UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Object instance segmentation using recurrent models

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Juan Carlos Morales Vega

In partial fulfilment

of the requirements for the degree in

Telecommunication Technologies and Services ENGINEERING

Advisor: Ferran Marqués

Barcelona, June 2017

Abstract

This work intends to expand the Polygon-RNN model for object segmentation [1], able to create a polygon outlining an object using a CNN and a RNN, by trying several modifications. Several models and approaches have been explored. Out of them, the one that has shown the best IOU has been a joint model combining CNNs and a RNN. While the accuracy when running the model in default mode prediction only reaches 55.14% (which in fact is quite high), when running in beam search mode the accuracy has increased up to 59,94%. Once the final model was chosen, it was implemented in the LabelMe annotation tool, a web tool that can be used for annotators to segment instances of objects. This way the annotation process is made in a semi-automatic manner, making the work much faster and easier for the annotator. The annotator has to select a bounding box around the object and just slightly correct the resultant prediction to obtain the final annotation. The qualitative results have proven to be very precise.

Resum

Aquest treball pretén expandir el model de segmentació d'objectes Polygon-RNN [1], capaç de crear un polígon al voltant d'un objecte usant una CNN i una RNN, provant diferents modificacions. S'han provat diversos models i enfocaments. D'entre ells, el que ha mostrat la major IOU ha estat un model conjunt combinant CNNs i una RNN. Si bé la precisió del model al executar-se en la manera de predicció per defecte només arriba a 55.14% (que, de fet, és força elevada), quan s'executa en la manera de predicció beam search la precisió s'incrementa fins a un 59.94%. Una vegada que es va escollir el model final, va ser implementat en l'eina d'anotació LabelMe, una eina web que pot ser usada per anotadors per segmentar instàncies d'objectes. D'aquesta manera el procés d'anotació es fa de forma semiautomàtica, fent la feina molt més ràpid i senzill per l'anotador. L'anotador ha de seleccionar una bounding box al voltant de l'objecte i simplement corregir lleugerament la predicció resultant per obtenir l'anotació final. Els resultats qualitatius han demostrat ser bastant precisos.

Resumen

Este trabajo pretende expandir el modelo de segmentación de objetos Polygon-RNN [1], capaz de crear un polígono alrededor de un objeto usando una CNN y una RNN, probando diferentes modificaciones. Se han probado diversos modelos y enfoques. De entre ellos, el que ha mostrado la mayor IOU ha sido un modelo conjunto combinando CNNs y una RNN. Si bien la precisión del modelo al ejecutarse en el modo de predicción por defecto solo alcanza 55.14% (que, de hecho, es bastante elevada), cuando se ejecuta en modo beam search la precisión se incrementa hasta un 59.94%. Una vez que se escogió el modelo final, fue implementado en la herramienta de anotación LabelMe, una herramienta web que puede ser usada por anotadores para segmentar instancias de objetos. De esta manera el proceso de anotación se hace de forma semiautomática, haciendo el trabajo mucho más rápido y sencillo para el anotador. El anotador tiene que seleccionar una bounding box alrededor del objeto y simplemente corregir ligeramente la predicción resultante para obtener la anotación final. Los resultados cualitativos han demostrado ser bastante precisos.

Acknowledgements

Firstly, I would like to thank my thesis supervisor, Sanja Fidler of the University of Toronto. She was always willing to answer my questions and has always been interested in my progress, helping me to decide the next steps in the thesis.

I would also like to thank Lluis Castrejón and Kaustav Kundu, from the Machine Learning department of the University of Toronto. They developed the original model and they have provided very valuable help any time I needed it.

I also want to thank my thesis advisor in the UPC, since he accepted to be my advisor despite being very busy.

Revision history and approval record

Revision	Date	Purpose
0	06/06/2017	Document creation
1	24/06/2017	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Juan Carlos Morales Vega	juan_carleitor@hotmail.com
Sanja Fidler	sanja.fidler@gmail.com
Ferrán Marques Acosta	ferran.marques@upc.edu

Written by:		Reviewed and approved by:	
Date	06/06/2017	Date	24/06/2017
Name	Juan Carlos Morales Vega	Name	Sanja Fidler
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables:

1. Introduction

1.1. Objectives

To be able to have big and useful datasets to train powerful neural networks some human work is needed. Annotators spend many hours classifying and labeling the data. One of the most typical tasks is object segmentation inside an image, where annotators have to create a polygon outlining the objects and then assign a class tag. Taking into account the high number of points that need to be used to create a precise mask around the object, it would be a great improvement to create a way to make the process automatic and faster. To achieve this, a neural network able to predict the boundaries of an object as a polygon is going to be used.

To make this model available to anyone, it is going to be implemented in a web tool. The user can select a bounding box around the desired object in an image and the model will predict the polygon, allowing the user to accept it or modify it by correcting individual points by simply clicking and dragging with the mouse. This way, the time spent in the annotation process will be reduced.

1.2. Requirements and specifications

This model is able to segment automatically an object inside a bounding box with a high Intersection Over Union (IOU) accuracy over a human-made ground truth. The automatic prediction intends to be as close as possible with respect to how would the real human-made annotation be. By doing that, we are making sure that the number of corrections by the human annotator are kept to a minimum.

Since the final system requires the model to be run in a web annotation tool, a server is needed. The annotator cannot wait long between selecting the bounding box and getting the prediction, so speed is a concern. To minimize this problem is highly recommendable for the server to has a good connection speed and, especially, a powerful GPU for the model to be run in (Nvidia Titan-X was used for the testing). As for now, only one client at a time is allowed to use the tool. The reason of that is because all of the tests have been carried on in a single GPU that can execute the model only once at a time. However, the system can be easily scaled by simply having several GPUs in the server and making very minor changes in the client to make every new connection to take a different GPU. Another way is neglecting the speed and just run the model over every selected bounding box sequentially, but I cannot recommend it.

Once the server is configured, the tool can be run in a normal web browser.

1.3. Methods and procedures

This work is the continuation of a previous project carried out by Lluis Castrejon, Kaustav Kundu, Raquel Urtasun and Sanja Fidler at the University of Toronto [1]. In their paper, they propose an approach for object segmentation using a Recurrent Neural Network to predict a polygon.

To build the model I have used TensorFlow, a very powerful tool to build, run and train neural networks developed very recently by Google Brain. The main programming language used has been Python, since is the one that has the most detailed TensorFlow

documentation. As for the hardware, all the tests have run in a powerful Nvidia Titan-X graphic card. Finally, to implement the model as a web tool, I used the LabelMe annotation tool developed at MIT as a template.

The methodology used during the project was, for each of the steps of the work, as follows: First, a learning stage was required to get familiar with the different tools (for example, TensorFlow when I arrived and the LabelMe annotation tool in the final steps of the project). The next stage was programming the necessary parts and testing them individually to check their correct behavior and partial results. Finally, many tests are carried to check that everything is working properly. Basically, the two final steps after the learning stage can be defined as trial and error stages where not only bugs were corrected, but the configuration for the whole system sometimes needed to be changed to make improvements (and in some of the cases the modifications were discarded because of poor results).

Since the model is so big and there are many people working and submitting jobs in the same cluster of GPUs, the number of times that a whole model can be tested in one day are very limited (from one to three, depending of the circumstances of the cluster and the type of test). During the mean time I worked in programming the next things, although the progress slowed down a little. Of course, partial tests over certain functionalities are much faster, so they are always preferred over full tests since a lot of them can be carried out in a single day.

1.4. Work plan

Work Packages:

<table>
<tr><td>Project: Object instance segmentation using recurrent models</td><td colspan="2">WP ref: 1.1</td></tr>
<tr><td>Major constituent: Self-learning</td><td colspan="2">Sheet 5 of 7</td></tr>
<tr><td rowspan="2">Short description:
Self-learning to gain some knowledge.</td><td colspan="2">Planned start date: 01/02/2017
Planned end date: 08/02/2017</td></tr>
<tr><td colspan="2">Start event: Learn Python
End event: Learn TensorFlow</td></tr>
<tr><td>Internal task T1: Learn Python
Internal task T2: Lear the basis of neural networks, convolutional neural networks and recurrent neural networks
Internal task T3: Learn TensorFlow</td><td>Deliverables:
None</td><td>Dates:
None</td></tr>
</table>

<table>
<tr><td>Project: Object instance segmentation using recurrent models</td><td colspan="2">WP ref: 1.2.1</td></tr>
<tr><td>Major constituent: Research</td><td colspan="2">Sheet 5 of 7</td></tr>
<tr><td rowspan="2">Short description:
Being able to understand the previous model and design a single network able to recognize object instances in a recurrent manner.</td><td colspan="2">Planned start date: 09/02/2017
Planned end date: 20/03/2017</td></tr>
<tr><td colspan="2">Start event: Read about the subject and the previous model
End event: Design network</td></tr>
<tr><td>Internal task T1: Understand previous model
Internal task T2: Design CNN
Internal task T3: Design RNN</td><td>Deliverables:
None</td><td>Dates:
None</td></tr>
</table>

Project: Object instance segmentation using recurrent models	WP ref: 1.2.2	
Major constituent: Python and TensorFlow programming	Sheet 5 of 7	
Short description: Implementing the designed networks using Python and TensorFlow and combine the two of them in a single network.	Planned start date: 20/03/2017 Planned end date: 01/05/2017	
	Start event: Start programming End event: End programming	
Internal task T1: Program the networks Internal task T2: Combine the networks	Deliverables: None	Dates: None

Project: Object instance segmentation using recurrent models	WP ref: 1.3	
Major constituent: GPU simulation	Sheet 5 of 7	
Short description: Test the network and its behavior	Planned start date: 24/04/2017 Planned end date: 10/05/2017	
	Start event: Start simulations End event: End simulations	
Internal task T1: test the network	Deliverables: None	Dates: None

Project: Object instance segmentation using recurrent models	WP ref: 1.4.1	
Major constituent: Learning and coding	Sheet 6 of 7	
Short description: Understand how the current annotation web tool works and making modifications to run the new model in it.	Planned start date: 02/05/2017 Planned end date: 07/06/2017	
	Start event: Learn how the web annotation tool works End event: Implement the new model in the tool	
Internal task T1: Learn how the LabelMe annotation tool works Internal task T2: Make changes to the tool	Deliverables: None	Dates: None

Project: Object instance segmentation using recurrent models	WP ref: 1.4.2	
Major constituent: Debugging	Sheet 6 of 7	
Short description: Test the annotation tool to find errors and correct them.	Planned start date: 07/06/2017 Planned end date: 28/06/2017	
	Start event: Start debugging End event: Finish debugging	
Internal task T1: Debug the web annotation tool	Deliverables: None	Dates: None

Milestones

WP#	Task#	Short title	Milestone / deliverable	Date (week)
1.1	1	Learn Python	Milestone	02/02/2017
1.1	2	Learn about CNN and RNN	Milestone	06/02/2017
1.1	3	Learn TensorFlow	Milestone	08/02/2017
1.2.1	1	Understand previous model	Milestone	01/03/2017
1.2.1	2	Design CNN	Milestone	10/03/2017
1.2.1	3	Design RNN	Milestone	20/03/2017
1.2.2	1	Program the networks	Milestone	24/04/2017
1.2.2	2	Combine the networks	Milestone	01/05/2017
1.3	1	Test the network	Milestone	10/05/2017
1.4.1	1	Understand annotation tool	Milestone	15/05/2017
1.4.1	2	Change annotation tool	Milestone	07/06/2017
1.4.2	1	Debug annotation tool	Milestone	28/06/2017

Gantt diagram

It is important to notice that, since it is a research project, everything was in constant change, so the Gantt diagram has not been strictly followed and acted only as a guideline (as an example, many tests for the network were being carried out even while debugging the annotation tool).

