Airbus Ship Detection

The 2024/25/1 semester assignment for the VITMAV45 course

Author names:
Botond Horváth (TRYLVW)
Martin Jakab (FX6A7J)
Tamás Fébert (UQ8MSF)

Introduction

We have chosen the Airbus Ship Detection challenge as our homework, because we wanted to try an image segmentation task. This challenge is a quite popular kaggle task that earned the attention of many data scientists around the world. It was a really versatile challenge, because there were not so many tasks with the goal of detecting ships with rotated bounding boxes, which was a merely discussed topic at the time. Therefore the kaggle team encoded the assignment to a mask detection challenge, so we also built our model that way. In this competition the assignment is given as satellite images and encoded ship locations in a csv file. The main goal is to locate ships in images, and put an aligned bounding box segment around the ships you locate. Many images do not contain ships, and those that do may contain multiple ships. Ships within and across images may differ in size (sometimes significantly) and be located in open sea, at docks, marinas, etc. These images can hinder our model in the training, because the waves or clouds can look like small ships on a satellite image. In this paper we try to present our approach to solve the problem, and want to discuss our experiences and difficulties in our way to the solutions. Finally, we want to point out possibilities to improve our model furthermore, based on our experiences.

Short summary of our achievements

We trained our model on balanced training data with augmentations using a U-net architectured network. The training parameters were the result of a hyperparameter optimization, where Intersection over Union (IoU) score was the goal. The best result we got is a 0.6809 value in the IoU_Metric. The GPU we used was the colab L4 GPU, which had both enough RAM and GPU capacity for bigger batch sizes.

Solution

Milestone1 - Preparing and loading data

The first steps (preparing and loading the data) were implemented - and can be found in the AirbusShipdetection.ipynb notebook.
The initial step in the Airbus Ship Detection project involves authenticating access to Kaggle's datasets and preparing the required files for further analysis. This can be done by uploading the kaggle.json file, which authenticates us and then we can download the dataset. The dataset is quite large and it takes approximately 30GB of disc space. After

downloading it to the /content folder in the session we decompressed it to access the content. After downloading and decompressing, we checked whether all necessary files are present with a ls command. To test images too, we created a show_images function to plot 3 images each from the train and test directories. We counted the rows of the provided csv file, to estimate the number of images we have to deal with - and found more than 200000 images. This step had another purpose too - to distinguish between images that contain ships and those that do not. In the notebook we even show some of these csv data using the head function.
Here we decided to split it to train, test and validation data (80% train, 10% test, 10% split), but later - in the model training phase - we changed the values of the splitting.
In the previous step we noticed that the locations of the ships are coded in a strange format, so we had to take some time to figure out what it is and what it means. After confirming the sizes of the input images, we converted the given, Run-length-encoding formatted data into ship masks with a function - searched for an image, which contained a ship and then plotted the result.
At this point, we were planning to use PyTorch and its embedded methods to complete this task, so we have created a custom dataset class, called “AirbusShipDataset”. Custom datasets work seamlessly with PyTorch’s data pipeline, enabling batching, shuffling, and augmentation.

Milestone2 - Training the model

The training of the model and its preparation is located in the Keras_solution.ipynb notebook.

Switching to tensorflow.keras

Instead of using PyTorch functions and dataset, we have decided to switch to tensorflow.keras.
Keras offers an extensive library of pre-implemented functionalities tailored to segmentation tasks, such as the ability to use specific loss functions like binary_crossentropy with smooth labeling and metrics like Intersection over Union (IoU) or Dice Coefficient. These metrics are either not directly available in PyTorch or require significant custom implementation. With Keras, these features are available out-of-the-box, allowing us to focus more on model design and data experimentation.
Furthermore, the high-level API of Keras enables faster prototyping of architectures such as U-Net or other encoder-decoder models, which are crucial for pixel-wise segmentation in this challenge. Keras also integrates seamlessly with TensorFlow, which provides an efficient backend for GPU acceleration and enhanced dataset handling using tf.data pipelines. These pipelines make it easier to manage large image datasets and optimize training processes.
So switching to Keras aligned better with our longer-term goals.

Directories and mask variables

In this notebook, we have collected most of the necessary libraries at the second cell, to make it more structured.
Similarly to how we have done the downloading of the data in the preparation notebook, we included those steps in the model training notebook too. The training and testing image

containing directories got their own variable and we sorted the images in the training folder to look for duplicates.
This was the step, where we found out that if an image contains multiple ships - its ID appears more than one time in the csv file.

