POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Capsule Networks as alternative to DCNN

Relatore
prof. Elio Piccolo

Giuseppe Lillo
matricola: 243296

Anno accademico 2017 – 2018

Contents

List of Figures 4

List of Tables 6

1 Introduction 7
- 1.1 Thesis motivation . . . 7
- 1.2 Thesis goal . . . 7

2 Deep Learning Concepts 9
- 2.1 Artificial Neural Networks . . . 9
  - 2.1.1 Artificial Neuron . . . 10
  - 2.1.2 Learning algorithms . . . 10
  - 2.1.3 Backpropagation algorithm . . . 12
  - 2.1.4 Training process . . . 12
- 2.2 Convolutional Neural Networks . . . 13
  - 2.2.1 Convolutional layers . . . 13
  - 2.2.2 Pooling layers . . . 13

3 CNNs applications 15
- 3.1 Image classification . . . 15
  - 3.1.1 Evaluation . . . 16
- 3.2 Image segmentation . . . 16
  - 3.2.1 Evaluation . . . 16
- 3.3 Generative Adversarial Networks . . . 17
  - 3.3.1 Evaluation . . . 18

4 Capsule Networks 21
- 4.1 Motivation . . . 21
  - 4.1.1 Inverse graphics . . . 21
- 4.2 Routing-by-agreement algorithm . . . 22

4.3 CapsNet architecture 23
4.4 CapsNet results 24

5 Experiments and results using Capsule Networks 27
5.1 Image classification 27
5.1.1 Training 29
5.1.2 Results 29
5.2 Image segmentation 36
5.2.1 SegCaps 36
5.2.2 Training 37
5.2.3 Results 38
5.3 Generative Adversarial Networks 47
5.3.1 Training 48
5.3.2 Results 49

6 Conclusion and future work 53

Bibliography 55

List of Figures

2.1 A 3-layer ANN. The hidden layer is composed of 3 neurons. 9
2.2 An artificial neuron with 2 inputs. 10
2.3 Gradient descent method. 11
2.4 A convolutional layer. 14
2.5 Max pooling layer. 14

3.1 An example of image segmentation with SegNet. 17
3.2 An example of image generation from the original GAN paper. . . . 18

4.1 Routing-by-agreement algorithm. 23
4.2 Capsule Network architecture. 24
4.3 Capsule Network decoder. 24
4.4 Capsule Networks against CNN baseline on MNIST. 25
4.5 Features encoded by the activation vector of the DigitCaps. 25

5.1 Some samples taken from the COIL-100 dataset 28
5.2 CNN: Training and test metrics 31
5.3 CapsNet: Training and test metrics 32
5.4 CapsNet: Manipulation of output capsules' instantiation parameters. 33
5.5 CapsNet: Manipulation of output capsules' instantiation parameters. 33
5.6 CapsNet: Manipulation of output capsules' instantiation parameters. 34
5.7 CapsNet: Manipulation of output capsules' instantiation parameters. 34
5.8 CapsNet: Manipulation of output capsules' instantiation parameters. 35
5.9 CapsNet: Manipulation of output capsules' instantiation parameters. 35
5.10 SegCaps architecture. 36
5.11 SegCaps segmentation results. 37
5.12 UFBA dental dataset samples. 39
5.13 U-Net training. First row represents training data, second row validation data. First column is loss, second is dice score. 40

5.14 Tiramisu training. First row represents training data, second row validation data. First column is loss, second is dice score. 40
5.15 SegCaps training. First row represents training data, second row validation data. First column is loss, second is dice score. 41
5.16 1: **U-Net** Dice: 0.897, Jaccard: 0.813; 2: **Tiramisu** Dice: 0.901, Jaccard: 0.820; 3: **SegCaps** Dice: 0.877, Jaccard: 0.781 42
5.17 1: **U-Net** Dice: 0.923, Jaccard: 0.856; 2: **Tiramisu** Dice: 0.918, Jaccard: 0.849; 3: **SegCaps** Dice: 0.899, Jaccard: 0.816 43
5.18 1: **U-Net** Dice: 0.927, Jaccard: 0.864; 2: **Tiramisu** Dice: 0.908, Jaccard: 0.831; 3: **SegCaps** Dice: 0.888, Jaccard: 0.799 44
5.19 1: **U-Net** Dice: 0.650, Jaccard: 0.482; 2: **Tiramisu** Dice: 0.812, Jaccard: 0.683; 3: **SegCaps** Dice: 0.900, Jaccard: 0.819 45
5.20 1: **U-Net** Dice: 0.694, Jaccard: 0.532; 2: **Tiramisu** Dice: 0.699, Jaccard: 0.538; 3: **SegCaps** Dice: 0.8669, Jaccard: 0.763 46
5.21 CapsNet generated images 50
5.22 DCGAN generated images 51

List of Tables

5.1 Capsule Network for image classification. 28
5.2 Convolutional Neural Network for image classification. 29
5.3 Segmentation models average test scores 38
5.4 DCGAN generator 47
5.5 CNN discriminator 48
5.6 Capsule Network discriminator 48
5.7 CapsNet discriminator 50
5.8 CNN discriminator 50

Chapter 1

Introduction

1.1 Thesis motivation

Convolutional Neural Network gained lots of success in 2012 thanks to AlexNet, a Neural Network capable of good performance on the ImageNet dataset. Since then numerous variants of CNNs have been developed, pushing the limits of the architecture for image classification.

At the end of 2017 S. Sabour, N. Frosst and G. Hinton published a paper named *"Dynamic Routing Between Capsules"*, in which a new architecture for object recognition was proposed: the **Capsule Network**. The idea behind this architecture is to give the classifier a deeper understanding of the objects contained inside the image, encoding the entities that form the object inside vectors of instantiation parameters, representing the pose of the entity. This is achieved using groups of neurons, named *capsules*, representing the entities present in the image. Moreover, a new algorithm for the training of layers of capsules is presented. The *"Routing-by-Agreement"* algorithm is an iterative algorithm which "routes" the activations from one layer of capsules to the next, based on the prediction that each capsule of one layer makes about the pose of the entity encoded in the capsules of the next layer.

1.2 Thesis goal

The original paper about Capsule Networks only describes results obtained in small and simple datasets such as MNIST and CIFAR-10. The goal of this thesis is to verify the possible applications of Capsule Networks in place of Convolutional Neural Networks. CNNs are used today in a variety of applications; in this thesis

we will discuss about Image Classification, Image Segmentation and Generative Adversarial Networks. The experiments presented in this thesis concern aspects of Capsule Networks that were never discussed in the original paper, including big input images, different datasets, time of training and different applications.

This project has been developed in collaboration with AddFor Srl company, which has provided important support ando collaboration.