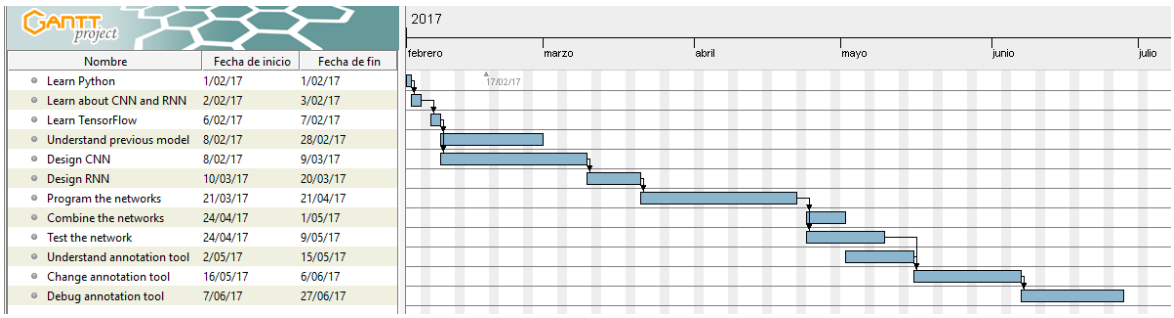

Figure 1: Gantt diagram

1.5. Deviations from the initial plan

In the initial plan, the purpose was to explore the Polygon-RNN model and to make several improvements. However, it was decided that it would also be a good idea to have the model implemented as a user-friendly tool. To achieve this, it was decided to fuse the model with the LabelMe annotation tool developed by Bryan Russell and Antonio Torralba at MIT. Because of that, a final step was added to the project. This final step has been divided in three different tasks: Understand annotation tool, change annotation tool and debug annotation tool, as it can be appreciated in the Gantt diagram.

1.6. Incidences

- Due to the size of the network, the training time has slowed down the development process and the number of tests that can be done in one day.
- Solving the natural bugs that have been appearing while programming has taken time, especially one that happened when restoring a Tensorflow checkpoint that took nearly two weeks in being solved.
- Despite those natural problems, some tasks have been completed sooner than expected.

2. State of the art of the technology used or applied in this thesis:

2.1. Introduction to Neural Networks

Since this project uses a neural network approach, a brief theoretical background about this field is required to understand it.

2.1.1. Perceptron

The perceptron is the most basic model of a neuron. It is a mathematical model that takes several inputs and generates a single output. A single perceptron is illustrated in the next figure:

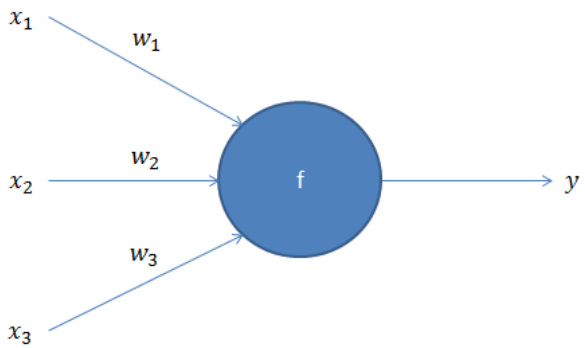

Figure 2: Perceptron

The equation that models the behavior of this basic unit is the following:

$$y = f(z) = f\left(\sum_i w_i x_i + b\right)$$

Here, the x_i are the inputs, the w_i are the weights, b is the bias term, y is the output and f is called the activation function, which is usually a non-linear function. The most used activation function is the rectified linear unit (ReLU) that can be mathematically expressed as $f(x) = \max(0, x)$:

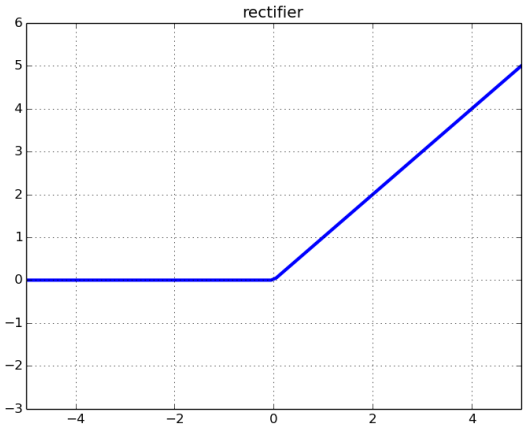

Figure 3: ReLU function

2.1.2. Neural Networks

Combining several neurons in parallel makes a layer. Layers are then stacked one after the other to create a neural network, as in can be seen in the following image:

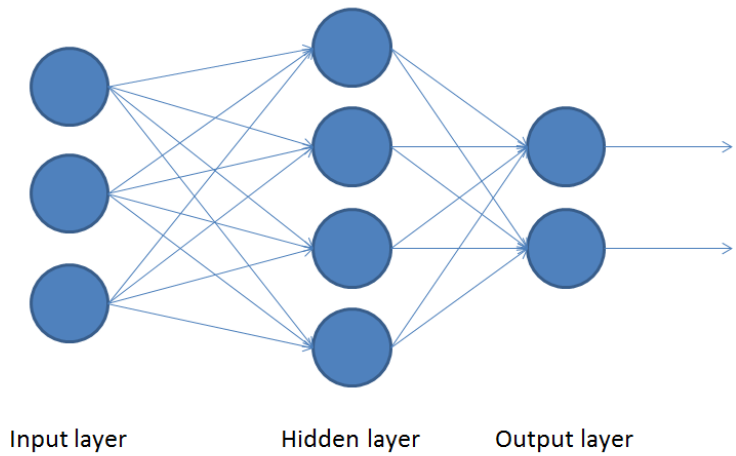

Figure 4: Fully connected neural network

As it can be appreciated, every neuron of a previous layer is connected with all of the neurons in the next layer. This is the most basic architecture that a neural network can have and it is known as a fully connected neural network (FC). At the output layer, instead of using a ReLU function, it is usual to have a target function. For example, when dealing with classification problems, one of the most famous target functions is softmax, which interprets all of the outputs as a normalized probability distribution:

$$\hat{y}_i = \frac{z_i}{\sum_j e^{z_j}}$$

Why using neural networks instead of other approaches? Neural networks with at least one hidden layer have been shown to be universal approximators that can compute any function [2] in a n-dimensional vector space by simply having enough neurons within the layers and by properly adjusting their weights and biases. While choosing manually the parameters is not a realistic task, what make neural networks interesting is that they can learn them by themselves. To do that, we define a cost function (also called loss function)

that evaluates how the model is failing when using it over a labeled dataset. Typical cost functions are the L2 loss:

$$C = \sum_i \left(y_i - \hat{y}_i(\vec{x})\right)^2$$

Or the cross entropy loss:

$$C = \sum_i \left(-y_i \cdot log\left(\hat{y}_i(\vec{x})\right) - (1 - y_i) \cdot log\left(1 - \hat{y}_i(\vec{x})\right)\right)$$

In both of them y_i is the target value, also called label, $\hat{y}_i$ is the prediction and $\vec{x}$ is the input vector. It is fairly easy to compute the gradient of the cost function with respect the parameters backpropagating it using the chain rule of partial derivatives. The gradient is, by definition, a vector function that points towards increasing values of the function, so if we want to decrease the cost we just have to make use of the chain rule once the gradients are computed:

$$dC = \frac{\partial C}{\partial v_i} \cdot dv_i$$

Where v_i is a trainable parameter of the model (w or b). We want $dC < 0$, so we arbitrarily choose:

$$dv_i = -\eta \cdot \frac{\partial C}{\partial v_i}$$

Which makes dC to be always negative. The final step is updating the parameters by simply doing $v_{i,t+1} = v_{i,t} + dv_i$. The final equation is:

$$v_{i,t+1} = v_{i,t} - \eta \frac{\partial C}{\partial v_i}$$

Here, η is a positive parameter called the learning rate that controls the speed of the learning and the stability of the process. A very low learning rate would make the training process very slow while a very high one will make the model simply behave as random. This technique is called Stochastic Gradient Descent (SGD) and it is the simplest way to update the parameters and make the network learn.

By showing the training data to a neural network model several times and repeating the algorithm that has just been explained, said network will improve its performance. A full pass through all of the datapoints in the training dataset is called an epoch. The training algorithm becomes more efficient if, instead of updating the parameters for every datapoint, the cost is averaged over several datapoints:

$$C = \frac{1}{n} \sum_{j=1}^{n} C_j$$

Where n is called the batch size.

Other optimization algorithms that can be found in neural network works can include ADAM or Nesterov Momentum among others. ADAM, which is the one used in this work, computes adaptive learning rates for the parameters and keeps an exponentially decaying average of past square and plain gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_t$$

$$u_t = \beta_2 u_{t-1} + (1-\beta_2)\nabla_t^2$$
$$v_{i,t+1} = v_{i,t} - \frac{\eta}{\sqrt{u_t} + \varepsilon} m_t$$

Usually, the values for β_1, β_2 and ε are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 1e-8$.

One problem when dealing with neural networks that can be a cause of concern is overfitting. Since the model is shown several times the same datapoints, it could simply learn by heart the training dataset if it is not big enough. This will cause the model not to generalize well to other data apart from the training dataset, making the network useless. To reduce the effect of this problem, a technique called regularization is introduced. Regularization adds a new term to the cost that tends to reduce the values of the weights, making the model to prefer lower weights over higher weights. Qualitatively, this is the same as introducing a forgetting effect that prevents the model to simply learn by heart the training data and makes it to rely more in the semantic information. The L2 regularization loss is the most used one:

$$RL = \sum_i w_i^2$$

2.1.3. Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a special type of neural networks designed to work especially well when dealing with images. Instead of using a linear filter, they use a set of small 3D filters that are applied to 3D input data (width x height x channels) such as RGB images using a convolution operation. This way the spatial local information is better preserved than in the fully connected approach. Moreover, since the number of parameters is fewer than in a fully convolutional neural network, they are also faster to train and to test. The depth of the filter is always the same as the depth of the input layer (the convolution is only in 2D) and the number of filters determine the depth of the output layer. Moreover, the 2D slices of the filters are square. CNNs introduce two different terms to describe how the convolution is done: stride and padding. Stride is how many pixels is the filter moved with respect to its previous position when doing the convolution. Padding is the process of adding 0s to the sides of the image to make it possible to have and output with the same 2D size of the input. The usual values for these parameters are filter size 3 x 3, stride 1 and padding 1 (cover the image with only one layer of 0s)

The following image taken from [3] illustrates the process in a 2D slice:

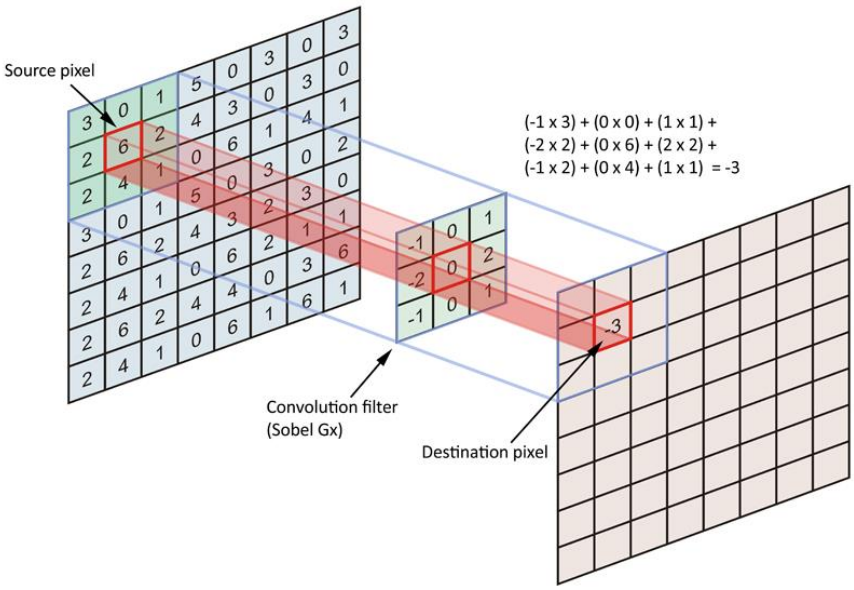

Figure 5: 2D convolutional filter

A good animation of the convolution process can be found in [4].

