FlowCloud: An Image Segmentation Pipeline for Edge Devices in the Cloud

Daniel Casado Herraez

***Abstract*—Image segmentation is a computer vision field that has been widely explored for different applications such as disease detection, self-driving vehicles and aerial map creation. It's main concept is to extract information from the image by identifying different regions that correspond to different objects or classes. However, computing power plays an important role on being able to perform this pixelwise inference, thus being a limiting factor for small devices (wearables, phones, surveillance cameras...). This project solves this limitation for internet-connected edge devices by taking the image segmentation inference step to a remote GPU-capable machine and transmitting the encoded video via a TCP client-server network. The data flow pipeline is presented from the video capturing step to the image segmentation.**

I. Introduction

Image segmentation has proven useful in numerous applications including medical diagnosis, Wang et al. 2021, self-driving, Kaymak and Uçar 2019 vehicles, precision agriculture and geo-sensing thanks to the valuable information it provides from different regions in the image. Classical approaches exist for segmenting pictures, however recent advances in deep learning during the past decade have allowed more generalizable and accurate ways of achieving pixelwise classification. One of the main challenges that appear when applying convolutional neural networks for image segmentation is the amount of computing power required to perform fast inference in real time. The bottleneck is even larger for small devices that cannot have GPU computing capabilities due to physical limitations. This research will approach this limitation by transferring the inference workload from the small edge device to a remote GPU server, increasing the computation frequency and expanding the boundaries of edge computing for small internet-connected devices.

The report will be structured as follows. Section II overviews the most common image segmentation approaches and existing solutions for CNN inference in edge-computing devices, as well as a brief introduction to distributed systems network models, Section III outlines the structure of the proposed solution, followed by the implementation in Section IV. The method is evaluated in Section V and the results are discussed in Section VI. The conclusion and future research will finally be summarized in Section VII. It is important to note in advance that due to the high number of components in the project, the main focus is on the implementation and qualitative measures rather than on obtaining optimal objective results for each of its parts.

II. Related Work & Background

Image segmentation techniques may be divided into three main branches (Yuheng and Hao 2017), classical, unsupervised clustering, and deep learning methods. The first are usually based on thresholding certain attributes of the image such as the brightness or a color channel, as well as obtaining the edges of the figures in the image and infer the objects from those features. Their main disadvantage is the low generalization characteristics, as the parameters would need to be adjusted based on the environment conditions. As for unsupervised clustering, one of the most common ones is K-means clustering, where all the pixels are grouped into similar clusters, resulting in a fast and simple algorithm. However it has some disadvantages regarding its generalization on non-convex pixel groups.

A. Image Segmentation Using Deep Learning

Neural network-based learning dates back to Hubel and Wiesel 1968. They introduced how the visual processing in the monkey's visual cortex starts with simple structures. Fukushima 1980 developed this research into a visual pattern recognition neural network, establishing the based for future research in the field of CNNs. Some years later, LeCun et al. 1989, developed a training procedure for the network using backpropagation.

However, it was not until 2014 that Shelhamer, Long, and Darrell 2014 developed the first image segmentation neural network by replacing the fully connected classification layers into convolution layers. This network outputted the heatmap binary mask each class, with the size of the input image. One of the main characteristics of this FCN segmentation approach was the compatibility with other models.

Further research in the field of medical image segmentation developed an encoder-decoder image segmentation implementation called U-Net, (Ronneberger, Fischer, and Brox 2015). This architecture received as an input an image, encoded it using convolution with ReLU loss function and maxpooling, and decoded the result with up-convolution. The main disadvantage of this approach is the dimensional information lost in the process of convolution-deconvolution. This issue is approached by alternatives such as Segnet (Badrinarayanan, Kendall, and Cipolla 2015), and DeepLab (Chen et al. 2017).

B. Edge Devices & Deep Learning

In the last decade, GPU-capable edge devices such as Nvidia Jetson series and Coral AI boards are becoming increasingly available. Other application specific hardware is being developed and neural networks are being adapted to small devices.

However their capabilities are still limited by their size, which require as low memory and number of computations as possible.