Chapter 2

Deep Learning Concepts

2.1 Artificial Neural Networks

An **Artificial Neural Network**(ANN) is a computational model that tries to approximate a function f^*[1], which maps an input vector $\boldsymbol{x}$ to an output vector $\boldsymbol{y}$. Artificial Neural Networks derive their name and structure from the biological neural networks. They are composed of **artificial neurons** organized in layers. The first layer is called the *input layer* and it receives the input vector $\boldsymbol{x}$, the last layer is called the *output layer* and returns the *output vector*; the layers in between the input and output layers are called *hidden layers*. In a standard ANN architecture each neuron of each layer is directly connected to each neuron of the next layer, and a **weight** is associated to each connection.

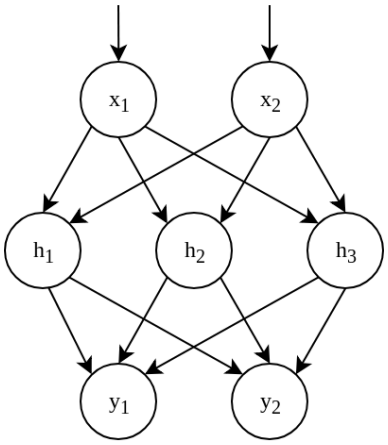

Figure 2.1: A 3-layer ANN. The hidden layer is composed of 3 neurons.

ANNs can be classified into **Feedforward Networks** and **Recurrent Networks**. In the first case the information flows directly from the input layer to next layers, until it reaches the output layer. Recurrent Networks instead contain also feedback connections. In this thesis we will discuss only about Feedforward Networks.

ANNs are also usually called *Deep Neural Networks*, indicating that the number of hidden layers is more than two, making the network "deep".

2.1.1 Artificial Neuron

The artificial neuron is the basic unit of computation inside a Neural Network. Each neuron has a **bias** associated to it. Every neuron of each hidden and output layer performs a weighted sum of its inputs. At the result is added the bias, forming the output $\boldsymbol{z}$. Before passing the output to the next layer, an **activation function** is performed on $\boldsymbol{z}$. Common activation functions used are *binary step*, *sigmoid*, *tanh*, *ReLu*.

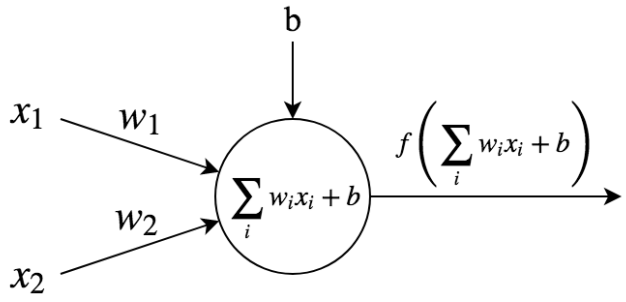

Figure 2.2: An artificial neuron with 2 inputs.

2.1.2 Learning algorithms

The function that the ANN tries to approximate is associated to a particular **task**. The most common tasks delegated to a Neural Network are **regression** and **classification**. For the regression task, the ANN is asked to predict a numerical value associated to the input. For the classification task, the ANN is asked to classify the input in one of the predefined categories.

Each task is strictly linked with the input data provided to the network. As a matter of fact, in order to execute its task and approximate as accurately as possible the function associated to the task, the Artificial Neural Network needs to *learn* the weights and bias of each neuron. An ANN is called a *learning algorithm* because

it learns to approximate a function based on some data. This set of data is called a **training set**. Learning algorithms can be classified into **unsupervised** and **supervised** learning algorithms. Unsupervised algorithms learn particular properties of the training data, while supervised algorithms learn from data associated to a **target**, which can be a numerical value or the class associated to the input.

We need some kind of way to describe how far the output of the ANN is from the desired output. This is achieved defining a **cost function**, also called **loss function**. This function gives us the estimate of how our network is performing compared to the original training data. An high value corresponds to a bad model, while an high value corresponds to a model that has learned correctly the training data. Common loss functions used include *mean absolute error*, *mean squared error*, *cross entropy*.

In order to improve the performance of the ANN, we need to *minimize* the cost function. This is done by calculating the **gradient** of the loss function, which describes how the function's output reacts to small changes in the input. The direction of the gradient points to the direction of steepest increase in the function, therefore to minimize the function we need to take small steps to the opposite direction of the gradient. The **learning rate** regulates how big the steps are. This method of minimizing a function is called **gradient descent**. Over time several methods have been developed to improve gradient descent, such as *stochastic gradient descent*, *gradient descent with momentum*, *adam*.

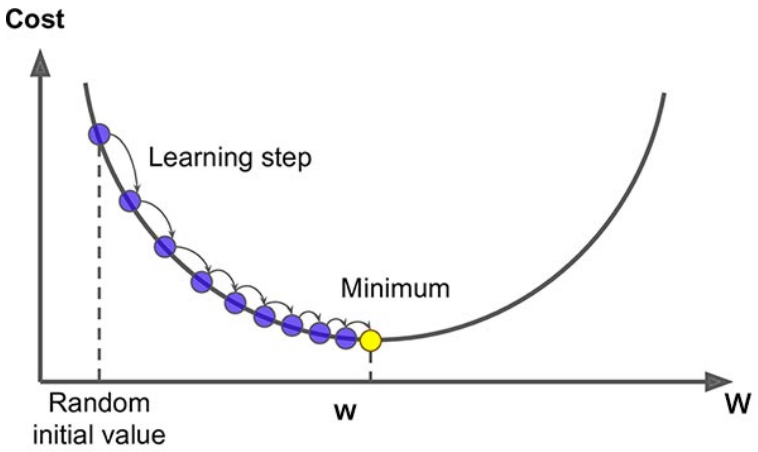

Figure 2.3: Gradient descent method. Source: https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning

2.1.3 Backpropagation algorithm

While the problem of minimizing the loss function is solved through gradient descent, calculating the gradient of the cost function of an ANN is not a trivial task, because of the structure of the network. We need to compute the derivative of the loss function with respect to each of the weights and biases of the network. For neurons of the output layers this is straightforward because they have a target and the loss function applies directly to their output, meanwhile for hidden neurons the derivative of the loss function depends from the parameters of the previous and following layers.

A solution to this problem comes from the **Backpropagation algorithm**[3]. Using the *chain rule*, which allows the computation of the derivative of composed functions, the error at the output layer is *propagated backward* to each of the hidden neurons, allowing the evaluation of the gradient through the entire network. Each of the weights and bias is then updated with a value that is the negative of the partial derivate of the error function with respect to the parameter, multiplied by the learning rate.

