Jesse Conijn, 4493516

What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?

Introduction

In the busy world of computer vision, where machines can see and understand the visual world with astonishing precision, lies an intriguing question: How do we quantify uncertainty in the predictions made by deep learning models? Even though the most advanced Deep Learning networks are able to learn powerful representations with the highest accuracy, these mappings are often taken blindly and assumed to be accurate, which is not always the case. And in situations like a self-driving car that's navigating through a busy intersection, a medical imaging system diagnosing diseases from scans, or a surveillance system identifying anomalies in crowded spaces, the stakes are high and the need for reliable and trustworthy predictions is significant.

Imagine a world where Deep Learning models not only provide accurate predictions but also express their level of confidence in those predictions. As the paper 'What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?' states: "Understanding what a model does not know is a critical part of many machine learning systems." This added layer of uncertainty estimation opens doors to a lot of applications, from safer autonomous systems to more reliable medical diagnoses.

But what kind of uncertainties are truly necessary for effective Bayesian Deep Learning in computer vision? How can this be implemented in the existing DenseNet architecture? And what actually are the benefits of implementing these uncertainties? These questions are exactly what Yarin Gal and Alex Kendall tried to answer.

In this blog, I'll try to explain what kind of uncertainties are suited to add into the existing DenseNet architecture. I will also give an overview of how the DenseNet architecture works, and ...

In Bayesian modeling, uncertainties can broadly be categorized into two main types: *aleatoric* and *epistemic* uncertainties. As I mentioned, these uncertainties play a crucial role in Bayesian deep learning, offering insights into model confidence, generalization and robustness.

If you think of typical 'noise' or uncertainty, most would think of examples like variations in lighting conditions, occlusions, sensor noise, motion noise, etc. This is what *Aleatoric uncertainty* is. Aleatoric uncertainty, often referred to as data uncertainty, captures the inherent variability in the observed data. Take for example flipping a coin and predicting

either HEADS or TAILS. This uncertainty is random and is part of the natural processes of what we are observing. This type of uncertainty cannot be reduced even if more data were to be collected. By modeling aleatoric uncertainty with a probability distribution, Bayesian deep learning frameworks can account for the inherent randomness in the data distribution, leading to more accurate predictions and risk-aware decision making.

Epistemic uncertainty, on the other hand, refers to the lack of knowledge or understanding in the model parameters. This lack of knowledge comes from many sources. Often referred to as *model uncertainty,* it reflects the uncertainty about the model's structure, architecture, or the underlying process that generates the data. In contrast to aleatoric uncertainty, epistemic uncertainty can be reduced by gathering more data or improving the model's architecture. If we have more information, we can take more measurements, conduct more tests and "buy" more information. By quantifying epistemic uncertainty, Bayesian deep learning approaches can give information about the model's reliability and can identify areas where the model can improve it's efforts. In other words, we need to learn how this thing works so there is uncertainty about its operation.

Fig. 1 shows some of the characteristics of both aleatoric and epistemic uncertainty. The overlap both uncertainties have can be seen as "parametric uncertainty", measurement error" or "process error".