A way to approach this problem is to build neural networks that fit in small devices (Véstias 2019), meaning that the number of parameters must be reduced. This can be achieved by reducing the input size such as EdgeCNN, Yang et al. 2019, or reducing the network arithmetic complexity performing quantization, Wu and Huang 2019.

One of the main variables considered in evaluating such approaches is the size of the model, computed as:

$$Size(in\ MB) = tensor\ size(B) * n_{parameters} \quad (1)$$

Commonly float32 is used, which takes up 4B, however lower precision has been successfully been used by Wen et al. 2021 leading to compact architectures.

Another indicator is the number of floating point operations (FLOPs). This is a measure of performance for computing platforms, it involves either addition, subtraction, multiplication or division of floating-point numbers. Bigger networks will require higher number of FLOPs.

More specifically, When referring to image segmentation, EdgeSegNet (Z. Q. Lin, Chwyl, and Wong 2019) and AttendSeg (Wen et al. 2021) have obtained outstanding results on these two measurements. However, such methods sometimes compromise the accuracy of the network.

A comparison of some architectures according to this criteria is displayed in Table I.

It can be observed the good accuracy of the compact methods, and future research on the area will lead to further improvements. The reduced number of operations and memory makes them a great fit for small devices. However, for the cases in which accuracy is a must, higher definition pictures need to be used, and bigger networks still show greater advantages if run in GPU resources that suit their needs. Hence their implementation in cloud computing services being an immediate and effective move to make.

TABLE I
SIZE, FLOPS AND ACCURACY COMPARISON FOR IMAGE SEGMENTATION.

Model	Parameters (M)	Memory (MB)	FLOPs (M)	Accuracy
EdgeSegNet	7	28.3	39	89.2
AttendSeg	1.2	1.19	4	89.9
ResNet50	35	140	28,000	91.4
DeepLabV3-ResNet50	42	168	51,000	92.4

C. Distributed Systems

The term 'Cloud Computing' refers to the on-demand availability of computing resources without the user having to actively manage them. This brings deep learning to another level, as it is no longer necessary to own expensive GPUs to train and develop neural networks. One can simply connect to a virtual machine in a cloud provider (Google Cloud, AWS, Microsoft Azure) and take advantage of its powerful computers. For those with more resources, it can even be more feasible to host a private GPU server to work on deep learning from different locations.

In order to be able to work remotely on these systems, a simple OpenSSH client-server could work to be able to access the machine. Then, the deep learning model can be trained and developed on the remote computer without the need of developing locally.

The main complexity comes when establishing a more complex client-server connection, where a data pipeline needs to be created for streaming real-time video information into the cloud. This leads to the need of understanding the basics of video encoding and a networking system.

1) Video Encoding: The easiest way to send data in a visual pipeline is to divide the live video into multiple compressed frames, and send them individually over the network. This is similar to the procedure of Motion JPEG (MJPEG). It captures JPEG images from the video, and encodes them as a bytestream that when played will simulate the effect of motion.

A more advanced and efficient technique is H.264 where motion estimation and quantization take place. This method helps obtaining a reduced bandwidth, helpful for reducing the amount of data that is being sent over the network, but at the same time, keeping the video essential information.

An important point to consider before encoding the video will be the video format, which in a wide variety of cases is YUV.

2) OSI Network Model: The OSI Network Model is a concept that specifies the different layers that a network distributed system should have. It is convenient to understand its main components with some examples to develop a data transmission pipeline between the layers. It can be divided into:

1) **Physical layer**: The lowest layer. It is responsible for connection between internet nodes and is based on the cables (RS232, optic fiber) or wireless technology (wifi, bluetooth) that make it possible.
2) **Data link layer**: Ensures a reliable connection for data transfer between two connected nodes. (Ethernet).
3) **Network layer**: Structures and manages the network nodes by addressing the packages to the corresponding addresses and divides (if sender) or reassembles (if receiver) data segments. (IPv4, IPv6). An important network protocol in this work will be RTP (Real-Time Protocol), used for streaming media over IP networks. It can run over TCP or UDP networks and it is in charge of delivering video packets.
4) **Transport layer**: Establishes a reliable transmission of data between points of a network, breaking the segments into chunks that can be used by lower layers (if sender) or reassembling them (if receiver) to be used by higher layers. (TCP and UDP).
5) **Session layer**: Manages communication channels (sessions) between network nodes. (SCP, P2P).
6) **Presentation layer**: Sets the agreement between how devices encode/decode, compress/decompress and encrypt/decrypt their data. (JPEG, MJPEG, H264).
7) **Application layer**: The highest level of the OSI model. It includes end-user applications and APIs that enable

the exchange of information. (email clients, HTTP, FTP).

