=loc_los
                   val_regr_loss=regr_loss.item(), val_loss_objectness=los
                   val_loss_rpn_box_reg = loss_rpn_box_reg.item())
    log.report_avgs(e+1)
checkpoint ={
    "state_dict": model.state_dict(),
    "optimizer": optimizer.state_dict(),
    }
save_checkpoint(checkpoint, filename=OUR_FAST_R_CNN_MODEL_File)
log.plot_epochs(['trn_loss', 'val_loss'])

## MODEL TRAINING AND SAVING PART FINISH
```



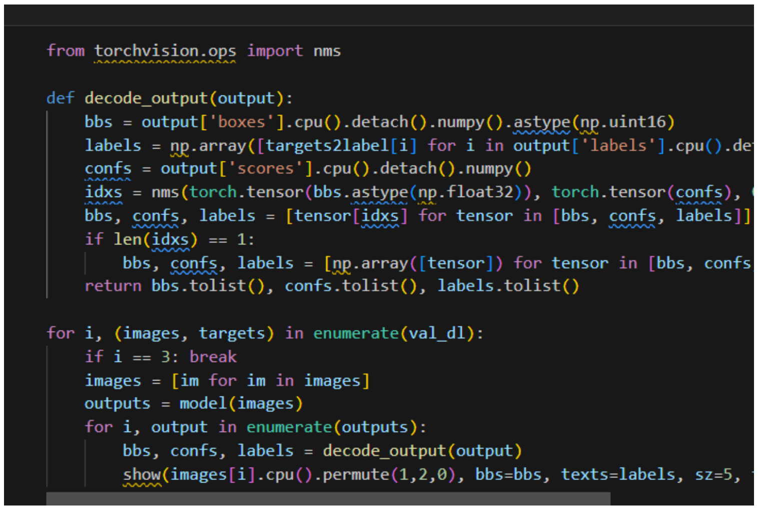
Step 4: Inference and Prediction

- **Model Loading:** We load the saved model from the file.



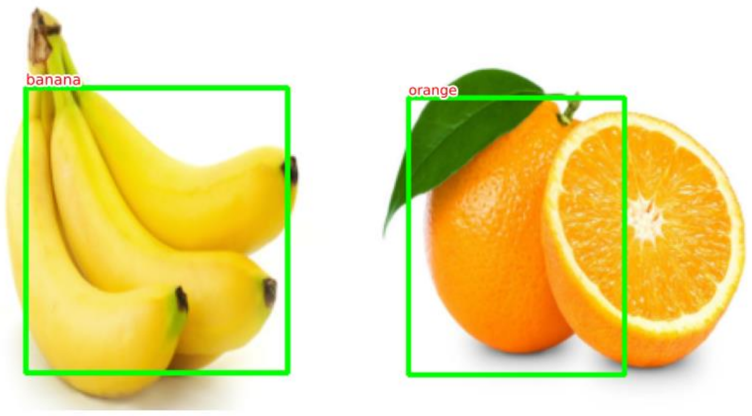
```
LOAD_MODEL = True
model = get_model().to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=0.005,
                            weight_decay=5e-4, momentum=0.9)
load_checkpoint(torch.load(OUR_FAST_R_CNN_MODEL_File), model, optimizer
model.eval()
```

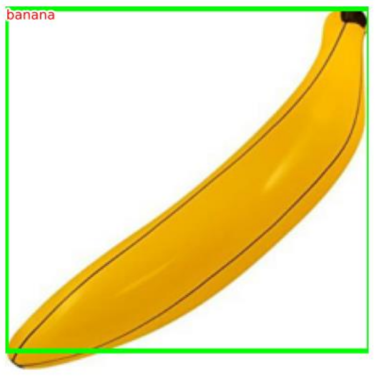
- **Image Prediction:** We provide a sample image to the loaded model for inference.

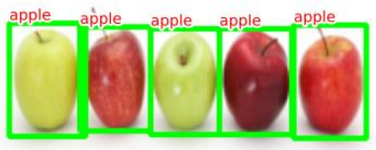


```
from torchvision.ops import nms

def decode_output(output):
    bbs = output['boxes'].cpu().detach().numpy().astype(np.uint16)
    labels = np.array([targets2label[i] for i in output['labels'].cpu().de
    confs = output['scores'].cpu().detach().numpy()
    idxs = nms(torch.tensor(bbs.astype(np.float32)), torch.tensor(confs),
    bbs, confs, labels = [tensor[idxs] for tensor in [bbs, confs, labels]]
    if len(idxs) == 1:
        bbs, confs, labels = [np.array([tensor]) for tensor in [bbs, confs
    return bbs.tolist(), confs.tolist(), labels.tolist()

for i, (images, targets) in enumerate(val_dl):
    if i == 3: break
    images = [im for im in images]
    outputs = model(images)
    for i, output in enumerate(outputs):
        bbs, confs, labels = decode_output(output)
        show(images[i].cpu().permute(1,2,0), bbs=bbs, texts=labels, sz=5,
```

- **Visualization:** The model's predictions (bounding boxes and class labels) overlaid on the image.







4. What is CNN?

Convolution Neural Networks (CNNs) have emerged as a powerful tool in the field of artificial intelligence (AI). These deep learning models are designed to **mimic the human brain's visual processing abilities, enabling them to analyze and interpret complex visual data**.

Padding in Convolutional Neural Networks

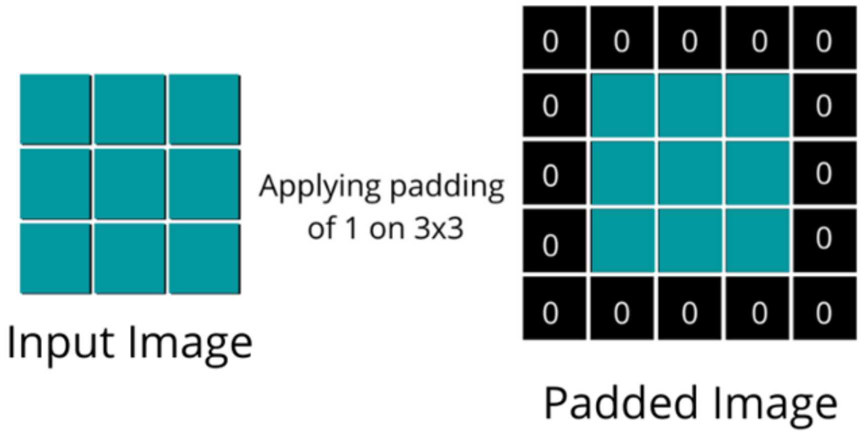
When performing a convolution operation on an image, adding padding can help **preserve the spatial dimensions of the input throughout the network layers**. Without padding, the output dimensions become smaller than the input. By adding a layer of zeros around the image border, we can apply the filter to more positions, maintaining the output dimensions.

