Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs

Athanasios Raptakis **Rene Snajder**

Abstract

In this project the task of semantic image segmentation with Deep Learning is studied. Our goal is to reproduce state-of-art techniques like the Deep Convolutional Nets, Atrous Convolution and Fully Connected CRFs presented in [1,2]. First, we discuss the intuition behind 'atrous convolution'[6], convolution with upsampled filters. The advantage of Atrous convolution is that it allows to explicitly control the resolution at which feature responses are computed within Deep Convolutional Neural Networks. It also allows us to effectively enlarge the field of view of filters to incorporate larger context without increasing the number of parameters or the amount of computation[1]. Second, we use the technique of Atrous spatial pyramid pooling (ASPP)[7,8] to robustly segment objects at multiple scales. ASPP performs Atrous convolution in an incoming convolutional feature layer with filters at multiple sampling rates and effective fields-of-views. In this way objects and image context at multiple scales are captured. In the end, we use a fully connected Conditional Random Field (CRF)[4,5] in order to improve the localization of object boundaries at the final DCNN layer , which improves the localization performance both qualitatively and quantitatively. Two Semantic Segmentation experiments where run and their results were evaluated during this project by calculating the iou metric. The first experiment is about semantic segmentation based on the PASCAL VOC2012 Segmentation competition [15,17]. The second experiment is about semantic segmentation where individual parts of objects are detected based on VOC2010 Person Part dataset [12]. While on our experiments the calculated iou accuracy is far worse than the original Deeplab publication the quality of the segmentation is overal good. Also the proposed crf method increases our iou the expected $2 - 5\%$. We believe that the discrepancy between our performance and the original publication happens for three main reasons. The first reason is that the authours of the original paper only roughly outline the proposed architecture without mentioning the details of their implementation. For example they do not mention how deep each atrous convolutional layer goes or if they used Dropout during training etc. Second we think that given addequate time to run the experiments our calculated iou and the generalisation performance of our DCNN would be much closer to the original (the original publication runs at least 10000 epochs while our own experiments run only for only a few tens of epochs due to time and hardware limitations).

Contents

Author's Contribution

Both authors, Athanasios Raptakis (AR) and Rene Snajder (RS), contributed equally to the practical work. As for the manuscript, the contriutions were:

- Abstract: AR
- Introduction and Motivation: AR
- Experiments: AR and RS
- Results: RS
- Discussion: RS

1 Introduction and Motivation

Deep Convolutional Neural Networks (DCNNs) [1,2,9] have greatly increased the performance of computer vision systems on a broad array of high-level problems, including image classification, object detection and semantic Segmentation. In the past conputer vision systems where based on hand crafted features made by specialised and experienced engineeres[2]. These methods typically wouldn't generalise well and a high level of expertise and knowledge on the task at hand was needed in order to create an efficient and robust algorithm. With the introduction of DCNNs those features are learned by the system. The problem is formulated as an optimisation problem which minimises a Loss function given some properly annotated training examples by using back probagation algorithm. A DCNN [2] consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a DCNN typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers. DCNNs trained in an end-to-end manner have delivered strikingly better results than systems relying on hand-crafted features. Essential to this success is the built-in invariance of DCNNs to local image transformations, which allows them to learn increasingly abstract data representations [3]. While this invariance is desirable for classification tasks it is unfavorable for dense prediction tasks such as semantic segmentation[3]. The main motivation for combining Deep Convolutional Nets, Atrous Convolution and Fully Connected CRFs as a solution to the semantic segmentation problem is to tackle the three main problems of classic DCNNs namely (1) reduced feature resolution, (2) existence of objects at multiple scales and (3) reduced localization accuracy due to DCNN invariance.

1. The first problem of classic DCNNs is caused by the repeated combination of max-pooling and downsampling ('striding') performed at consecutive layers of DCNNs [1]. This results in feature maps with significantly reduced spatial resolution. In order to overcome this problem atrous convolution and is used[6]. Atrous convolution is equivalent with a simple convolution with a filter which is equivalent to inserting holes between nonzero filter taps. Atrous convolution can effectively enlarge the field of view without increasing the number of parameters or the amount of computation.

2. The second challenge is caused by the existence of objects at multiple scales [1]. In order to overcome this problem we can apply multiple Atrous convolution filters with complementary effective fields of view. This allows us to capture objects as well as useful image context at multiple scales. This techinique is called "atrous spatial pyramid pooling" (ASPP)[7,8].

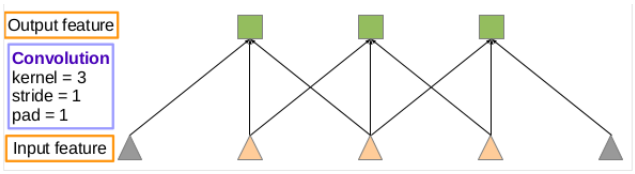
3. The third challenge is that DCNNs designed for object classification require invariance to spatial transformations [1]. This unfortunatly limites the spatial accuracy of a DCNN in dense semantic segmentation tasks.In order to tackle this problem, the Deeplab researchers have used a fully-connected Conditional Random Field (CRF) to capture fine details after the final layer [4,5].

1.1 Atrous Convolution and Atrous Spatial Pyramid Pooling

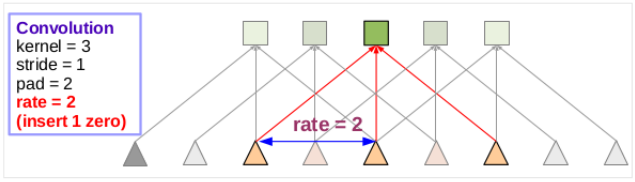
Typically DCNNs use spatially small convolution kernels [1,9] (typically 3×3) in order to minimize computation and the number of parameters used. Atrous convolution [6] with rate r introduces $r-1$ zeros between consecutive filter values, enlarging the kernel size of a $k \times k$ filter to $k_e = k + (k-1)(r-1)$ without increasing the number of parameters or the amount of computation. In this way the algorithm can control the field-of-view and finds the best trade-off between accurate localization (small field-of-view) and context assimilation (large field-of-view). Atrous convolution allows us to compute the responses of any layer at any desirable resolution. Considering one-dimensional signals first, the output $y[i]$ of atrous convolution of a 1-D input signal $x[i]$ with a filter $w[k]$ of length K is defined as:

$$y[i] = \sum_{k=1}^{K} x[i + r \cdot k] w[k]. \tag{1}$$

The rate parameter r corresponds to the stride with which we sample the input signal. Standard convolution is a special case for rate $r = 1$.



(a) Sparse feature extraction



(b) Dense feature extraction

Figure 1: Atrous convolution in 1-D. (a) Sparse feature extraction with standard convolution on a low resolution input feature map. (b) Dense feature extraction with atrous convolution with rate $r = 2$, applied on a high resolution input feature map.

Atrous Convolution is easily expanded in 2-D[1]. Given an image, we assume that we first have a downsampling operation that reduces the resolution by a factor of 2, and then perform a convolution with a kernel. This is equivalent to obtaining responses at only 1/4 of the image positions. In order to compute responses at all image positions we can convolve the full resolution image with a filter 'with holes'. This is equivalent to upsampling the original filter by a factor of 2, and introduce zeros in between filter values. As a result the effective filter size increases and we only need to take into account the non-zero filter values. As a result, both the number of filter parameters and the number of operations per position stay constant. This method allows for explicit control over the spatial resolution of neural network feature responses.

