Assignment 4
Computer Vision and Deep Learning
Mamoona Birkhez Shami
mamoona.b.shami@ntnu.no

Delivery Deadline:
Friday, March 18th, 23:59PM.
This assignment counts 4% of your final grade.
You are allowed to work in groups of 2 persons.

Introduction. In this assignment, we will take a look at object detection with convolutional neural networks. You will get familiar with common metrics used to evaluate object detection models. Then, we will take a deep dive into the SSD300 architecture, a lightweight real-time object detection model.

Compute resources: For task 4, utilizing our GPU resources will save you a significant amount of time (see assignment 3 for intro).

Starter Code. We provide you starter code for the programming tasks. You are required to use the provided files, but you are allowed to create any additional files for each of the subtasks. You can download the starter code from: https://github.com/TDT4265-tutorial/TDT4265_StarterCode.

Report outline. We've included a jupyter notebook as a skeleton for your report, such that you won't use too much time on creating your report. Remember to export the jupyter notebook to PDF before submitting it to blackboard. You're not required to use this report skeleton, and you can write your report in whatever program you'd like (markdown, latex, word etc), as long as you deliver the report as a PDF file.

Recommended reading.

1. Section 4.2 of The PASCAL Visual Object Classes (VOC) Challenge. This is the official description of how to calculate the mean average precision for object detection tasks.
2. Jonathan Hui's blog post about mean average precision. This blog post is very short, but gives a very simple explanation of mean average precision.
3. SSD Blog Post by Jonathan Hui: We recommend everyone to read this before starting on the SSD tasks.
4. SSD: Single Shot MultiBox Detector. We recommend you to read the blog post before reading the paper.
5. PyTorch Tutorial for SSD. (This implementation is a little different from our starter code).

Delivery We ask you to follow these guidelines:

- **Report:** Deliver your answers as a **single PDF file**. Include all tasks in the report, and mark it clearly with the task you are answering (Task 1.a, Task1.b, Task 2.c etc). There is no need to include your code in the report.
- **Plots in report:** For the plots in the report, ensure that they are large and easily readable. You might want to use the "ylim" function in the matplotlib package to "zoom" in on your plots. Label the different graphs such that it is easy for us to see which graphs correspond to the train, validation and test set.
- **Source code:** Upload your code as a zip file. In the assignment starter code, we have included a script (`create_submission_zip.py`) to create your delivery zip. **Please use this**, as this will structure the zipfile as we expect. (Run this from the same folder as all the python files).

  To use the script, simply run: `python3 create_submission_zip.py`
- **Upload to blackboard:** Upload the ZIP file with your source code and the report to blackboard before the delivery deadline.

- The delivered code is taken into account with the evaluation. Ensure your code is well documented and as readable as possible.

Any group who does not follow these guidelines or delivers late will be subtracted in points.

Task 1. Object Detection Metrics

For evaluating image classification methods, accuracy is an easy and good metric. For object detection, measuring performance is a more complicated task. First, we need to ensure that we localize the object correctly with a good bounding box. Once we have detected the object, we want to make sure that the detected object is classified correctly. When the PASCAL VOC dataset was proposed, they introduced a standard metric to measure the performance of object detection architectures. This metric was *mean average precision (mAP)*.

Take a look at the recommended resources, and **answer the following in your report:**

(a) [0.05*pts*] Explain what the Intersection over Union is and how we can find it for two bounding boxes. Illustrate it with a drawing.

(b) [0.05*pts*] Write down the equation of precision and recall, and shortly explain what a true positive and false positive is.

(c) [0.15*pts*] Given the following precision and recall curve for the two classes, what is the mean average precision?