There are two types of padding:

- **Valid Padding:** No padding is added, and the filter is applied only to valid positions within the image, resulting in smaller output dimensions.
- **Same Padding:** The image is padded with zeros, ensuring that the output dimensions after convolution are the same as the input dimensions.

Padding serves two main purposes:

- **Dimension Preservation:** Padding helps maintain the spatial dimensions of the input image as it progresses through the network layers.
- **Border Information:** Padding allows the network to consider information from the edges of the image, preventing the loss of important details that may be located at the border.



Stride in Convolutional Neural Networks

Stride is a parameter in convolutional operations that determines **how much the filter moves across the input matrix**. A stride of 1 means the filter moves one pixel at a time, while a stride of 2 causes the filter to jump two pixels at a time.

The effect of the stride value can be summarized as follows:

- **Dimensionality Reduction**: Using larger stride values leads to smaller output dimensions, resulting in dimensionality reduction.
- **Computation Speed**: Larger strides can speed up computation since the filter is applied fewer times.
- **Model Capacity**: However, larger strides may cause the model to lose detailed information because the filter skips over some pixels. This can potentially reduce

model accuracy, especially when capturing fine-grained details is important.

In the convolution operation formula:

$$(n \times n) * (f \times f) = \left\lfloor \left(\frac{n-f}{s}+1\right) \right\rfloor \times \left\lfloor \left(\frac{n-f}{s}+1\right) \right\rfloor$$

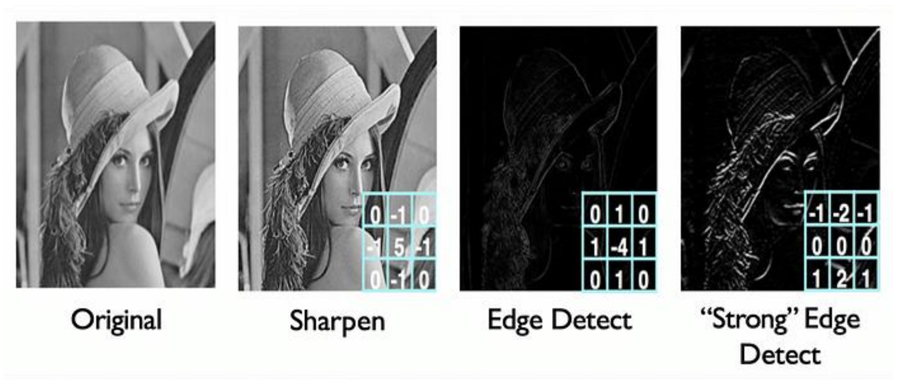
where:

- n x n represents the image size,
- f x f represents the filter size,
- s represents the stride length,
- ⌊ ⌋ denotes the floor function, which rounds down to the nearest integer.

CNN Layer Types

Convolution layers:

Convolutional layers are the primary building blocks of CNNs. These layers perform the **convolution operation**, where filters are applied to input images to extract features. Each filter is a small matrix of weights that is convolved with the input image, producing a feature map. By sliding the filter across the image and computing the dot product at each position, the convolutional layer captures different features at different locations. The number of filters in a convolutional layer determines the number of feature maps it produces. This allows the network to learn multiple features simultaneously. Furthermore, convolutional layers can have different kernel sizes, which dictate the receptive field of the filters. Larger kernel sizes capture more global features, while smaller kernel sizes focus on finer details.



Convolution Operation

- So, the convolution operation is responsible for detecting edges and features from the images

Fully Connected Layers (FC) / Dense Layer:

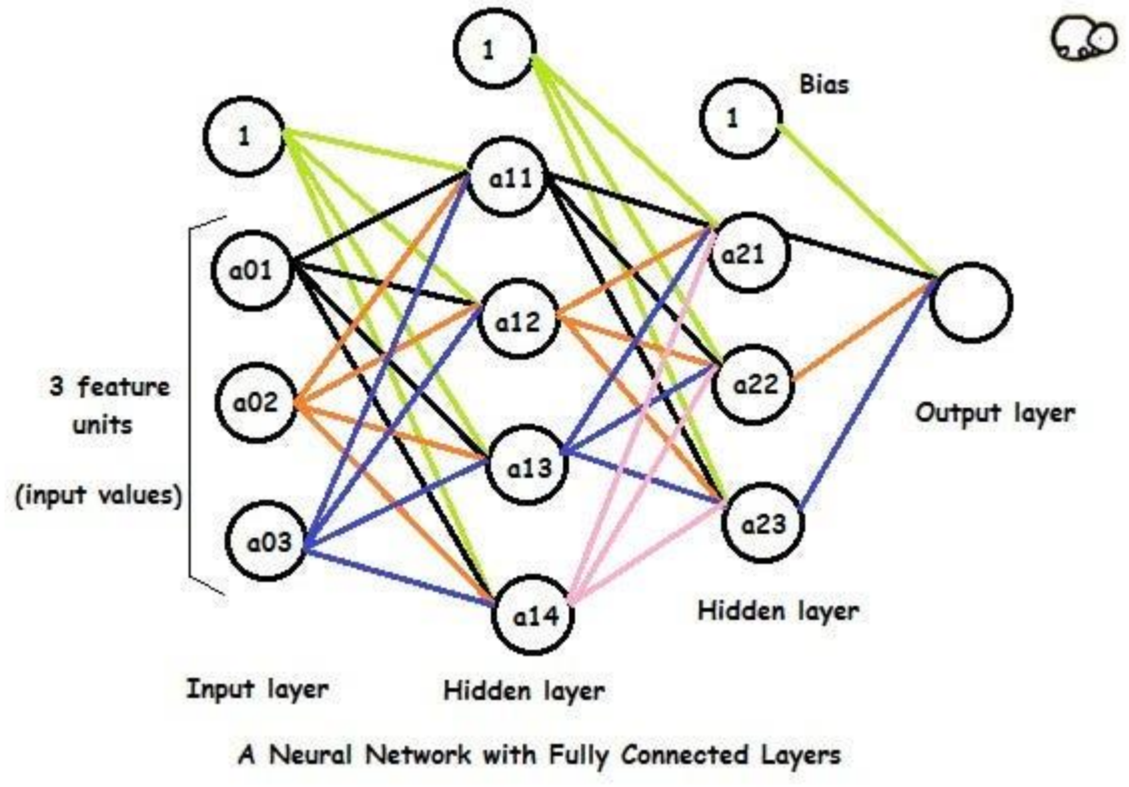
Fully connected (FC) layers, also known as Dense layers, are an important component of artificial neural networks. They are typically located at the end of a network and play a crucial role in generating final predictions.

FC layers establish connections between every neuron in the previous layer and each neuron in the current layer. In other words, every node in the preceding layer is linked to every node in the FC layer. This connectivity pattern enables information flow and computation across the network.