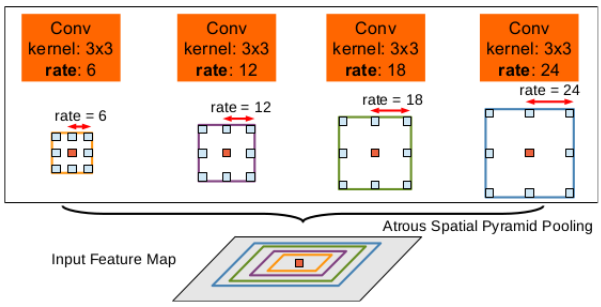


Figure 2: Multiple parallel 2-D Atrous Convolution filters with different rates can be combined in order to determine the class of the center pixel[1].

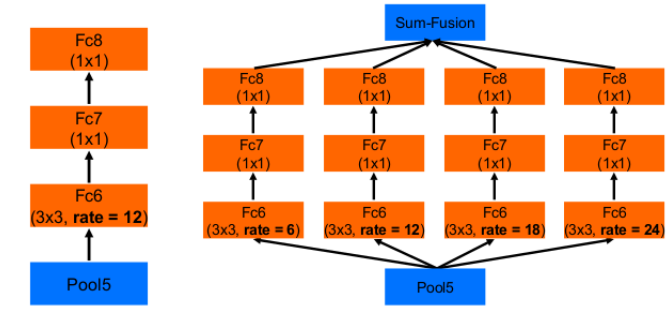


Figure 3: DeepLab-ASPP[1,7,8] adds multiple Atrus Convolution filters with different rates to capture objects and context at multiple scales[1]. A hybrid net approach can be employed simply by adding the Atrous layers after Pool5 of a pretrained VGG-16 net and changing stride of Pool5 to 1. Then the Atrus layers are fine-tuned by training all the hybrid network together.

In practice, we can take advantage of Atrus convolution by replacing convolutional layers with Atrus layers [1]. This would result in a DCNN output of arbitrarily high resolution. For example, in order to double the spatial density of computed feature responses in the VGG-16[9], we can set the stride to 1 for the last pooling or convolutional layer which decreases the resolution, and then replace all subsequent convolutional layers with atrous convolutional layers having rate $r = 2$. If we follow this method in all our network we could compute feature responses at the original image resolution. Alternatively, in order to avoid the high computational cost a hybrid approach can be taken like the one used in this project by following the original DeepLab publication.

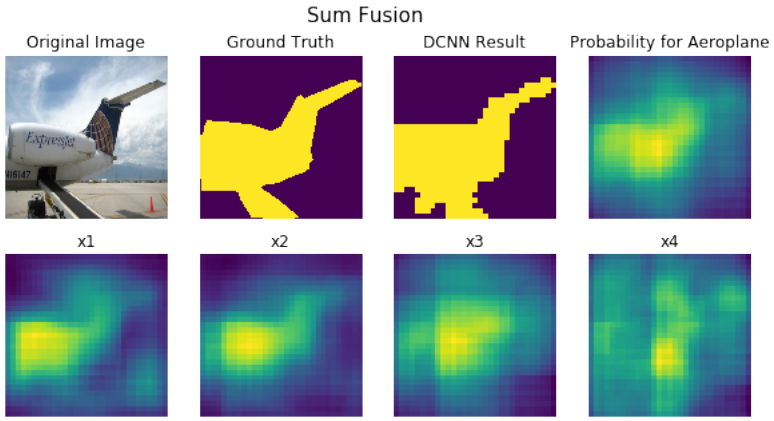


Figure 4: Illustration of ASPP and Sum fusion operation[7,8]. x1, x2, x3, x4 are the individual responses of the parrallel Atrus layers with rates = 2, 4, 8, 12 respectively. We can observe that while x1 captures local information, x4 captures more context. The Output image is the sum of the four individual responses.

1.2 Structured Prediction with Fully-Connected Conditional Random Fields for Accurate Boundary Recovery

In order to improve multi-class image segmentation and labeling tasks scientists often use Conditional Random Fields[4]. The advantage of the Conditional random fields is that they take into acount the structure of the input image thus making a structured predition. Conditional random fields usually belong to two main categories depending on the connectivity of their graph. The first category is the region-level CRF which takes into acount only local interaction of small regions, for example dense pairwise connectivity. The second category assumes pixel-level models. These models are considerably larger and have only permitted sparse graph structures. Fully connected CRF models

are defined on the complete set of pixels in an image. The resulting graphs have billions of edges, making traditional inference algorithms impractical. At [4] a highly efficient approximate inference algorithm for fully connected CRF models is proposed in which the pairwise edge potentials are defined by a linear combination of Gaussian kernels.

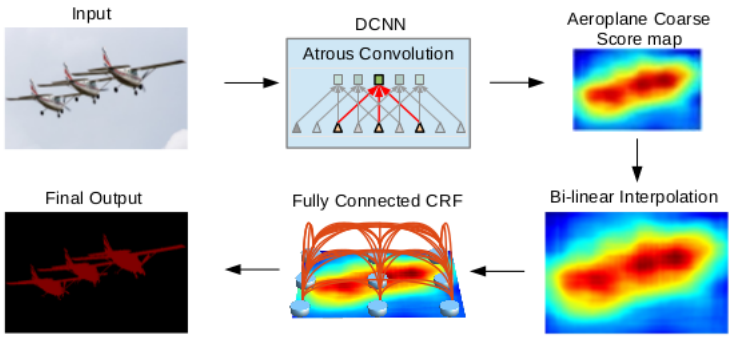


Figure 5: Illustration of the proposed DeepLab pipeline. The result of a DCNN (ex. VGG-16 [9]) is given as an input to four parallel atrous convolution layers to reduce the degree of signal downsampling (from 32x down to 8x). Then after ASPP and sum-fusion[7,8], bilinear interpolation enlarges the feature maps to the original image resolution. A fully connected CRF is then applied to refine the segmentation result and better capture the object boundaries.

1.2.1 The Fully Connected CRF Model

Consider a random field X defined over a set of variables $\{X_1, ..., X_N\}$. The domain of each variable is a set of labels $L = \{l_1, l_2, ..., l_k\}$. Consider also a random field I defined over variables $\{I_1, ..., I_N\}$. I ranges over possible input images of size N and X ranges over possible pixel-level image labelings. I_j is the color vector of pixel j and X_j is the label assigned to pixel j. A conditional random field (I, X) is characterized by a Gibbs distribution

$$P(X|I) = \frac{1}{Z(I)} exp(-\sum_{c \in C_G} \Phi_c(X|I)) \tag{2}$$

where $G = (V, E)$ is a graph on X and each clique c in a set of cliques C_G in G induces a potential φ_c [24]. The Gibbs energy of a labeling $x \in L^N$ is $E(x|I) = \sum_{c \in C_G} \varphi_c(x_c|I))$. The maximum a posteriori (MAP) labeling of the random field is $x^* = argmax_{x \in L^N} P(x|I)$. For convenience we will omit the conditioning and use $\psi_c(x_c)$ to denote $\varphi_c(x_c|I)$. In the fully connected pairwise CRF model, G is the complete graph on X and C_G is the set of all unary and pairwise cliques. The corresponding Gibbs energy is

$$E(x) = \sum_i \psi_u(x_i) + \sum_{i<j} \psi_p(x_i, x_j), \tag{3}$$

where i and j range from 1 to N . The unary potential $\psi_u(x_i)$ is computed independently for each pixel by a classifier that produces a distribution over the label assignment x_i given image features. The unary potential incorporates shape, texture, location, and color descriptors. Since the output of the unary classifier for each pixel is produced independently from the outputs of the classifiers for other pixels, the MAP labeling produced by the unary classifiers alone is generally noisy and inconsistent. The pairwise potentials have the form:

$$\psi_p(x_i, x_j) = \mu(x_i, x_j) \underbrace{\sum_{m=1}^{K} w^{(m)} k^{(m)}(f_i, f_j)}_{k(f_i, f_j)} \tag{4}$$

where each $k^{(m)}$ is a Gaussian kernel $k^{(m)}(f_i, f_j) = exp(-\frac{1}{2}(f_i - f_j)^T \Lambda^{(m)}(f_i - f_j)$, where vectors f_i and f_j are feature vectors for pixels i and j in an arbitrary feature space, $w^{(m)}$ are linear

combination weights, and μ is a label compatibility function. Each kernel $k^{(m)}$ is characterized by a symmetric, positive-definite precision matrix $\Lambda^{(m)}$, which defines its shape.
For multi-class image segmentation and labeling we use contrast-sensitive two-kernel potentials, defined in terms of the color vectors I_i and I_j and positions p_i and p_j :

$$k(f_i, f_j) = w^{(1)} \underbrace{exp(-\frac{|p_i - p_j|^2}{2\theta_\alpha^2} - \frac{|I_i - I_j|^2}{2\theta_\beta^2})}_{\text{(appearance kernel)}} + w^{(2)} \underbrace{exp(-\frac{|p_i - p_j|^2}{2\theta_\gamma^2})}_{\text{(smoothness kernel)}} \tag{5}$$

The appearance kernel is inspired by the observation that nearby pixels with similar color are likely to be in the same class. The degrees of nearness and similarity are controlled by parameters θ_α and θ_β. The smoothness kernel removes small isolated regions [25]. A simple label compatibility function μ is given by the Potts model, $\mu(x_i, x_j) = [x_i \neq x_j]$. It introduces a penalty for nearby similar pixels that are assigned different labels.

1.2.2 Efficient Inference in Fully Connected CRFs

According to [4] we can perform efficient inference of the fully connected CRF by using Mean field approximation algorithm. Mean field approximation[4] to the CRF is an iterative message passing algorithm for approximate inference.Briefly, each iteration of Mean Field Approximation algorithm performs a message passing step, a compatibility transform, and at last a local update. Both the compatibility transform and the local update run in linear time and are highly efficient. The computational bottleneck is message passing. Mean field approximation is based on the following key methods in order to solve the inference problem efficiently:

1. Message passing in the CRF can be performed using Gaussian filtering in feature space.[4]
2. The permutohedral lattice is used to whitten and transform the feature space.[4]
3. In the transformed feature space the high-dimensional convolution can be separated into a sequence of one-dimensional convolutions along the axes of the lattice[5].

The above enable us to utilize highly efficient approximations for high-dimensional filtering, which reduce the complexity of message passing from quadratic to linear, resulting in an approximate inference algorithm for fully connected CRFs that is linear in the number of variables N and sublinear in the number of edges in the model[4,5].

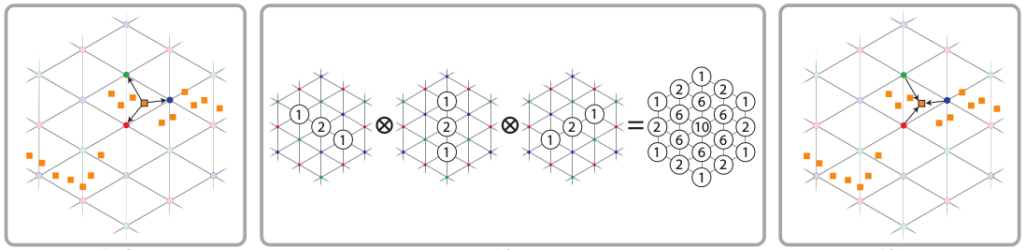


Figure 6: To perform a high-dimensional Gaussian filter using the permutohedral lattice, first the position vectors are embedded in a hyperplane. Then, each input value splats onto the vertices of its enclosing simplex using barycentric weights. Next, lattice points blur their values with nearby lattice points using a separable filter. Finally, the space is sliced at each input position using the same barycentric weights to interpolate output values.[5]

2 Experiments

In this section we will describe our own re-implementation of the architecture, our experiments with various datasets, as well as design decisions, improvements, and results.

2.1 Experimental setup

Durin this project we first re-implemented the described architecture solely based on the description in the DeepLab publication[1], as well as referenced works. Our re-implementation is based on the pytorch framework and written entirely in python. We chose VGG-16[9] as a basis for our DCNN implementation. We concentrated our efforts on reproducing the results on two segmentation problems also found in the original publication: the VOC2012 20-class[15] classification problem, and the VOC2010 7-class person parts classification[12].

2.2 Dataset description

2.2.1 VOC2012

The PASCAL Visual Object Classses Challenge[15], was a yearly competition held to benchmark object segmentation and classification methods. The training data and ground truth is publically available for download[15][23].The dataset contains a total of 17,125 images, depicting objects from 19 classes plus background. On the original dataset only 2,913 of these images are annotated for semantic segmentation. However, we used the augmented VOC dataset which provides annotation for a total of 10.582 training images (available from the previous years) and 1449 validation images. The ground truth is provided as an indexed image (PNG format), where the color index indicates the object class. Images have varying dimensions, with the larger of the dimensions always being 500 pixels. Borders between classes are labeled with a void class (class number 255) in the ground truth. We followed the example of the original publication and ignored the border class during training. The twenty classes used for the segmentation problem can be seen in the following table:

Vehicles	Household	Animals	Other
Aeroplane	Bottle	Bird	Person
Bicycle	Chair	Cat	
Boat	Dining table	Cow	
Bus	Potted plant	Dog	
Car	Sofa	Horse	
Motorbike	TV/Monitor	Sheep	
Train			

Figure 7: Object classes for semantic Segmentation Task at VOC2012 Dataset.

2.2.2 VOC2010 Person Part

This dataset is based on an additional set of ground truth images containing not only object classes, but focusing on classifying individual parts of objects [12]. Since all 19 foreground classes are split into further components, this results in 195 individual classes. Since the VOC2012 dataset contains all images from previous years, the original images for this ground truth are all in the VOC2012 dataset as well. In order to achieve comparable results to the DeepLab paper, we applied the same simplification to the classification task: we focused on only those images containing persons, and also reduced the number of parts by grouping bodyparts together. Hence the classes `hair`, `head`, `left ear`, `left eye`, `left eyebrow`, `right ear`, `right eye`, `right eye brow`, `mouth`, `nose` are grouped to make the class `head`; `left foot`, `left lower leg`, `right foot`, `right lower leg` are grouped to make the class `lower leg`; `left upper arm` and `right upper arm` are grouped to make the class `upper arm`; and `neck` and `torso` are grouped to make the group `torso`.

The ground truth of this dataset has been provided as matlab files (`.mat` format). The first step to integrating this ground truth into our code was therefore to convert these files to an image format that

can be read from python. We therefore wrote a script that is part of our submission and can be found in file `util/create_VOC_Person_dataset.py`. This script takes care of the file conversion as well as the simplification of parts. It furthermore filters out all ground truth images that do not contain a person. We utilize the function `loadmat` from the `scipy.io` module to read the matlab files. The internal data structure of the matlab files was undocumented and had therefore been deduced through trial and error. Instead of a PNG file where the color indices represent the classes, we store a grayscale TIFF file where the gray value represents the class. This is less convenient for visual inspection of the ground truth, but entirely compatible with our data loader.