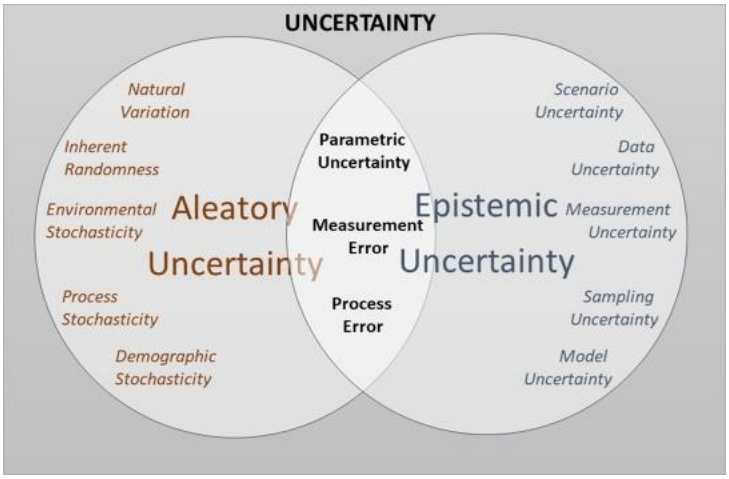

Figure 1: Characteristics of aleatoric and epistemic uncertainty

Using deep learning for Computer Vision purposes obviously makes for specific types of uncertainties. Fig. 2 illustrates the difference between aleatoric and epistemic uncertainty within Computer Vision when it comes to semantic segmentation. Here you can see in (d) that aleatoric uncertainty is increased on object boundaries and for objects far from the camera. Epistemic uncertainty on the other hand, seen in (e), exhibits increased epistemic uncertainty for semantically and visually challenging pixels.

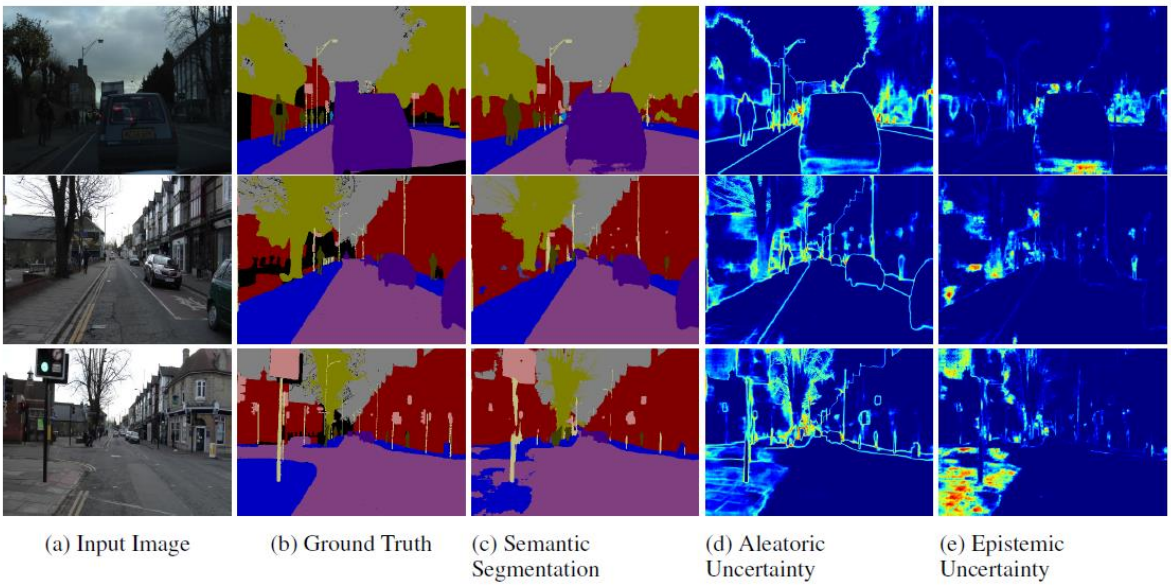

Figure 2: Illustration of the difference between aleatoric and epistemic uncertainty

In this blog we want to test the influence of modeling aleatoric uncertainty in a Bayesian deep learning framework, as well as epistemic uncertainty and lastly combining both. For this we use the DenseNet architecture, so before we move any further, let's explore how the DenseNet architecture works exactly.

DenseNet

In deep learning, Convolutional Neural Networks (CNNs) have been the backbone of numerous breakthroughs in computer vision tasks, from image classification to object detection. But the problems arise when CNN's want more layers and become deeper. Gradients between layers are constantly being computed and when the amount of layers become *so big*, they gradients can get vanished before reaching the other side. And that's where DenseNet comes in. Instead of just making the information flow in a linear fashion, it makes sure that they simply connect every layer directly with each other.

Imagine building a pathway through which information flows in a neural network. Traditional methods create one-way streets, where each layer passes information only to the next layer. But DenseNet takes a different approach by connecting every layer to all others, like a web of interconnected roads. This dense connectivity makes sure information can travel freely, leading to better learning and more accurate predictions. Figure 3 shows a visualization of the connections between layers within DenseNet.