Precision and recall curve for class 1:
$\text{Precision}_1 = [1.0, 1.0, 1.0, 0.5, 0.20]$
$\text{Recall}_1 = [0.05, 0.1, 0.4, 0.7, 1.0]$

Precision and recall curve for class 2:
$\text{Precision}_2 = [1.0, 0.80, 0.60, 0.5, 0.20]$
$\text{Recall}_2 = [0.3, 0.4, 0.5, 0.7, 1.0]$

Hint: To calculate this, find the precision for the following recall levels: 0.0, 0.1, 0.2, ... 0.9, 1.0.

Task 2. Implementing Mean Average Precision

Now, we will implement mean average precision for a *single class* [1]. In the assignment files, there is a .json file for both a set of predicted bounding boxes and a set of ground truth bounding boxes. We have provided you a set of tests to simplify the debugging steps for this task. For each subtask you will finish, you can run tests.py and it will check if the function is correct. These tests are written with simple checks and we can't guarantee that they cover every edge case.

Implement the following:

(a) [0.15*pts*] Implement a function that takes two bounding boxes, then finds the intersection over union. This is the function `calculate_iou` in task2.py

(b) [0.15*pts*] Implement a function that computes the precision given the number of true positives, false positives, and false negatives. This is the function `calculate_precision` in task2.py.

Implement a function that computes the recall given the number of true positives, false positives, and false negatives. This is the function `calculate_recall` in task2.py

[1] We simplify the task to handle a single class, as this reduces the amount of code significantly. If you want to extend this code for a multi-class object detection algorithm, you find the average precision for each class and take the mean over all of these to get the mean average precision.

(c) [0.3*pts*] Implement a function that takes in a set of predicted bounding boxes, a set of ground truth bounding boxes, and a given intersection of union threshold, then finds the optimal match for each bounding box. This is the function `get_all_box_matches` in task2.py.

Note, each box can only have a single match and it should be matched with the bounding box with the highest intersection of union ratio. If there is no match that has a higher IoU than the IoU threshold then the bounding box has no match.

(d) [0.25*pts*] Implement a function that calculates the true positives, false positives, and false negatives for a single image. Furthermore, implement a function that finds the precision and recall for all images in a dataset. Finally, implement a function that computes the precision recall curve. This are the functions `calculate_individual_image_result`, `calculate_precision_recall_all_images`, and `get_precision_recall_curve` in task2.py.

(e) [0.3*pts*] Implement a function that computes the mean average precision given a list of precisions and recalls. This is the function `calculate_mean_average_precision` in task2.py

Note, do not change the recall levels used to compute the mAP. This is what is used in standard object detection benchmarks.

If you run task2.py as a python file, the printed mean average precision over the dataset should be 0.9066 if you use a IoU threshold of 0.5.

(f) [0.05*pts*] **In report:** Put the plot of your final precision-recall curve in your report.

If you run task2.py, it will save the precision-recall curve to precision_recall_curve.png

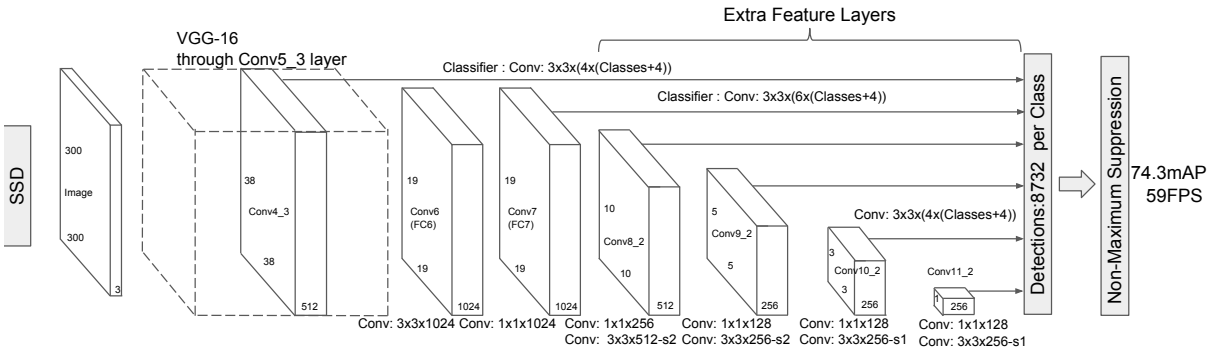

Figure 1: The SSD architecture presented in the original paper.