2.3 Data loading

To load the data we used the torchvision package. We wrote an implementation of the `torch.utils.data.Dataset` class. This implementation can be found as part of our submission in the file `include/data.py`. The constructor of the dataset finds the segmentation ground truth images and stores their file names in a list. In the implementation of `getitem` we then load segmentation ground truth and source image, perform dataset augmentation and transformation to torch tensor, and then return a tuple consisting of normalized source image, ground truth, downscaled ground truth, and original source image. The normalized source image and downscaled segmentation ground truth are needed as an input for the DCNN, while we need the not-normalized source image and the full-size ground truth are required for the CRF.

Unfortunately we encountered a the following problem while attempting to train this data: In order to do batchwise computation and use the GPU memory efficiently, we needed the images to be the same size during training, so they could be collated to 4 dimensional tensors. We were unable to ascertain how the authors of the DeepLab paper worked around this issue, so we considered three different options: i) pad images to a 500x500 resolution, ii) interpolate images to a common resolution, iii) perform computations on individual images in sequence and add up the loss for images from one batch. Option (i) would increase memory requirements per image and thus reduce batch size. This would make gradients noisier than they alread are. We attempted option (iii), but found that the increase in training time due to having to compute the forward step in the network in sequence was too much of a performance hit, and not feasible. Hence, we chose option (ii) and downscaled all images to a resolution of 250x250. This corresponds to a 2x downsampling along one axis, and less than that along the other. This resolution was chosen as a compromise between retaining as much detail as possible, while being able to keep a large enough batch size to avoid noisy gradients. We fully expected this downsampling to reduce the performance of our network compared to the original publication.

2.3.1 Dataset Normalisation

According to [13] normalising the input of a neural network results in faster convergence of the training process. By following the methodology at [13] we first need to center and then scale our dataset in order to make it look like a normal distribution with zero mean and unit variance. We first calculate the mean and the standard deviation for each color channel (Red, Green and Blue) over all the images in our training and validation sets. The input images are given to the neural network after we first apply the following transformation at every color channel:

$$\begin{aligned} &\text{Given mean: } (M_1, ..., M_n) \text{ and std: } (S_1, .., S_n) \text{ for } n \text{ channels,} \\ &I(channel) = \frac{(I(channel) - mean(channel))}{std(channel)} \end{aligned} \tag{6}$$

the same centering and scaling transformation must be applied to our test set as well.

2.3.2 Dataset augmentation

Since the number of training examples is relatively low, we performed some dataset augmentation. We implemented horizontal flipping of images, as well as salt-and-pepper noise added to the original images to increase the number of training examples by 4. Following the experiments in the DeepLab paper, we also implemented random rescaling of input images as dataset augmentation. This option was disabled whenever we trained with downsampled images, and enabled whenever we trained on training image in their original size. The code for dataset augmentation can be found in `include/data.py`.

2.4 DCNN Architecture

The deep convolutional neural network used for feature extraction is implemented as an extension to `nn.Module` and can be found in the file `include/network.py`. It implements a modified version of VGG-16[9] and is initialized with the pre-trained weights of that network. We use `torchvision.models.vgg16` with `pretrained=True` to download the network, but only use the features extraction layers, since the fully connected layers for object classification are irrelevant to our task. We initially implemented the network the way it was described in the DeepLab publication[1], and had to make some guesses where the paper was imprecise. In a later experimental stage, we downloaded the definition of the caffee model used by the authors of DeepLab and inspected it for discrepancies with the text. We found several differences between the network as the authors had described it and we had implemented it, and the network as the authors published. These differences will be detailed later in this report. We finally adjusted our network to more closely resemble the network published by the authors based on these findings, in order to get more comparable results. The final results listed in the section Results of this report have been produced with this final version of the architecture.

2.4.1 DCNN Version 1

Here we describe our first attempt at modelling the DCNN the way it was described in the paper, before we inspected the caffee model published by the authors. The basis of the DCNN is the VGG-16[9] network. The following changes have been made to the VGG-16[9] network:

- all convolution layers received adequate padding, such that the output size is the same as the input size
- the convolution layers 4 and 5 have been changed to atrous convolution with a scaling of 2
- the pooling layers 4 and 5 have been changed to have a stride of 1
- the pooling layers 4 and 5 received padding of 1 pixel on the right and bottom side, to ensure that the output size is the same as the input size
- the fully connected layers representing the classifier at the end have been replaced with the pyramedial atrous convolution

Since asynchronous padding is not implemented in pytorch, but needed in the cases where there is a 2x2 filter with stride 1, we implemented a function called `padForPooling` to take care of this.

The pyramedial atrous convolution at the end consists of 4 lanes. Since the number of features per lane was not documented in the DeepLab publication, we made our own guess based on the number of classes, number of output features from the feature extraction, and compromised that with the memory availability on our GPUs. Thus, the first layer had 256, the second layer 64, and the last layer as many as there are classes. Since we already downscaled our input images, we used the ASPP-S[1] (small FOV) version of the pyramedial layer, meaning that the lanes were scaled by 2,4,8, and 12 respectively.

2.4.2 DCNN Version 2

With version 1 of our DCNN we saw good performance on the training data, however it failed to generalize and did not perform well on the validation dataset. After many checks to make sure we followed the descriptions in the publication to the letter, we downloaded the caffee model published by the authors and had a closer look. To do so, we used the tool netscope [9] to visualize the `.prototxt` file of the caffee model. We found several differences to our interpretation of the paper, leading to the following adjustments made to version 1:

- the kernel size of the max pooling layers are all changed to 3x3, but the stride remains unchanged. This means pooling layers 1, 2, and 3 also require 1 pixel padding.
- the convolutional layer 4 is not an atrous convolution layer, despite the publication describing it as such. The pooling layer 4, however, does have a stride of 1.
- the number of features in the pyramedial atrous convolution part are much larger than anticipated, with 1024 in both the first and second layer.

- dropout regularization is added for the first and second layer of the pyramedial atrous convolution part. This was never stated in the paper.

With these changes implemented, we believe our network to now be identical to the one published by the authors.

2.4.3 DCNN with Batch Normalization

In order to improve upon the results of our experiments, we later decided to experiment with an implementation of the network that uses batch normalization. Batch normalization are additional layers in the architecture of the neural network, that normalize the activations of a layer not unlike how the input data was normalized at the beginning. Since these layers also require parameters to be learned, this increases the memory requirement of the network. Furthermore, since adding batch normalization changes the behavior of the network, the pretrained parameters of the VGG16 network would no longer be applicable. Fortunately, torchvision provides a version of the VGG16 network that includes batch normalization layer and has been trained on it. The implemetation of said network can be found in `include/hybrid_net_batchnorm.py`.

2.5 CRF implementation

The authors of the DeepLab publication did not implement the CRF, and instead utilized the publically available C++ implementation denseCRF [4,5]. Since we were highly interested in the workings of the CRF, we initially decided to attempt our own implementation. Below we list the stages of experimentation with implementing the CRF.

2.5.1 Naive implementation

As our first attempt at implementing the CRF, we created a pure python implementation of Algorithm 1 from the referenced publication [4,5]. This implementation would perform message passing by convolution as in Algorithm 2 of said paper. We did not expect this naive approach to perform well, however thought that it would provide us with a some first results and a better understanding of the method. Unfortunately, it performed even worse than expected and took about 10 minutes per image. Due to that, the code is longer part of the submission.