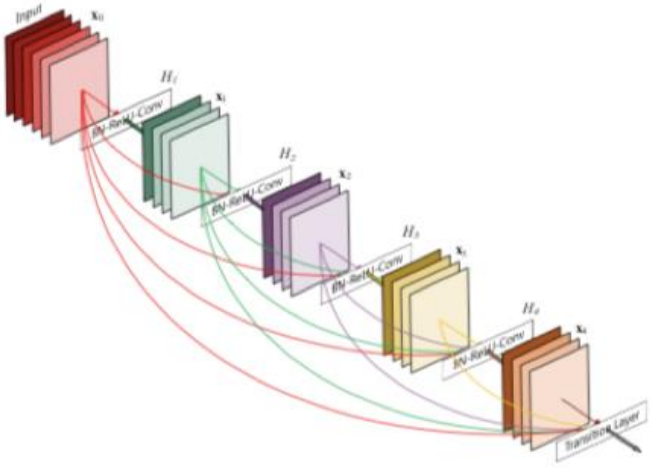

Figure 3: Densenet with 5 layers

One would say that by making the connectivity even denser than before, it would give more parameters than a traditional CNN. But this is actually the other way around, as there is no need to learn redundant feature maps.

Another problem with very deep networks was the problems to train, because of the mentioned flow of information and gradients. DenseNets solve this issue since each layer has direct access to the gradients from the loss function and the original input image.

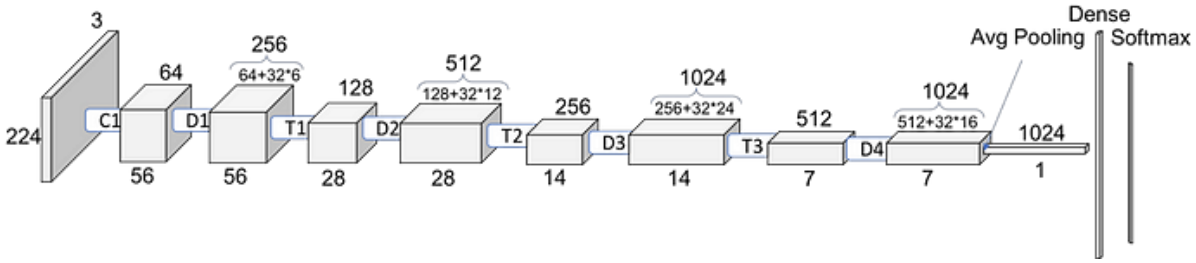

Figure 4: Dense Blocks and Transition Blocks

The architecture

So let's break down the DenseNet architecture. How is it made exactly? Say we start with an image that's 224x224 with RGB values. The initial stage of the network is a Convolution Layer, with size 7x7 and stride 2. This has an output of 112x112 with 64 different features. Next it goes into a Max Pooling Layer with a kernel of size 3x3 and

stride 2, that gives us an output of 64 features of size 56x56. So now the network has officially started. What's next?

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Figure 5: Different versions of DenseNet and their outputs

The whole network is made out of two fundamental building blocks: Dense Blocks and Transition Layers. These are shown in (Fig. 4), where the Dense Blocks are shown with a D and Transition Layers with a T. The Dense Blocks are the blocks that do most of the work. These Dense Blocks consist of a certain number of layers, that's different for each version of the DenseNet (Fig 5). The numbers under each volume represent the sizes of the width and depth, whereas the number on top represents the feature maps dimension.