CNN often have max-pooling layers between convolutional layers to reduce the dimensionality of the data coming from the previous layer. Max pooling is the process of selecting only the maximum value inside a n x n grid and pass it to the next layer. As in the convolution part, the size of the filter size and the stride are also defined, being 2 x 2 filters and stride 2 the most common options that can be found in the literature. Sometimes, instead of max-pooling, the convolutional filters are chosen to use stride 2, which also reduces the dimensionality in the width and height dimension by 2. The next image taken from [4] illustrates perfectly the process of max-pooling:

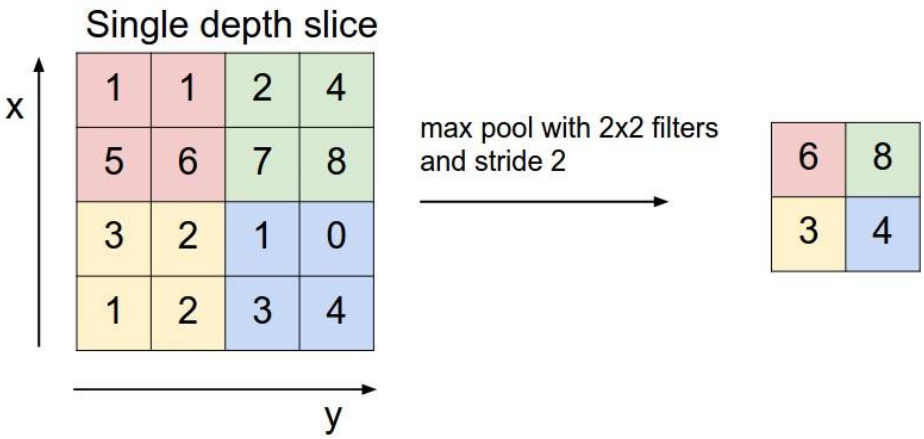

Figure 6: Max-pooling

In classification tasks, once several convolutional and max pooling layers have been applied, usually the output is flattened and is passed through a few fully connected layers to get the final prediction. The following image taken from the MathWorks webpage [5] illustrates a full CNN that makes use of all the techniques explained until now:

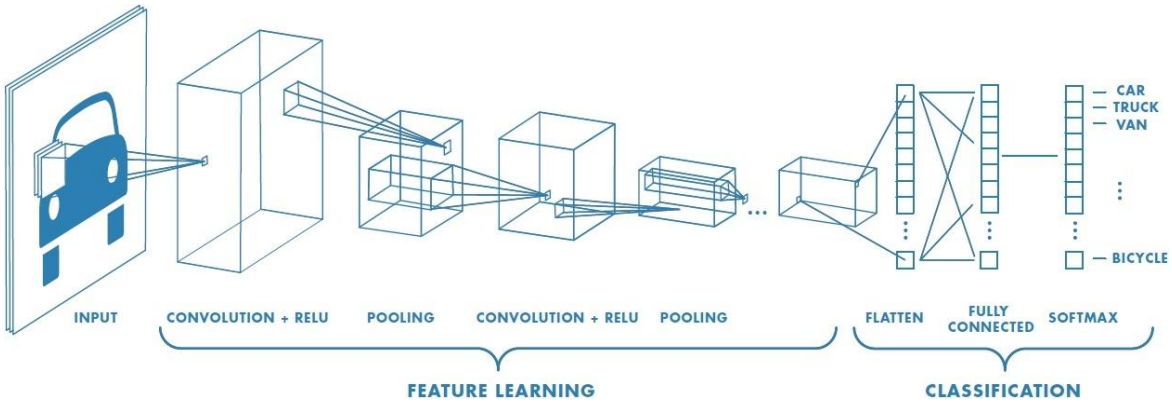

Figure 7: CNN from start to end

2.1.4. Recurrent Neural Networks

Recurrent Neural Networks (RNN) are neural networks that have loops, which means that they have memory. A RNN unit is equivalent to several copies to the same network stacked one after the other. At each time step the network receives two different inputs: The new data and the output of the previous step. The two of them are concatenated and after being passed through the filter, traditionally a tanh activation function is used. The two following images taken from [6] illustrate this:

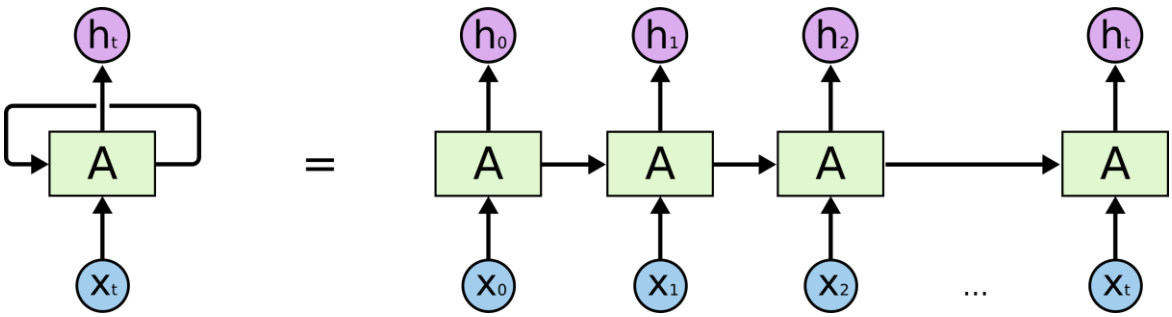

Figure 8: RNN unit

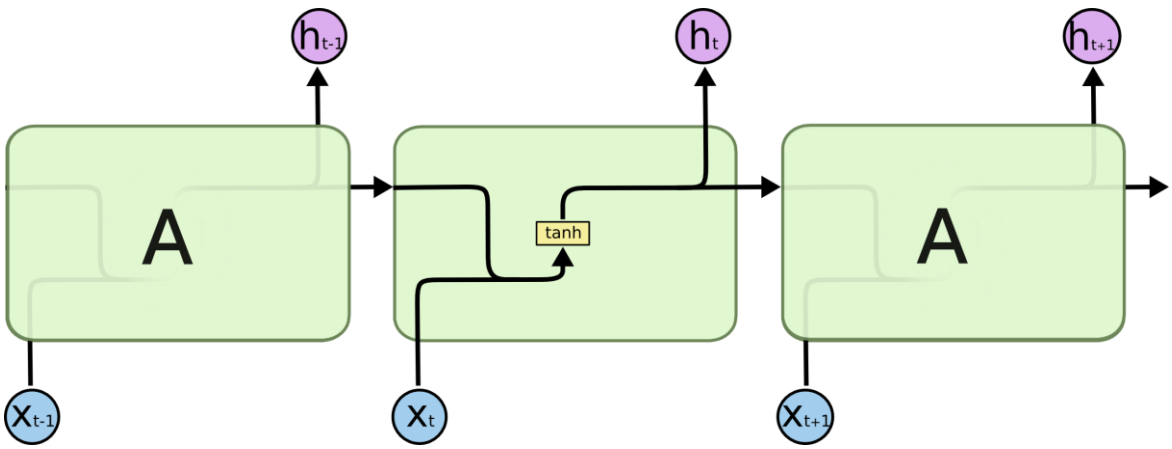

Figure 9: Inside of a RNN unit

Since the length of the sequence can be very large, the backpropagation can become very slow when passing the information through all the time steps. Because of this,

usually a maximum length is set or other techniques different from normal backpropagation are used.

2.2. Introduction to image segmentation

Image segmentation is the process of dividing a giving image into different parts according to a criteria that is learnt by a neural network. Each one of the selected regions is labelled in a certain category (i.e. car, person, TV, door, etc.). Image segmentation has proven to be really useful for different tasks such as locate objects, locate boundaries or even understand scenes, which is a key point to develop advanced AI models.

Broadly speaking, image segmentation can be divided in two different types: Semantic segmentation and instance segmentation. Semantic segmentation tries to label each pixel in an image in a category corresponding to the object to which that pixel belongs (i.e. a pixel belonging to a car will be labelled as car; a pixel belonging to a bed will be labelled as bed, etc.). However, semantic segmentation is not able to distinguish between different objects of the same category. This means that if the image contains a lot of cars without having a separation between them, they will all be labelled as the same entity. On the other hand, image segmentation is just the opposite. It is able to detect individual objects inside an image and label separately, but it does not tell to which class each object belongs.

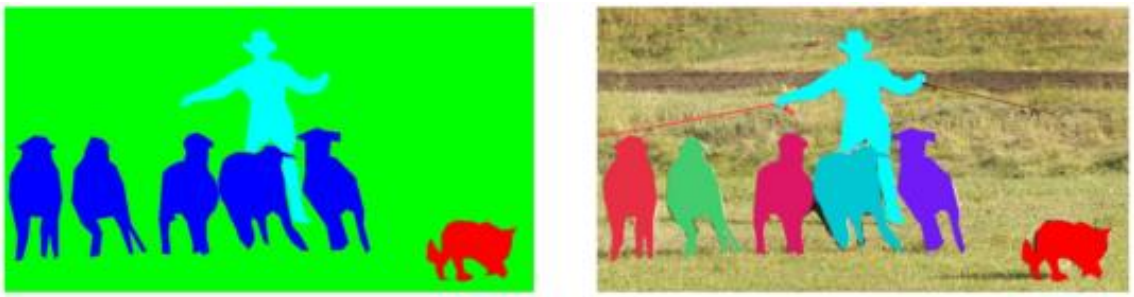

Figure 10: Semantic vs instance segmentation

In the left image, it can be appreciated how the model can differentiate between background, person, dog and sheep, but all the sheep are labeled indistinguishably. In the right image, the model is only detecting objects by their shape without telling which class is each. As it can be quickly understood simply by seeing the images, these two segmentation types are both necessary at the same time to develop powerful scene understanding models.

This work focuses mainly on instance segmentation and tries to use it to help annotators segmenting images to obtain better datasets to train more powerful models.

2.3. Related work

Convolutional Neural Networks (CNN) have proven to be really effective when dealing with images in recent years [7]. In [8] and [9] semantic segmentation is carried out using CNNs for pixel labelling ([8] makes use of superpixels while [9] relies on a skip architecture and deconvolution layers to label each pixel). Because of this increasing necessity of powerful CNN models, many pretrained models that yields very good results have been created. Examples include GoogLeNet [10], AlexNet [7], ResNet [11] or VGG [12]. The VGG, which has been used in this work, is quite easy to implement and yields very good results consuming few resources. It is composed of 11 to 19 convolutional

layers (the one used has 16 layers) and ends with a softmax classifier, as in can be appreciated in Table 2 in [12]. Skip connections has also proven to be very useful. In [9], Long et al. use skip connections to do semantic segmentation over an image. Moreover, since in a VGG the first layers generate information regarding the style of the image while the deepest layers tend to generate semantic information about the image, using skip connections has yielded to the creation of very interesting models such as image style transfer [13].

There are many different image segmentation models that have produced good results and can be used as relevant baselines when doing image segmentation, such as DeepMask [14] or more recently SharpMask, which improves the recall of DeepMask in the COCO dataset by 10-20% [15].

Recurrent Neural Networks (RNN) or, particularly, Long Short Term Memory networks (LSTMs) [6][16] are able to deal efficiently with many problems that need to be solved in several steps, emulating the reasoning human process. Clear examples can be found in machine translation [17] or speech recognition [18].

Moreover, RNNs can also be used for image segmentation. In [19], several layers of RNN do horizontal and vertical sweep across an image to get a good semantic segmentation model. In [20], the authors use stacked layers using a modified version of a CLSTM (Convolutional LSTM) called BDC-LSTM (Bi-Directional Convolutional LSTM) to segment 3-D biomedical images. Another example of image segmentation can be found in [21], where P instances of a CNN that share the same parameters are run sequentially over an image, refining and improving the prediction of a single forward pass through the CNN. In [1], the original Polygon-RNN model in which this project is based is explained. A RNN is applied over different layers of a VGG to predict a polygon outlining an object.

3. Methodology / project development:

3.1. Architecture

At first, before deciding what kind of architecture I should use for my model, I spent some time learning about the polygon-RNN model and the techniques the authors used in it. The model they used is, in fact, composed of two separated models that work as follows:

The first one is in charge of the first point prediction, in other words, the start point of the polygon. It is composed of a VGG-16 from which some of the layers are taken, using skip connections, as the input for two additional convolutional layers that predict the first point as a probability density. This point is then fed into the second model. The second model also uses a VGG-16 and skip connections, but this time the output is connected to an RNN. Also, the RNN takes as an input the first point predicted in the first model. In each time step, the RNN predicts the next point of the polygon. The model has been proved to work very well when evaluated in the Cityscapes dataset.

However, despite offering very good results, the model cannot be easily implemented in other systems since it is divided in two different parts. Due to being split, the system needs to allocate memory for the two parts, which is a bit wasteful since the two of them have a very similar structure based on a VGG. Moreover, since the second part needs the first point prediction which is generated in the first part, the running time is increased.

Due to this problem, the first thing I have taken care of has been creating a single joint model that can be easily implemented in other systems. The final chosen structure is as follows:

First, to preprocess the image to extract its features, the same VGG-16 architecture has been used since it is easy to implement and does not need a lot of resources. Since a VGG is used the input is scaled to a (224 x 224 x 3) image. Skip connections have also been used since they tend to yield better results than simply taking the output of the VGG, as it has been indicated in the literature. The layers used for the skip connections can be found in Figure 11.