2.5.2 Python implementation of Permutohedral Lattice

In an attempt to improve the performance of the CRF, while still being able to write our own implementation of the CRF, we decided to use the Permutohedral Lattice method described above as an efficient way to approximate message passing. Since implementing the Permutohedral Lattice seemed out of scope for our project, we included the public python implementation by Ido Freeman [20]. This increased the performance significantly, to about 10 seconds per image. Unfortunately, this was still orders of magnitude slower from what C++ implementation of denseCRF can accomplish. As a proof of our efforts, however, we include this implementation as part of our submission. It can be found in the file `include/crf_numpy.py` but was not used to create our final results.

2.5.3 DenseCRF

Since the performance of our python implementation was insufficient, there was no other choice than to use the publically available C++ implementation denseCRF. In order to integrate it seamlessly into our python scripts, we used the the python wrapper written by Lucas Beyer [21]. The code that constructs the denseCRF object, prepares the filters, and starts the inference can be found in function `applyDenseCRF` in the file`include/crf.py`.

2.6 Performance evaluation measure

In order to efficiently evaluate our system it is imperative that we use a mixture of quantitative and qualitative methods. While the quality of the semantic segmentation can be evaluated simply by comparing the resulting segmentation with the "ground truth", we need a strict mathematical metric to evaluate our model's performance quantitatively. While many researchers use metrics like "Precision" and "Recal" in order to evaluate the performance of their systems we followed the IoU

metric (intersection over union) proposed at [16]. Also this makes the comparison with the original DeepNet paper very easy.

2.6.1 IoU - Intersection over Union metric

In order to define IoU metric[16] we need to describe first some fundamental ideas broadly used by the Computer Vision community. When the problem at hand is classifying the pixels into two classes (i.e. positive and negative), let **True Positive** be the number of pixels classified correctly as belonging to the positive class, **False Positive** the number of pixels misclassified as belonging to the positive class, **False Negative** the number of pixels misclassified as belonging to the negative class and finally **True Negative** the number of pixels classified correctly as belonging to the negative class. Then, the IoU metric can be defined as:

$$\boldsymbol{IoU} = \frac{\boldsymbol{True\ Positive}}{\boldsymbol{True\ Positive + False\ Positive + False\ Negative}} \tag{7}$$

The above definition can be very easilly expanded from the binary classification to the multi-class classification example. If C_{gt} is the set of occuring classes in ground truth and C_{seg} is the set of occuring classes in the segmentation result then the **IoU** is simply the mean IOU calculated over all the classes $c \in \{C_{gt} \cup C_{seg}\}$. Pixels marked 'void' in the ground truth (i.e. those around the border of an object that are marked as neither an object class or background) are excluded from this measure.

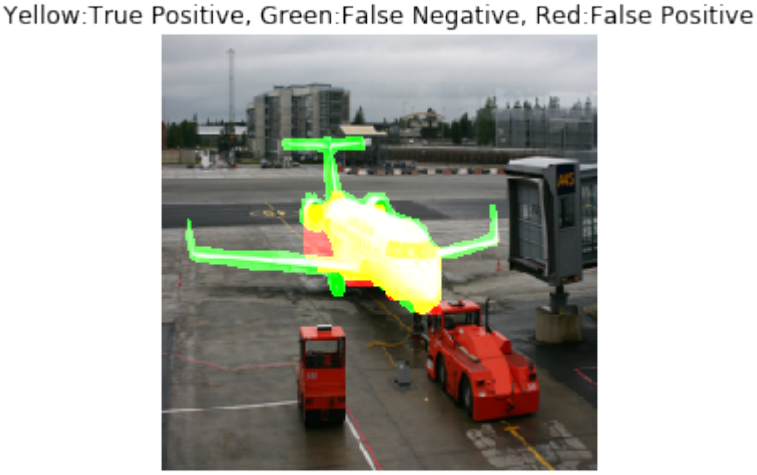


Figure 8: Illustration of True Positive, False Negative and False Positive for class Aeroplane

2.7 DCNN Training

2.7.1 Model Validation

For the VOC2012 20-part problem, we obtained a training set and a validation set as it was provided by the publishers of the dataset. We therefore did not need to do any further splitting. For the VOC2010 7-part people-parts problem, we removed 500 images for validation, 250 for testing, and kept the remaining 2834 images for training. The split was performed at random, but a constant seed was set before the split, such that we could stop and restart training at any time while still retaining the same sets and ensuring the training is not biased towards validation or test set.

As previously stated, training was performed with reduced image size in order to allow for larger batch sizes. Validation was performed using the full image size in order to include all details into the prediction. After each training epoch, validation was performed on the entire validation set by computing the IoU on the resulting predictions. We then stored the version parameters of the model with the highest validation set on the disk and overwrote it once parameters with a higher IoU were found.

2.7.2 Loss functions

The DeepLab publication[1] specifies as a loss function the **unweighted CrossEntropy** loss summed up over all pixels. This loss function is defined as:

$$\text{Unweighted Cross Entropy: } -\sum_{i}^{N}\sum_{c}^{M} y_{i,c}\log(p_{i,c}) \tag{8}$$

where N is the number of pixels, M is the number of classes, $y_{i,c}$ is 1 if and only if pixel i belongs to class c and 0 otherwise, and $p_{i,c}$ is the probability with which pixel i is assigned to class c by the neural network. We quickly observed, however, that the VOC dataset suffers from highly unbalanced classes. That is, most images consist of a large amount of background, and only few foreground objects. Furthermore, in the 20-class problem most images contain only objects from a few classes and no objects from other classes. Unweighted CrossEntropy does not perform well for such unbalanced datasets. For that purpose, in our work we tried two other loss functions that attempt to compensate for this imbalance.

The first loss function we tried is **Sørenson Dice Loss**, defined as:

$$\text{Sorensen Dice Loss: } -\frac{2\sum_{i}^{N}\sum_{c}^{M} p_{i,c}y_{i,c}}{\sum_{i}^{N}\sum_{c}^{M} p_{i,c}p_{i,c} + \sum_{i}^{N}\sum_{c}^{M} y_{i,c}y_{i,c}}$$

This loss function is motivated similarly to the IoU measure discussed previously, except that it takes the uncertainty of a prediction into account. Like IoU, Sørenson Dice similarity has a range from 0 to 1. Therefore, Sørenson Dice Loss has a range from -1 to 0. This loss function is often considered the gold standard for image segmentation tasks. In our experiments, we used the implementation of Sørenson Dice loss available from the pytorch-inferno package. Unfortunately we encountered several problems: First, since Sørenson Dice loss has a different range of values from Cross Entropy, the learning rates and other optimizer parameters documented in the DeepLab paper were no longer applicable. Due to time and hardware constraints we were unable to run enough experiments to find optimizer parameters that work well with this loss function. Secondly, pytorch provides a well optimized C++ implementation of Cross Entropy with an implementation of the gradient as well. The Sørenson Dice loss implementation in the inferno package is a pure python implementation and depends on auto differentiation. Computing it is hence much slower. Finally, it requires that the ground truth is available as a one-hot matrix, wheras Cross Entropy works with a matrix of class labes and is thus more efficient.