Task 3. Theory

Take a look at the recommended resources, and **answer the following in your report:**

(a) [0.1*pts*] The SSD architecture produces a fixed-size number of bounding boxes and a score for each bounding box. When performing inference with SSD, we need to filter out a set of overlapping boxes. What is this filtering operation called?

(b) [0.1*pts*] The SSD architecture predicts bounding boxes at multiple scales to enable the network to detect objects of different sizes.

Is the following true or false: Predictions from the deeper layers in SSD are responsible to detect small objects.

(c) [0.15*pts*] SSD use k number of "anchors" [2] with different aspect ratios at each spatial location in a feature map to predict c class scores and 4 offsets relative to the original box shape.

Why do they use different bounding box aspect ratios at the same spatial location?

(d) [0.1*pts*] What is the main difference between SSD and YOLOv1/v2 (The YOLO version they refer to in the SSD paper)?

(e) [0.15*pts*] Given a SSD framework, where the first scale the network predicts at is at the last feature map with a resolution of 38×38 ($H \times W$). For each anchor location, we place 6 different anchors with different aspect ratios. How many anchors boxes do we have in total for this feature map?

(f) [0.15*pts*] The network outlined in the previous subtask predicts at multiple resolutions, specifically 38×38, 19×19, 10×10, 5×5, 3×3 and 1×1. It uses 6 different aspect ratios at each location in every feature map as anchors. How many anchors boxes do we have in total for the entire network?

Task 4. Implementing Single Shot Detector

For this task, we provide a large code base with an SSD implementation. Most of the code is finished, except the backbone network. Your task is to implement the backbone which should output the **six** different feature maps that SSD use to predict bounding boxes (see Figure 1). Table 1 shows the backbone network you will implement.

This code base is significantly larger then what we've given you before. To get you started, take a look in the readme.md file in the repository: Note that for this task we will only use a train and validation set (no test set).

[2] An anchor (often called a prior) is a bounding box placed at fixed locations in the image. SSD outputs a class score and an offset to shift the anchor to predict if there is an object close to this anchor, which gives us the predicted bounding box.

(a) [0.6*pts*] Implement the CNN in Table 1. The only code required to change is located in `ssd/modeling/backbones/basic.py`.

(b) [0.3*pts*] Train your model on the MNIST Object detection dataset. For information about the dataset, see: https://github.com/hukkelas/MNIST-ObjectDetection.

You should expect a mean average precision (mAP) of about 75-77% if you train for 6000 iterations. **(report)** Include a plot of "total_loss" (You can take a screenshot from the tensorboard, or use the provided jupyter notebook).

(report) Finally, report your mean average precision (mAP).

(c) [0.6*pts*] Use what you learned from assignment 3 to improve your model. Reach a mean average precision of 85% within 10K gradient descent iterations (or get as close as possible - we will give partial points).

(report) Describe what improvements you did to reach the accuracy and report the final mAP.

Some tips for improving your model:

- Look back at assignment 3 and think about what improved your model. These additions will probably improve this model as well!
- You can change other parts of the code as well (for example, augmentation, optimizers, learning rate, etc.)
- Be careful to not add too much data augmentation. Our reference implementation used no data augmentation. (Some might help though!)
- The model in Table 1 has somewhere in the neighbourhood of 2M parameters. The original backbone (VGG) had 130M. Maybe adding more layers or filters per layer might work?

(d) [0.15*pts*] Given a 300x300 image, the feature map with sxs resolution divides the image into sxs 'tiles' and places anchor boxes in the center of those tiles. The calculation for anchor boxes in the starter code is slightly different from the original paper.

