import matplotlib 1.pdf

anonymous marking enabled

Submission date: 02-Aug-2024 01:15AM (UTC+0500)
Submission ID: 2425928599
File name: import_matplotlib_1.pdf (178.39K)
Word count: 3032
Character count: 16264

```
6
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from collections import Counter
from torch.utils.data import DataLoader
import torch
import torch.nn as nn

config = [
    (32, 3, 1),
    (64, 3, 2),
    ["list", 1],
    (128, 3, 2),
    ["list", 2],
    (256, 3, 2),
    ["list", 8],
    (512, 3, 2),
    ["list", 8],
    (1024, 3, 2),
    ["list", 4],

    (512, 1, 1),
    (1024, 3, 1),
    "sp",
    (256, 1, 1),
    "us",
    (256, 1, 1),
    (512, 3, 1),
    "sp",
    (128, 1, 1),
```

```
    "us",
    (128, 1, 1),
    (256, 3, 1),
    "sp",
]

class blockcnn(nn.Module):
  def __init__(self, inchnel, outchnel, bnact=True, **kwargs):
    super(blockcnn, self).__init__()
    self.conv = nn.Conv2d(inchnel, outchnel, bias=not bnact, **kwargs)
    self.bn = nn.BatchNorm2d(outchnel)
    self.leaky = nn.LeakyReLU(0.1)
    self.use_bnact = bnact

  def send(self, x):
    if self.use_bnact:
      return self.leaky(self.bn(self.conv(x)))
    else:
      return self.conv(x)

class residul(nn.Module):
  def __init__(self, chnel, use_residual=True, repeatnum=1):
    super(residul, self).__init__()
    self.layers = nn.ModuleList()

    for repeat in range(repeatnum):
      self.layers += [
        nn.Sequential(
          blockcnn(chnel, chnel//2, kernel=1),
```

```
            blockcnn(chnel//2, chnel, kernel=3, padding=1)
        )
    ]

    self.use_residual = use_residual
    self.repeatnum = repeatnum

  def send(self, x):
    for layer in self.layers:
      if self.use_residual:
        x = x + layer(x)
      else:
        x = layer(x)

    return x
class predictionscl(nn.Module):
  def __init__(self, inchnel, classno):
    super(predictionscl, self).__init__()
    self.pred = nn.Sequential(
        blockcnn(inchnel, 2 * inchnel, kernel=3, padding=1),
        blockcnn(2 * inchnel, (classno + 5) * 3, bnact=False, kernel=1), box which in total is 3
    )
    self.classno = classno

  def send(self, x):
    return (
        self.pred(x)
        .reshape(x.shape[0], 3, self.classno + 5, x.shape[2], x.shape[3])
        .permute(0, 1, 3, 4, 2)
```

```
        )
class yolov3(nn.Module):
  def __init__(self, inchnel=3, classno=20):
    super(yolov3, self).__init__()
    self.classno = classno
    self.inchnel = inchnel
    self.layers = self.convlayr()

  def send(self, x):
    outputs = []
    route = []

    for layer in self.layers:
      if instance(layer, predictionscl):
        outputs.append(layer(x))
        continue

      x = layer(x)

      if instance(layer, residul) and layer.repeatnum == 8:
        route.append(x)

      elif instance(layer, nn.Upsample):
        x = torch.cat([x, route[-1]], dim=1)
        route.pop()

    return outputs

  def convlayr(self):
```

```
layers = nn.ModuleList()
inchnel = self.inchnel

for module in config:
 if instance(module, tuple):
  outchnel, kernel, stride = module
  layers.append(blockcnn(
    inchnel,
    outchnel,
    kernel=kernel,
    stride=stride,
    padding=1 if kernel == 3 else 0
  ))
  inchnel = outchnel

 elif instance(module, list):
  repeatnum = module[1]
  layers.append(residul(inchnel, repeatnum=repeatnum))

 elif instance(module, str):
  if module == "sp":
   layers += [
    residul(inchnel, use_residual=False, repeatnum=1),
    blockcnn(inchnel, inchnel//2, kernel=1),
    predictionscl(inchnel//2, classno = self.classno)
   ]
   inchnel = inchnel // 2

  elif module == "us":
```

```
        layers.append(nn.Upsample(scale_factor=2))
        inchnel = inchnel * 3

    return layers

classno = 20
imag = 416
model = yolov3(classno=classno)