III. Proposal

After studying the existing solutions for image segmentation, and introducing some background on cloud computing, it is clear how important it can be to develop an efficient image segmentation framework. The main trade-off that arises is between latency and accuracy. On the one hand, computing in the cloud, while it provides high accuracy results, it has the disadvantage of causing data transmission delay. On the other hand, computing on smaller devices requires no data transmission, thus no network latency, but the neural networks need to be modified to fit on the device. Assuming an immediate and high accuracy solution is needed for an edge device, and that it has connection to the internet, this work provides a solution for high precision inference procedure in the cloud. The main features of the work are summarized as follows:

- Image segmentation: An application to a specific domain has been selected, with an appropriate CNN architecture, and fine-tuned accordingly.
- Data pipeline: In order to transmit data over the internet to a remote GPU server, four main steps have been implemented: 1) data capture from edge device, which in this case will be simulated by an Nvidia Jetson Nano, 2) video encoding and transmission over the network 3) video packets decoding on the GPU server 4) data processing with image segmentation.

IV. Implementation

A. Image Segmentation

The main steps for implementing the image segmentation component of the project are the following:

1) Define specific application domain: The application selected for this project will be autonomous driving. Even though self-driving vehicles can fit larger GPU devices and do not necessarily require cloud inference, there is a large research in the field with plenty of data to train and verify the image segmentation network.
2) Understand available programming tools: Most popular CNN development tools are TensorFlow and PyTorch. Even though there are some differences in the computations, most functionalities are available in both. PyTorch was selected arbitrarily for the project.
3) Find a training dataset: Finding a training dataset is specific to the application domain. In autonomous driving, there are two well-known ones. Cityscapes is based on Germany. KITTI is based on the United States and contains data from numerous sensors. As current location is based on Spain, the similarities are bigger so Cityscapes dataset is being used in order to avoid limiting further development.
4) Implement network in PyTorch: Without having previous knowledge on implementation of a CNN in the PyTorch framework, it is convenient to to start with implementing a simple model and then narrow it down to the more specific domain of the project.
5) Deploy and test on cloud: After the image segmentation network is trained and runs with decent performance, it is necessary to test it in remote servers while verifying its performance.

1) Dataset: The Cityscapes dataset (Cordts et al. 2016) is divided into two components, gtFine (241MB) and leftImg8bit (11GB), that should be downloaded independently. The former contains the labelled masks, while the latter has the original images. The masks are of three types: 'color' (the mask will have RGB colors of the classes), 'instanceID' (labelling of each independent instance), 'labelID' (labelling of each class). Only the 'labelID' masks will be used in this project. There are a total of 35 classes. Each pixel in the image will correspond to one of those classes with values between 0 and 35. An example of the images contained in the dataset can be observed in Fig. 1.

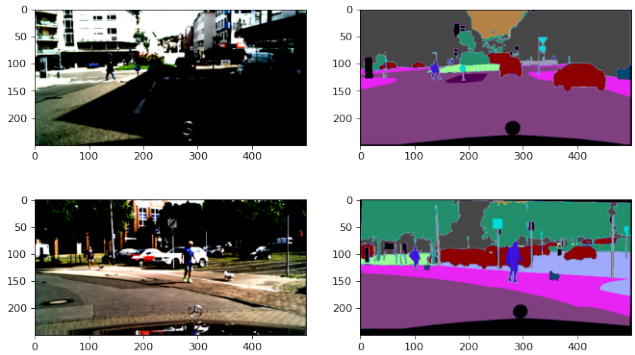

Fig. 1. Cityscapes dataset sample training images.