- Calculate the pixel values of the 25 center points for the anchor boxes for the feature map with resolution 5x5 using strides(L26) from `configs/ssd300.py` (Strides is the number of pixels (in image space) between each spatial position in the feature map or the size of the 'tile').
- Number of anchor boxes calculated is 2+2 per aspect ratio. Size for anchor boxes is calculated as follows:
  - square of side min_size
  - square of side $\sqrt{min\_size * next\_min\_size}$
  - $[min\_size * \sqrt{aspect\_ratio}, min\_size / \sqrt{aspect\_ratio}]$
  - $[min\_size / \sqrt{aspect\_ratio}, min\_size * \sqrt{aspect\_ratio}]$

  Calculate the sizes of the anchor boxes using the aspect ratios and min_sizes. **Note:** Remember the size cannot be bigger than image size.
- Then visualize the priors for this layer using script provided for you in `notebooks/visualize_priors`. See if the centers and the sizes of the anchor boxes match your calculation.

(report) Write pixel values for 25 center points and shapes of 6 anchor boxes in report and show the plot for visualization.

(e) [0.05*pts*] Try out your model on a couple of images. **(report)** Run demo.py on the images in the `demo/mnist` folder and include the classified images in your report.

Do this by typing the following:
`python demo.py configs/ssd300.py demo/mnist demo/mnist_output`

(report) Were there any digits your model was not able to detect?

(f) [0.15*pts*] This last task is to prepare you for the project and test a model on the PASCAL VOC dataset. VOC consists of 21 different classes and we will train a SSD network with VGG16 as the backbone. This backbone is already pre-trained on the ImageNet dataset (classification).

We've included documentation on how to download the VOC dataset in the readme.md file in the starter code.

Use the config file "voc_vgg.py" to train a VGG16 model on the VOC dataset. Train the model for 5000 gradient descent iterations (the same as "iter" that prints in the starter code).

Test your model on a couple of images, by running the demo.py file:
`python demo.py configs/voc_vgg.py demo/voc demo/voc_output`

(report) In your report, include the plot of "total_loss", your final mAP on the validation set and the images you tested with demo.py.

Note that no modification to the code should be required to finish this task.

Table 1: Illustration of a custom SSD CNN backbone. The number of filters specifies the number of filters/kernels in a convolutional layer. Each convolutional layer has a filter size of 3×3 and padding of 1. Each MaxPool2D layer has a stride of 2 and a kernel size of 2×2 Note that output_channels[i] is a variable that you can set in the config file (By default this is: [128, 256, 128, 128, 64, 64]). * **The last convolution** should have a stride of 1, padding of 0, and a kernel size of 3×3

Is Output	Layer Type	Number of Filters	Stride
	Conv2D	32	1
	ReLU	–	–
	MaxPool2D	–	2
	Conv2D	64	1
	ReLU	–	–
	MaxPool2D	–	2
	Conv2D	64	1
	ReLU	–	–
	Conv2D	output_channels[0]	2
Yes - Resolution: 38×38	ReLU	–	–
	ReLU	–	–
	Conv2D	128	1
	ReLU	–	–
	Conv2D	output_channels[1]	2
Yes - Resolution: 19×19	ReLU	–	–
	ReLU	–	–
	Conv2D	256	1
	ReLU	–	–
	Conv2D	output_channels[2]	2
Yes - Resolution: 10×10	ReLU	–	–
	ReLU	–	–
	Conv2D	128	1
	ReLU	–	–
	Conv2D	output_channels[3]	2
Yes - Resolution: 5×5	ReLU	–	–
	ReLU	–	–
	Conv2D	128	1
	ReLU	–	–
	Conv2D	output_channels[4]	2
Yes - Resolution: 3×3	ReLU	–	–
	ReLU	–	–
	Conv2D	128	1
	ReLU	–	–
	Conv2D	output_channels[5]	1
Yes - Resolution: 1×1 *	ReLU	–	–