x = torch.randn((2, 3, imag, imag))
out = model(x)
assert model(x)[0].shape == (2, 3, imag//32, imag//32, classno + 5)
assert model(x)[1].shape == (2, 3, imag//16, imag//16, classno + 5)
assert model(x)[2].shape == (2, 3, imag//8, imag//8, classno + 5)
import cv2
import torch

deviceuse = "cuda" if torch.cuda.is_available() else "cpu"

worker = 4
batchsize= 32
imag = 416
classno = 20
learing = 1e-5
epochsno = 80
threshold = 0.8
ioumap = 0.5
iounms = 0.45
sp = [imag // 32, imag // 16, imag // 8]
```

```
IMG_DIR = "/kaggle/input/pascalvoc-yolo/images"
LABEL_DIR = "/kaggle/input/pascalvoc-yolo/labels"

ANCHOR = [
    [(0.28, 0.22), (0.38, 0.48), (0.9, 0.78)],
    [(0.07, 0.15), (0.15, 0.11), (0.14, 0.29)],
    [(0.02, 0.03), (0.04, 0.07), (0.08, 0.06)],
]

classpascal = [
    "aeroplane",
    "bicycle",
    "bird",
    "boat",
    "bottle",
    "bus",
    "car",
    "cat",
    "chair",
    "cow",
    "diningtable",
    "dog",
    "horse",
    "motorbike",
    "person",
    "pottedplant",
    "sheep",
    "sofa",
    "train",
```

```
    "tvmonitor"
]
def widthheight(box_A, box_B):

    intersection = torch.min(box_A[..., 0], box_B[..., 0]) * torch.min(
        box_A[..., 1], box_B[..., 1]
    )
    union = (
        box_A[..., 0] * box_A[..., 1] + box_B[..., 0] * box_B[..., 1] - intersection
    )
    return intersection / union
def interction_union(preds_box, label_box, formate_box="midpoint"):

    if formate_box == "midpoint":
        boa1_a1 = preds_box[..., 0:1] - preds_box[..., 2:3] / 2
        boa1_b1 = preds_box[..., 1:2] - preds_box[..., 3:4] / 2
        boa1_a2 = preds_box[..., 0:1] + preds_box[..., 2:3] / 2
        boa1_b2 = preds_box[..., 1:2] + preds_box[..., 3:4] / 2
        boa2_a1 = label_box[..., 0:1] - label_box[..., 2:3] / 2
        boa2_b1 = label_box[..., 1:2] - label_box[..., 3:4] / 2
        boa2_a2 = label_box[..., 0:1] + label_box[..., 2:3] / 2
        boa2_b2 = label_box[..., 1:2] + label_box[..., 3:4] / 2

    if formate_box == "corners":
        boa1_a1 = preds_box[..., 0:1]
        boa1_b1 = preds_box[..., 1:2]
        boa1_a2 = preds_box[..., 2:3]
        boa1_b2 = preds_box[..., 3:4]
        boa2_a1 = label_box[..., 0:1]
```

```
        boa2_b1 = label_box[..., 1:2]
        boa2_a2 = label_box[..., 2:3]
        boa2_b2 = label_box[..., 3:4]

    a1 = torch.max(boa1_a1, boa2_a1)
    b1 = torch.max(boa1_b1, boa2_b1)
    a2 = torch.min(boa1_a2, boa2_a2)
    b2 = torch.min(boa1_b2, boa2_b2)

    intersection = (a2 - a1).clamp(0) * (b2 - b1).clamp(0)
    boa1_area = abs((boa1_a2 - boa1_a1) * (boa1_b2 - boa1_b1))
    boa2_area = abs((boa2_a2 - boa2_a1) * (boa2_b2 - boa2_b1))

    return intersection / (boa1_area + boa2_area - intersection + 1e-6)
def non_suppression(boxx, iou_threshold, threshold, formate_box="corners"):

    assert type(boxx) == list

    boxx = [box for box in boxx if box[1] > threshold]
    boxx = sorted(boxx, key=lambda x: x[1], reverse=True)
    boxx_after_nms = []

    while boxx:
        chosen_box = boxx.pop(0)

        boxx = [
            box
            for box in boxx
            if box[0] != chosen_box[0]
```

```
            or interction_union(
                torch.tensor(chosen_box[2:]),
                torch.tensor(box[2:]),
                formate_box=formate_box,
            )
            < iou_threshold
        ]

        boxx_after_nms.append(chosen_box)

    return boxx_after_nms
import numpy as np
import os
import pandas as pd
import torch

from torch.utils.data import Dataset, DataLoader
from PIL import Image, ImageFile

ImageFile.LOAD_TRUNCATED_IMAGES = True
class yolo_dataset(Dataset):
  def __init__(self, csv_file, img_dir, label_dir, anchor,
               imag=416, sp=[13,26,52], c=20, transform=None):
    self.annotations = pd.read_csv(csv_file)
    self.img_dir = img_dir