In the context of Convolutional Neural Networks (CNNs), FC layers are commonly employed after convolutional and pooling layers. Their primary purpose is to transform the two-dimensional spatial structure of the data into a one-dimensional vector, which can then be processed for classification tasks.

During the training process, the weights and biases in FC layers are adjusted and refined, ensuring that they adapt to the specific problem being addressed. This adaptability allows the network to learn and improve its performance over time.

The number of neurons in the final FC layer is typically determined by the number of output classes in a classification problem. For example, if the task is to classify handwritten digits into ten different classes (0 to 9), the final FC layer would consist of ten neurons. Each neuron would produce a score indicating the likelihood of the input belonging to its corresponding class.



Convolution layer vs Fully Connected layer

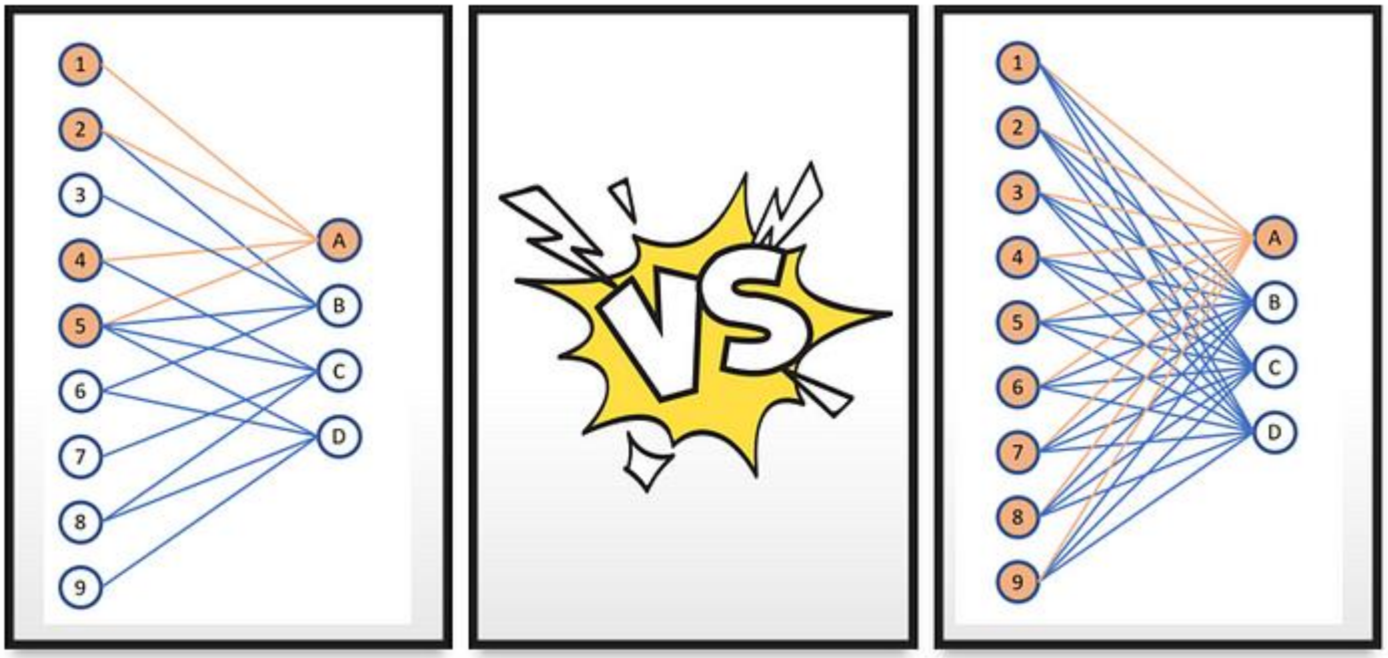


Image by Towards Data Science

Pooling layers:

Pooling layers play a crucial role in **reducing the spatial dimensions of the feature maps.** The most common pooling operation is **max pooling**, which selects the **maximum value within a local neighborhood and discards the rest.** This downsampling process helps to extract the most salient features while discarding redundant or irrelevant information.

Max pooling also introduces a degree of translation invariance, as the maximum value represents the presence of a feature regardless of its precise location. This property allows CNNs to be robust to small translations and increases their ability to generalize to new examples.

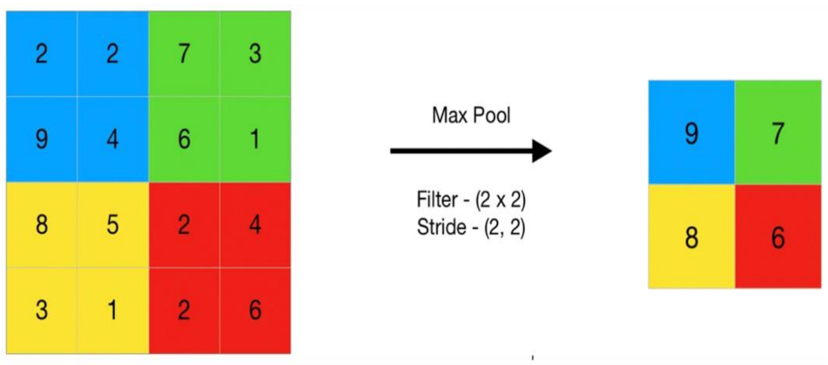
Purpose of pooling

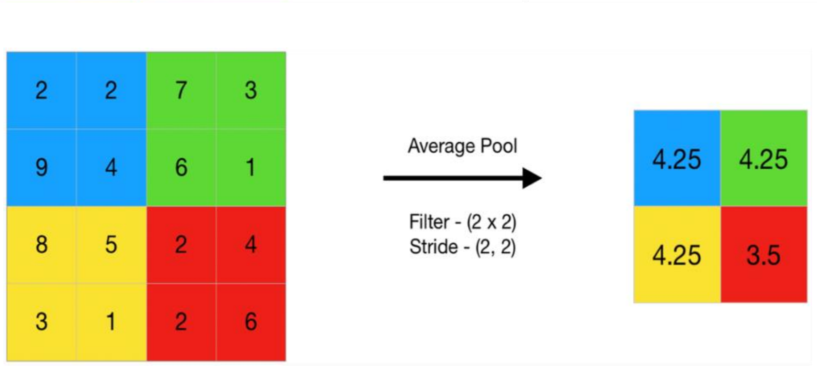
— reduce the dimension of the feature maps.

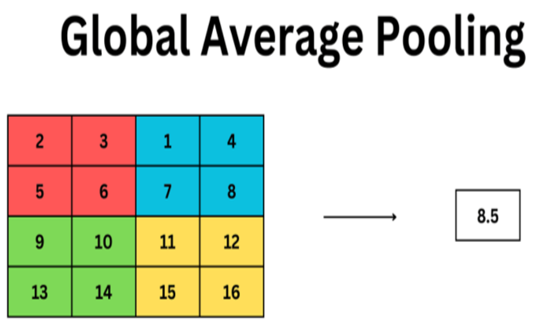
— enhance features.