2) Architecture Selection: Despite the focus of the project not being the selection of the best architecture for cloud image segmentation, it is convenient to select a CNN that will output meaningful results so that they can be compared when deployed in an edge device versus in the cloud. Table II shows some of the available pretrained models in Pytorch. DeepLabV3's atrous convolution (Chen et al. 2017) provides an accurate solution that reduces the effects of scale in the input. It was selected with ResNet50 (He et al. 2015) backbone due to the high score ratings, as well as for the high number of parameters that will make a big difference when deploying it in the cloud and in an edge device.

TABLE II
PYTORCH IMAGE SEGMENTATION ARCHITECTURES COMPARISON.

Network	Accuracy (%)
FCN ResNet50	91.4
FCN ResNet101	91.9
DeepLabV3 ResNet50	92.4
DeepLabV3 ResNet101	92.4
DeepLabV3 MobileNetV3	91.2
LR-ASPP MobileNetV3	91.2

3) Loss Computation: One of the most important parts of training an image segmentation network is understanding how the loss is computed, which must be done pixelwise. Therefore, if the input image has dimensions 250x500, the output with predictions for 35 classes will have size 35x250x500.

This can be confusing at first, due to the fact that the Cityscapes masks have dimensions 250x500 and all class labels are pixels in the image. However, it can be solved in two different ways: 1) introducing an intermediate step to adapt the masks to the output size of 35x250x500 using one hot encoding. This will solve this problem by setting one layer for each class, and assigning the label locations a probability of 1. 2) alternatively, selecting a loss function that can take multidimensional labels, such as the cross-entropy cost function.

4) Transfer Learning: Provided models by Pytorch have been pretrained for 21 classes on the COCO dataset (T.-Y. Lin et al. 2014) and are ready to use with specific implementations by simple fine tuning. Fine-tuning refers to the step of replacing the pixel classification layer with a custom one for the required number of classes, freeze the other layers, and perform training to adjust it to the required classes. In the case of Cityscapes the number of classes is 35, so the last convolution layer is replaced to output 35 binary masks, one per class.

Fine tuning was performed on the DeepLabV3-ResNet50 network (Figures 2, 3), obtaining a training pixelwise accuracy of 91.5% (computed at every step), and validation accuracy of 88.4% (computed after each epoch). Generalization is enhanced as noted by Bradley and Gomez-Uribe 2021, by using stochastic gradient descent when training.

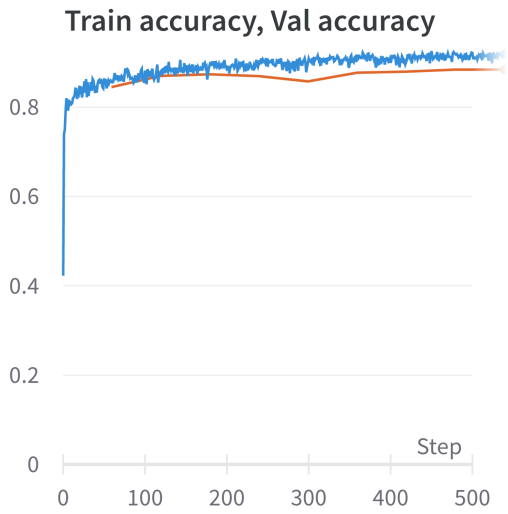

Fig. 2. Training and validation accuracy chart.

B. Data Pipeline

Real time video streaming over the internet will be explained using a summary of certain components in the OSI model described earlier, which makes it easier to understand.

First of all, before entering the network, the edge device captures real-time data from the camera in a raw YUV video format variant. This video is encoded as a byte-stream into H.264 frames, more efficient than MJPEG, and then packaged into RTP packets that can be sent over a TCP server. By this

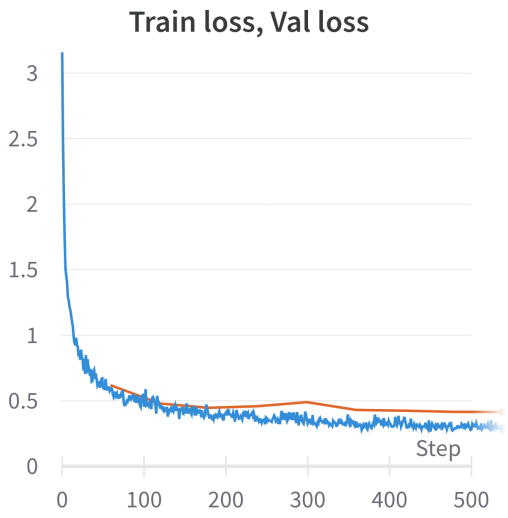

Fig. 3. Training and validation loss chart.

point, the video data is ready to be sent to a remote server. This process has been illustrated in Figure 4 (left).