	ImageId	EncodedPixels
0	00003e153.jpg	NaN
1	0001124c7.jpg	NaN
2	000155de5.jpg	264661 17 265429 33 266197 33 266965 33 267733...
3	000194a2d.jpg	360486 1 361252 4 362019 5 362785 8 363552 10 ...
4	000194a2d.jpg	51834 9 52602 9 53370 9 54138 9 54906 9 55674 ...
5	000194a2d.jpg	198320 10 199088 10 199856 10 200624 10 201392...
6	000194a2d.jpg	55683 1 56451 1 57219 1 57987 1 58755 1 59523 ...

We converted the run length encoding formats into numpy arrays, where number 0 stood for the background pixels and number 1 for the pixels that are part of a ship.
We have separated the unique image indexes and then split the images into train and validation parts (75% - 25%), because testing images were already part of the project. Then created training and validation data frames by merging the masks with the split subsets of trainIndexes and validationIndexes. Balancing the data in our binary segmentation task seemed important because it directly affects the model's ability to learn and generalize effectively. Images may either contain multiple ships, a single ship, or no ships at all. If the dataset is heavily imbalanced, such as having a majority of images with only water and very few with ships, the model is likely to become biased toward predicting the majority class. For example, the model might overly predict "no ships" because it sees this outcome far more frequently during training, leading to poor performance on images that actually contain ships. To avoid this - we grouped and counted images based on the number of ships they contained.

shipCountsInGroups	count
1	1200
2	1200
3	1200
4	1200
6	1200
5	1200
7	1200
0	400

dtype: int64

The ships that contain more than seven ships, were grouped with the ones that have seven. (Because there were few of them and this looks better visually.)
Also, the dataset is really large and most of the images do not contain ships at all - so we have created an equalized dataset from it, where each category is represented by 1200 images, except the no-ship images - there we have kept only 400. This way the model can learn more efficiently the constellations of ships.
We decided to use data augmentation in our project, because it helps the model generalize and guess better on unseen data. By applying transformations such as rotations, flips, scaling, cropping, or changes in brightness and contrast, data augmentation creates new training samples that are variations of the original images.
Before augmenting the equalized data, we made a generator function, which loads images and masks in pairs(in batches). Then we checked whether it is working on a single batch. The next step was defining the augmentation parameters.
The ImageDataGenerator class is compatible with keras and has many modifying options, so we chose that.

```python
# Preparing image data generator arguments
generatorParameters = dict(
    rotation_range=10,        # The maximum range (in degrees) to randomly rotate the image during augmentation. A value of 10 means the image can be rotated up to +/-10 degrees.
    width_shift_range=0.05,  # Fraction of the total image width to randomly shift the image horizontally. 0.05 means up to 5% of the image width.
    height_shift_range=0.05, # Fraction of the total image height to randomly shift the image vertically. 0.05 means up to 5% of the image height.
    horizontal_flip=True,    # Randomly flips the image horizontally (left-to-right) during augmentation.
    vertical_flip=True,      # Randomly flips the image vertically (top-to-bottom) during augmentation.
    fill_mode='nearest',     # Determines how to fill pixels introduced by rotations or shifts. 'nearest' uses the nearest pixel value, which can reduce artifacts at edges.
    data_format='channels_last' # Specifies the data format of the image (shape convention).'channels_last' means the image dimensions are (height, width, channels),which is standard
)
```

The image shows the augmentation parameters and their values that we used.
There are more optional parameters, but not all of them seemed necessary in our case.
We have paired the images and masks earlier, so in the next step we created a function, where their augmented versions are being paired too - finally we have tested the dimensions and type of this newly created data.

Hyperparameter optimization(Keras-Tuner)

As we have progressed, using tensorflow.keras - the logical solution to a hyperparameter optimizer tool, seemed like keras-tuner. This is a library designed specifically to simplify and automate hyperparameter optimization for TensorFlow/Keras models.
We chose the following parameters to modify during optimization:

- Batch size: A range from 4 to 16 with a step of 4, which allows exploration of smaller to moderate batch sizes. A default of 8 is the midpoint for balancing computational efficiency and model performance, especially on hardware like GPUs with limited memory in environments like Google Colab.
- Gaussian noise: Gaussian noise adds random noise sampled from a normal distribution to the input data during training. This acts as a regularization technique, preventing the model from overfitting to the training data and making it more robust to small perturbations in the input. The range from 0.0 to 0.2 with a step of 0.05 reflects a gradual exploration of noise intensity.
- Upsampling mode: Upsampling is the process of increasing the spatial resolution of feature maps in a neural network.
  SIMPLE: Basic interpolation methods that are computationally light but may lose some detail.
  DECONV: Transposed convolution (also called deconvolution) learns the upsampling operation and can produce sharper and more detailed outputs, but it is computationally more expensive.
  The choice between Simple and Deconv provides flexibility to balance simplicity and performance.
- Net scaling: This refers to downscaling the input images before feeding them into the network. It reduces computational requirements and helps focus the model on coarse-grained patterns, potentially speeding up training and reducing overfitting.
  A value of 0 means no scaling (use the original resolution).
  Higher values (2, 4) represent greater reductions in resolution, such as halving or quartering the dimensions.
  Including scaling options (2 and 4) allows the tuner to explore whether reducing resolution benefits generalization by focusing on high-level features rather than pixel-level details.

The actual search of the optimizer's goal was to find the max validation IoU_Metric value and we had to try several times to make it work well.(Sadly some of the good starting ones were lost too.) These are the parameters, we used for the tuner:

```
tuner = kt.RandomSearch(
    build_model,
    objective=kt.Objective("val_IOU_Metric", direction="max"),
    max_trials=7,  # Number of trials
    executions_per_trial=1,  # How many times to train each model
    directory='tuner_results',
    project_name='ship_segmentation'
)
```

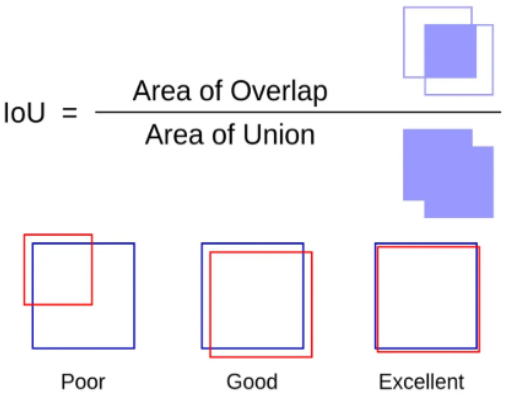
After the correct settings, this is how we started the search with the keras-tuner:

```
augmentedGeneration = createAugmentations(createAndPairMasksToImages(equalizedTrainDataFrame))
# Perform the search
tuner.search(augmentedGeneration,
             steps_per_epoch=100,
             epochs=5,
             validation_data=(validationX, validationY),
             #validation_steps=validationX.shape[0] // BATCH_SIZE,
             callbacks=callbacks_list)
```

```
Epoch 1/5
100/100 ━━━━━━━━━━━━━━━━━━━━ 89s 767ms/step - f1__metric: 0.0033 - iou__metric: 0.0108 - loss: 0.4663 - val_f1__metric: 9.3371e-09 - val_iou__metric: 0.1673 - v
Epoch 2/5
100/100 ━━━━━━━━━━━━━━━━━━━━ 69s 700ms/step - f1__metric: 1.8081e-12 - iou__metric: 0.0470 - loss: 0.0514 - val_f1__metric: 3.0209e-09 - val_iou__metric: 0.3052
Epoch 3/5
100/100 ━━━━━━━━━━━━━━━━━━━━ 68s 690ms/step - f1__metric: 1.8230e-13 - iou__metric: 0.0676 - loss: 0.0450 - val_f1__metric: 1.4375e-11 - val_iou__metric: 0.5699
```

We ran it for 7 trials, each containing 5 epochs with 100 steps. More than 5 epochs seemed unnecessary, because good hyperparameters are usually detectable when monitoring the search for 2 or 3 epochs per trial.

Evaluation metrics(IOU, IOU_BCE, IOU Loss, F1)



To evaluate our model, we chose the Intersection over Union metric along with a custom loss function. We also monitored the F1 score of the model, but that was not the goal.
We implemented these functions in a separate cell and used them during the model training.

IOU_Metric:

- **intersection** computes the pixel-wise overlap between the predicted and true masks.
- **union** calculates the total area covered by both masks (overlap plus the non-overlapping areas).
- The IoU score is then averaged across the batch using **reduce_mean.**

IOU_Loss:

- Derived from the IoU metric but is designed to be minimized during training.
- Instead of maximizing IoU (as in the metric), the loss formulation subtracts the IoU score from 1, turning the problem into a minimization task.