Types of pooling

— max pooling, Average pooling , Global Pooling

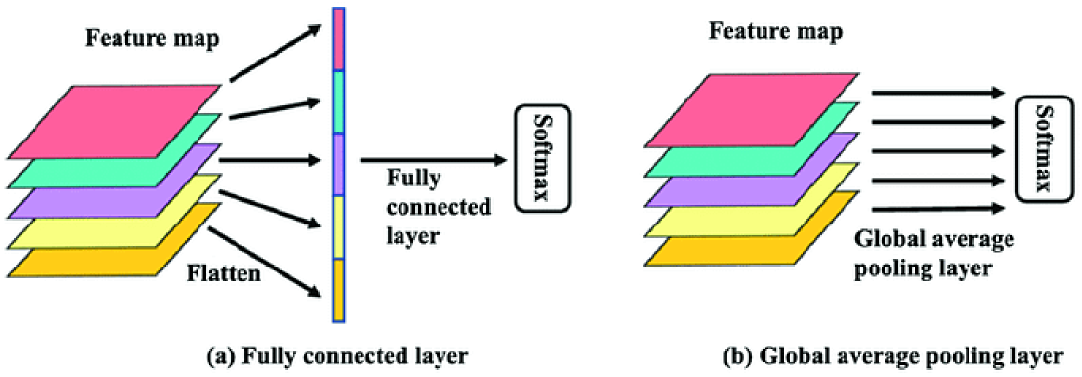






Max pooling, Average Pooling & Global Average Pooling

Fully connected layer vs Global average pooling layer


Feature map
Flatten
Fully connected layer
Softmax
(a) Fully connected layer
Feature map
Global average pooling layer
Softmax
(b) Global average pooling layer

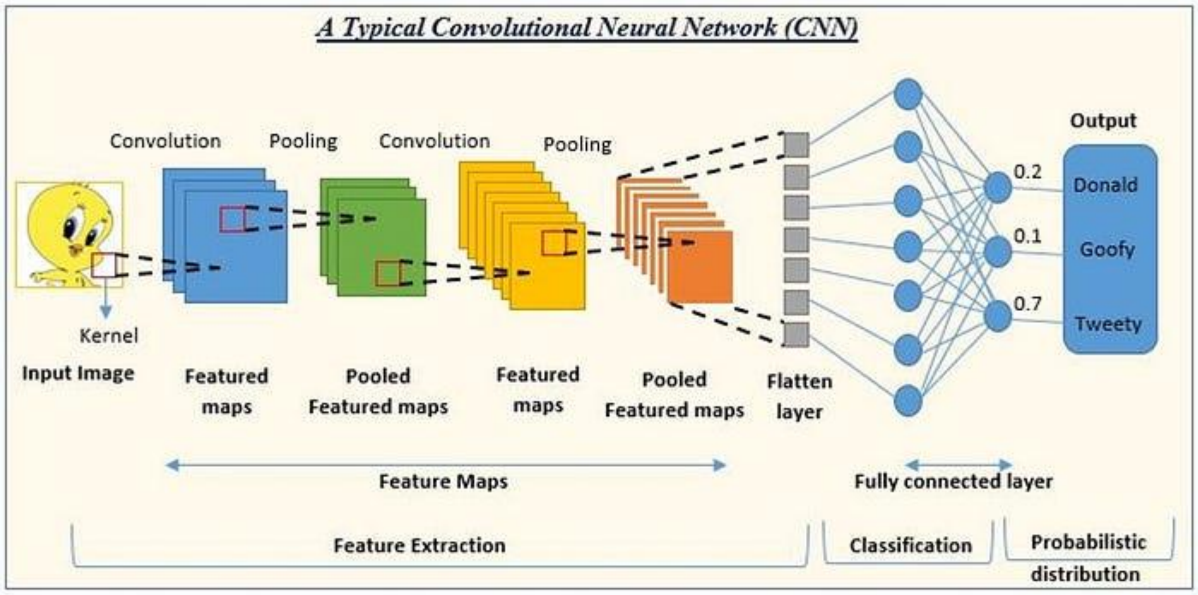
Different Types of CNN Architecture

- LeNet, 1998
- AlexNet, 2012
- VGGNet, 2014
- ResNet, 2015

Applications of CNNs

CNNs have revolutionized a wide range of industries by achieving state-of-the-art performance in various tasks. In computer vision, CNNs have been extensively used for image classification, object detection, and image segmentation. They have also been employed in natural language processing for tasks such as sentiment analysis, text classification, and machine translation. Furthermore, CNNs have found applications in medical imaging, autonomous vehicles, robotics, and many more domains.

CNN for image classification



The diagram illustrates the flow of information through the network, starting from the input layer and progressing towards the output layer. Each layer in the network performs specific operations on the input data to extract meaningful features and make predictions.

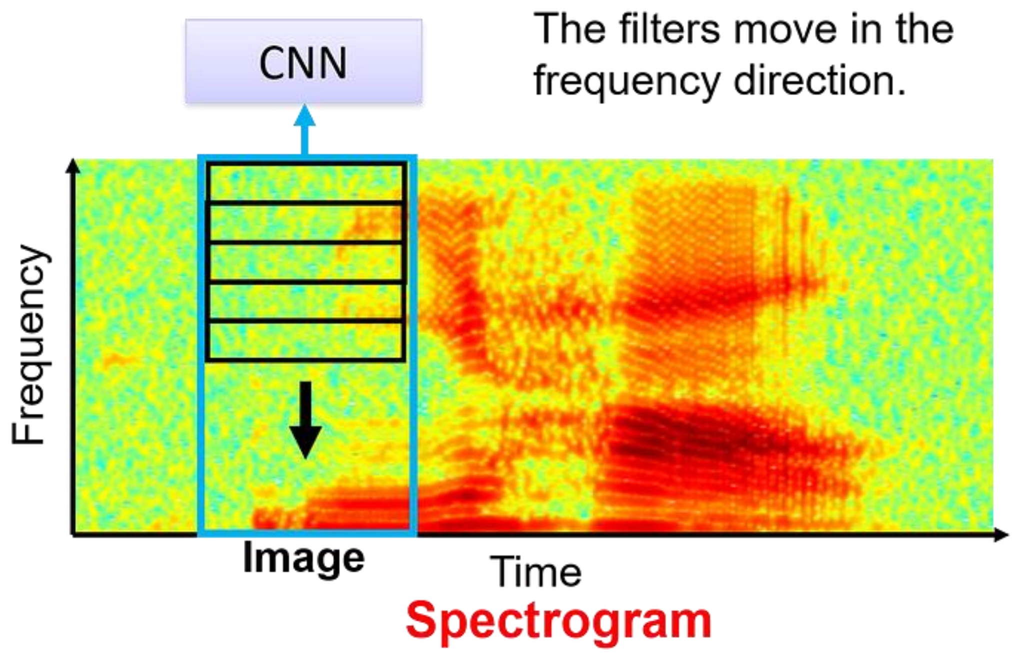
The network consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply filters to the input data, performing feature extraction by detecting patterns and structures in the image. Pooling layers downsample the data, reducing its spatial dimensions and aiding in dimensionality reduction.