2.1.4 Training process

ANNs need to see the training data several times in order to fully understand the patterns between the input and the target. The process of training is usually split up in **epochs**. The training process starts with the **forward pass**, where the output of each neuron is calculated and the output of the last layer is compared to the target. This is repeated for each sample inside the **batch**, a subset of the entire training set. The total error of a batch is calculated as the mean error over each sample in the batch. At the end of the forward pass, the **backward pass** is executed, consisting in the computation of the new weights and biases through the backpropagation algorithm. The entire process is reapeated for each batch in the training set, marking one **epoch**. In order for the network to learn correctly, several epochs are usually needed. Selecting the right number of batches and epochs is a process of trial and error.

Particular care must be taken in avoiding the phenomenon of **overfitting**, which happens when the network is trained too much and it is unable to perform well on data outside the training set. For the evaluation of the performance of a model are usually used a **validation** and **test set**, including samples not present in the training set. The first one is used during the training phase for the evaluation of the model when tuning the **hyperparameters** such as batch size, number of epochs, learning rate, while the second one is used to provide an unbiased evaluation of the

final model.

2.2 Convolutional Neural Networks

A **Convolutional Neural Network**(CNN) is an Artificial Neural Network architecture specialized for processing data with a grid-like topology[1]. One particular application of CNNs that has been greatly successful is images, which can be seen as a grid of pixels, leading them to become the de facto standard for image processing in the Deep Learning area. The name "convolutional" comes from the mathematical operation **convolution**, used instead of the matrix multiplication of ANNs.

The need for CNNs is given by the difficulty to scale to large images of regular ANNs, due to the nature of full connectivity. To solve this problem, CNNs introduce the convolutional layer, that together with pooling layers form a scalable architecture.

2.2.1 Convolutional layers

Convolutional layers are the core of CNNs. The weights of a neuron inside a convolutional layer are called **kernels** or **filters**, and they are organized in 3 dimensions: *width*, *height*, *channels*. Width and height are usually small integers, while the channels extend through the full depth of the input volume, that is an input image or another convolutional layer. Each filter is slided across the width and height of the input, computing the element-wise multiplication bewtween the filter and the input; this constitutes the operation of *convolution*. Convolving a filter over all the input creates a **feature map**, which gives the localized response of the input to the filter. A convolutional layer is composed of different filters of the same size, so the output of this layer comprises a number of stacked feature maps. The peculiarity of convolutional layers is the connectivity among layers: in fact a neuron in a layer $l + 1$ is connected only to neurons in layer l that are in its **receptive field** of *height* x *width*. Just like ANNs, convolutional layers also apply a nonlinear activation function to the output of the convolutions. This kind of design, consisting of **local connectivity** through receptive field and **parameter sharing** through shared filters across all the input, allows for substantial improvements to the scalability of CNNs.

2.2.2 Pooling layers

Common CNNs implementations insert **pooling layers** in-between convolutional layers. The goal of a pooling layer is to reduce the size of the layers, resulting

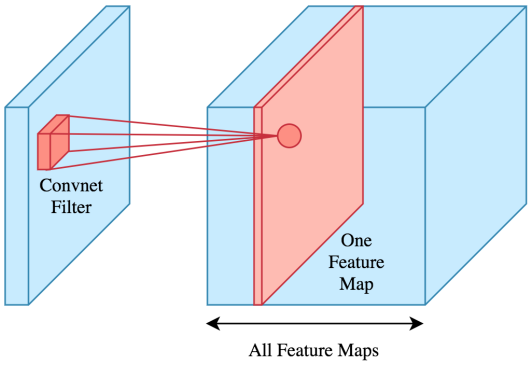

Figure 2.4: A convolutional layer. Source https://brilliant.org/wiki/convolutional-neural-network/

in fewer parameters to train. It replaces the output of a convolutional layer at a certain location with a summary, applying a function over a spatial rectangle $p_h \times p_w$ to each channel. A popular pooling function is **max pooling**, which computes the maximum inside the rectangle.

Another benefit given by pooling is the **invariance** to small translations of the input. This happens because pooling computes a local summary of the input, so small perturbations will not affect the output of pooling. This is useful because allows the network to identify more easily some feature in the input. However the application of pooling layers loses information about the exact location of the features inside the input, and this can be an important factor for particular images. We will see in chapter 4 how pooling will be replaced inside Capsule Networks.

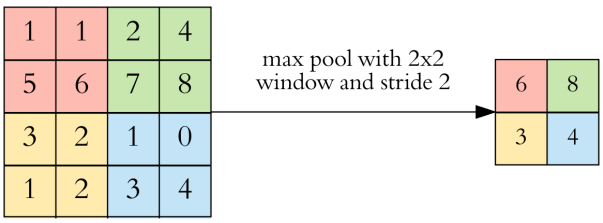

Figure 2.5: Max pooling layer. Source https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2

Chapter 3

CNNs applications

In this chapter we will present some applications of Convolutional Neural Networks. CNNs were originally developed for image classification, specifically for handwritten digit recognition [6]. The charateristics of Convolutional Networks were then successfully applied to a variety of tasks involving images. Image segmentation is a natural follow-up for application of convolutional models. In 2014 Ian Goodfellow came up with the idea of training two different networks one against the other, resulting in a Generative Adversarial Network. CNNs quickly found their place in both of the networks present in the GAN architecture, generating realistic images. In general, these days Convolutional Neural Networks represent the standard for any Computer Vision task.

In this chapter we will also describe the methods for evaluation of the algorithms presented, an important step to benchmark the differents models available.

3.1 Image classification

Image classification refers to the task of assigning an input image to an output label representing a class. Common datasets for image classification include MNIST[11], a database of handwritten digits, and ImageNet[12], a database of hundreds of thousands images of 1000 different objects. Classical Machine Learning algorithms such as Support Vector Machine and K-Nearest Neighbors have been used for image classification, however since the pubblication of the LeNet[11] architecture, CNNs have become the standard approach.

3.1.1 Evaluation

Just like any other type of classifier, the evaluation of an image classifier is based on its ability to recognize the class of the input image. Therefore, classifiers can be ranked evaluating the **error rate** calculated over a test set.

3.2 Image segmentation

Image segmentation consists in partitioning an input image into different segments. Image segmentation is often referred to as *semantic segmentation*, as the term semantic refers to the attribution of each pixel of the image to a particular class. In this way an image segmentation algorithm is able to distinguish the outline of different objects present inside an image.

Many methods are available for image segmentation, however here we will concentrate on methods involving CNNs. In 2014 CNNs for image segmentation were popularized by the Fully Convolutional Networks[7]. The application of pooling in CNN-based architectures for segmentation removes the precise spatial information inside the image. To overcome this problem, architectures like U-Net[8] and SegNet[9] adopted an encoder-decoder structure, in which the encoder learns a representation of the input into low resolution feature maps, and the decoder upsamples them into full input resolution feature maps. This kind of strategy is also used inside SegCaps[10], the architecture for image segmentation based on Capsule Networks that we will explore in chapter 5.