For the above mentioned reasons, we decided to go with **weighted Cross Entropy** loss instead. This is simply an adaptation from regular Cross Entropy loss. Here, the absence of a class in the prediction reduces the influence this prediction has on the loss, giving more importantce to the classes actually predicted in the training image. It is defined as:

$$\text{Weighted Cross Entropy: } -\sum_{i}^{N} w_c \sum_{c}^{M} y_{i,c}\log(p_{i,c})$$

$$\text{where:} w_c = \frac{N - \sum_{i}^{N} p_{i,c}}{\sum_{i}^{N} p_{i,c}}$$

2.7.3 Stochastic Gradient Descent (SGD)

In order to train our models we need to find the best weights W of our DCNN model that minimise one of the loss functions (L) proposed above. In order to train our models we used Stochastic Gradient Descent method (SGD)[11, 22]. SGD is a gradient based, itterative optimisation method which shuffles the training data and then calculates the gradient of the loss by evaluating L over a "batch" of random training examples of size N. At each epoch (one epoch is the execution of the

algorithm over all training examples) the training examples are resuffled and new batches are formed, this is what introduces stochasticity in the algorithm. Furthermore, during training with SGD we used momentum and weight decay with the parameters proposed in the original publication. According to [14, 22] momentum can increase the speed of training because it damps the size of the steps along directions of high Loss curvature thus yielding a larger effective learning rate along directions of low curvature. Weight decay according to [10, 22] improves generalisation by suppressing any irrelevant components of the weight vectors by choosing the smallest vector that solves the learning problem and also it suppresses oscilations of the gradient by supressing the static noise on the targets. Stochastic gradient descent with momentum and weight decay[22] can be presented as follows:

Choose an initial vector of parameters w_0 and learning rate $\eta_0 \in \Re$, momentum $\mu \in \Re$ and weight decay $\theta \in \Re$. Repeat for every step t until algorithm converges:

1. $t \leftarrow t + 1$
2. Randomly select a batch and return the corresponding gradient $g_t \leftarrow \nabla L_t(W_{t-1})$.
3. $m_t \leftarrow \mu \cdot m_{t-1} + \eta_t \cdot g_t$
4. $w_t \leftarrow w_{t-1} - m_t - \eta_t \cdot \theta \cdot m_t$

2.7.4 SGD Hyperparameters and Learning Rate Schedulling

The learning rate η, momentum μ, weight decay θ and batch size N of SGD algorithm are hyperparameters of our optimisation algorithm which means that we have to set them before the optimisation algorithm begins. During the training we used a momentum $\mu = 0.9$ a weight decay $\theta = 0.0005$ and a batchsize of $N = 10$. For the learning rate we used a polynomial schedulling strategie[1] for η:

$$\eta(epoch) = \eta_0 \cdot (1 - \frac{current\ epoch}{total\ epochs})^{0.9} \tag{9}$$

Because we use pretrained weights for the VGG part of our network we mainly want to finetune the last Atrus convolution layers[1]. For this reason a small $\eta_0 = 0.001$ is used for the VGG layers and a larger $\eta_0 = 0.01$ is used for the Atrus convolutional part.

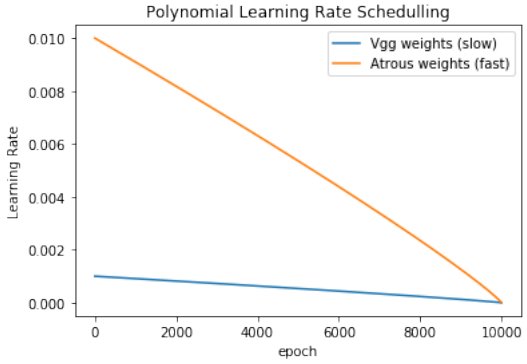


Figure 9: Polynomial Learning rate Schedulling with two different rates (fast/slow) for each parrameter groop

2.8 CRF Hyperparameter Training

The performance of the CRF largely depends on the choice of hyperparameters. These hyperparameters are:

- θ_α the size of the pixel distance kernel part of the bilateral appearance kernel
- θ_β the size of the color similarity kernel part of the bilateral appearance kernel
- θ_γ the size of the gaussian smoothness kernel
- w_1 the weight of the bilateral appearence kernel

- w_2 the weight of the gaussian smoothness kernel

The denseCRF implementation allows for computation of the CRF's gradient with respect to these parameters. In the DeepLab paper, however, the hyperparameters were chosen via a gridsearch using part of the dataset. In our implementation, we performed a similar grid search. The implementation can be found as the function `gridSearchCRFParameters` in `include/crf.py`. The function iterates the given torchvision data loader once, and runs prediction with the given DCNN. For each prediction, a grid search is performed for the hyperparameters of the bilateral appearance kernel, while the parameters of the smoothness kernel are constant and set to the same values documented in the DeepLab paper ($w_2 = 3, \theta_\gamma = 3$). The ranges searched for the remaining parameters are: $w_1 \in \{3, 4, 5, 6\}, \theta_\alpha \in \{30, 40, 50, 60, 70, 80, 90, 100\}, \theta_\beta \in \{3, 4, 5, 6\}$. For each combination of hyperparameters, the CRF prediction is performed based on the unaries provided by the prediction of the DCNN. Crossentropy loss is computed on these new predictions and compared to the previously computed crossentropy of the DCNN prediction. The improvement for each combination of parameter is stored and returned, such that a calling method can not only pick the combination with the best improvement, but also evaluate how much improvement can be provided and how strongly it differs based on the choice of hyperparameters.

3 Results

3.1 VOC2012 20 class problem

To measure the hyperparameters for the CRF, we picked 100 random images from the validation set and performed gridsearch on those. The performance values shown in Table 1 have then been evaluated from the rest of the validation set, seeing how ground truth for the test set has not been made available. The best CRF parameters found by us ($\sigma_\alpha = 70, \sigma_\beta = 3, \sigma_\gamma = 3, w_1 = 5, w_2 = 3$) differ from the ones published with the code as part of the DeepLab paper ($\sigma_\alpha = 70, \sigma_\beta = 5, \sigma_\gamma = 3, w_1 = 5, w_2 = 3$).

3.1.1 Performance

Class	IOU without CRF	IOU with CRF
Background	84.40	84.87
Aeroplane	59.42	64.42
Bicycle	19.15	20.01
Bird	40.73	43.88
Boat	32.26	35.38
Bottle	24.28	25.22
Bus	58.87	60.38
Car	58.72	61.91
Cat	59.06	63.82
Chair	11.47	10.53
Cow	35.17	37.60
Diningtable	23.71	23.79
Dog	48.12	51.49
Horse	35.51	35.66
Motorbike	47.94	50.32
Person	60.94	64.68
Pottedplant	23.72	21.60
Sheep	42.76	45.71
Sofa	22.10	22.00
Train	52.75	56.33
Mean	41.96	43.96
DeepLab	68.96	71.57

Table 1: IOU measured on the VOC2012 validation set for the 20 class problem, with application of the CRF and without application of the CRF.

In Figure 10 we can see an example result. One can see how the CRF manages to recover details and snap the object boundaries tighter to the actual object, yielding an overall improvement on the IOU.