    self.label_dir = label_dir
    self.transform = transform
    self.sp = sp
```

```
    self.anchor = torch.tensor(anchor[0] + anchor[1] + anchor[2]) # For all 3 scales
    self.num_anchor = self.anchor.shape[0]
    self.num_anchor_per_scale = self.num_anchor // 3

    self.cp = cp

    self.ignore_iou_thresh = 0.5

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

  def __getitem__(self, index):
    label_path = os.path.join(self.label_dir, self.annotations.iloc[index, 1])
    boxx = np.roll(np.loadtxt(fname=label_path, delimiter=" ", ndmin=2), 4, axis=1).tolist() # np.roll with
shift 4 on axis 1: [class, x, y, w, h] --> [x, y, w, h, class]

    img_path = os.path.join(self.img_dir, self.annotations.iloc[index, 0])
    image = Image.open(img_path)

    if self.transform:
      image = self.transform(image)

    targets = [torch.zeros((self.num_anchor // 3, sp, sp, 6)) for sp in self.sp] # 6 because objectness score,
bounding box coordinates (x, y, w, h), class label

    for box in boxx:
      iou_anchor = widthheight(torch.tensor(box[2:4]), self.anchor) # IOU from height and width
      anchor_indices = iou_anchor.argsort(descending=True, dim=0) # sporting sucht that the first is the
best anchor
```

```
    x, y, width, height, class_label = box
    has_anchor = [False, False, False] # Make sure there is an anchor for each of three scales for each bounding box

    for anchor_idx in anchor_indices:
     scale_idx = anchor_idx // self.num_anchor_per_scale # scale_idx is either 0,1,2: 0-->13a13, 1:-->26a26, 2:-->52x52
     anchor_on_scale = anchor_idx % self.num_anchor_per_scale # In each scale, choosing the anchor thats either 0,1,2

     sp = self.sp[scale_idx]
     i, j = int(sp*y), int(sp*x)
     tokenanker = targets[scale_idx][anchor_on_scale, i, j, 0]

     if not tokenanker and not has_anchor[scale_idx]:
      targets[scale_idx][anchor_on_scale, i, j, 0] = 1
      x_cell, y_cell = sp*x - j, sp*y - i # 6.5 - 6 = 0.5 such that they are between [0,1]
      width_cell, height_cell = (
        width*sp, # sp=13, width=0.5, 6.5
        height*sp
      )

      box_coordinates = torch.tensor([x_cell, y_cell, width_cell, height_cell])

      targets[scale_idx][anchor_on_scale, i, j, 1:5] = box_coordinates
      targets[scale_idx][anchor_on_scale, i, j, 5] = int(class_label)
      has_anchor[scale_idx] = True

     # Even if the same grid shares another anchor having iou>ignore_iou_thresh then,
     elif not tokenanker and iou_anchor[anchor_idx] > self.ignore_iou_thresh:
```

```
            targets[scale_idx][anchor_on_scale, i, j, 0] = -1 # ignore this prediction

    return image, tuple(targets)

import torchvision.transforms as transforms
transform = transforms.Compose([transforms.Resize((416, 416)), transforms.ToTensor()])
def get_loaders(train_csv_path, test_csv_path):

    train_dataset = yolo_dataset(
        train_csv_path,
        transform=transform,
        sp=[imag // 32, imag // 16, imag // 8],
        img_dir=IMG_DIR,
        label_dir=LABEL_DIR,
        anchor=ANCHOR,
    )
    test_dataset = yolo_dataset(
        test_csv_path,
        transform=transform,
        sp=[imag // 32, imag // 16, imag // 8],
        img_dir=IMG_DIR,
        label_dir=LABEL_DIR,
        anchor=ANCHOR,
    )
    train_loader = DataLoader(
        dataset=train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        drop_last=False,
```

```
    )
    test_loader = DataLoader(
        dataset=test_dataset,
        batch_size=BATCH_SIZE,
        shuffle=False,
        drop_last=False,
    )

    return train_loader, test_loader
def mean_average_precision(
    pred_boxes, true_boxes, iou_threshold=0.5, formate_box="midpoint", classno=4
):
    # list storing all AP for respective classes
    average_precisions = []