IOU_BCE:

- Combines the IoU loss with Binary Crossentropy, weighted by a factor alpha.
- BCE measures the pixel-wise difference between the predicted and ground truth masks, treating each pixel independently.

- BCE with IoU loss balances the strengths of both. BCE ensures the model captures pixel-level details, while IoU encourages the model to focus on the overall structure and spatial correctness of the predictions.

U-net model architecture

We designed a U-Net model for image segmentation tasks with built-in flexibility for preprocessing and architecture customization. The model-building function, designed for hyperparameter tuning using Keras Tuner, allows exploration of various configurations, such as batch size, noise regularization, and upsampling methods. The U-Net architecture, renowned for its effectiveness in segmentation tasks, follows a typical encoder-decoder pattern with skip connections to preserve spatial information across layers.

The input preprocessing phase supports optional downscaling using average pooling, which can reduce computational costs for large input images. It also includes Gaussian noise and batch normalization layers. The noise layer acts as a regularizer, improving the model's robustness to variations in input data, while batch normalization speeds up convergence during training and reduces sensitivity to initialization.

In the contracting path, the model uses convolutional blocks with ReLU activation and max-pooling to downsample and extract features. Conversely, the expanding path reconstructs the spatial resolution using either Conv2DTranspose (deconvolution) or UpSampling2D, selected based on the hyperparameter UPSAMPLE_MODE. Skip connections bridge the encoder and decoder layers, ensuring the model retains fine-grained spatial details.

The final segmentation layer applies a sigmoid activation to produce a binary mask, suitable for segmentation tasks. An edge cropping and zero-padding mechanism compensates for border effects introduced during convolution operations. The output can optionally be upscaled back to the original resolution if downscaling was applied during preprocessing.

Callback functions

We added callback functions to the training of the network to make it more efficient.

Tensorboard(Not a callback, but is great to visualize the learning):

Tensorboard is used to monitor training progress and visualize metrics like loss and accuracy in real-time. It records data such as the evolution of the loss function, learning rates, and even model graph visualizations to a log directory that can be explored interactively using the TensorBoard tool.

ModelCheckpoint:

The ModelCheckpoint callback saves the model weights whenever the specified metric (in this case **val_IOU_Metric**) reaches a new high. The weights are saved to a file, which ensures that the best-performing model during training is preserved.
This callback prevents losing the best state of the model due to subsequent degradation in performance.

ReduceLROnPlateu:

The ReduceLROnPlateu adjusts the learning rate dynamically when the monitored metric stops improving. It reduces the learning rate by a specified factor (here, 0.5) after the model fails to improve for a defined patience period (3 epochs).

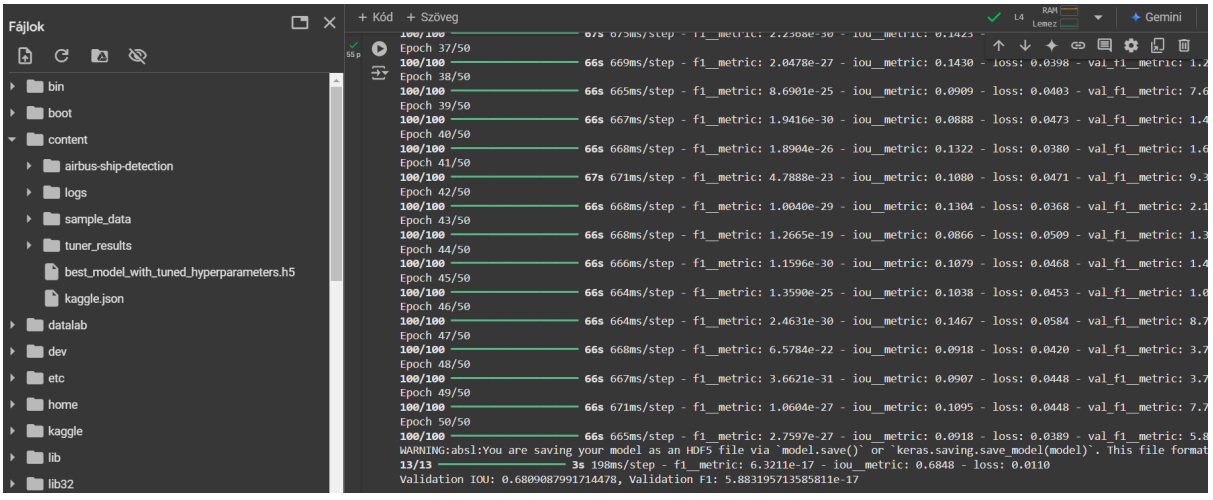
EarlyStopping:

Early Stopping halts training when the monitored metric (val_IOU_Metric) does not improve for a set number of epochs (patience of 4 in this case). This prevents the model from overfitting to the training data and saves computational resources by terminating unproductive training phases.