3.2.1 Evaluation

Image segmentation algorithms outputs an image of the same size as the input, where each pixel corresponds to a class.

The evaluation of these algorithms must take in account the overlap between the prediction output and the target mask of each class. This is done by the **Jaccard index** or **Intersection over Uniov** (**IoU**):

$$\text{Jaccard Index} = \frac{|\text{target} \cap \text{prediction}|}{|\text{target} \cup \text{prediction}|}$$

It measures the number of pixels present in both the prediction and the ground truth, divided by the total number of pixels present across the target mask and the prediction.

A similar metric is given by the **Dice coefficient**:

$$\text{Dice coefficient} = \frac{2\,|\text{target} \cap \text{prediction}|}{|\text{target}| + |\text{prediction}|}$$

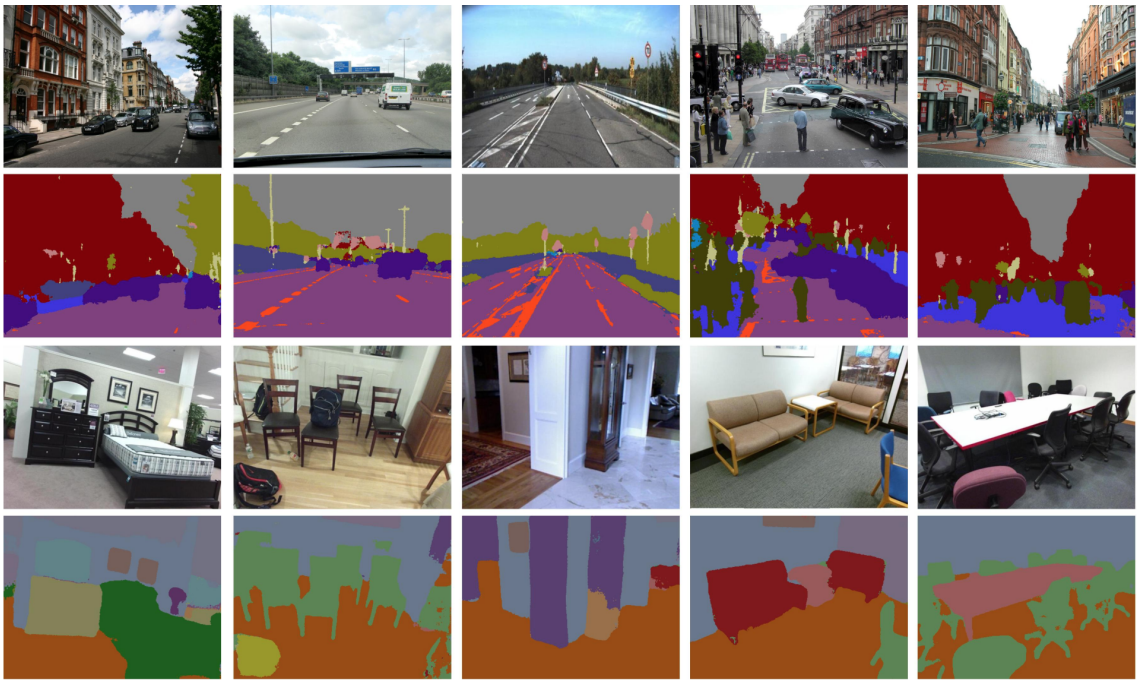

Figure 3.1: An example of image segmentation with SegNet.

It measures the number of pixels present in both the prediction and the ground truth multiplied by two, divided by the total cardinality of pixels of the target and the prediction.

3.3 Generative Adversarial Networks

Generative Adversarial Networks were introduced by Ian Goodfellow in 2014[13]. This architecture consists in two different models being trained simultaneously: a generative model G, called the Generator, learns the distribution of the data starting from a latent vector sampled from a prior distribution, while a discriminative model D, called the Discriminator, learns to distinguish the original data from the generated data. CNNs are commonly used both in G and D for image-generating GANs. The Generator is created mixing Convolutional layers with Transposed Convolutions, an operation that does the opposite of a normal convolution. The Discriminator is usually implemented through a classic CNN. In 2015, Deep Convolutional Generative Adversarial Networks (DCGANs) came out as a class of CNNs for image generation, detailing certain architectural constraints, such as the use of LeakyReLU and the removal of fully connected layers.[14]. In chapter 5, we will compare a DCGAN architecture against a GAN using CapsNets as discriminator,

on a dataset consisting of 20 objects depicted in different viewpoints.

Figure 3.2: An example of image generation from the original GAN paper. The last column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set.

3.3.1 Evaluation

The task of evaluating a GAN model is not as easy as for a regular Deep Learning model. Being a generative model, comparing the quality of images produced is a highly subjective matter. The argument of finding good metrics for evaluation of GAN models is still under debate, however some standard approaches exist.

Inception Score (**IS**)[15] tries to measure the performance of a GAN through the quality of the generated images and their diversity. To measure the quality, the images are classified using an Inception-v3 model[16][17], a particularly successful CNN architecture, trained on ImageNet, then the entropy related to the prediction is evaluated. Low entropy correponds to a highly predictable gerated image, stating that this is an high quality image. This is exactly as evaluating the conditional label distribution $p(y|x)$. To measure the diversity of the images the marginal label probability $p(y)$ is evaluated. High entropy corresponds to the absence of a dominant class. KL-divergence is used to measure the divergence between the two distributions; **higher** scores of IS are better because they imply a larger divergence.

$$\mathrm{IS}(x) = \exp(\mathbb{E}_x\Big[KL\Big(p(y|x)||p(y)\Big)\Big])$$

Fréchet Inception Distance (**FID**)[18] builds upon the IS, using the Inception network to extract features from an intermediate hidden layer, the 2048-dimensional activations of the *pool3* layer. These features are modelled using a multivariate Gaussian distribution with mean μ and covariance Σ for both the generated and

real images. Final FID score is calculated comparing the means and summing all the diagonal elements of the covariance matrices. **Lower** FID scores corresponds to better image quality and diversity.

$$\text{FID} = ||\mu_r - \mu_g||^2 + Tr(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{1/2})$$

Other common metrics used are **precision**, for measuring the quality, **recall** for the capacity to generate any sample in the training set, and **F1 score** to give a summary of precision and recall in one statistic.

Chapter 4

Capsule Networks

In this chapter we will describe the main concepts about Capsule Networks, starting from the motivations behind this idea and the drawbacks of CNNs, then analyzing the architecture in detail.

4.1 Motivation

A **capsule** is a group of neurons whose activity vector represents the instantiation parameters of a specific type of entity such as an object or an object part[4].