    # used for numerical stability later on
    epsilon = 1e-6

    for c in range(classno):
        detections = []
        ground_truths = []

        for detection in pred_boxes:
            if detection[1] == c:
                detections.append(detection)

        for true_box in true_boxes:
            if true_box[1] == c:
                ground_truths.append(true_box)
```

```
amount_boxx = Counter([gt[0] for gt in ground_truths])

for key, val in amount_boxx.items():
    amount_boxx[key] = torch.zeros(val)

# sort by box probabilities which is index 2
detections.sort(key=lambda x: x[2], reverse=True)
TP = torch.zeros((len(detections)))
FP = torch.zeros((len(detections)))
total_true_boxx = len(ground_truths)

# If none exists for this class then we can safely skip
if total_true_boxx == 0:
    continue

for detection_idx, detection in enumerate(detections):
    # Only take out the ground_truths that have the same
    # training idx as detection
    ground_truth_img = [
        bbox for bbox in ground_truths if bbox[0] == detection[0]
    ]

    num_gts = len(ground_truth_img)
    best_iou = 0

    for idx, gt in enumerate(ground_truth_img):
        iou = interction_union(
            torch.tensor(detection[3:]),
```

```
            torch.tensor(gt[3:]),
            formate_box=formate_box,
        )

        if iou > best_iou:
            best_iou = iou
            best_gt_idx = idx

    if best_iou > iou_threshold:
        # only detect ground truth detection once
        if amount_boxx[detection[0]][best_gt_idx] == 0:
            # true positive and add this bounding box to seen
            TP[detection_idx] = 1
            amount_boxx[detection[0]][best_gt_idx] = 1
        else:
            FP[detection_idx] = 1

    # if IOU is lower then the detection is a false positive
    else:
        FP[detection_idx] = 1

TP_cumsum = torch.cumsum(TP, dim=0)
FP_cumsum = torch.cumsum(FP, dim=0)
recalls = TP_cumsum / (total_true_boxx + epsilon)
precisions = TP_cumsum / (TP_cumsum + FP_cumsum + epsilon)
precisions = torch.cat((torch.tensor([1]), precisions))
recalls = torch.cat((torch.tensor([0]), recalls))
# torch.trapz for numerical integration
average_precisions.append(torch.trapz(precisions, recalls))
```

```
    return sum(average_precisions) / len(average_precisions)
def get_evaluation_boxx(
    loader,
    model,
    iou_threshold,
    anchor,
    threshold,
    formate_box="midpoint",
    device="cuda" if torch.cuda.is_available() else "cpu",
):
    # make sure model is in eval before get boxx
    model.eval()
    train_idx = 0
    all_pred_boxes = []
    all_true_boxes = []
    for batch_idx, (x, labels) in enumerate(loader):
        x = x.float().to(device)

        with torch.no_grad():
            predictions = model(x)

        batch_size = x.shape[0]
        boxx = [[] for _ in range(batch_size)]
        for i in range(3):
            sp = predictions[i].shape[2] # grid cell size for each predictions
            anchor = torch.tensor([*anchor[i]]).to(device) * sp # anchor for each grid, prediction type
            i_scale = boxx_cell( # get boxx for each image in the batch
                predictions[i], anchor, sp=sp, is_preds=True
```

```
            )
            for idx, (box) in enumerate(i_scale): # for each image, append the bbox to corr. boxx[idx]
                boxx[idx] += box

        # we just want one bbox for each label, not one for each scale
        true_boxx = boxx_cell(
            labels[2], anchor, sp=sp, is_preds=False
        )

        for idx in range(batch_size):
            nms_boxes = non_suppression(
                boxx[idx],
                iou_threshold=iou_threshold,
                threshold=threshold,
                formate_box=formate_box,
            )

            for nms_box in nms_boxes:
                all_pred_boxes.append([train_idx] + nms_box)

            for box in true_boxx[idx]:
                if box[1] > threshold:
                    all_true_boxes.append([train_idx] + box)

            train_idx += 1

    model.train()
    return all_pred_boxes, all_true_boxes
def boxx_cell(predictions, anchor, sp, is_preds=True):
```

```
    batchsize= predictions.shape[0]
    num_anchor = len(anchor)
    box_predictions = predictions[..., 1:5]
    if is_preds:
        anchor = anchor.reshape(1, len(anchor), 1, 1, 2)
        box_predictions[..., 0:2] = torch.sigmoid(box_predictions[..., 0:2])
        box_predictions[..., 2:] = torch.exp(box_predictions[..., 2:]) * anchor
        scores = torch.sigmoid(predictions[..., 0:1])
        best_class = torch.argmax(predictions[..., 5:], dim=-1).unsqueeze(-1)
    else:
        scores = predictions[..., 0:1]
        best_class = predictions[..., 5:6]


    cell_indices = (
        torch.arange(sp)
        .repeat(predictions.shape[0], 3, sp, 1)
        .unsqueeze(-1)
        .to(predictions.device)
    )
    x = 1 / sp * (box_predictions[..., 0:1] + cell_indices)
    y = 1 / sp * (box_predictions[..., 1:2] + cell_indices.permute(0, 1, 3, 2, 4))
    w_h = 1 / sp * box_predictions[..., 2:4]
    converted_boxx = torch.cat((best_class, scores, x, y, w_h), dim=-1).reshape(BATCH_SIZE, num_anchor
* sp * sp, 6)
    return converted_boxx.tolist()
class YoloLoss(nn.Module):
  def __init__(self):
    super(YoloLoss, self).__init__()
```

```
    self.mse = nn.MSELoss() # For bounding box loss
    self.bce = nn.BCEWithLogitsLoss() # For multi-label prediction: Binary cross entropy
    self.entropy = nn.CrossEntropyLoss() # For classification
    self.sigmoid = nn.Sigmoid()

    # Constants for significance of obj, or no obj.
    self.lambda_class = 1
    self.lambda_noobj = 10
    self.lambda_obj = 1
    self.lambda_box = 10

  def send(self, predictions, target, anchor):
    obj = target[..., 0] == 1
    noobj = target[..., 0] == 0

    no_object_loss = self.bce(
        (predictions[..., 0:1][noobj]), (target[..., 0:1][noobj])
    )

    anchor = anchor.reshape(1,3,1,1,2)

    box_preds = torch.cat([self.sigmoid(predictions[..., 1:3]), torch.exp(predictions[..., 3:5]) * anchor],
dim=-1)

    ious = interction_union(box_preds[obj], target[..., 1:5][obj]).detach()

    object_loss = self.bce(
        (predictions[..., 0:1][obj]), (ious * target[..., 0:1][obj])
    )
```

```
        predictions[..., 1:3] = self.sigmoid(predictions[..., 1:3])
        target[..., 3:5] = torch.log(
            (1e-6 + target[..., 3:5] / anchor)
        )

        box_loss = self.mse(predictions[..., 1:5][obj], target[..., 1:5][obj])

        class_loss = self.entropy(
            (predictions[..., 5:][obj]), (target[..., 5][obj].long())
        )

        return(
            self.lambda_box * box_loss
            + self.lambda_obj * object_loss
            + self.lambda_noobj * no_object_loss
            + self.lambda_class * class_loss
        )

def plot_image(image, boxes):

    cmap = plt.get_cmap("tab20b")
    class_labels = classpascal
    colors = [cmap(i) for i in np.linspace(0, 1, len(class_labels))]
    im = np.array(image)
    height, width, _ = im.shape

    # Create figure and axes
    fig, ax = plt.subplots(1)
```

```
# Display the image
ax.imshow(im)


# Create a Rectangle patch
for box in boxes:
    assert len(box) == 6,
    class_pred = box[0]
    box = box[2:]
    upper_left_x = box[0] - box[2] / 2
    upper_left_y = box[1] - box[3] / 2
    rect = patches.Rectangle(
        (upper_left_x * width, upper_left_y * height),
        box[2] * width,
        box[3] * height,
        linewidth=2,
        edgecolor=colors[int(class_pred)],
        facecolor="none",
    )
    # Add the patch to the Axes