Now, the edge device needs to be connected to the internet using a **physical layer**, which in this case was an RJ45 cable, thus the **data link layer** connection is ensured using Ethernet Protocol. So that data can be sent to the GPU server, it must be transmitted over a **network layer** with a public Internet Protocol (IP) address. It is important to note that the IP address corresponds to the address where the video wants to be streamed, in this case, the edge device's IP. This address must be public and readable by external devices, so port forwarding in the router might be required.

To establish a reliable data transmission in the **transport layer**, two alternatives are possible. The User Datagram Protocol (UDP) is a fast data transfer method that tolerates packet loss. However, HTTP application layer only supports the Transport Control Protocol (TCP), that has higher latency but connection is more reliable than UDP and does not tolerate packet loss. One type or another could be chosen for different applications, but as the main idea of this work was to be able to implement the pipeline in Google Cloud, which runs over HTTP, the transport layer used will be based on TCP.

Once the data packets have been sent through the TCP server, the client will receive them encoded inside RTP packets. The client will decompress the packages and decode the H.264 stream to raw video readable by OpenCV.

Finally, the client will be able to perform image segmentation on the incoming data stream, Figure 4 (left), and either send a response back to the edge device, or process and keep the data in the cloud. Note that the term 'client' comes from its role in the TCP network as the receiver of the video data, strictly speaking, when interacting with it as a Virtual Machine it would be the 'GPU server', therefore referred to as such.

As it is observed, the last three layers of the OSI model are not being used. This is because in this work, no data is being presented to the end user and no peer-to-peer connection needs to be established. The corresponding layers can be observed

in Table III.

TABLE III
OSI MODEL LAYERS USED IN THIS WORK

Layer	Used
Physical Layer	RJ45
Data Link Layer	Ethernet
Network Layer	IPv4
Transport Layer	TCP
Session Layer	-
Presentation Layer	-
Application Layer	-

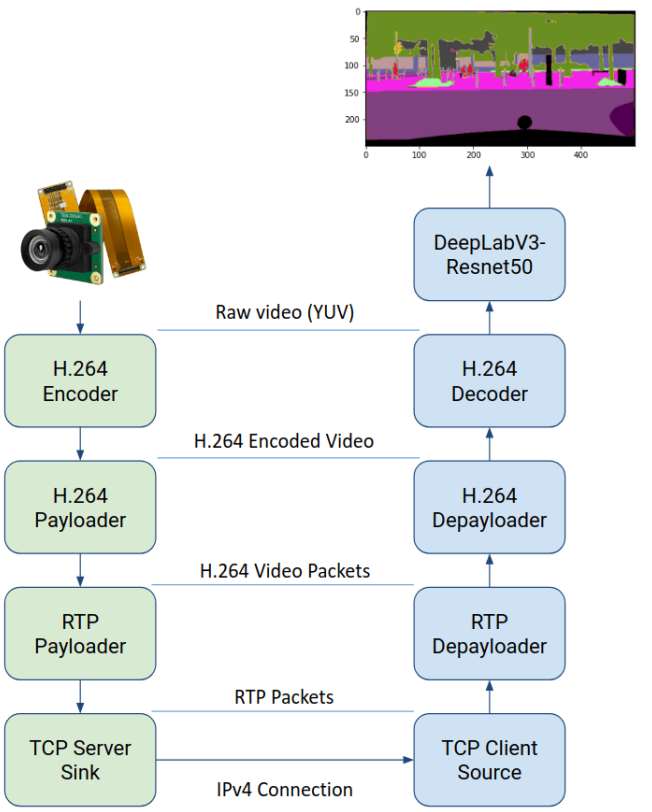

Fig. 4. Video transmision via TCP server (left branch) and image segmentation in the TCP client