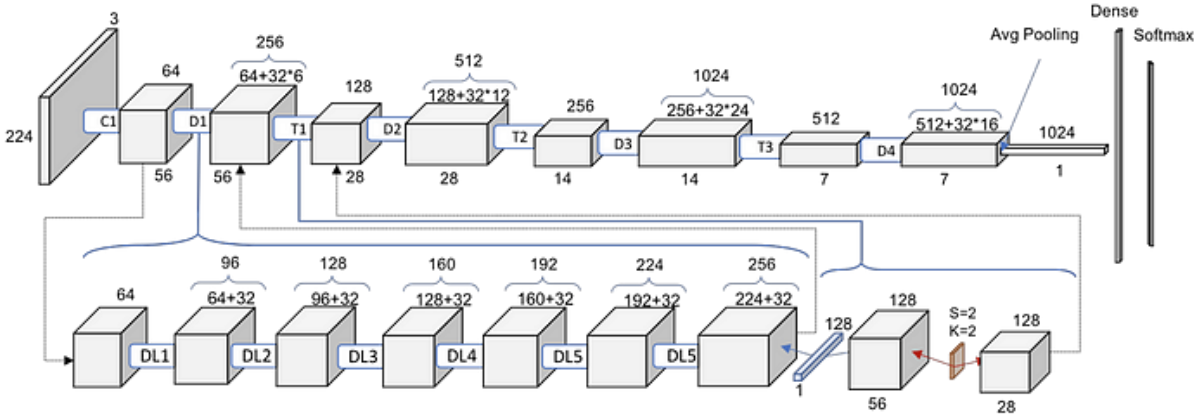

Figure 6: One level deeper in DenseNet-121

Dense Blocks

If we go a little bit deeper into those Blocks, we can see a bit more clear what happening. As depicted in Fig. 6, You see that those Blocks are separated into more of those volumes. But what are all those then? Well, these are the Dense Layers. A dense layer is

a type of neural network layer where each neuron is connected to every neuron in the preceding layer, forming a fully connected network. Okay, but then what are those layers then? So, we can go one last step deeper.

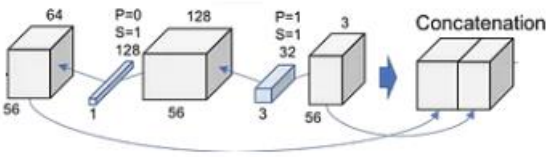

Figure 7: Dense Layer

Dense layers

Finally, we've arrived. Inside the lowest level, the dense layer. In the new deeper level, the dense layer, we can actually see how this behavior of adding 32 times the number of layers is achieved. This Dense Layer consist of 2 convolution layers (the thinner light blue volumes), a 1x1 convolution layer (the bottleneck layer) and a 3x3 convolution layer. Say, in the first Dense Block, in the first Dense Layer, we start with 64 feature maps with a size of 56x56. We feed this to the convolution layer, where it does a Batch Normalization, followed by a ReLU activation function. Then over all these feature maps we first perform a convolution of 1x1 (padding = 0, stride = 1). Then lastly, we do a Drop Out, to prevent overfitting during training. This 1x1 convolution layer then has an output of the same size as the input (56x56) with 128 feature maps. We then do the same thing, but with a 3x3 convolution layer (padding = 1, stride = 1), where we end up with 32 feature maps. Now FINALLY, these 32 feature maps are concatenated by the initial feature maps that was the input of the layer.

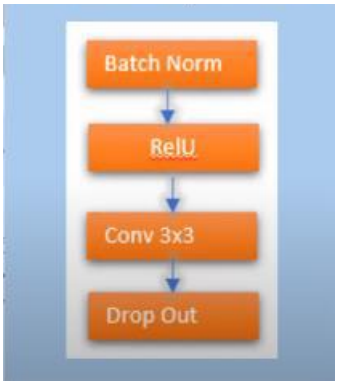

Figure 8: Convolution Layer

But now something interesting is happening. In Figure 9 is the whole network shown. So we've had the dense layer, where in each layer 32 features are added to the input of the block. Quick calculation gives us 64 + 32 = 96 features. Then if we go to the next layer, this whole circus begins again, and again 32 features are added. In this particular DenseNet-121 version, in each block there are 6 Dense Layers. When each time 32

features are added, in the end you'll end up with 64 + 32 * 6 = 256 feature maps!! Ah, so that's how this "magical adding of feature maps" works.