The idea of capsules was first described by Hinton et al. in *Transforming Auto-encoders*[5]. In this paper they describe how Convolutional Neural Networks are capable of recognizing objects but incapable of knowing their position in space. This is due to the usage of Pooling layers, which are capable of giving the model the translational invariance, as discussed in section 2.2.2. Instead of aiming for viewpoint invariance, artificial neural networks models should aim for viewpoint equivariance. This can be done by the usage of capsules, which represent a single entity present in the image, and outputs a vector of instantiation parameters representing the characteristics of the entity, together with the probability that this entity is present within its limited domain [5].

4.1.1 Inverse graphics

In a talk given by Geoffrey Hinton at MIT in 2014 [19], he described how the human brain is capable of rendering the visual information received by the eyes, creating a parse tree from the objects and their parts. While computer graphics programs contruct the image starting from geometric representation of the entities, building an hierarchical model through viewpoint-invariant matrices, the human brain does

exactly the opposite, leading to the term **inverse graphics**. The representations of the entities that the brain constructs do not depend from the viewpoints. To construct a learning algorithm that matches the capabilities of the human brain and be able to become viewpoint equivariant, inverse graphics should be the main goal.

4.2 Routing-by-agreement algorithm

As low level capsules represent basic entities of an object, we need a way to transfer information to the appropriate parent capsule representing the right whole in the next layer. This is made possible by a dynamic routing algorithm, the **routing-by-agreement**, which decides where the outputs from low level capsules should go to the next layer through an iterative process. This mechanism is based on a prediction given by the low level capsule for the the instatiation parameter of the higher level capsule. This prediction is calculated through a transformation matrix, and when multiple predictions from low level capsules agree with the output of the high level capsule, this gets activated.

The probability of existence of the entity represented by the capsule is given by the length of output vector. A non-linear **squashing function** is used to ensure that the length stays in range 0 to 1, shrinking the short vectors to almost zero and the long vectors to almost one.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

where $\mathbf{v}_j$ is the vector output from capsule j and $\mathbf{s}_j$ its input vector. $\mathbf{s}_j$ deriving from a previous layer of capsules is calculated through a weighted sum over all the prediction vectors $\hat{\mathbf{u}}_{j|i}$, obtained multiplying the output $\mathbf{u}_i$ of a capsule i by a learned transformation matrix $\mathbf{W}_{ij}$.

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$$

$$\mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$$

c_{ij} are the coupling coefficients determined by the iterations of the dynamic routing. Coupling coefficients between one capsule and all the others in the next layer sum to 1 and they derive from a routing softmax.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})}$$

b_{ij} represent the log prior probabilities that capsule i should be coupled to capsule j. For some cases like MNIST they are initially set to 0 so at the first iteration all the prediction gets sent to all the parent capsules. Coupling coefficients are then refined through a iterative process, measuring the agreement a_{ij} between the current output $\mathbf{v}_j$ of the parent capsule $j$and the prediction $\hat{\mathbf{u}}_{j|i}$ given by capsule i. This is calculated by the scalar product of those two.

$$a_{ij} = \mathbf{v}_j \cdot \hat{\mathbf{u}}_{j|i}$$

b_{ij}s gets updated with the agreement a_{ij}, and leads to an update of the coupling coefficients.

The full routing-by-agreement algorithm is shown in figure 4.1

procedure ROUTING($\hat{\boldsymbol{u}}_{j|i}$, r, l)
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
    for r iterations **do**
        for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$
        for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
        for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$
        for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
    return $\mathbf{v}_j$

Figure 4.1: Routing-by-agreement algorithm.

4.3 CapsNet architecture

The original Capsule Network architecture is shown in figure 4.2. It has been developed to work with the MNIST[11] dataset. It uses a first layer of convolution with 256, 9×9 filters, with stride 1 and ReLU activation. This layers in needed to transform the activities of local feature detectors into *primary* capsules. The second layers constitutes the Primary Capsules, which is a convolutional capsule layer with 32 channels of 8D capsules, organized in 6×6 grids. This is done by applying 8 9×9 filters with stride 2 to the 256 feature maps in input from the previous layer. Each of the capsule in the same 6×6 grid shares the weights with each other. The application of the squashing function functions as a non-linearity for this layer. The next layer is the DigitCaps, which contains one 16D capsule per class and constitutes the output layer. The increase in capsule dimension is justified by the increase in complexity of the entities represented by the capsules in the layer above. The routing-by-agreement algorithms happens between the PrimaryCaps and the DigitCaps.

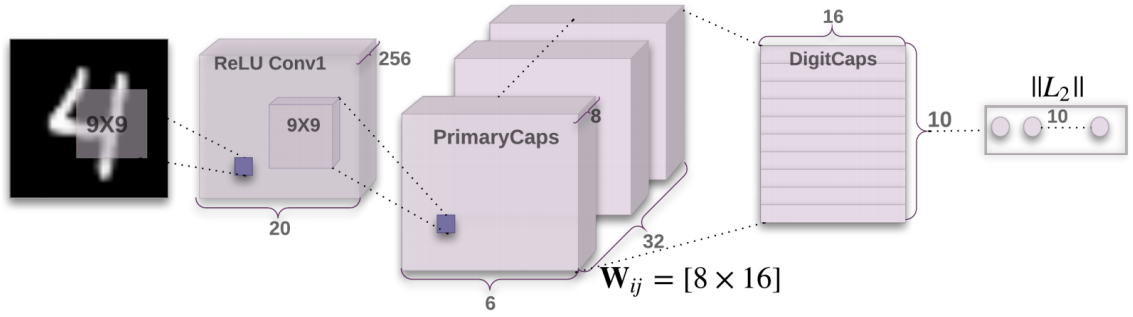

Figure 4.2: Capsule Network architecture.

A decoder is attached to the DigitCaps layer to reconstruct the the input image, in order to encourage the digit capsules to encode the instantiation parameters (figure 4.3). This is done by masking all but the output vector of the correct digit capsule.

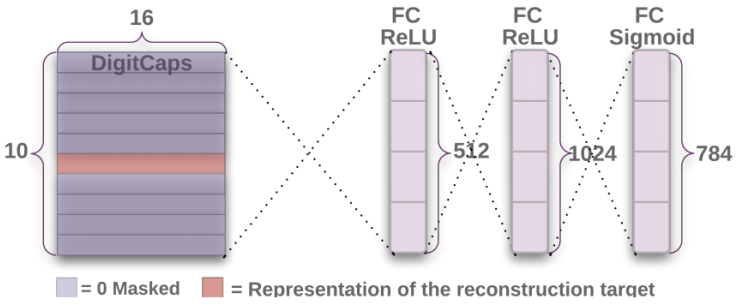

Figure 4.3: Capsule Network decoder.