The diagram also highlights the use of padding, which is the addition of extra pixels around the border of the image. Padding helps maintain the spatial dimensions of the input throughout the network layers, ensuring that important information at the image edges is not lost.

Another technique shown in the diagram is the use of strides. Strides determine the number of pixels by which the filter moves across the input data. Larger strides result in smaller output dimensions and faster computation, but they may lead to a loss of detailed information.

Weights and biases are learned during the training process, allowing the network to adapt to the specific problem it aims to solve. The output of each layer is passed to the next layer until the final output layer, which produces the network's prediction.

CNN for the Speech recognition

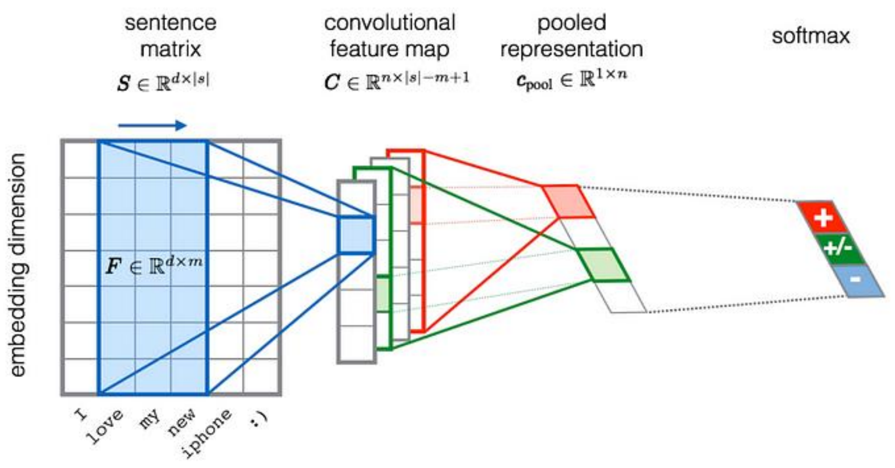


Speech recognition using Convolutional Neural Networks (CNN) is a technique where a CNN is trained to recognize and understand spoken words or phrases. The network consists of convolutional layers, pooling layers, and fully connected layers. These layers work together to process and extract important features from input data, specifically spectrograms of spoken words or phrases.

The CNN employs techniques like padding and strides. Padding helps maintain the size of the input data throughout the layers, while strides determine the movement of filters across the data.

The diagram showcases how the CNN processes spectrograms, extracting relevant features and patterns to identify spoken words. Through training, the CNN learns to associate specific spectrogram patterns with corresponding words or phrases.

CNN for the text classification


sentence matrix
$S \in \mathbb{R}^{d \times |s|}$
convolutional feature map
$C \in \mathbb{R}^{n \times |s|-m+1}$
pooled representation
$c_{pool} \in \mathbb{R}^{1 \times n}$
softmax
embedding dimension
$F \in \mathbb{R}^{d \times m}$
+
+/-
-
I
love
my
new
iphone
:)

Text classification is the process of categorizing or labeling text data into predefined categories or classes. Text classification typically involves several key components:

1. Data Preprocessing: Textual data is cleaned and transformed to remove noise, punctuation, and irrelevant information. It may involve techniques like tokenization, stemming, or lemmatization to standardize the text.

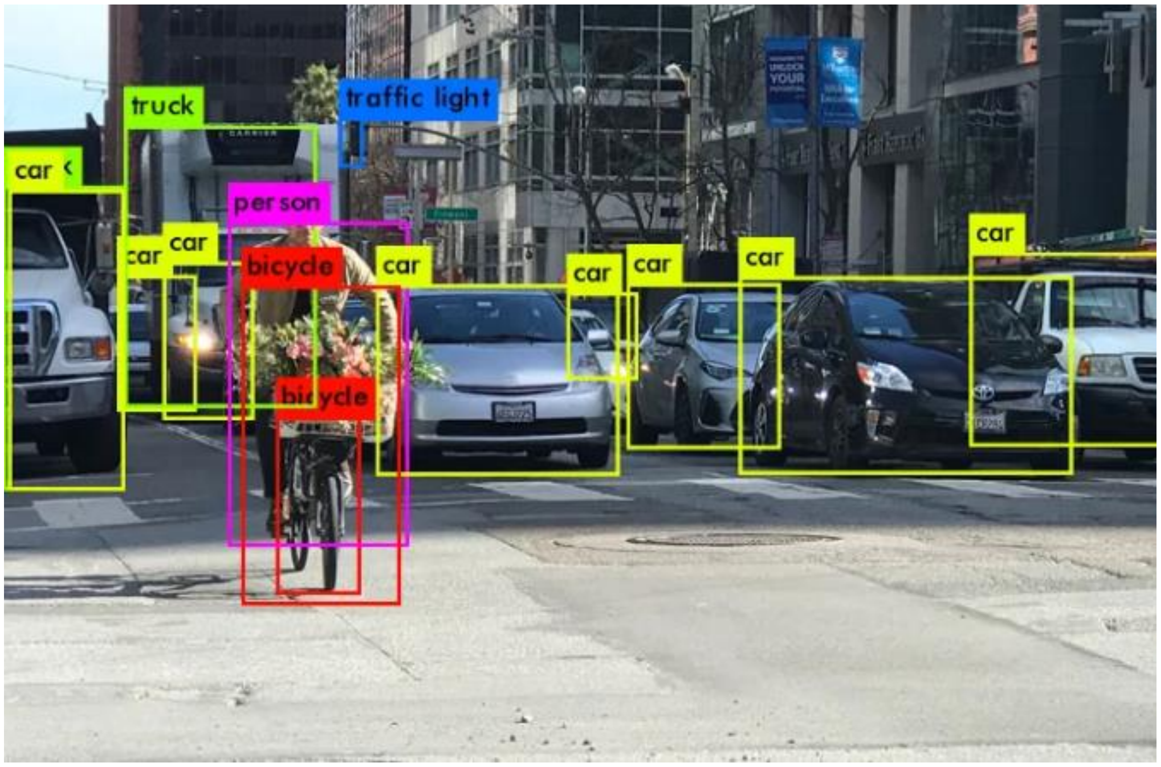
2. Feature Extraction: Relevant features are extracted from the preprocessed text to represent the content in a numerical format that machine learning algorithms can understand. This can involve methods like Bag-of-Words, TF-IDF, or word embeddings like Word2Vec or GloVe.