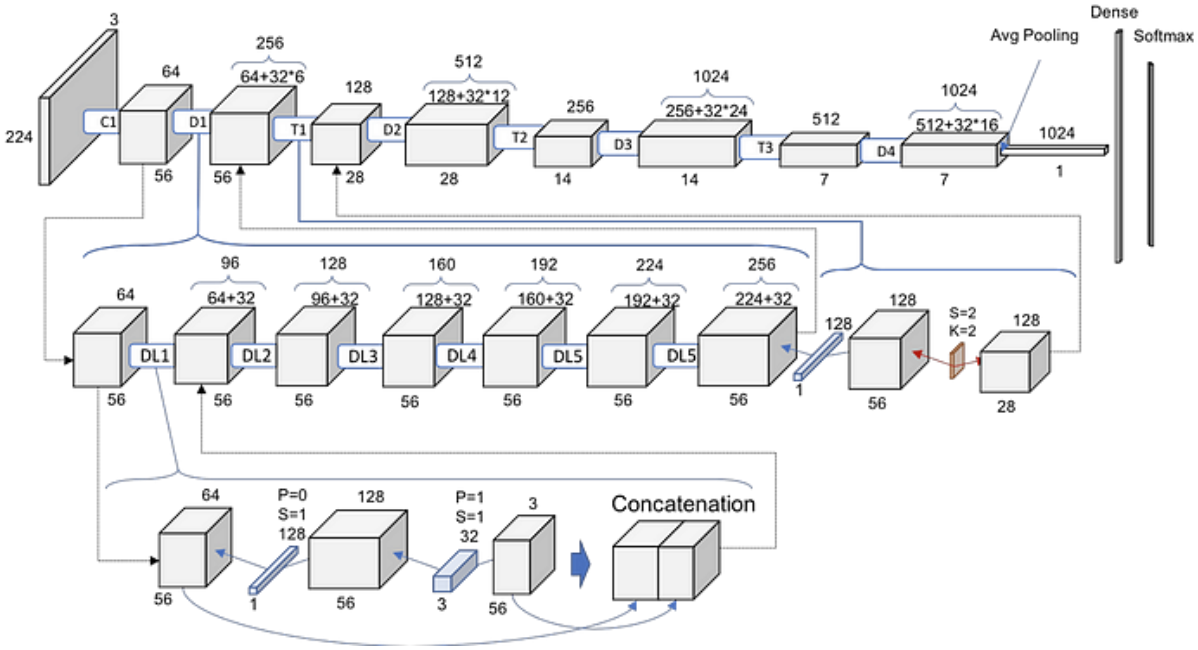

Figure 9: Total DenseNet architecture

Transition layers

At the end of a Dense Block, you'll find a Transition Layer (Fig 10). These are mostly to reduce the spatial dimensions of feature maps. It achieves down sampling by first using average pooling, following a convolution of stride 2, where it reduces the size of the feature maps.

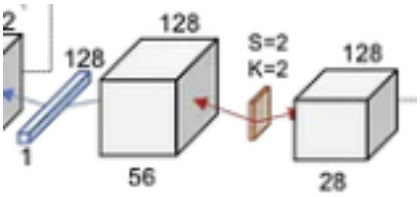

Figure 10: Transition Layer

So there you have it! Each time the feature maps go into a Dense Block, it goes through all these different layers where the amount of feature maps increase, but the image sizes get smaller. In the particular version of DenseNet-121, when giving input images of 224x224, after the initial stage it starts out with 64 feature maps of size 56x56, and with each Block making the amount of feature maps bigger and the size of the images smaller, it ends up with a group of 1024 feature maps of size 7x7. Wow!

Last stage

In the last stage of the DenseNet it performes a 7x7 Average Pooling, so it leaves us with 1024 1x1 feature maps. And there you have it! This is then put into a fully connected layer, and that's basically how the DenseNet architecture works!

Advantages

- Better performance with fewer parameters: DenseNet can achieve better accuracy in image classification tasks with fewer parameters.
- Robust to overfitting: DenseNet's architecture helps prevent overfitting, which means that it is less likely to make incorrect predictions based on the training data.
- Strong feature propagation: DenseNet can learn complex representations of images, which can lead to more accurate predictions and better performance in tasks such as image classification.

Disadvantages

- Increased computational cost: DenseNet can be slower and more computationally expensive to train and run compared to other architectures.
- Memory-intensive: DenseNet may require more memory to store intermediate feature maps, which can limit the size of the images that can be processed and make it less practical for use on certain devices.
- Over-reliance on earlier layers: Dense connections may result in limited capacity to learn more complex features in later layers, which may impact its ability to learn certain types of features.