4.4 CapsNet results

CapsNet for MNIST has been trained using a *margin loss*, one for each digit capsule k, to allow the presence of multiple digits inside the same input image. This margin uses the length of the instantiation vector to represent the probability that a capsule's entity exists.

$$L_k = T_k \max(0, m^+ - ||\mathbf{v}_k|| + \lambda(1 - T_k)\max(0, ||\mathbf{v}_k - m^-||)^2$$

$T_k = 1$ when a digit of class k is present, while $m^+ = 0.9$ $m^+ = 0.1$. $\lambda = 0.5$ is a down-weighting parameter for absent digit classes that stops the initial learning from shrinking the lengths of the activity vectors of all the digit capsules. The loss

for the reconstruction is given by the sum of the squared differences between the output of the decoder and the input's pixel intensities. Total loss used for training is the sum of all margin losses for each digit, with the addition of the reconstruction loss, scaled down by 0.0005.

On MNIST the CapsNet is trained against a CNN baseline, consisting of three convolutional layers of 256, 256, 128 channels, each with 5x5 kernels and stride of 1. The last convolutional layers are followed by two fully connected layers of size 328, 192. The output layer is a 10 class softmax layer, using dropout and a cross-entropy loss. This baseline is designed to achieve the best performance on MNIST while keeping the computation cost as close as to CapsNet. In terms of number of parameters the baseline has 35.4M while CapsNet has 8.2M parameters and 6.8M parameters without the reconstruction subnetwork. Figure 4.4, shows the results, where a CapsNet using the reconstruction and three iterations of routing is able to outperform the baseline, reaching a 0.25% test error achieved only by deeper convolutional architectures.

Method	Routing	Reconstruction	MNIST (%)	MultiMNIST (%)
Baseline	-	-	0.39	8.1
CapsNet	1	no	$0.34_{\pm 0.032}$	-
CapsNet	1	yes	$0.29_{\pm 0.011}$	7.5
CapsNet	3	no	$0.35_{\pm 0.036}$	-
CapsNet	3	yes	$\mathbf{0.25}_{\pm 0.005}$	**5.2**

Figure 4.4: Capsule Networks against CNN baseline on MNIST.

Thanks to the decoder it is possible to visualize what the 16 dimensional vector given by the DigitCaps layer is encoding, introducting perturbations to the parameters of the vector and feeding it to the decoder network. Figure 4.5 shows some features encoded by the parameters.

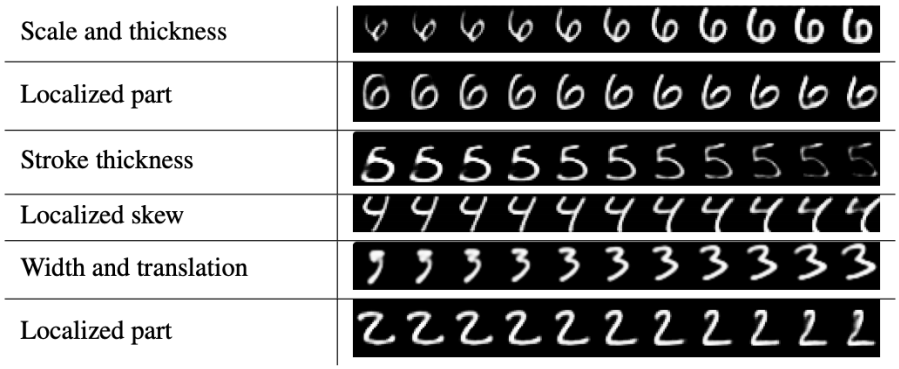

Figure 4.5: Features encoded by the activation vector of the DigitCaps.

Training a CapsNet and a CNN on padded and translated MNIST achieving both around 99% test accuracy, and testing on the affNIST dataset (containing affine transformation of MNIST samples) gave 79% test accuracy for the CapsNet model, while only 66% for the CNN model.

Using the same architecture for MNIST on smallNORB achieved a 2.7% test error rate, on par with other state of the art models.

Chapter 5

Experiments and results using Capsule Networks

Throughout this chapter the methods used for the experiments on Capsule Networks will be described. Each section describes the dataset and the architectures analyzed, then the results obtained.

Each of the tests performed in this chapter has been done using the Python programming language and Keras Application Programming Interface[20]. Keras provides an high-level API for neural networks, allowing for fast experimentation. It runs on top of TensorFlow[21] and Python. A Nvidia GTX 1080 with 8GB of memory has been used to conduct all tests.

As a side note, unless otherwise indicated each of the neural networks layers described in this chapter use a ReLU activation.

5.1 Image classification

The experiments with Capsule Networks regarding image classification are done using the **COIL-100** dataset[22](see figure 5.1). It consists of 128×128 RGB images of 100 different objects, each represented in 72 different angles, for a total of 7200 samples.

The tests presented in this chapter aim to evaluate the performance of CapsNets in modeling the instantiation parameters of objects present in the input, using this dataset to exploit their charateristics. The original dataset has been rescaled into 32×32 images, and divided in training and test set. The training set contains only one third of the total images, while the testing is done on the remaining two thirds of the dataset. This particular choice has been made to evaluate the ability of the

Figure 5.1: Some samples taken from the COIL-100 dataset

model to **generalize** on new viewpoints.

The architecture of the Capsule Network used for this task is similar to the one presented to classify MNIST images[4], but in order to model additional complexity of the dataset, the channels of Primary Capsules layer have been increased to 64 (see table 5.1). The decoder is still present to perform regularization.

Layer	**Description**
Input	$32 \times 32 \times 3$ input shape
Convolution	256 filters, 9×9 kernel, stride 1
Primary Capsules	8-dimensional capsules, 64 channels, 9×9 kernel, stride 2
Output Capsules	100 16-dimensional capsules, 3 routing iterations
Decoder dense 1	512 neurons
Decoder dense 2	1024 neurons
Reshape	$32 \times 32 \times 3$ output shape

Table 5.1: Capsule Network for image classification.

This CapsNet is compared against a Convolutional Neural Network using max pooling. The architecture is similar to the one used to compare CNNs against CapsNets in the original Dynamic Routing paper. It consists of a series of convolutional layers, interleaved with max pooling layers. A dropout layer is placed after the fully

connected classifier to prevent overfitting (see table 5.2).

Layer	Description
Input	$32 \times 32 \times 3$ input shape
Convolution 1	256 filters, 5×5 kernel, stride 1
Max pooling	2×2 window
Convolution 2	256 filters, 5×5 kernel, stride 1
Max pooling	2×2 window
Convolution 3	128 filters, 4×4 kernel, stride 1
Max pooling	2×2 window
Fully connected 1	328 neurons
Fully connected 2	192 neurons
Dropout	0.4 rate
Output	100 neuron, sigmoid

Table 5.2: Convolutional Neural Network for image classification.