    ax.add_patch(rect)
    plt.text(
        upper_left_x * width,
        upper_left_y * height,
        s=class_labels[int(class_pred)],
        color="white",
        verticalalignment="top",
        bbox={"color": colors[int(class_pred)], "pad": 0},
    )
```

```
    plt.show()
# Instantiate the model
model = yolov3(classno=classno).to(deviceuse)

# Compile the model
optimizer = torch.optim.Adam(
    model.parameters(), lr=learing
)
loss_fn = YoloLoss()

# Scaler
scaler = torch.cuda.amp.GradScaler()

# Train-Test Loader
train_loader, test_loader = get_loaders(
    train_csv_path='/kaggle/input/pascalvoc-yolo/test.csv', test_csv_path='/kaggle/input/pascalvoc-
yolo/test.csv'
)

# anchor
scaled_anchor = (
    torch.tensor(ANCHOR) * torch.tensor([13,26,52]).unsqueeze(1).unsqueeze(1).repeat(1,3,2)
).to(deviceuse)
# Save test loader to a file
torch.save(test_loader, '/kaggle/working/test_loader.pth')

import torch.optim as optim

from tqdm import tqdm
```

```
import time

history_loss = [] # To plot the epoch vs. loss

for epoch in tqdm(range(epochsno), desc="Epochs"):
 model.train()

 losses = []

 start_time = time.time() # Start time of the epoch

 for batch_idx, (x,y) in enumerate(train_loader):
  x = x.to(deviceuse)
  y0, y1, y2 = (y[0].to(deviceuse),
          y[1].to(deviceuse),
          y[2].to(deviceuse))

  # context manager is used in PyTorch to automatically handle mixed-precision computations on CUDA-enabled GPUs
  with torch.cuda.amp.autocast():
   out = model(x)
   loss = (
     loss_fn(out[0], y0, scaled_anchors[0])
     + loss_fn(out[1], y1, scaled_anchors[1])
     + loss_fn(out[2], y2, scaled_anchors[2])
   )

  losses.append(loss.item())
```

```
    optimizer.zero_grad()
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()

  end_time = time.time()  # End time of the epoch
  epoch_duration = end_time - start_time  # Duration of the epoch

  history_loss.append(sum(losses)/len(losses))

  if (epoch+1) % 10 == 0:
    # Print the epoch duration
    tqdm.write(f"Epoch {epoch+1} completed in {epoch_duration:.2f} seconds")

    # Print the loss and accuracy for training and validation data
    print(f"Epoch [{epoch+1}/{epochsno}], "
          f"Loss: {sum(losses)/len(losses):.4f}")

    # save the model after every 10 epoch
    torch.save(model.state_dict(), f'/kaggle/working/Yolov3_epoch{epoch+1}.pth')

model.eval()
x, y = next(iter(test_loader))
x = x.float().to(deviceuse)

with torch.no_grad():
    out = model(x)
    boxx = [[] for _ in range(x.shape[0])]
    batch_size, A, sp, _, _ = out[0].shape
```

```
anchor = torch.tensor([*ANCHOR[0]]).to(deviceuse) * sp
i_scale = boxx_cell(
    out[0], anchor, sp=sp, is_preds=True
)
for idx, (box) in enumerate(i_scale):
    boxx[idx] += box


for i in range(batch_size):
    nms_boxes = non_suppression(
        boxx[i], iou_threshold=0.5, threshold=0.6, formate_box="midpoint",
    )
    plot_image(x[i].permute(1,2,0).detach().cpu(), nms_boxes)
```

import matplotlib 1.pdf

ORIGINALITY REPORT

41%	40%	7%	11%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	huggingface.co Internet Source	31%
2	discuss.pytorch.org Internet Source	4%
3	origin.geeksforgeeks.org Internet Source	3%
4	blog.csdn.net Internet Source	1%
5	Submitted to University of Sheffield Student Paper	1%
6	Submitted to Bilkent University Student Paper	1%
7	chowdera.com Internet Source	1%
8	Submitted to unibuc Student Paper	<1%
9	Submitted to City University of Hong Kong Student Paper	<1%

10	Submitted to University College London Student Paper	<1%
11	noisrucer.github.io Internet Source	<1%
12	Eklas Hossain. "Machine Learning Crash Course for Engineers", Springer Science and Business Media LLC, 2024 Publication	<1%

Exclude quotes	On	Exclude matches	< 5 words
Exclude bibliography	On		

import matplotlib 1.pdf