Uncertainties

So I've talked a lot about the DenseNet architecture, but how do you actually implement the uncertainties that I started with into this model? Well, here we have to look at the Drop Out mostly, that's, like I said earlier, put into the end of each convolution layer inside the dense layers. That's it! So remove that and you would get a result of a DenseNet architecture without the Drop Out, and add it, and you have implemented some epistemic uncertainty!

Look at it like this: As I mentioned, epistemic uncertainty reflects the model's uncertainty about its own parameters, which comes from limited training data. To tackle

this, you could decide to incorporate dropout regularization. Dropout is a technique where randomly selected neurons are temporarily set as non-active during training, effectively simulating training multiple different models with subsets of the neurons active. So implementing this drop out is like placing a prior distribution over a model's weights, and then trying to capture how much these weights vary when given some data. This simulates a sort of randomness.

To integrate aleatoric uncertainty is a little bit more complicated. This is modeled by placing a distribution over the *output* of the model. In they paper they give the example of when the outputs might be corrupted with Gaussian noise. In this case, we are interested in learning the noise's variance as a function of different inputs. So to do this, modify the output layer of the DenseNet to predict both the segmentation masks and the uncertainty associated with each pixel in the mask. Instead of outputting a single segmentation mask per pixel, the model now predicts a distribution of potential values along with their associated uncertainties.

As I will mention later, because of unforeseen circumstances, it has been hard to actually get all the results that we initially hoped to get when reproducing this paper. In the paper itself, the thing that was to be reproduced was the data shown in Table 1. Here you can see value that implementing uncertainties into the DenseNet architecture has, whether it's aleatoric uncertainty, epistemic uncertainty, or a combination of both. Adding epistemic uncertainty has given a 0.1% increase in accuracy where aleatoric uncertainty gives a 0.3% increase. A combination of both gives a 0.4% increase in accuracy, which shows the value of adding this to a DenseNet architecture.

CamVid	IoU
SegNet [28]	46.4
FCN-8 [29]	57.0
DeepLab-LFOV [24]	61.6
Bayesian SegNet [22]	63.1
Dilation8 [30]	65.3
Dilation8 + FSO [31]	66.1
DenseNet [20]	66.9
This work:	
DenseNet (Our Implementation)	67.1
+ Aleatoric Uncertainty	67.4
+ Epistemic Uncertainty	67.2
+ Aleatoric & Epistemic	**67.5**

Table 1: Camvid dataset for road scene segmentation

As my own task within the group was to implement the DenseNet architecture itself, I have tried to at least create a class for all the aspects of the DenseNet architecture. In order to actually train the model I had to do a preprocessing of the images and the labels, where I made them suitable as input for the DenseNet model. Lastly there was the training part. Implementing all this on my own without making too much use of other people's code was challenging, and in the end I had too many setbacks and things that didn't work like expected that unfortunately I wasn't able to get any results that I can

show in this blog. Hopefully in the future I am able to work more on this project and finish at least the proper implementation of the DenseNet architecture.

Contributions

During this project I encountered some setbacks. When starting the Reproducibility project, our group was already one of 3 members instead of 4 like the other groups. Normally this should not necessarily make that much of a difference. Only I was from the start awkwardly clear to predict how the contributions in this project were not going to be equal. After a while, when I was the one that put most effort into the meetings, into leading the project, preparing, dividing the workload etc. Without putting in too much details, in the end I needed to finish this project by myself, which is not an easy task when it is supposed to be done by 4 people. So this is my effort at delivering the best that is possible within what I'm capable of and within the time that I have apart from other courses.

Conclusion

DenseNet is a powerful deep learning architecture that can improve the performance of convolutional neural networks by using dense connectivity between layers. The two types of uncertainties, epistemic and aleatoric uncertainty, can be integrated into the Bayesian model to make it become more aware of its own limitations and gains a better understanding of its uncertainty, which can improve accuracy in the model's results.

[1] G. Huang, Z. Liu and L. van der Maaten, "Densely Connected Convolutional Networks," 2018.