Next, the image features are used as an input for the first point prediction part. This part consists of two convolutional layer and two fully connected layers. The first convolutional layer is directly connected to the image features and tries to predict features correspondent to an edges map. The second one is connected to the edges features and tries to predict features correspondent to a vertices map. These two layers are followed by a ReLU non-linear function. The first fully connected layer takes the edges features as an input and tries to predict the edges. The second one takes the vertices features as an input and tries to predict the vertices. A sigmoid function is applied to the output of each of the fully connected layers, which allows this output to be interpreted as a probability distribution over the individual pixels. The way they are connected and the size of each layer are depicted in Figure 12.

The final part is the polygon prediction, which takes as an input the predicted first point and the previously computed features of the image. As in the original work, a two layer Convolutional LSTM [22] network have been used. The reason is that using a convolutional approach yields way better results when processing images since convolutional networks preserve better the spatial information. Using a LSTM instead of a normal RNN makes the network able to remember long term information, which is very

useful when dealing with long polygons. This part outputs one point for each step in the RNN in the default mode of operation. The way the layers are connected can be depicted in Figure 12. Batch normalization is used after each layer.

In the following image, the structure of the VGG-16 is shown (the original image has been taken from [23]):

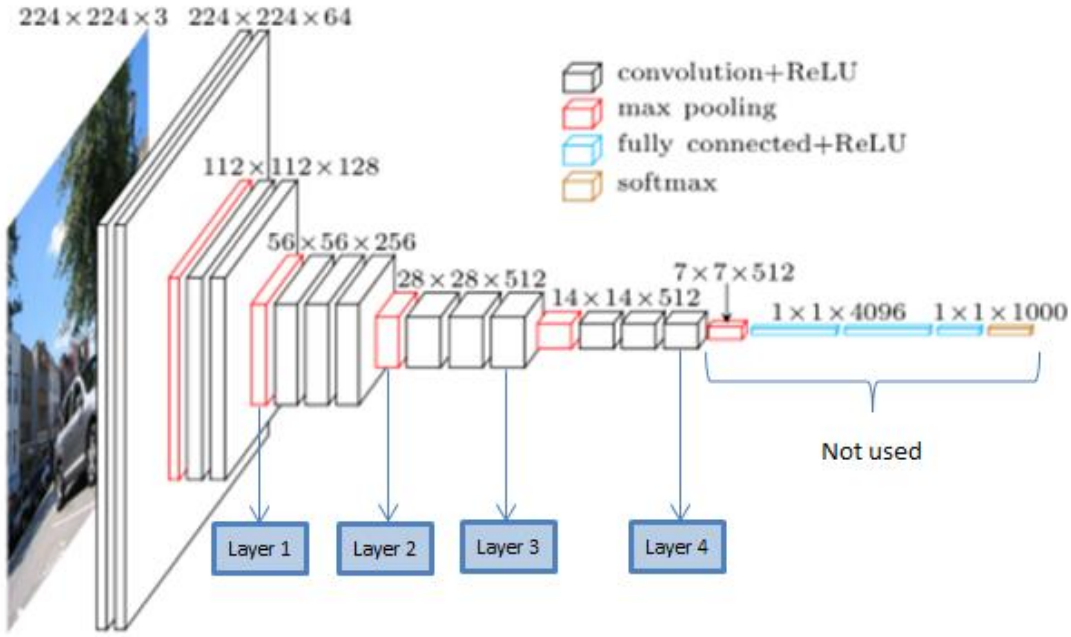

Figure 11: VGG-16 architecture

In the following image, the architecture of the joint model is depicted:

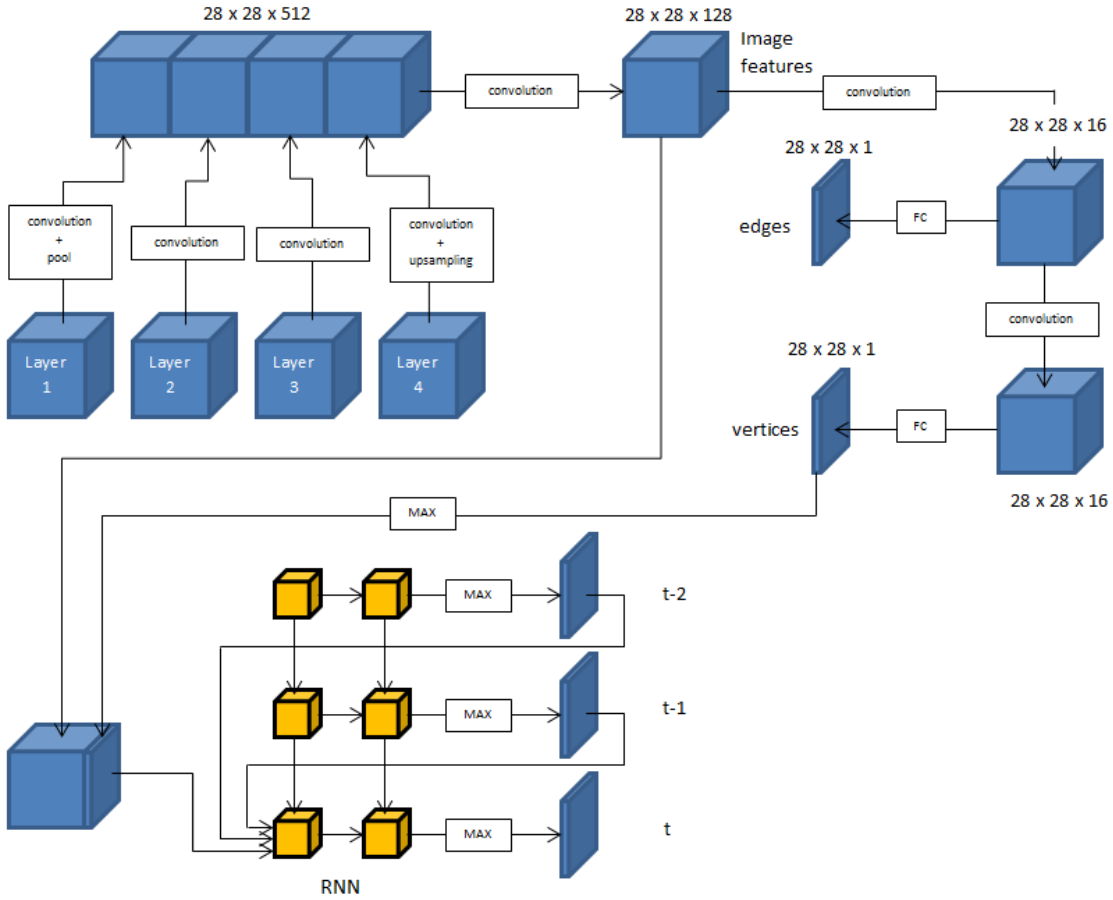

Figure 12: Architecture of the joint model

All the convolutions use a 3 x 3 filter size and stride 1. The edges layer that is apparently not connected to anything is used during training to compute one of the terms of the loss.

I also tried to fuse this model with the Yolo model [24][25]. This would allow the model not to take crops of the image with size 224 x 224, but take a whole image and use the Yolo predictions as the inputs for this model. Ideally, after obtaining the Yolo prediction, it would be expanded by a 15% in every direction to minimize the noise of the prediction and then resize it to 224 x 224. What is more, Yolo is extremely fast and it can be run at 67 frames per second, what means that the model would not suffer from a slowdown. However, after testing it, I decided not to implement it in the final model. While Yolo is a very powerful and real time object detection model, it is a bit noisy and tends to predict some false positives. The purpose of this work is to make an object segmentation model that is able to be implemented in any place to help annotators do their work. What is required in the model is mostly to have a high precision in its predictions, so it would be meaningless to obtain the bounding boxes through Yolo if the annotator has to be checking them and deleting the ones that are incorrect. Since the process of selecting the bounding boxes is not very time-consuming, the decision was to leave this part in the hands of the annotator. Some examples of Yolo predictions can be found in the results section of this work.

3.2. Training

To train the network, it was run in default prediction mode predicting one point at every time step. The optimizer used has been the ADAM optimizer with a learning rate of $1e-4$ and the rest of parameters as in the default implementation by TensorFlow. Regularization loss has also been used for the weights, using a L2 loss and parametrized by a regularization strength of $1e-5$. The batch size used is 8.

The loss function is as follows:

$$C = a \cdot \sum_{batch} \sum_{i=1}^{784} \left(-c \cdot y_{e_i} \cdot log\left(\widehat{y_{e_i}}\right) - \left(1 - y_{e_i}\right) \cdot log\left(1 - \widehat{y_{e_i}}\right) - c \cdot y_{v_i} \cdot log\left(\widehat{y_{v_i}}\right) - \left(1 - y_{v_i}\right) \cdot log\left(1 - \widehat{y_{v_i}}\right)\right) + b \cdot \sum_{batch} \sum_{t} \sum_{i=1}^{785} \left(-y_{v_{t,i}} \cdot log\left(\widehat{y_{v_{t,i}}}\right) - \left(1 - y_{v_{t,i}}\right) \cdot log\left(1 - \widehat{y_{v_{t,i}}}\right)\right) + \lambda \cdot RL$$

The first line corresponds to the loss of the first point prediction, the second line corresponds to the RNN loss and the third line is the regularization loss. Here, $\hat{y}$ is the prediction, y is the target, RL is the regularization loss, the subindex e references the edges, the subindex v references the vertices and the subindex t is the time step. The subindex i runs over the pixels of the prediction (28 x 28, with and additional position in the RNN to detect the end of the polygon). As it can be appreciated, there are multiple cross entropy terms weighted by several parameters (a, b and c).

During training, instead of using directly the predicted vertices when feeding the RNN, the first point is chosen as the closest ground truth to the predicted point. This creates a natural order for the target ground truth polygon, which is reordered to adapt to this "random" choice for the first point. For the next points, directly the next ground truth point is the one which is fed into the next RNN time step regardless of the prediction.

As for the parameters, simply by seeing the loss function it is clear that the cost associated with the RNN is roughly $t/2$ times higher than the cost associated to the first vertex prediction. Because of this, a has been included to be able to increase that cost (otherwise, the entire training over the VGG will be due to the RNN only). The parameter b has been added for further control over this issue. Moreover, the ground truth for the edges and vertices is a binary map where the majority of values are zeros. Since there are very few ones, the parameter c was added to increase the cost of the ones (only in the first point prediction). The results in this documents have been obtained using $a = 3$, $b = 1$ and $c = 3$.

Before starting to train the model in the way that has just been explained, the model was pretrained for a few epochs using only the RNN part ($a = 0$ and $b = 1$), which should give results similar to the ones obtained with the original model. The reason of doing that was to test if the RNN part was behaving as it should.

3.3. Testing

All the tests have been done using the cityscapes dataset using the cities of Frankfurt, Lindau and Munster. The measures were taken using a prediction script and then

evaluating the accuracy as the Intersection Over Union over the ground truth with an evaluation script.

While testing in the default mode, the most probable point in the prediction for the first vertex and in the prediction in the RNN is the one that is fed to the next step.

A beam search algorithm has also been implemented. Beam search have been used is many applications involving RNNs with very good results, especially in machine translation [26] and sequence to sequence models in general [27]. While testing in the beam search mode, the network tracks the most probable k sequences and then outputs the most probable one (where k is a parameter that can be freely set in the options of the model). The algorithm is as follows:

After getting the CNN features and the first point prediction, instead of taking the top 1 best point, the top k points are stored. Then, the CNN features are replicated k times and for each of these copies one of the k points is used as the first point. Moreover, the log-probability of each one of the chosen points is stored in a vector. Reached this point, the model has initialized the k paths. Once the RNN has finished predicting the probability map for the next point, for each of the k already existing paths, the top k most probable points are stored again. This way, we have now k^2 possible paths to follow. The mean log-probabilities for each one of the k^2 paths are computed and stored in a temporal variable. Among the k^2 paths, the top k with the highest mean log-probability are chosen and the log-probability vector is updated with the log-probabilities of those k paths. Then, the model predicts again the next k^2 most probable points and the process is repeated until the maximum length of the polygon is reached.

It is important to say that, when a polygon has already been closed, its mean log-probability is not updated, using always the one that it had in the previous time step. Moreover, that path does not contribute with k different new paths but with only one: The same closed polygon path that already exists. That means that if, for example, among the k paths, two of them have already predicted an end of sequence (EOS), the number of paths that are going to be evaluated in the beam search are not k^2 but $k \cdot (k-2)+2$. This is the main reason of why the model uses the mean log-probability and not the log-probability itself. Otherwise the shortest paths will always be preferred since the log-probability is always a negative number.