Model training and accuracy

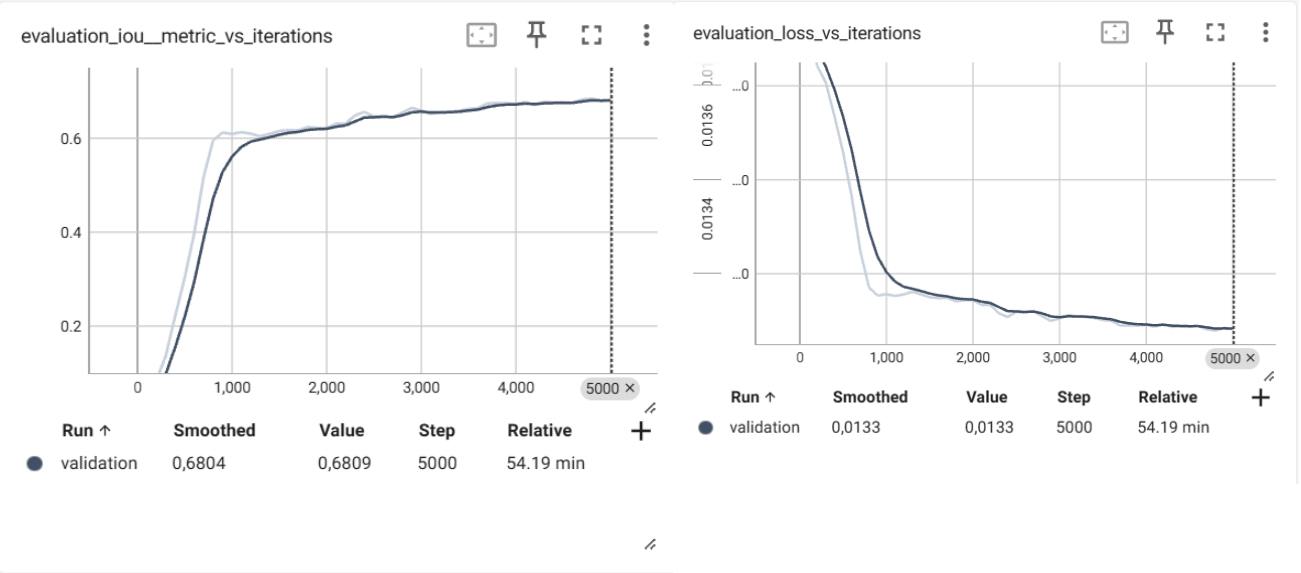
```
# Train the model using the optimal hyperparameters
history = best_model.fit(
    augmentedGeneration,
    steps_per_epoch=100,
    validation_data=(validationX, validationY),
    epochs=50,
    callbacks=callbacks_list
)
```

After the hyperparameter optimization, training the model was quite simple, because we just had to retrieve the best parameters from the tuner.
We had to call the build_model function and then define the length and add the callbacks to the process.
This was obviously not the first try (we tried earlier with different GPU-s and epoch sizes), but finally we trained the model for 50 epochs.



Results

During the model training we included the tensorboard callback to collect data from the optimizations and training.
When the training was finished, we could visually see the final statistics from the model. The IoU score and loss during the training:



The highest IoU score we have reached was 0.6809.
The F1 score of the model is low, because the focus of the optimization was the IoU.

Conclusion and future works

Intersection over Union would probably work better if the objects would cover a significant portion of the image.
F1-score may be more accurate for small, sparse objects like ships.
In the future this project might be worth retraining using F1-score as its central(focused) metric.

Links, where we found the IoU images:

https://idiotdeveloper.com/what-is-intersection-over-union-iou/
http://ronny.rest/tutorials/module/localization_001/iou/

AI usage:

We used ChatGPT and the embedded Gemini AI in google colab mostly to correct code parts by explaining the cause of errors. While choosing the ImageDataGenerator arguments, we also used them to explain the parameters better, because their documentation was not very precise.