V. Evaluation and Results

The following experiments for evaluation were performed:

1) Perform image segmentation directly in Google Cloud VM instance (Belgium, Intel Xeon CPU @ 2.20GHz, 8GB ram), Jetson Nano in low power mode (Spain, NVIDIA Maxwell architecture with 128 CUDA cores, 4GB ram), local computer (Spain, NVIDIA MX250 2GB with 384 CUDA cores and I7-10510U CPU @1.80GHz, 16GB ram), and remote GPU server (Japan, NVIDIA RTX2080 with 4,352 CUDA cores and I9-10900X CPU @3.7GHz, 128GB ram). Measure performance based on framerate.
2) Setup TCP server from Jetson Nano to local computer, Google Cloud, and remote GPU server. Measure network delay.

A. Direct Segmentation in Machine

In order to see the performance of each of the devices, segmentation was performed directly into various tests machines. The specifications on the Google Cloud Machine were limited to the trial features, in which even though $300 could be spent in cloud products, adding GPU to the cloud instance was not allowed. Similar situation was given with Microsoft Azure and AWS, where GPUs were limited to premium users. To compensate this, KEIO laboratory servers were used. The delay consequences will be outlined later in the report.

Moreover, due to the lack of material resources to connect the Jetson Nano in a vehicle outdoors to the Internet, the behavior of the camera was simulated by streaming a car video file located in the memory of the Jetson Nano.

The results of the tests are outlined in Table IV. Furthermore, Figure 5 shows some sample runs on the test set with the segmentation mask label IDs transformed to the corresponding colors.

TABLE IV
IMAGE SEGMENTATION TESTS PERFORMED ON EACH MACHINE.

Test	Framerate (fps)
Google Cloud E2 (Xeon@2.20GHz, 8GB)	0.16
Jetson Nano (Maxwell 128 CUDA cores, 4GB)	0.6
Local PC (MX250 384 CUDA cores, I7-10510U@1.80GHz, 16GB)	4
GPU server (RTX20804,352 CUDA cores, I9-10900X@3.7GHz, 128GB)	18

Fig. 5. Sample image segmentation runs on fine-tuned DeepLabV3-ResNet50

B. TCP Data Pipeline

After measuring each of the computing machines individually, it is necessary to consider network delay between client and server. A way of carrying out these measurements would be to compare the timestamps of the received RTP packages from the camera device with the server timestamp. However this would require dealing with the complexity of modifying the used video streaming library (GStreamer), and synchronizing the client and server streams, which is out of the scope of the project. Subsequently a qualitative evaluation will be done for the approach.

Some notes to be considered are the RTP package size, 12 bytes per packet, and the H.264 encoding bandwidth, 2.5Mbit/s for a 480p video. Thus, having a higher bandwidth than that on the network will be enough for ingesting the video into the TCP server.

The network delays and TCP (sender/receiver) bandwidths for each of the test machines are presented in Table V.

TABLE V
NETWORK DELAY AND TCP BANDWIDTHS ON EACH MACHINE.

Device	Delay (ms)	Bandwidth (Mbit/s)
Local computer	1.5	313
Google Cloud VM	40	85
GPU Server	350	8.5

VI. DISCUSSION

After observing the evaluation results for the FlowCloud components, it is obvious how introducing a cloud server connection to an edge device can be a great option to perform inference using high accuracy heavy neural networks. Even the Nvidia Jetson Nano, whose hardware is oriented to deep learning processing, was not able to handle such a high processing power, with its results taking 6.25s to process a single frame, more than 10 times worse than using external GPU power.

Moreover, the network delay varies greatly when the client is far away from the server. In this case, the Jetson TCP server was located in Spain. The best result is for the local computer as it is in the same network as the server. Google Cloud instances located in Europe also offered good network delay and bandwidth for non-high-speed applications. Meanwhile, the GPU server in japan caused very high network delay, which is still better than the 6.25s mark per frame obtained from the Jetson Nano.