The following image illustrates the process. The final chosen path is drawn with green arrows. As it can be appreciated, its mean log-probability is -0.167, which is higher than the leftmost path, -0.2, and the rightmost path, -0.225. The paths that end with a red line are the ones that have not been chosen for the next step.

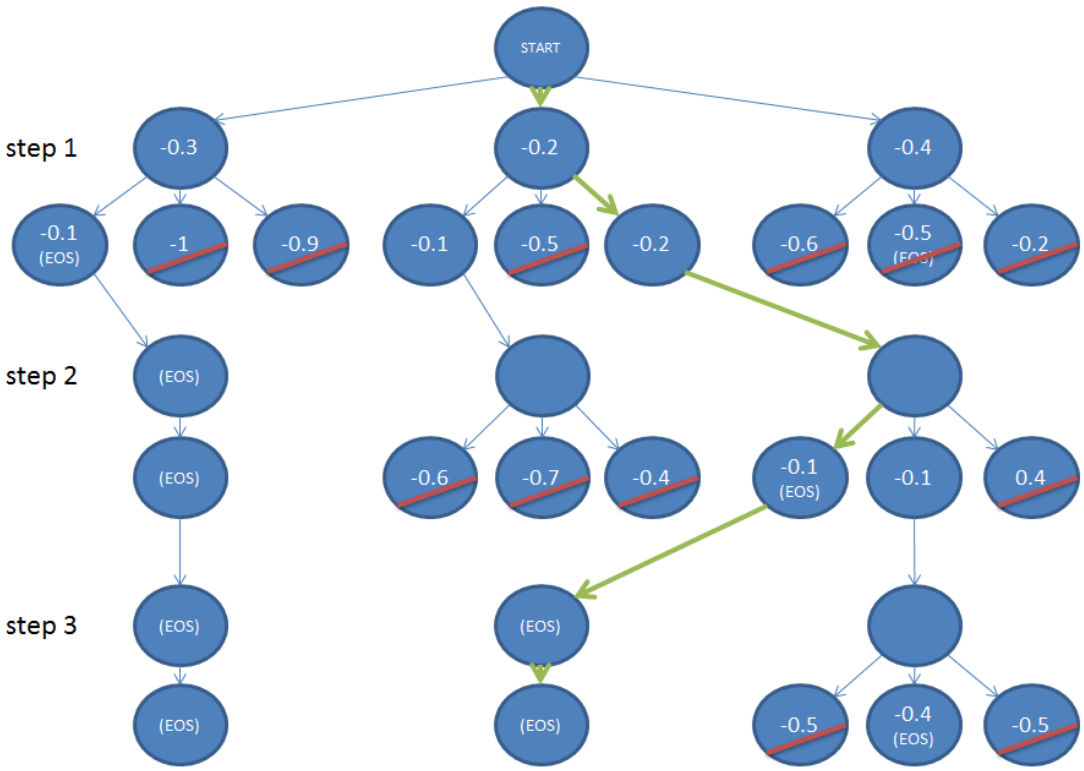

Figure 13: Beam search

3.4. Other approaches

After getting the results for the model that has just been explained (these results can be found in the results section), I thought about improving the accuracy by redesigning a little bit the architecture and the way the training was done. To avoid the model getting a decrease in the accuracy with respect to the original model when running in default mode prediction, I added one more convolutional layer between the image features and the edge features. This layer has 128 3x3 filters and a ReLU activation function. The output volume is then 28 x 28 x 128. This model was trained in two stages. In the first one, only the RNN part was trained ($a = 0, b = 1$), using as first point the first ground truth point. This first train was done end to end. In the second stage, only the first point part of the network was taken into account ($a = 3$, $b = 0$, $c = 3$). In this second stage, the backpropagation of the gradient is stopped before reaching the VGG. This way, since the first point part does not train end to end, the RNN part is avoiding getting noisier because of the double training that was done before in the plain joint model at a cost of having a little lower accuracy in the first point. The results for this extended model are shown in the results section. Despite showing a good performance when running in default mode, it is not as good as the plain joint model when running in beam search mode, so it was decided to keep the plain joint model in the final system.

3.5. Annotation tool

Once trained and tested, the model has been implemented in the LabelMe annotation tool. The annotation tool is too big to be explained in detail in this thesis, and since I am not the one who developed it, I will only make a brief introduction about how does it work before starting to talk about the modifications.

LabelMe is a web tool that can be used in many browser engines and has a friendly environment that make easy to create annotations over images and save them in an xml file. To annotate an object the process is the following: First, the annotator needs to create a bounding box around the object to annotate. Next, the tool zooms in the section of the image that has been selected. The crop that is zoomed is always a square, so if the selected bounding box is not a square (typical case), some treatment is done with its boundaries to make it a square without deforming the image. Once the bounding box has been zoomed in, the annotator can start to create a polygon around the object by simply clicking in the image and creating points. To close the polygon, the annotator has to left-click in the first introduced point or to right-click in any part of the image. Finally, the annotator has to introduce the class of the object to finish the annotation.

The annotations can be deleted and modified by dragging points whenever the annotator wants.

The following image shows the annotation tool:

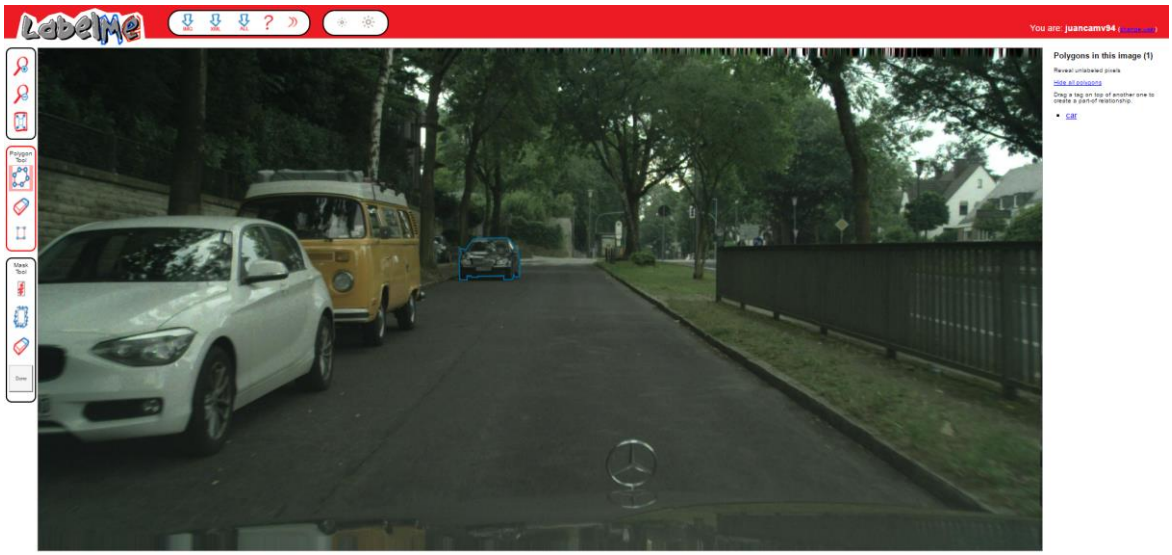

Figure 14: LabelMe annotation tool

After a few days understanding at a code-level how the tool worked, I started to make some modifications to fuse it with the model. The final system is composed of two different parts: The client and the server. The LabelMe annotation tool works as the client part of the system while all of the processing regarding the model is done remotely (server part).

The process works as follows: After selecting the bounding box in the annotation tool, the coordinates are written in an xml file along with a number that acts as a flag and the path of the image that is displayed in the web tool. In the server, a python script is periodically reading the flag in that file. Once the flag becomes 1, the model is run remotely over the image that is pointed out by the path stored in the xml file using the bounding box coordinates to compute the crop. Once finished, the points of the polygon are written back to the same xml file and the flag is overwritten with a 0. During this process, the annotation tool is periodically checking the xml file to detect when the flag change from 1 to 0. Once a 0 is detected, the points of the predicted polygon that are stored in the xml file are loaded by the annotation tool. The polygon is displayed and the tool asks the annotator to choose the class of the object to finish the annotation. The annotator can

choose four different actions: Accept the polygon, which stores it in the xml annotation file, delete polygon, which allows the user to make it manually from scratch, undo close polygon, which deletes the last segment of the polygon, allowing the user to continue manually from there, and modify polygon, which allows the annotator to click and drag points to modify the prediction.

In the original LabelMe annotation tool, the option to modify the prediction is only enabled when the polygon is already stored in the final xml file. Maintaining this would be annoying for the annotator since, if he wants to modify the polygon, he would need to first save it, then click in it again, select adjust polygon, make zoom manually, modify the polygon, accept it and zoom out. Another functionality was added to make this process simpler when LabelMe is working with the model. The polygon can be now modified without previously having to accept it in the already zoomed crop. Once the corrections are finished, the polygon is accepted and stored in the xml annotation file for the first time. This way, the modify polygon feature becomes many times faster and more user-friendly.

Another new functionality that has been added to the model is the “smart polygon” button. While the polygon is being modified in the zoomed crop, this button is displayed in the upper-left corner of the canvas where the image is. When the user clicks this button, the points that have been modified are sent back to the server. The way to do that is, again, using the xml file the same way as it was explained before, with the difference that, this time, the flag written is a 2 instead of a 1. When the server detects a 2 in the flag, it reads the points that have been written in the xml file and it runs the model using all of them as first points. After this process, the server has computed n different polygons (being n the number of points that have been modified before clicking the smart polygon button). All of these polygons are stored back into the xml file and the flag is overwritten with a 0. The tool computes a final polygon to be displayed taking parts of the predicted polygons and merging them. This merging algorithm, which is the core part of the smart polygon feature, works as follows:

The tool takes the two first polygons and computes a forward semipath for the first one and a backward semipath for the second one. A semipath is a fragment of the polygon that always starts at the first point of the first taken polygon and ends in the first point of the second taken polygon. Of course, in general, the first point of the second polygon is not a point that belongs to the first polygon. The distance of this point respect the edges of the first polygon is computed and one of the vertices of the first polygon is substituted for the first vertex of the second polygon. That way the start and end conditions of the semipath can be fulfilled. The next figure illustrates the process of creating the forward and backward semipaths:

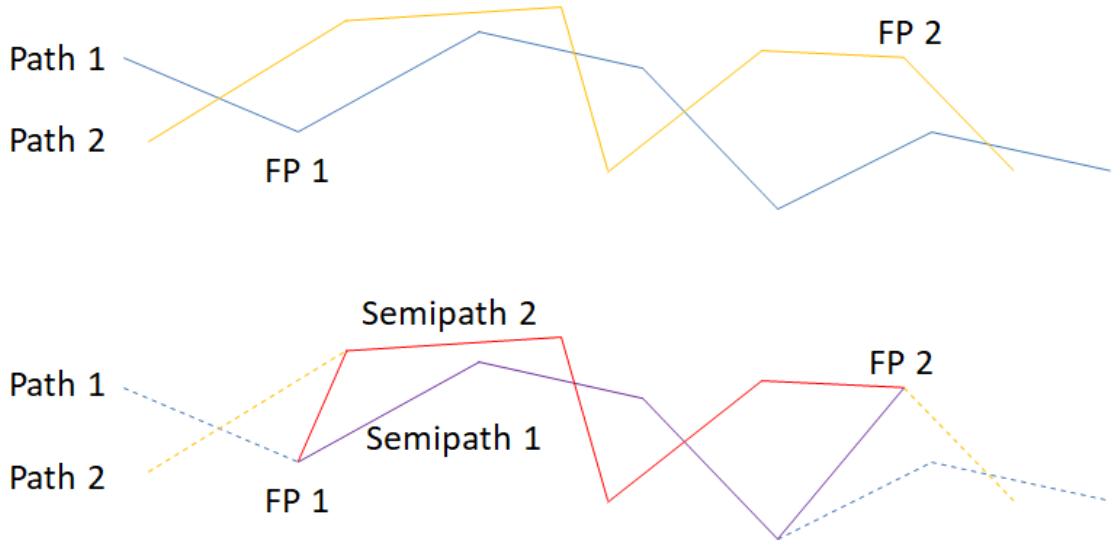

Figure 15: Semipath creation

Once these two semipaths are computed, they are merged using a very simple algorithm. Starting from one of the extremes, the second point of each semipath is chosen and the mid-point is computed for that diagonal. Next, the model chooses one of the two possible diagonals: the one that is formed with the second point of the first semipath and the third point of the second semipath or the one that is formed with the third point of the first semipath and the second point of the second semipath. The shortest one is chosen and the mid-point is computed again. The tool continues doing that process of taking diagonals and computing the midpoints until reaching the end of the semipaths. The next figure illustrates two semipaths and the computed merged path in red.