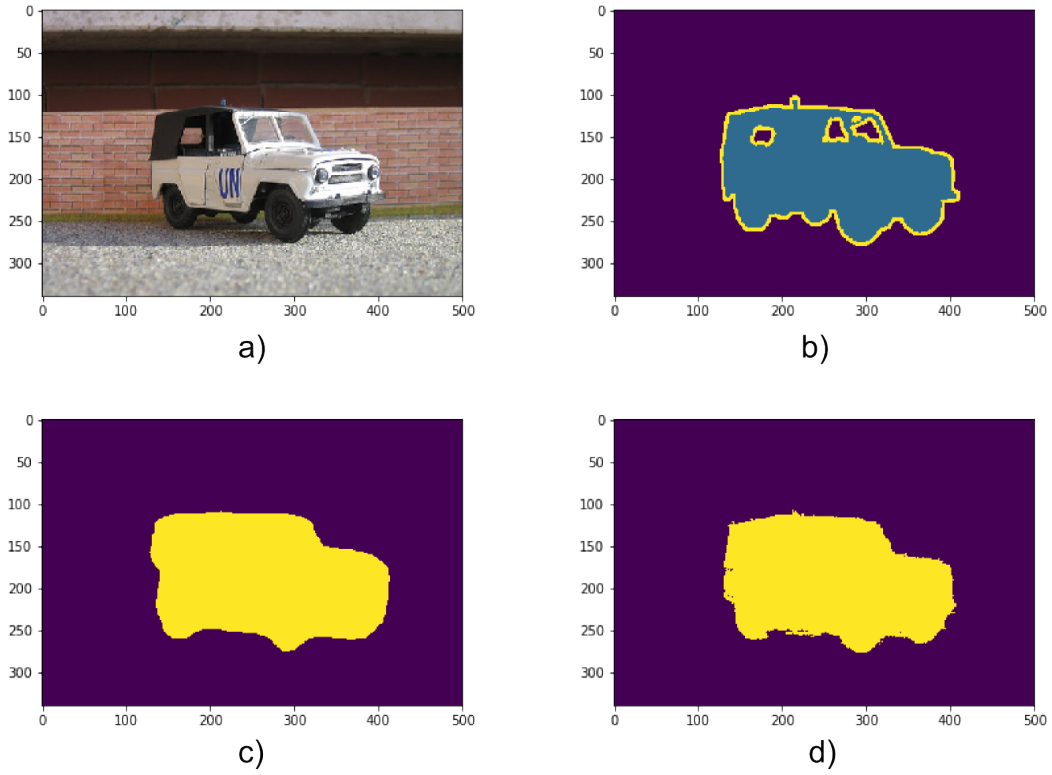


Figure 10: Example of how the CRF improves the prediction. a) original image, b) ground truth ima c) segmentation created by the DCNN, d) segmentation after the CRF has been applied

3.1.2 Failure states

The DCNN seemed to have problems mostly with objects that are not one solid piece, but rather contain gaps. These objects are chairs, plants and bycicles. Especially in these cases, where prediction probability is very low, the CRF does not improve but rather disimprove the result. Areas of low probability are smoothened out by the gaussian filter - sometimes to the point where objects disappear completely. An example of this can be seen in Figure 11

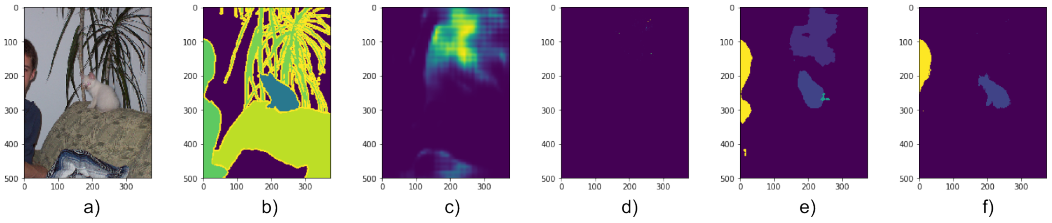


Figure 11: Example of a case in which the prediction without CRF is closer to the ground truth. a) original image, b) ground truth image, c) probability map for the class “pottedplant”, d) probability map of the class “pottedplant” after the CRF has been applied, e) segmentation created by the DCNN, f) segmentation after the CRF has been applied. We can see that the plant was recognized by the DCNN, but due to high uncertainty it was smoothed out by the gaussian filter of the CRF.

3.2 VOC2010 7 class person-part problem

For this problem, the best results were achieved using the DCNN with batch normalization, trained on the images at their full size (no downsampling) and full dataset augmentation. After approximately 50 epochs the validation error started to drop as the network appeared to have started overfitting. For the gridsearch of the hyperparameters of the CRF, we used 100 images from the validation set. The best CRF parameters found by us ($\sigma_\alpha = 70, \sigma_\beta = 5, \sigma_\gamma = 3, w_1 = 3, w_2 = 3$) differ from the ones published with the code as part of the DeepLab paper ($\sigma_\alpha = 70, \sigma_\beta = 5, \sigma_\gamma = 3, w_1 = 5, w_2 = 3$). The results shown in Table 2 are measured on the test set. The jupyter notebook performing the performance evaluation can be found in `CRF_VOC_2010_PersonPart.ipynb`.

3.2.1 Performance

Table 2 shows the performance of our network. As one can see, the performance is nearly identical to the one shown in the DeepLab paper. This is, however, considering that we used a network with batch normalization, which we would assume to perform better.

Class	IOU without CRF	IOU with CRF
Background	92.27	93.05
Head	73.3 0	77.98
Torso	67.82	69.88
Upper arm	58.15	59.60
Lower arm	50.59	54.74
Upper leg	50.62	52.68
Lower leg	43.24	45.40
Mean	62.19	64.79
DeepLab	-	64.94

Table 2: IOU measured on the VOC2010 7 class person-part problem, with application of the CRF and without application of the CRF, compared with the results published in the DeepLab paper

In Figure 12 we can see the influence of the CRF on the prediction. Not only is the prediction smoothened out where there are gaps in the initial prediction by the CRF, but also details are recovered such as strands of hairs that cannot even be seen in the ground truth.

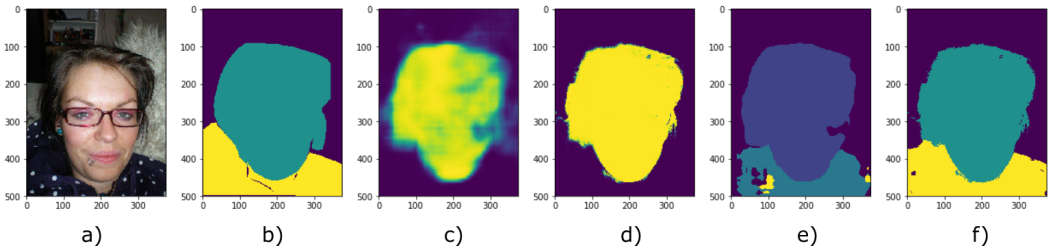


Figure 12: Example of how the CRF improves the prediction. a) original image, b) ground truth image, c) probability map for the class “head”, d) probability map of the class “head” after the CRF has been applied, e) segmentation created by the DCNN, f) segmentation after the CRF has been applied

3.2.2 Failure states

If the prediction created by the DCNN is not confident enough, the CRF can actually make the results less accurate by smoothing out instead of recovering details. This is especially true for very small objects. In Figure 13 we can observe this problem.

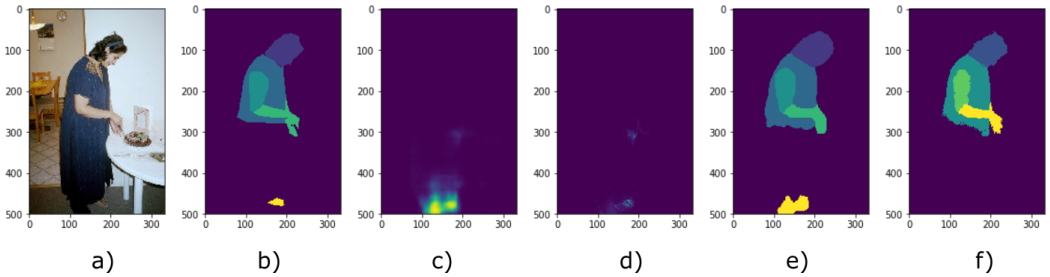


Figure 13: Example of a case in which the prediction without CRF is closer to the ground truth. a) original image, b) ground truth image, c) probability map for the class "lower leg", d) probability map of the class "lower leg" after the CRF has been applied, e) segmentation created by the DCNN, f) segmentation after the CRF has been applied. We can see that while the CRF recovered some detail on hands and head, the probability map for the lower legs has been smoothened out to the point where the feet disappeared.