3. Model Training: A classification model, such as a Naive Bayes, Support Vector Machine (SVM), or more advanced models like Recurrent Neural Networks (RNN) or Transformers, is trained on labeled data. The model learns patterns and relationships between features and their corresponding classes.

4. Model Evaluation and Tuning: The trained model is evaluated using evaluation metrics like accuracy, precision, recall, or F1-score. Hyperparameters may be tuned to optimize the model's performance.

5. Prediction and Deployment: Once the model is trained and evaluated, it can be used to predict the class of new, unseen text data. The model can be deployed in real-world applications to automatically classify text into relevant categories.

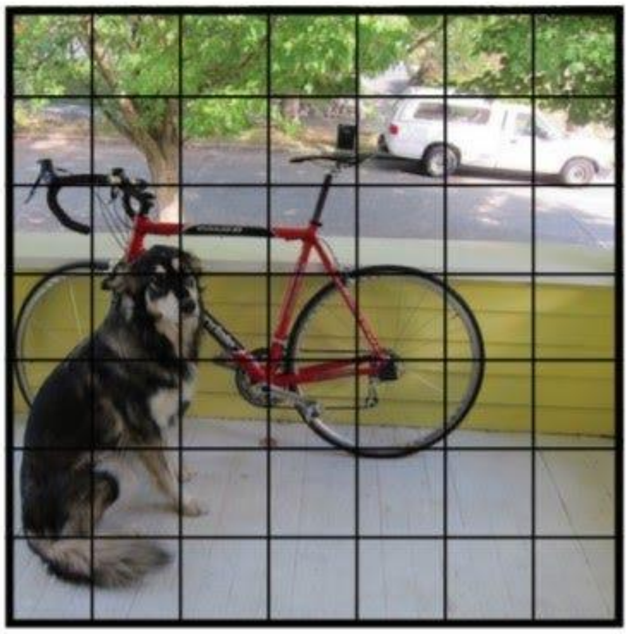
5. What is Yolo Network?



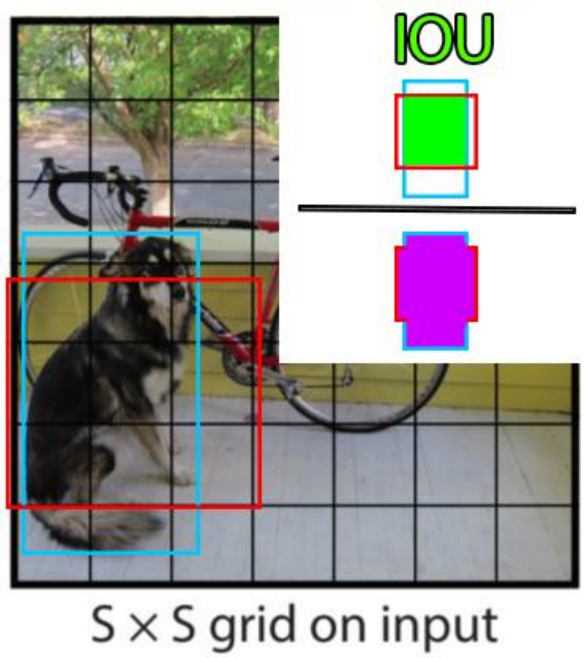
YOLO or You Only Look Once, is a popular real-time object detection algorithm. YOLO combines what was once a multi-step process, using a single neural network to perform both classification and prediction of bounding boxes for detected objects. As such, it is heavily optimized for detection performance and can run much faster than running two separate neural networks to detect and classify objects separately. It does this by repurposing traditional image classifiers to be used for the regression task of identifying bounding boxes for objects. This article will only look at YOLOv1, the first of the many iterations this architecture has gone through. Although the subsequent iterations feature numerous improvements, the basic idea behind the architecture stays the same. YOLOv1 referred to as just YOLO, can perform faster than real-time object detection at 45 frames per second, making it a great choice for applications that require real-time detection. It looks at the entire image at once, and only once — hence the name You Only Look Once — which allows it to capture the context of detected objects. This halves the number of false-positive detections it makes over R-CNNs which look at different parts of the image separately. Additionally, YOLO can generalize the representations of various objects, making it more applicable to a variety of new environments. Now that we have a general overview of YOLO, let's take a look at how it really works.

How Does YOLO Work?

YOLO is based on the idea of segmenting an image into smaller images. The image is split into a square grid of dimensions S×S, like so:

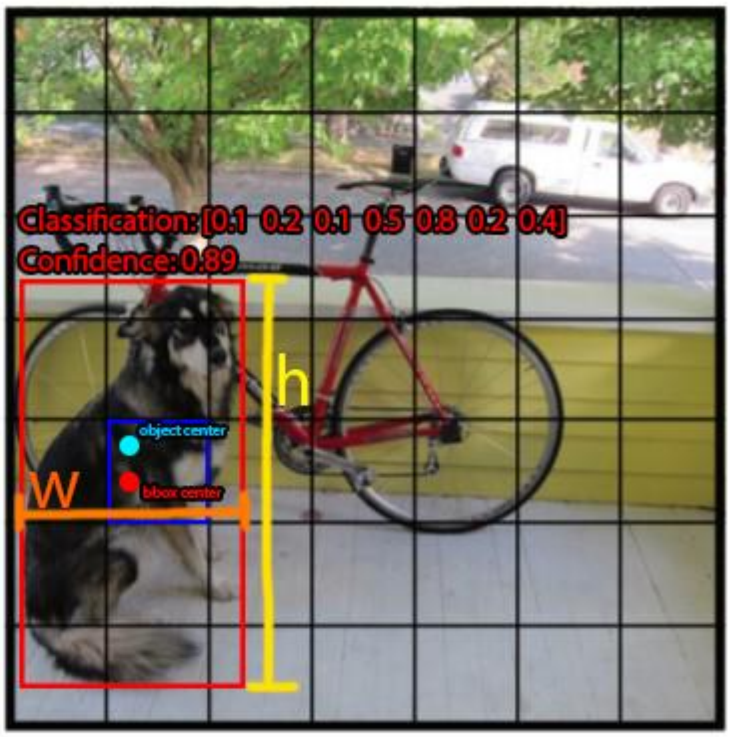


The cell in which the center of an object, for instance, the center of the dog, resides, is the cell responsible for detecting that object. Each cell will predict B bounding boxes and a confidence score for each box. The default for this architecture is for the model to predict two bounding boxes. The classification score will be from `0.0` to `1.0`, with`0.0` being the lowest confidence level and `1.0` being the highest; if no object exists in that cell, the confidence scores should be `0.0`, and if the model is completely certain of its prediction, the score should be `1.0`. These confidence levels capture the model's certainty that there exists an object in that cell and that the bounding box is accurate. Each of these bounding boxes is made up of 5 numbers: the x position, the y position, the width, the height, and the confidence. The coordinates `(x, y)` represent the location of the center of the predicted bounding box, and the width and height are fractions relative to the entire image size. The confidence represents the IOU between the predicted bounding box and the actual bounding box, referred to as the ground truth box. The IOU stands for Intersection Over Union and is the area of the intersection of the predicted and ground truth boxes divided by the area of the union of the same predicted and ground truth boxes.