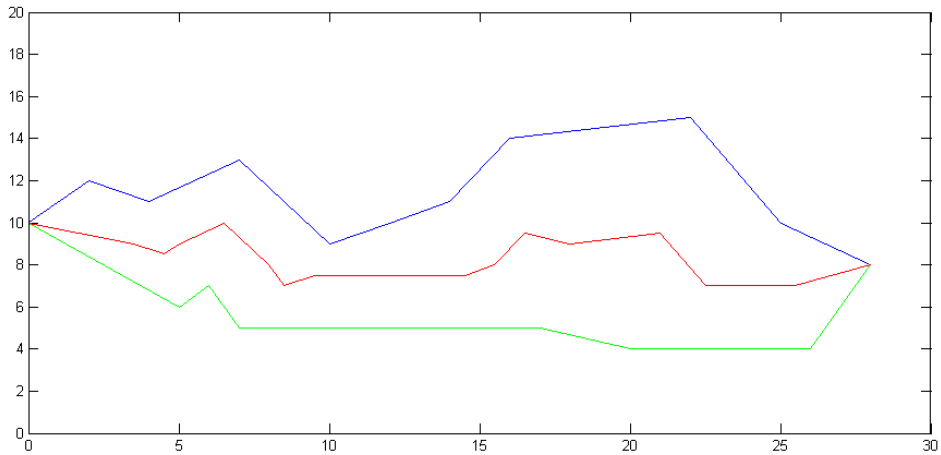

Figure 16: Merged path computation

This algorithm is a suboptimal one since it does not ensure the final path to be between the two semipaths, as it can be seen in the image below. However, it is extremely fast and the results are very close to an optimal solution.

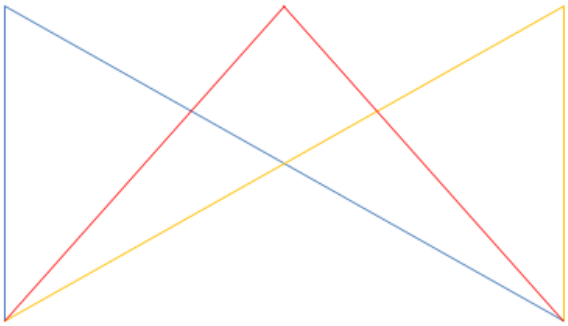

Figure 17: Merged path outside the boundaries of the semipaths

The whole process is then repeated using the second and third polygons, then using the third and fourth polygons, etc. The algorithm finishes when merging the semipaths of the last and first polygons. As it can be appreciated, the corrected points have been maintained and only the points in the middle have been recomputed.

Sometimes the annotator is going to modify points that are compressed in a small region of the polygon. This usually means that the parts that are far from the modified points are correct and that the annotator does not desire to change them. The algorithm also takes this into account. If there is a wide region of the polygon where the annotator has not made any correction (for example, a set of 30 consecutive points), the first predicted polygon is maintained in that region. The semipaths to reach that region are computed using one of the new predicted polygons and the first predicted polygon. The next image illustrates this:

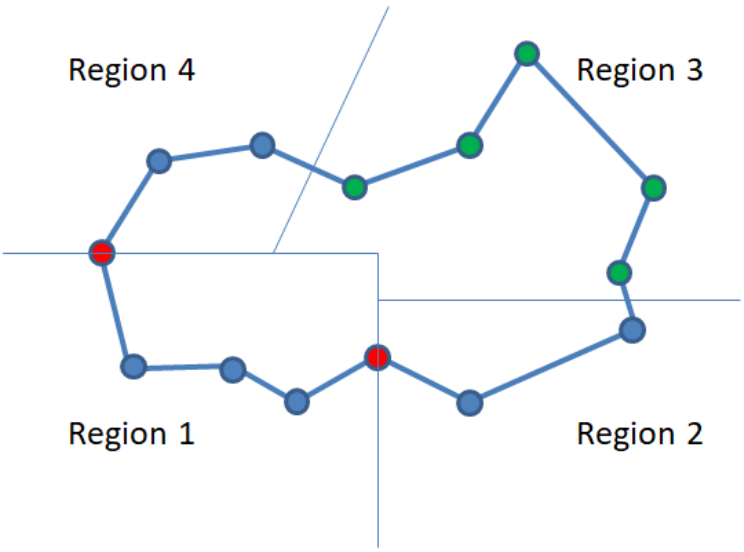

Figure 18: Regions in the merging algorithm

The points in red are the ones that the annotator has modified while the points in green are the ones that are far from the modified points (a distance of 2 points has been used as example). The points in blue are points that are going to be modified. In region 1, the new path will be the merging of the semipaths of the two new predicted polygons. In region 2, the new path will be the merging of the semipath of the new polygon predicted

using the bottom red vertex as the starting point and the semipath of the first predicted polygon (the one in the image). Region 4 is the same than region 2, but using in this case the semipath of the polygon predicted using the leftmost red vertex as the starting point. The points in region 3 will be maintained as they are.

Due to the process of computing the mid-points of the diagonal, the number of points of the final polygon is increased by a factor of 1.5. This adds a lot of redundancy to the polygon, so a filter is applied to remove this redundancy. If the next point is at the same position of the previous point, it is deleted. Also, if several consecutive points are aligned, the ones that are in the middle are removed since they do not contribute to the form of the polygon.

A second filter is applied to reduce the noise. If there is a very small region with a lot of consecutive points in it, they are removed and substituted be their mean point. This filter can erase some self-intersecting loops that would make the quality of the final polygon far worse.

Finally, a third filter is applied to further reduce the noise. This final filter removes all the points that lie inside the polygon, deleting large self-intersecting parts.

4. Results

All the results have been taken from evaluating different models in the Cityscapes dataset. The fraction of the dataset where the model has been tested in contains a total of 9784 instances taken from 500 images in the validation dataset, composed of the cities of Frankfurt, Lindau and Munster. The classes that the model is able to segment are “bicycle”, “bus”, “person”, “train”, “truck”, “motorcycle”, “car” and “rider”. The number of instances of each class can be viewed in the following table:

	bicycle	bus	person	train	truck	motorcycle	car	rider
instances	1129	98	3239	23	93	148	4517	537

Table 1: Number of instances per class in the test set

The accuracy has been measured using the Intersection Over Union (IOU) metric, which is a very well-known and vastly used metric to compute accuracies when dealing with object segmentation in images. It is a very simple metric that can be used regardless of the method used to obtain the predictions. The IOU can be defined as the area where the prediction and the ground truth are overlapping over the area resulting of the union of the prediction and the ground truth. The following image illustrates how this metric works:

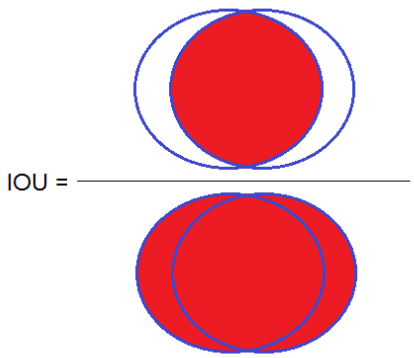

Figure 19: Intersection Over Union

The average of the IOU over each class and the total average IOU over all instances have been computed for different models.

First, the original model was run to have a baseline to compare with the rest of the results of the joint model. The acquired mean IOU over all classes has been as high as 58.89% (usually, a prediction with an IOU over 50% is considered a good prediction). When running the joint model in its default mode, the accuracy has decreased by more than 3%, obtaining an accuracy of 55.14%. It continues to be a good prediction but it is not as good as in the original model. To check why this was happening I did two more tests: In the first one I used the closest ground truth to the predicted point as the first point (just like it was done while training). In the second one I took directly the first ground truth point as the first point (which is equivalent to not use the first point part of the model at all). The results are the following:

Model	Bicycle	Bus	Person	Train	Truck	Motorcycle	Car	Rider	Mean
Original	50.35	59.74	64.05	53.89	60.05	51.94	70.17	60.94	58.89
Joint	46.35	59.65	62.64	51.06	48.17	47.29	65.91	60.02	55.14
Closest GT	48.70	59.27	63.16	53.17	50.20	50.21	67.10	60.33	56.52
First GT	48.75	61.89	63.45	55.73	55.02	48.69	68.40	59.62	57.70

Table 2: Comparison between the original and the joint model in default mode

The decrease in accuracy in the joint model choosing the first ground truth point with respect to the original model can be easily understood taking into account that the joint model tries to train end to end the first point part and the RNN part. Training the first point through the VGG as well is undoubtedly making the first point prediction more accurate, but it is also making the RNN noisier. However, the further decrease when choosing the closest ground truth is not fully understood. The model seems to be having a preference for certain first points. My guess is that, since before training the joint model the RNN part was pretrained for a few epochs separately using the first ground truth as the first point, the model could have learned some kind of dependence in this first point. One possibility is that the annotator that made the ground truth could have a preference annotating some objects starting from similar points (the head of the objects correspondent to the person or rider class or the car roof for the car class for example). This could create this kind of dependence in the RNN, lowering down the IOU a bit when selecting a random first point.

When running the network in beam search mode, the model receives a big boost. Just by using the most simple beam search algorithm (k=2), the accuracy is already increasing a lot, reaching 58.64% of mean accuracy, being only a little bit behind the original model. Using higher values of k boost the accuracy even further, reaching values of 59.32% for k=3 and 59.94% for k=4, which are a bit higher than the original model. Increasing k even further can continue increasing the accuracy, but the model starts to become too slow.

The results for the extended joint model described in section 3.4 are a little bit disappointing. When the model is run in default prediction mode, the accuracy is a bit higher than in the original model, reaching 55.95%, which is the expected outcome taking into account that not training the first point part end to end has prevented the RNN part to become noisy. However, when the model is run in beam search mode, the accuracy does not increase as much as when using the plain joint model. The accuracy obtained is 56.07% for k=2, 57.04% for k=3 and 57.72% for k=4. My guess is that, when training end to end the first point, the VGG learn some kind of edges dependences that introduce some noise in the RNN part and lower the accuracy when running in default mode. However this noise is probably useful for the beam search to find the most probable k points, making the beam search in the plain joint model better than in the extended joint model.

Model	Bicycle	Bus	Person	Train	Truck	Motorcycle	Car	Rider	Mean
Original	50.35	59.74	64.05	53.89	60.05	51.94	70.17	60.94	58.89
Joint default	46.35	59.65	62.64	51.06	48.17	47.29	65.91	60.02	55.14
Joint beam k=2	50.21	65.00	64.86	53.85	52.24	52.59	68.71	61.67	58.64
Joint beam k=3	51.32	63.60	65.56	51.89	57.04	51.83	70.22	62.93	59.32
Joint beam k=4	53.66	63.50	66.35	54.59	55.68	43.49	70.52	62.75	59.94
Extended default	47.61	54.08	63.06	49.65	56.57	51.34	66.53	58.79	55.95
Extended beam k=2	50.20	52.90	64.11	42.27	58.03	52.31	67.75	60.96	56.07
Extended beam k=3	50.64	55.66	64.75	46.15	57.06	52.95	67.64	61.49	57.04
Extended beam k=4	51.57	54.91	64.98	49.88	58.10	52.75	67.99	61.58	57.72

Table 3: Summary of results

Other results that have not been obtained by me can be found in the Polygon-RNN paper [1]. There, they reported an accuracy of 56.45% with DeepMask, 60.21% with SharpMask and 61.40% with the Polygon-RNN model. The difference between their and mine results when using the Polygon-RNN model is because they did not take only the first best point but the five best points. They ran the model over these five points to obtain five paths and they took the most probable one.

The following sets of images show some qualitative results where it can be seen how the different models are predicting the polygons for all the instances in an image. The colors used to fill the polygons are red, green and blue. Other colors are caused by the overlapping of polygons.