5.1.1 Training

The Capsule Network model has been trained using an Adam[23] optimizer with a learning rate of 0.0001. The same optimizer and learning rate is used for the Convolutional Neural Network. The loss used is the margin loss, with the reconstruction loss scaled by 0.0005; batch size is set to 32. For the CNN the categorical cross-entropy is used as loss, with a batch size of 32 as well. The training has been done using a Capsule Network implementation in the Keras framework by Xifen Guo.[24].

Figure 5.2 shows a stable training for the CNN, maintaining a test accuracy of around 0.85. Figure 5.3 shows training data for the CapsNet model, with classifier and decoder networks.

5.1.2 Results

The CapsNet model is able to achieve a test accuracy of **0.87625** after **79 epochs**. The best CNN model reaches **0.87021** after **767 epochs**.

Both model reach a good test accuracy of 0.80 after few epochs, 16 for the CNN and 19 for the CapsNet. However the CapsNet is quickly able to reach 0.86 after 47 epochs, while the CNN reaches the same performance after 123 epochs.

Overall the Capsule Network model outperforms the Convolutional Neural Network model by a very small margin, but is able to reach the peak of performance

after few epochs. At the end this convergence speed is counterbalanced by the slowness of the entire CapsNet architecture, which requires about 10 times the time of the CNN model to complete one epoch.

The decoder of the CapsNet architecture allows us to analyze the instantiation parameters of the capsules of the last layer. This is made possible by introducing perturbations in the 16-dimensional activation vector that each capsule outputs, and feeding it to the decoder network which is able to reconstruct the image from it. Figures 5.4, 5.5, 5.6 and 5.7, show some examples of activation vector manipulation for output capsules. Each row corresponds to one element of the vector, and the rows represent the final reconstruction when tweaking that single element. The perturbations are in range -0.5 to 0.5. It is possible to see how the capsule is able to model the different viewpoints of the input objects. In fact in each of those capsules there is at least one parameter out of the sixteen which encodes the perspective of the object. In particular, figures 5.6 and 5.7 show how the capsules associated to the most complex classes to learn are much more susceptible to parameters variation than the capsules associated to easier classes. This could be due to the the fact that the capsules try to model all the possible viewpoints for complex classes.

This is also confirmed by figures 5.8 and 5.9 which show the reconstruction for capsules associated to classes easier to learn. In fact although those classes present some unique details among the samples, those are not reflected into the reconstructions, which appears all the same for each parameter variation.