Here is an example of an IOU: the area of intersection of the ground truth and predicted box in green divided by the area of the union of the two boxes, in purple. This will be between 0 and 1, 0 if they don't overlap at all, and 1 if they are the same box. Therefore, a higher IOU is better as it is a more accurate prediction.

In addition to outputting bounding boxes and confidence scores, each cell predicts the class of the object. This class prediction is represented by a one-hot vector length C, the number of classes in the dataset. However, it is important to note that while each cell may predict any number of bounding boxes and confidence scores for those boxes, it only predicts one class. This is a limitation of the YOLO algorithm itself, and if there are multiple objects of different classes in one grid cell, the algorithm will fail to classify both correctly. Thus, each prediction from a grid cell will be of shape ***C* + *B* * *5***, where ***C*** is the number of classes and ***B*** is the number of predicted bounding boxes. ***B*** is multiplied by 5 here because it includes ***(x, y, w, h, confidence)*** for each box. Because there are $S \times S$ grid cells in each image, the overall prediction of the model is a tensor of shape $S \times S \times (C + B * 5)$.



S × S grid on input

Here is an example of the output of the model when only predicting a single bounding box per cell. In this image, the dog's true center is represented by the cyan circle labeled 'object center'; as such, the grid cell responsible for detecting and bounding the box is the one containing the cyan dot, highlighted in dark blue. The bounding box that the cell predicts is made up of 4 elements. The red dot represents the center of the bounding box, ***(x, y)***, and the width and height are represented by the orange and yellow markers respectively. It is important to note that the model predicts the center of the bounding box with widths and heights rather than top left and bottom right corner positions. The classification is represented by a one-hot, and in this trivial example, there are 7 different classes. The 5th class is the prediction and we can see that the model is quite certain of its prediction. Keep in mind that this is merely an example to show the kind of output that is possible and so the values may not be accurate to any real values. Below is another image of all the bounding boxes and class predictions that would actually be made and their final result.

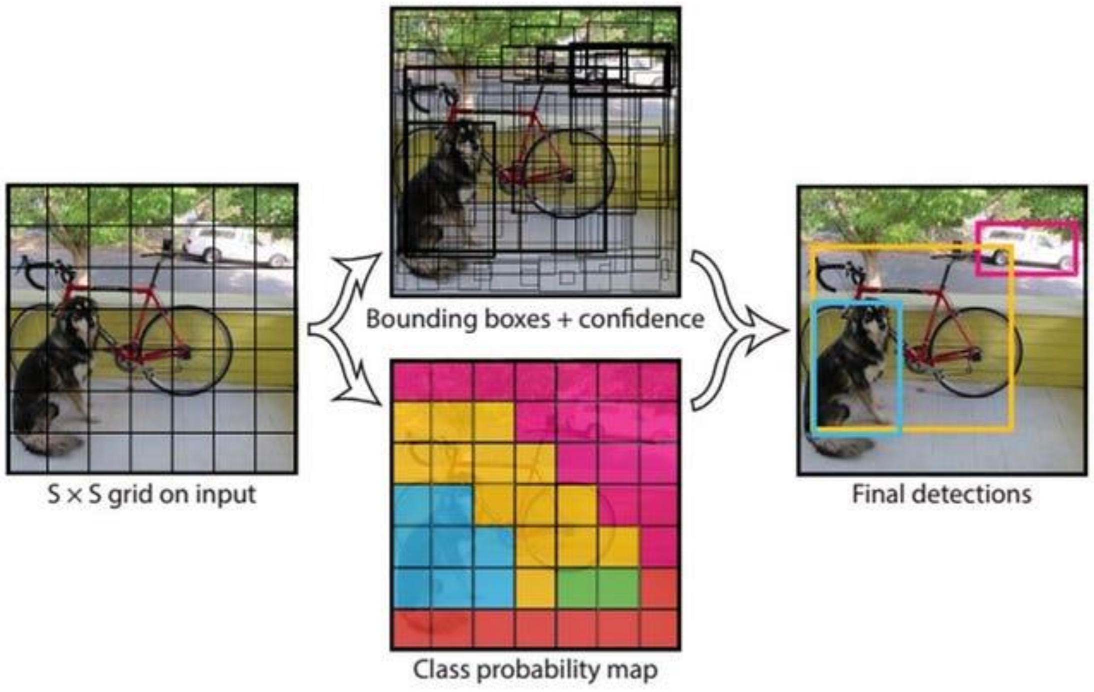


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

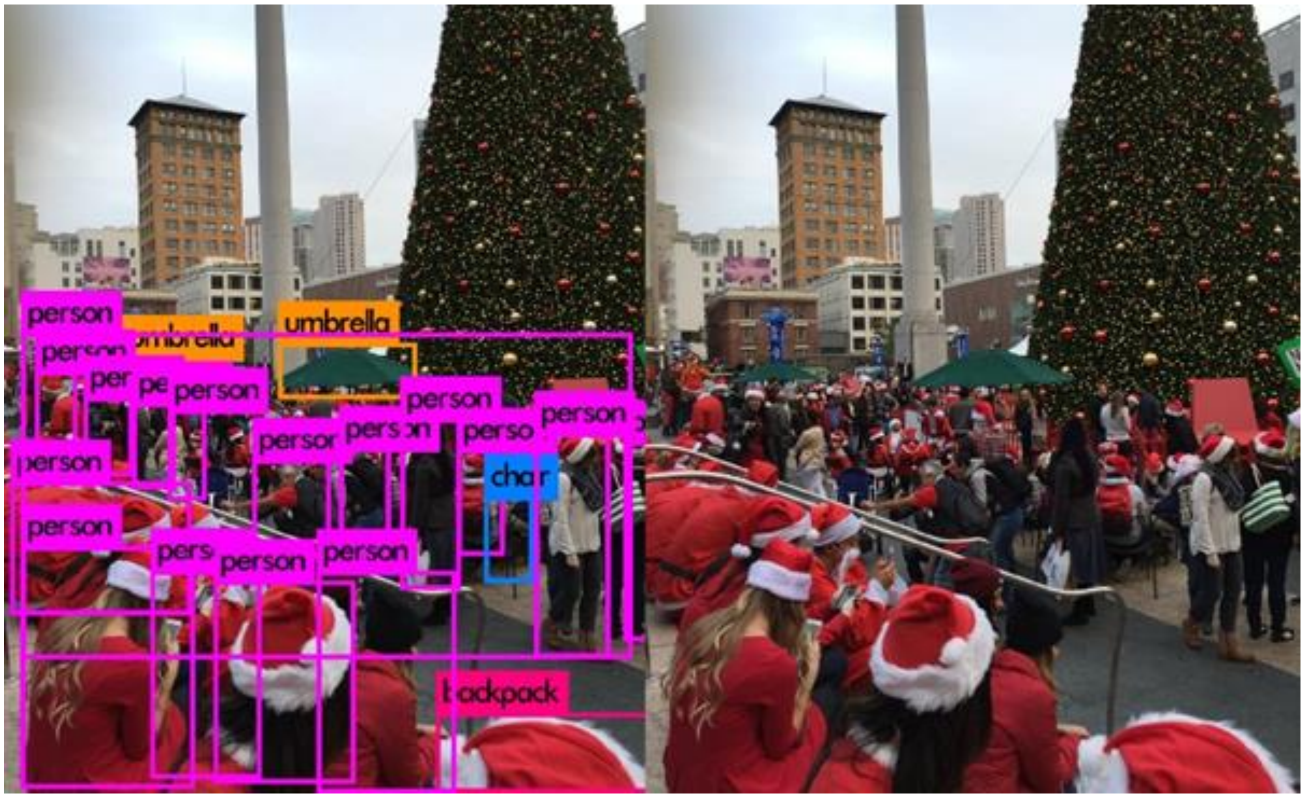
YOLO Architecture

The YOLO model is made up of three key components: the head, neck, and backbone. The backbone is the part of the network made up of convolutional layers to detect key features of an image and process them. The backbone is first trained on a classification dataset, such as ImageNet, and typically trained at a lower resolution than the final detection model, as detection requires finer details than classification. The neck uses the features from the convolution layers in the backbone with fully connected layers to make predictions on probabilities and bounding box coordinates. The head is the final output layer of the network which can be interchanged with other layers with the same input shape for transfer learning. As discussed earlier, the head is an $\boldsymbol{S \times S \times (C + B * 5)}$ tensor and is $\boldsymbol{7 \times 7 \times 30}$ in the original