Figure 20: Qualitative results 1

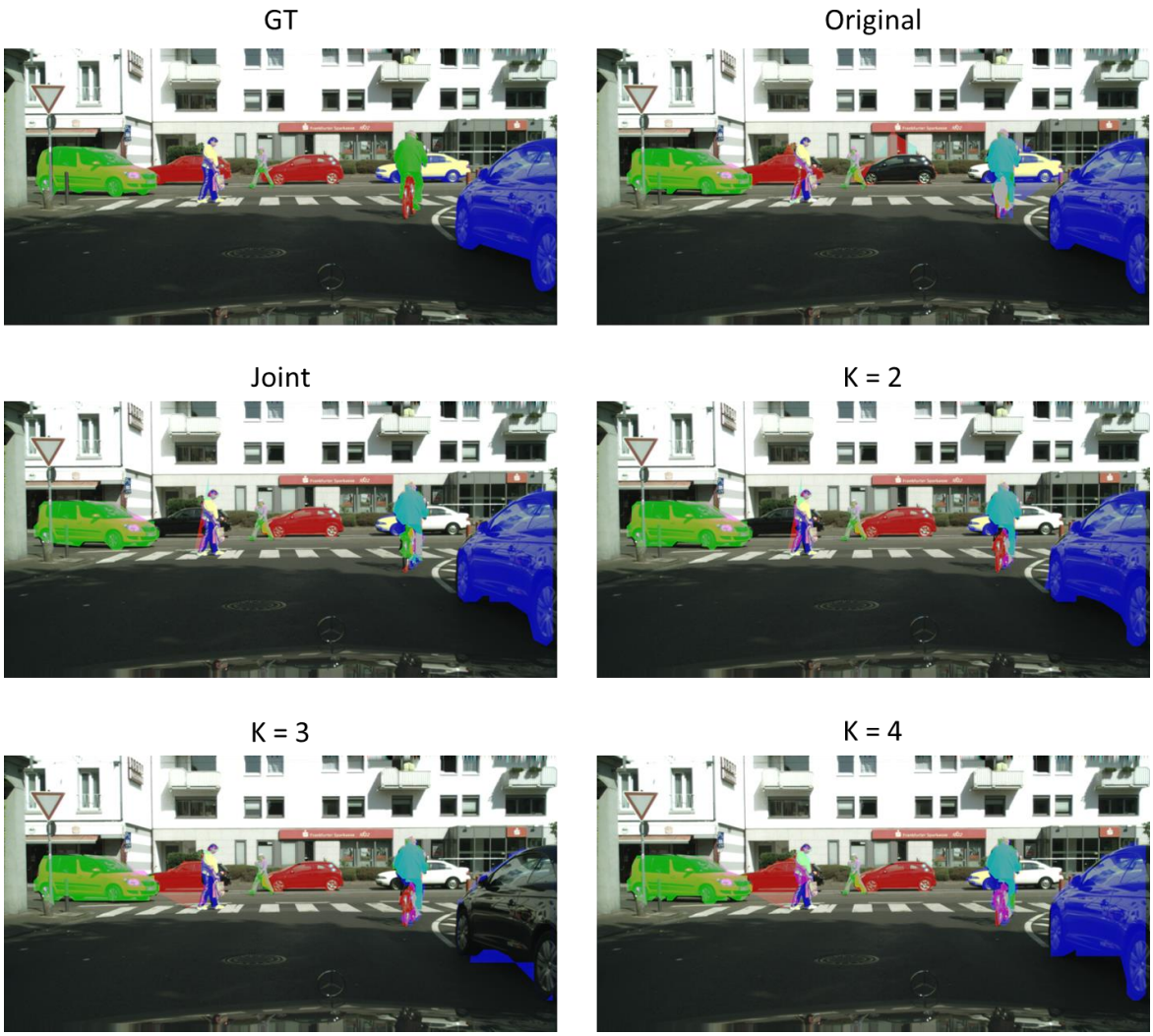

Figure 21: Qualitative results 2

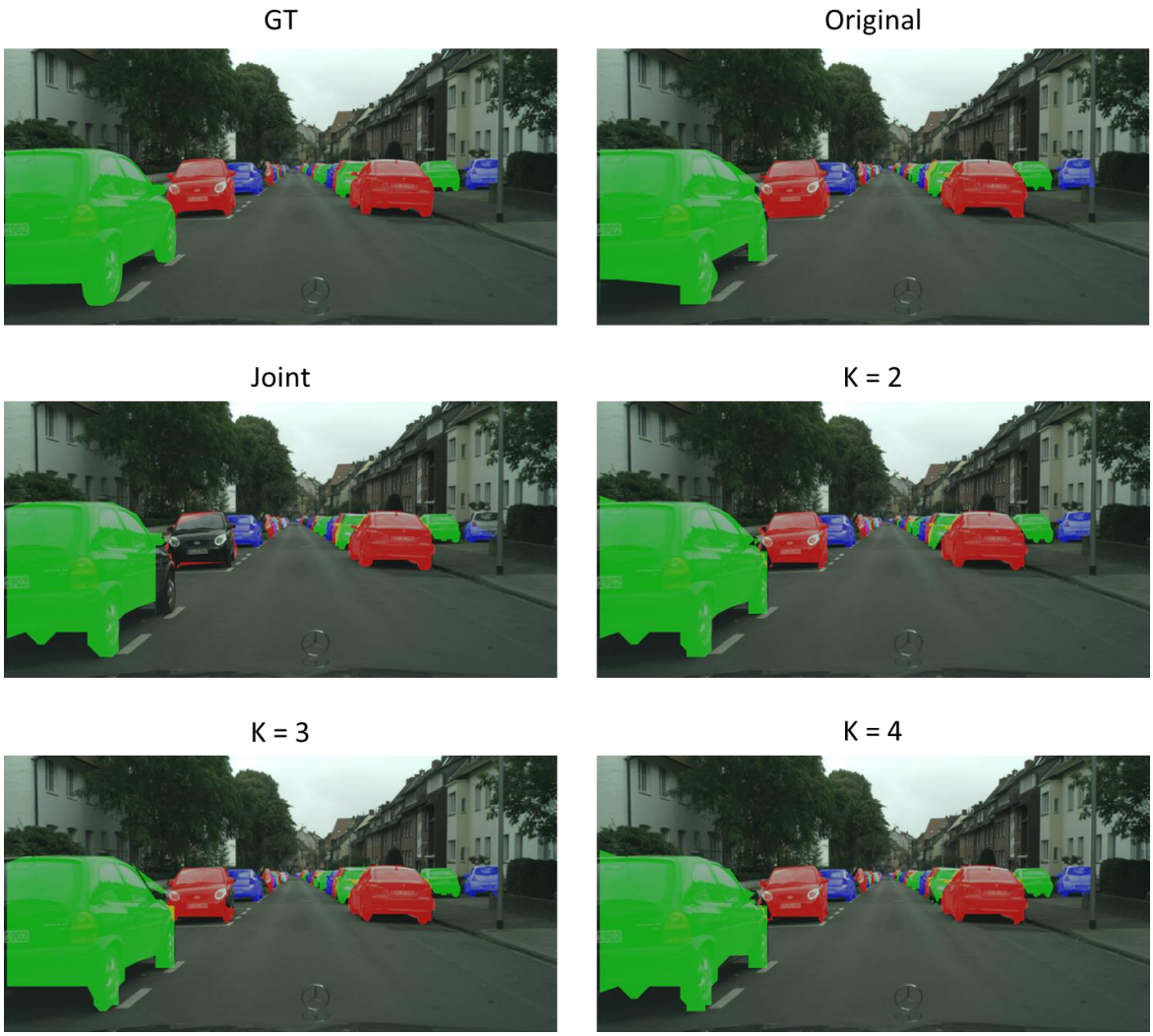

Figure 22: Qualitative results 3

It can be appreciated that the models with higher values of k tend to perform better when segmenting the objects. The models with k=3 and k=4 show slightly better results than the original model. However, as it can be appreciated in the k=3 model in Figure 21 when watching at the car in the right side of the image, a model with a higher k does not guarantee that it is going to work better in each and every case.

The joint model has been implemented in the LabelMe annotation tool just as it was explained in section 3.5. After selecting the bounding box and waiting for a few seconds, the prediction is displayed in the screen, where you can label the object and modify or delete the annotation. The next images illustrate the procedure and how the final result looks like. These images were taken using the joint model in beam search mode with k=2. In the first one, the bounding box is being selected. In the second one the automatic prediction is displayed. In the third one, the annotation has being slightly modified by using the modify polygon feature. In the fourth one the accepted prediction is shown in the whole image.