VII. CONCLUSION AND FUTURE WORK

This report has overviewed all the background and steps for a full implementation of a data pipeline, starting from the captured image in the Jetson Nano camera, going through video encoding and transport layer delivery, and ending at the image segmentation result of a CNN in a remote server. The main focus of the work has been the implementation of the entire system rather than optimizing for each particular feature.

A final evaluation has led to interesting results on how network delay can be compensated with processing power, and it can be concluded that the optimal results would be obtained by having a GPU server in a location close to the video transmitting device, reducing network delay.

Future work in this project would include a quantitative and more rigorous evaluation of the system delays, comparing it with the performance obtained from compact CNNs working directly in the edge device. Testing it with other CNNs and other deep learning applications is also an open door for evaluation of this system. In addition, a limitation of this approach is the required connectivity to the internet of the device. New lines of work could be to integrate the two discussed approaches into the system, so when there is network availability, cloud resources are used, and when there is not, a simple CNN is used for segmentation. Furthermore, other edge devices could be tested, with different internet connection capabilities and less computing power.

REFERENCES

Hubel, D. H. and T. N. Wiesel (Mar. 1968). *Receptive fields and functional architecture of monkey striate cortex*. DOI: 10.1113/jphysiol.1968.sp008455. URL: http://dx.doi.org/10.1113/jphysiol.1968.sp008455.

Fukushima, Kunihiko (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36, pp. 193–202.

LeCun, Y. et al. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4, pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

Lin, Tsung-Yi et al. (2014). "Microsoft COCO: Common Objects in Context". In: *arXiv preprint arXiv: Arxiv-1405.0312*.

Shelhamer, Evan, Jonathan Long, and Trevor Darrell (2014). "Fully Convolutional Networks for Semantic Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.4, pp. 640–651. DOI: 10.1109/TPAMI.2016.2572683.

Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla (2015). "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *arXiv preprint arXiv: Arxiv-1511.00561*.

He, Kaiming et al. (2015). "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv: Arxiv-1512.03385*.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox (2015). "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab et al. Cham: Springer International Publishing, pp. 234–241. ISBN: 978-3-319-24574-4.

Cordts, Marius et al. (2016). "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *arXiv preprint arXiv: Arxiv-1604.01685*.

Chen, Liang-Chieh et al. (2017). "Rethinking Atrous Convolution for Semantic Image Segmentation". In: *arXiv preprint arXiv: Arxiv-1706.05587*.

Yuheng, Song and Yan Hao (2017). "Image Segmentation Algorithms Overview". In: arXiv: 1707.02051.

Kaymak, Çağrı and Ayşegül Uçar (2019). "Semantic Image Segmentation for Autonomous Driving Using Fully Convolutional Networks". In: *2019 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pp. 1–8. DOI: 10.1109/IDAP.2019.8875923.

Lin, Zhong Qiu, Brendan Chwyl, and Alexander Wong (2019). "EdgeSegNet: A Compact Network for Semantic Segmentation". In: *arXiv preprint arXiv: Arxiv-1905.04222*.

Véstias, Mário (July 2019). "A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing". In: *Algorithms* 12, p. 154. DOI: 10.3390/a12080154.

Wu, Yueh-Chi and Chih- Tsun Huang (2019). “Efficient Dynamic Fixed-Point Quantization of CNN Inference Accelerators for Edge Devices”. In: *2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–4. DOI: 10.1109/VLSI-DAT.2019.8742040.

Yang, Shunzhi et al. (2019). “EdgeCNN: Convolutional Neural Network Classification Model with small inputs for Edge Computing”. In: *arXiv preprint arXiv: Arxiv-1909.13522*.

Bradley, Arwen V. and Carlos Alberto Gomez-Uribe (2021). “How Can Increased Randomness in Stochastic Gradient Descent Improve Generalization?” In: *arXiv preprint arXiv: Arxiv-2108.09507*.

Wang, Risheng et al. (2021). “Medical Image Segmentation Using Deep Learning: A Survey”. In: arXiv: 2009.13120.

Wen, Xiaoyu et al. (2021). “AttendSeg: A Tiny Attention Condenser Neural Network for Semantic Segmentation on the Edge”. In: *arXiv preprint arXiv: Arxiv-2104.14623*.