YOLO research paper with a split size ***S*** of 7, 20 classes ***C***, and 2 predicted bounding boxes ***B***. These three portions of the model work together to first extract key visual features from the image then classify and bound them.

Limitations of YOLO

YOLO can only predict a limited number of bounding boxes per grid cell, 2 in the original research paper. And though that number can be increased, only one class prediction can be made per cell, limiting the detections when multiple objects appear in a single grid cell. Thus, it struggles with bounding groups of small objects, such as flocks of birds, or multiple small objects of different classes.



Here you can see that only 5 people in the lower left-hand corner are detected by YOLO when there are at least 8 in the lower left-hand corner.

6. What is Faster-RCNN Network?

Faster R-CNN is an extension of Fast R-CNN. As its name suggests, Faster R-CNN is faster than Fast R-CNN thanks to the region proposal network (RPN).

Main Contributions

The main contributions in this paper are:

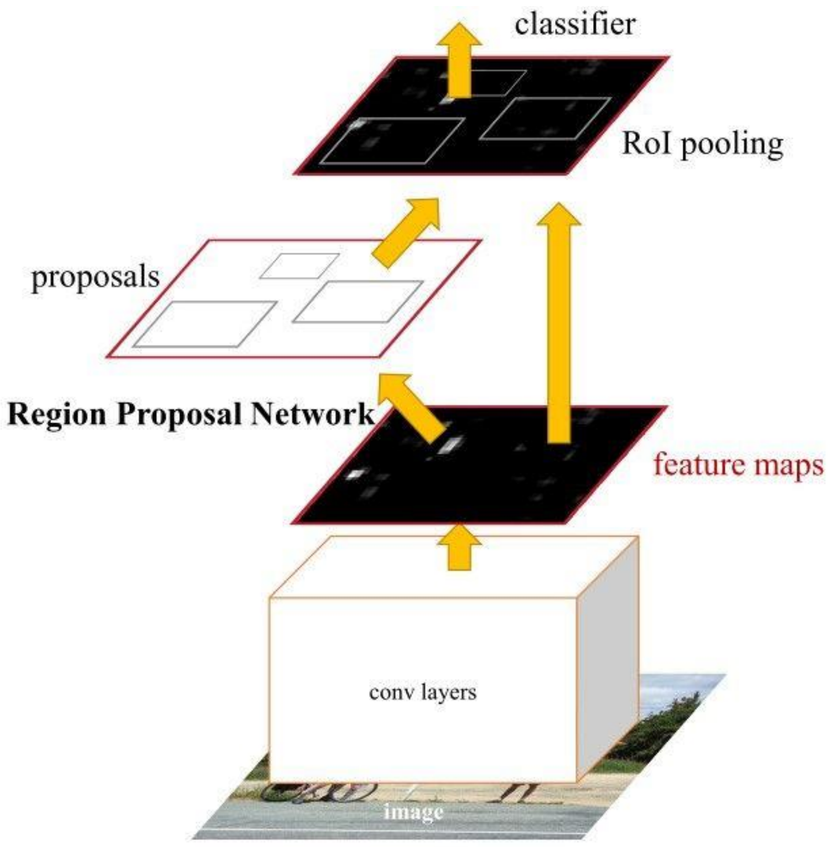
- Proposing region proposal network (RPN) which is a fully convolutional network that generates proposals with various scales and aspect ratios. The RPN implements the terminology of neural network with attention to tell the object detection (Fast R-CNN) where to look.
- Rather than using pyramids of images (i.e. multiple instances of the image but at different scales) or pyramids of filters (i.e. multiple filters with different sizes), this paper introduced the concept of anchor boxes. An anchor box is a reference box of a specific scale and aspect ratio. With multiple reference anchor boxes, then multiple scales and aspect ratios exist for the single region. This can be thought of as a pyramid of reference anchor boxes. Each region is then mapped to each reference anchor box, and thus detecting objects at different scales and aspect ratios.
- The convolutional computations are shared across the RPN and the Fast R-CNN. This reduces the computational time.

The architecture of Faster R-CNN is shown in the next figure. It consists of 2 modules:

I. **RPN:** For generating region proposals.

II. **Fast R-CNN:** For detecting objects in the proposed regions.

The RPN module is responsible for generating region proposals. It applies the concept of attention in neural networks, so it guides the Fast R-CNN detection module to where to look for objects in the image.



Note how the convolutional layers (e.g. computations) are shared across both the RPN and the Fast R-CNN modules.

The Faster R-CNN works as follows:

- The RPN generates region proposals.
- For all region proposals in the image, a fixed-length feature vector is extracted from each region using the ROI Pooling layer.
- The extracted feature vectors are then classified using the Fast R-CNN.
- The class scores of the detected objects in addition to their bounding-boxes are returned.

Report score: ___________

Instructor's signature: ___________