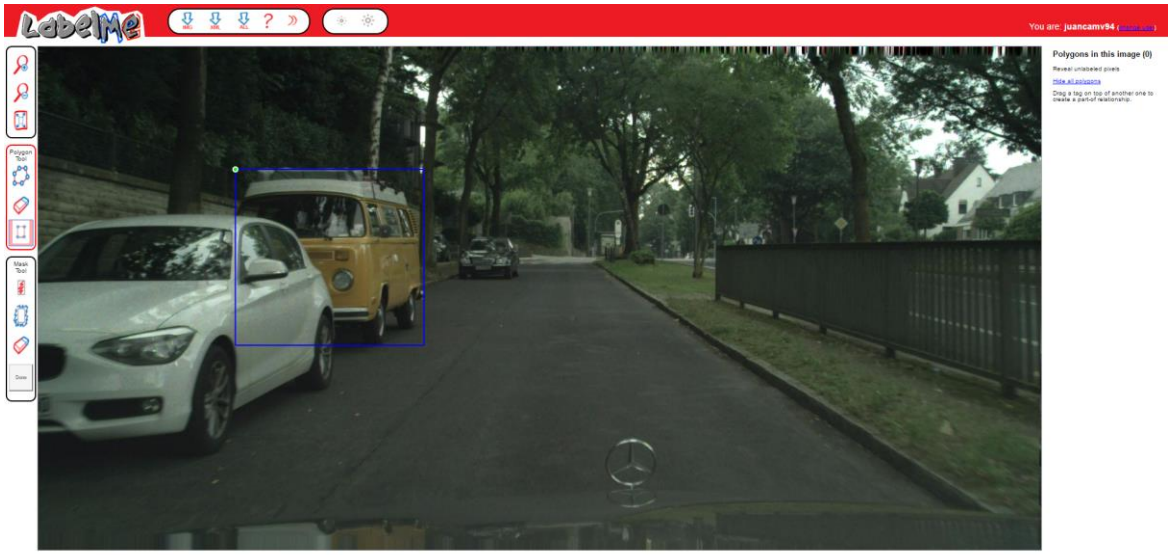

Figure 23: Selecting bounding box in LabelMe

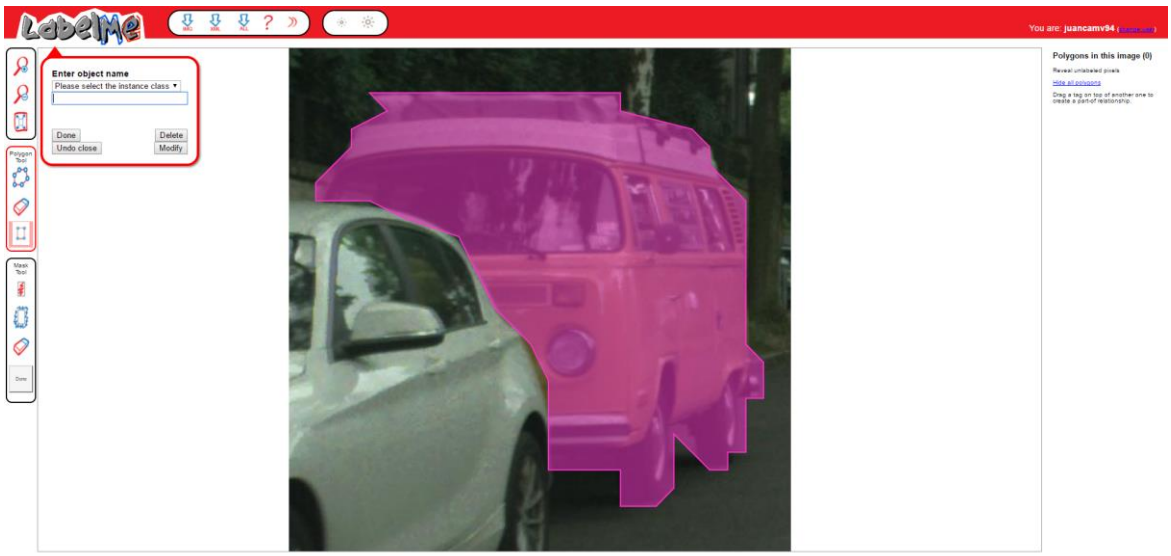

Figure 24: Getting prediction in LabelMe

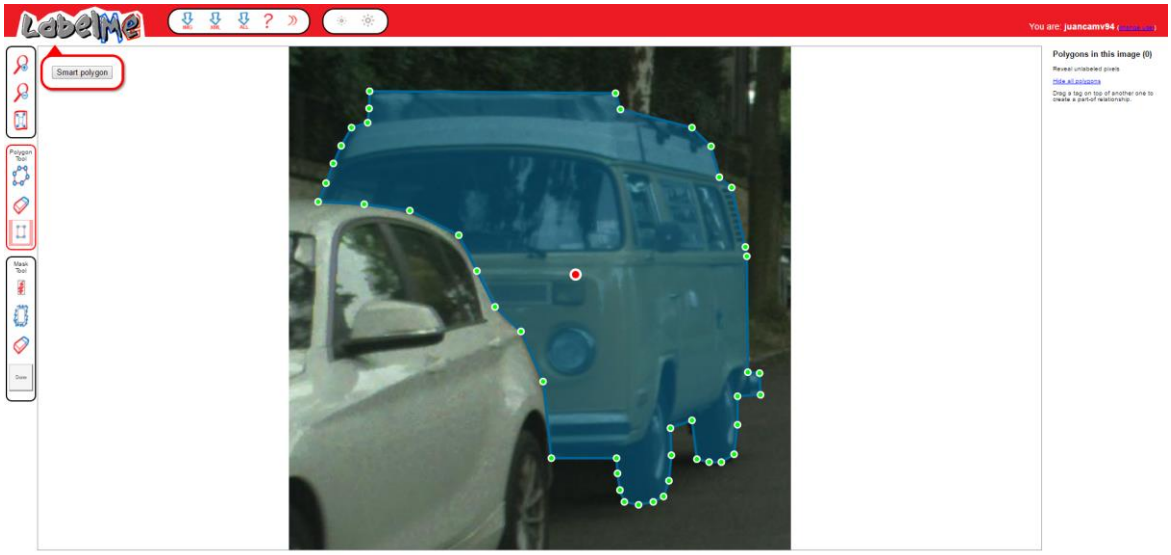

Figure 25: Modify polygon in LabelMe

Figure 26: Final annotation result in LabelMe

The smart polygon feature is also working properly and producing good results. The main drawback of the automatic predictions is that the model is predicting a polygon using a resolution of 28 x 28, while the zoomed crop has a resolution of 1024 x 1024. Because of this a quantization effect appears which causes the predictions not to be completely accurate.

To finish the presentation of the results, I will show some of the crops and predictions that resulted when applying the Yolo model over the image. Since I do not have a ground truth for the annotations in the Yolo predictions, I cannot provide a quantitative result. Because of this, the following images can only be interpreted qualitatively. The crop rows show the Yolo boxes and the mask rows show the model prediction over the correspondent Yolo box as a binary image.

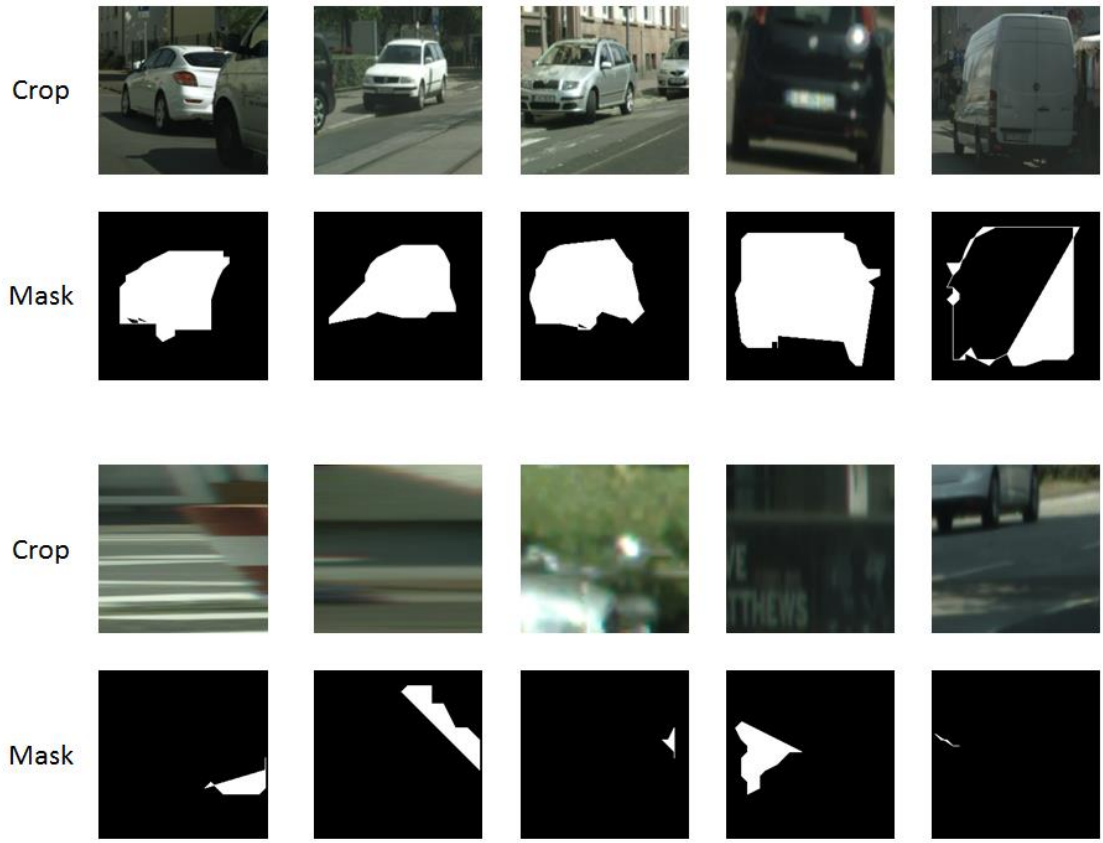

Figure 27: Yolo predictions

The first two rows show acceptable Yolo boxes. The first and last one are particularly good. The second one is a little too far from the car and the third and fourth do not cover completely the car. However, all of them are very reasonable and the model should be able to use them without problems since a human can easily recognize the object that is supposed to be segmented. The predicted masks show that, in fact, the model is able to recognize the objects. The last image, which shows a perfect Yolo box, is also showing a wrong prediction in the model: the polygon is self-intersecting and going around the car several times.

The second two rows show completely wrong and unacceptable boxes. In the first four not even a human can guess which object should be segmented in them, so the model is making a completely random prediction. In the last one is easy for a human to see that the box is trying to select a car, but it is so badly centered that the model does not seem to understand it. Because of predictions like these the Yolo model was decided not to be implemented in the final model.

5. Budget

The purpose of this thesis is not to create a physical prototype. Moreover, all the software that has been used is free for use and open-source. Because of this, the only thing that can be taken into account is the hours spent in the different tasks. The salary of a junior engineer has been estimated as 18€ per hour after checking in several webpages such as https://www.glassdoor.ca/Salaries/ (since I did the thesis in Canada, I considered a normal Canadian salary converted to euros). The budget divided per tasks can be summarized in the next table:

Task	Hours	Cost (€)
Lear Python	7	126
Lear about CNN and RNN	14	252
Learn TensorFlow	14	252
Understand previous model	35	630
Design CNN	119	2142
Design RNN	49	882
Program the networks	168	3024
Combine the networks	14	252
Test the network	42	756
Understand annotation tool	56	1008
Change annotation tool	112	2016
Debug annotation tool	105	1890
Total	735	13230

Table 4: Budget

In the table above, it has been estimated a mean of 7 hours of work in weekdays in the period between February 2nd and June 27th. The time for each task has been estimated according to the Gantt diagram.

6. Conclusions and future development:

This should include your summary, conclusions and recommendations.

In this work several models and approaches have been explored to be able to create a powerful tool for object segmentation based in neural networks that can be easily implemented in other systems. Out of those models, a single joint model with an IOU as high as 59.94% capable of predicting the object boundaries in a recurrent manner was chosen as the best one and was implemented in an online annotation tool to make it possible to be used by annotators. Other features such as being able to modify the prediction even before saving it were also implemented in the annotation tool to make the annotation procedure even simpler and more comfortable for the annotator. The qualitative results of the final system have proven to be very good.

To make the model better, several approaches could be taken. For example, it would be a good improvement if predictions such as the one depicted in the first row and final column of Figure 27 could be avoided. Some hard constrains can be implemented in the process of choosing the next point in the RNN to solve the problem. Also, other CNN networks such as ResNet could be implemented instead of the VGG to check the performance. It is not a good idea though to try to increase the performance by just increasing the complexity of the network since a low test time is required for the model to work in the annotation tool.

That said, the accuracy of the predictions in the annotation tool could be improved by adding a few deconvolution layers in the model after getting the prediction for the vertices. This way the quantization effect described in the results could be avoided, making the predictions more accurate.

Bibliography:

[1] L. Castrejon, K. Kundu, R. Urtasun, and S. Fidler, “Annotating Object Instances with a Polygon-RNN,” 2017.

[2] C. Debao, “Degree of approximation by superpositions of a sigmoidal function,” *Approx. Theory its Appl.*, vol. 9, no. 3, pp. 17–28, 1993.

[3] “When Parallelism Gets Tricky: Accelerating Floyd-Steinberg on the Mali GPU - Graphics & Multimedia blog - Graphics & Multimedia - ARM Community.” [Online]. Available: https://community.arm.com/graphics/b/blog/posts/when-parallelism-gets-tricky-accelerating-floyd-steinberg-on-the-mali-gpu. [Accessed: 06-Jun-2017].

[4] “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed: 06-Jun-2017].

[5] “Convolutional Neural Network - MATLAB & Simulink.” [Online]. Available: https://www.mathworks.com/discovery/convolutional-neural-network.html. [Accessed: 06-Jun-2017].

[6] “Understanding LSTM Networks—colah’s blog.” [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv. Neural Inf. Process. Syst.*, pp. 1–9, 2012.

[8] C. Couprie, L. Najman, and Y. Lecun, “for Scene Labeling,” *Pattern Anal. Mach. Intell. IEEE Trans.*, vol. 35, no. 8, pp. 1915–1929, 2013.

[9] E. Shelhamer, J. Long, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 640–651, 2017.

[10] C. Szegedy, W. Liu, Y. Jia, and P. Sermanet, “Going deeper with convolutions,” *arXiv Prepr. arXiv 1409.4842*, pp. 1–9, 2014.

[11] S. Wu, S. Zhong, and Y. Liu, “Deep residual learning for image steganalysis,” *Multimed. Tools Appl.*, pp. 1–17, 2017.

[12] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” pp. 1–14, 2014.

[13] L. A. Gatys, A. S. Ecker, and M. Bethge, “Image Style Transfer Using Convolutional Neural Networks,” *2016 IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 2414–2423, 2016.

[14] P. O. Pinheiro, R. Collobert, and P. Dollar, “Learning to Segment Object Candidates,” pp. 1–10, 2015.

[15] P. O. Pinheiro, T. Y. Lin, R. Collobert, and P. Doll??r, “Learning to refine object segments,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9905 LNCS, pp. 75–91, 2016.

[16] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks,” pp. 1–9, 2014.

[18] H. Soltau, H. Liao, and H. Sak, “Neural Speech Recognizer: Acoustic-to-Word LSTM Model for Large Vocabulary Speech Recognition,” 2016.

[19] F. Visin, M. Ciccone, A. Romero, K. Kastner, K. Cho, Y. Bengio, M. Matteucci, and A. Courville, “ReSeg: A Recurrent Neural Network-based Model for Semantic Segmentation,” 2015.

[20] J. Chen, L. Yang, Y. Zhang, M. Alber, and D. Z. Chen, “Combining Fully Convolutional and Recurrent Neural Networks for 3D Biomedical Image Segmentation,” no. Ii, 2016.

[21] P. Pinheiro and R. Collobert, “Recurrent convolutional neural networks for scene labeling,” *Proc. 31st Int. Conf. …*, vol. 32, no. June, pp. 82–90, 2014.

[22] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. Wong, and W. Woo, “Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting,” pp. 1–12, 2015.

[23] “A brief report of the Heuritech Deep Learning Meetup #5 |.” [Online]. Available: https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/. [Accessed: 06-Jun-2017].

[24] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” 2015.

[25] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” 2016.

[26] M. Freitag and Y. Al-Onaizan, “Beam Search Strategies for Neural Machine Translation,” 2017.

[27] S. Wiseman and A. M. Rush, “Sequence-to-Sequence Learning as Beam-Search Optimization,” 2016.

Glossary

ADAM: Adaptative Moment Estimation.
CLSTM: Convolutional Long Short Term Memory.
CNN: Convolutional Neural Network.
EOS: End Of Sequence.
FC: Fully Connected.
IOU: Intersection Over Union.
LSTM: Long Short Term Memory.
ReLU: Rectified Linear Unit.
RNN: Recurrent Neural Network.
SGD: Stochastic Gradient Descent.