3.3 Submission

Our code can be found under: `https://github.com/snajder-r/AML2018_project`

In the following we briefly describe the contents of each file:

- `include`: This directory contains python code written by us that is included in multiple python projects
  - `Utility_Functions.py`: Contains code for computation of validation measures, as well as checkpointing (computes validation measure and stores model to file if and only if the score is higher than the previous best)
  - `crf.py`: Contains the function that applies the CRF to a given image. Also contains the function that performs grid search over potential CRF hyperparameters.
  - `crf_numpy.py`: Contains the initial implementation of the CRF using a publically available python implementation of the permutohedral lattice. This code was not used productively and is only submitted as a proof of our efforts to create our own implementation of the CRF.
  - `data.py`: Implements the dataset that is given to the torchvision data loader. Loads raw images and ground truth, applies transformations, and performs dataset augmentation.
  - `network.py`: Contains our initial implementation of the DCNN. This code was not used to create the final results.
  - `hybrid_net.py`: Contains our implementation of the DCNN. Here, the DCNN is implemented in two parts, such that different learning rates can be provided for the feature extraction and classification part of the network. This net was used for the VOC2012 20-class problem.
  - `hybrid_net_batchnorm.py`: Contains the variant of our implementation of the DCNN that uses batch normalization. This net was used for the VOC2010 7-class person part problem.
- `TRAIN.ipynb`: This code was used to train the VOC2012 20-class problem
- `TRAIN_VAL_CrossEntropy_VOC_2010_Person.ipynb`: This code was used to train the VOC2010 7-class person part problem
- `DataNormalisation.ipynb`: This code was used to compute the empirical mean and standard deviation of our dataset, so that they can then be used for normalization of the images before training and prediction.
- `Create_TRAIN_VAL_Sets.ipynb`: This code was used to split the VOC2012 data into training and validation sets based on file lists
- `CRF_VOC_2012.ipynb`: This code invokes the hyperparameter search for the CRF, and evaluates performance of the VOC2012 20-class problem with and without CRF

- `CRF_VOC_2010_PersonPart.ipynb`: This code invokes the hyperparameter search for the CRF, and evaluates performance of the VOC2010 7-class person part problem with and without CRF
- `util/create_VOC_Person_dataset.py`: This code was used to transform the ground truth of the VOC2010 parts data into the required format, seeing how the data was provided as matlab files instead of images

4 Discussion

In this work we attempted to reproduce the results published by Chen et.al. implemented a convolutional neural network with a pyramedial atrous convolution architecture and fully connected random fields for the purpose of pixel classification. While for the 20-class problem we were unable to achieve a comparable performance, we attribute this to hardware limitations, having had to downscale images and being unable to train the network for long enough. In the 7-class problem the performance achieved is very much comparable with the one published by the original authors. The improvements provided by the CRF are comparable as well. We were able to confirm improvements of the IOU of about 2-5% with application of the CRF. The hyperparameters for the CRF found by performing grid-search over a validation set were similar to the ones published by the authors as well.

Given more time, there would have still been several possibilities to improve on the network architecture. For example, the use of Sørenson Dice loss as a loss function might have yielded better results for an unbalanced dataset such as the VOC2012 dataset, however we would have spent too much time finding the right hyperparameters for the optimizer when choosing a different loss function. Further dataset augmentation methods might have also helped to combat overfitting. Instead of rescaling the images, we also considered 0-padding each image to where they all have a 512×512 resolution. This would have allowed us to retain all the detail while still being able to perform batchwise computations on the GPU. With more time, we would have also liked to retrain the network for the 20-class problem using batch normalization in order to see whether we would achieve faster convergence and a better results.

Ultimately, we successfully confirmed the applicability of atrous convolutional layers, a pyramedial convolutional classifier, as well as the improvement to pixel classification provided by a fully connected CRF.

5 References

[1] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, Alan L. Yuille,“DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs”, [Online]. Available: https://arxiv.org/abs/1606.00915

[2] Alex Krizhevsky,Ilya Sutskever,Geoffrey E. Hinton,"ImageNet Classification with Deep Convolutional Neural Networks"

[3] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in ECCV, 2014.

[4] P. Krähenbühl and V. Koltun, “Efficient inference in fully connected crfs with gaussian edge potentials,” in NIPS, 2011. [Online]. Available: https://arxiv.org/abs/1210.5644

[5] Andrew Adams, Jongmin Baek, Abe Davis"Fast High-Dimensional Filtering Using the Permutohedral Lattice".[Online]. Available: https://graphics.stanford.edu/papers/permutohedral/

[6] M. Holschneider, R. Kronland-Martinet, J. Morlet, and P. Tchamitchian, “A real-time algorithm for signal analysis with the help of the wavelet transform,” in Wavelets: Time-Frequency Methods and Phase Space, 1989, pp. 289–297.

[7] S. Lazebnik, C. Schmid, and J. Ponce, “Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories,” in CVPR, 2006.

[8] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial pyramid pooling in deep convolutional networks for visual recognition,” in ECCV, 2014.

[9]K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in ICLR, 2015.

[10] A. Krogh and J. A. Hertz, “A Simple Weight Decay Can Improve Generalization,” in Advances in Neural Information Processing Systems 4, J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds. Morgan-Kaufmann, 1992, pp. 950–957.

[11] H. Robbins and S. Monro, “A Stochastic Approximation Method,” The Annals of Mathematical Statistics, vol. 22, no. 3, pp. 400–407, Sep. 1951.

[12] X. Chen, R. Mottaghi, X. Liu, S. Fidler, R. Urtasun, and A. Yuille, “Detect What You Can: Detecting and Representing Objects using Holistic Models and Body Parts,” in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014.

[13] Y. A. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, “Efficient backprop,” in Neural Networks: Tricks of the Trade, vol. 7700 LECTURE NO, 2012, pp. 9–48.

[14] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013, vol. 28, pp. 1139–1147.

[15] “Pascal VOC data sets.” [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/.

[16] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes Challenge: A Retrospective,” International Journal of Computer Vision, vol. 111, no. 1, pp. 98–136, Jan. 2015.

[17] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes (VOC) Challenge,” International Journal of Computer Vision, vol. 88, no. 2, pp. 303–338, Jun. 2010.

[18] “Netscope.” [Online]. Available: http://ethereon.github.io/netscope/quickstart.html.

[20] I. Freeman, “Python implementation of Permutohedral Lattice.” [Online]. Available: https://github.com/idofr/pymutohedral_lattice.

[21] L. Beyer, “PyDenseCRF.” [Online]. Available: https://github.com/lucasb-eyer/pydensecrf.git.

[22] Ilya Loshchilov,Frank Hutter, "Fixing Weight Decay Regularization in Adam".[Online]. Available:https://arxiv.org/abs/1711.05101

[23] Augmented PASCAL VOC 2012 Dataset with additional annotations [Online]. Available: https://www.dropbox.com/s/oeu149j8qtbs1x0/SegmentationClassAug.zip?dl=0

[24] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Proc. ICML, 2001. 3

[25]J. Shotton, J. M. Winn, C. Rother, and A. Criminisi. Textonboost for image understanding: Multi-class object recognition and segmentation by jointly modeling texture, layout, and context. IJCV, 81(1), 2009. 1, 3, 5, 6