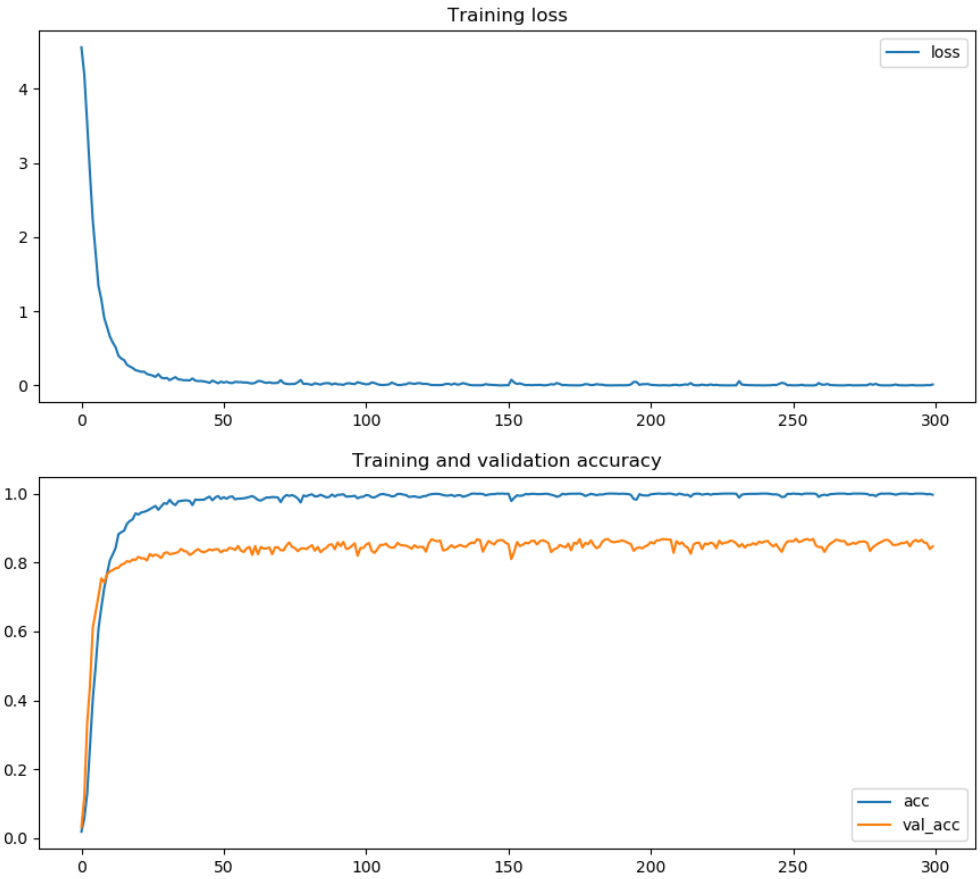

Figure 5.2: CNN: Training and test metrics

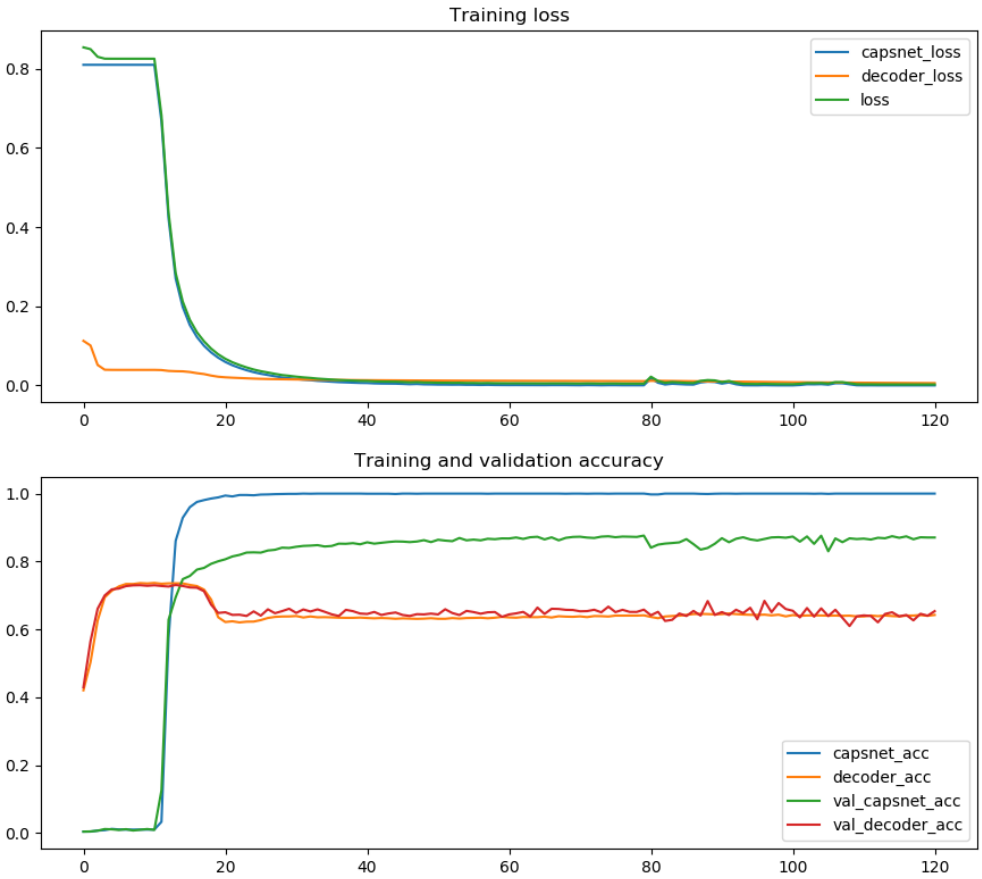

Figure 5.3: CapsNet: Training and test metrics

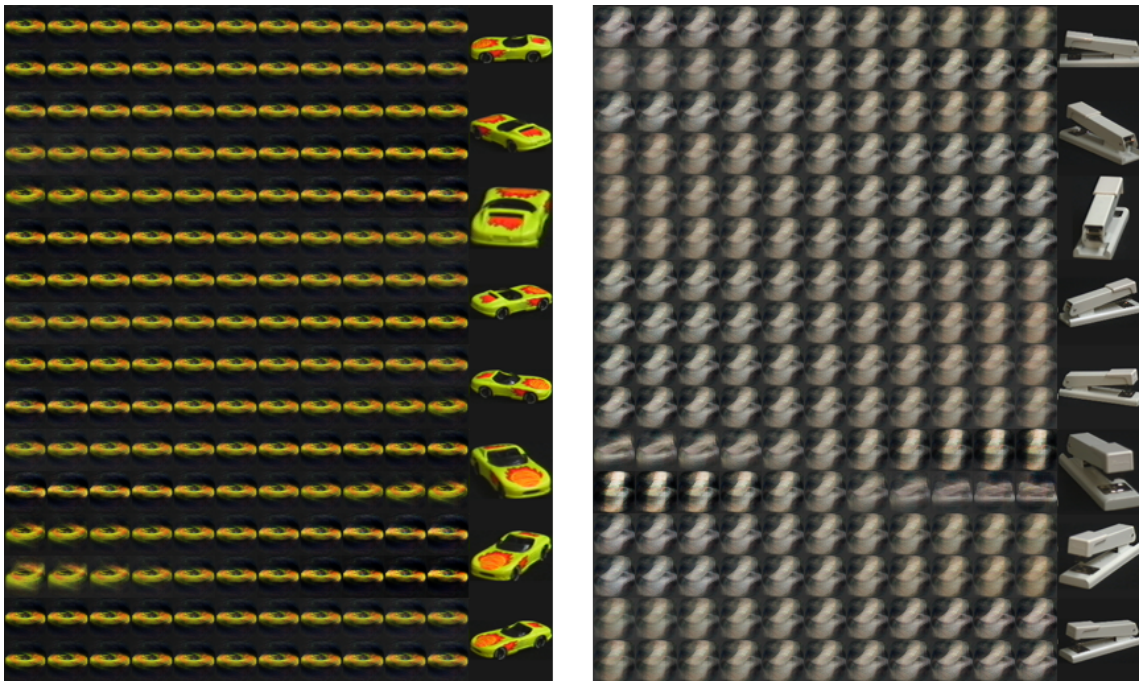

Figure 5.4: CapsNet: Manipulation of output capsules' instantiation parameters.

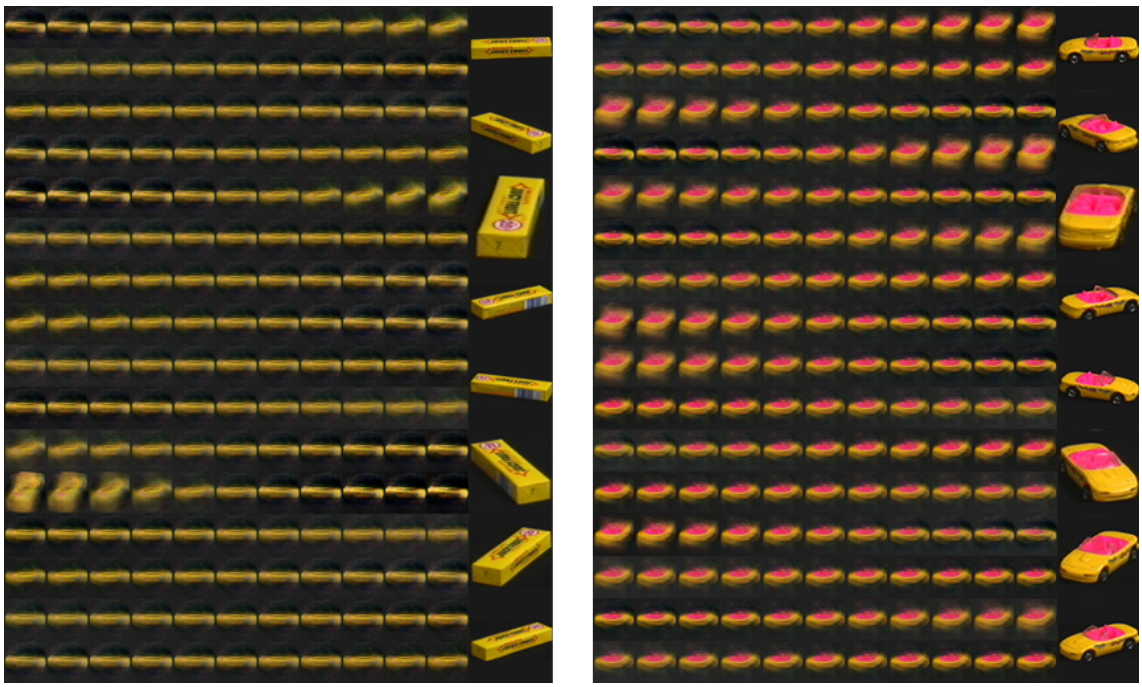

Figure 5.5: CapsNet: Manipulation of output capsules' instantiation parameters.

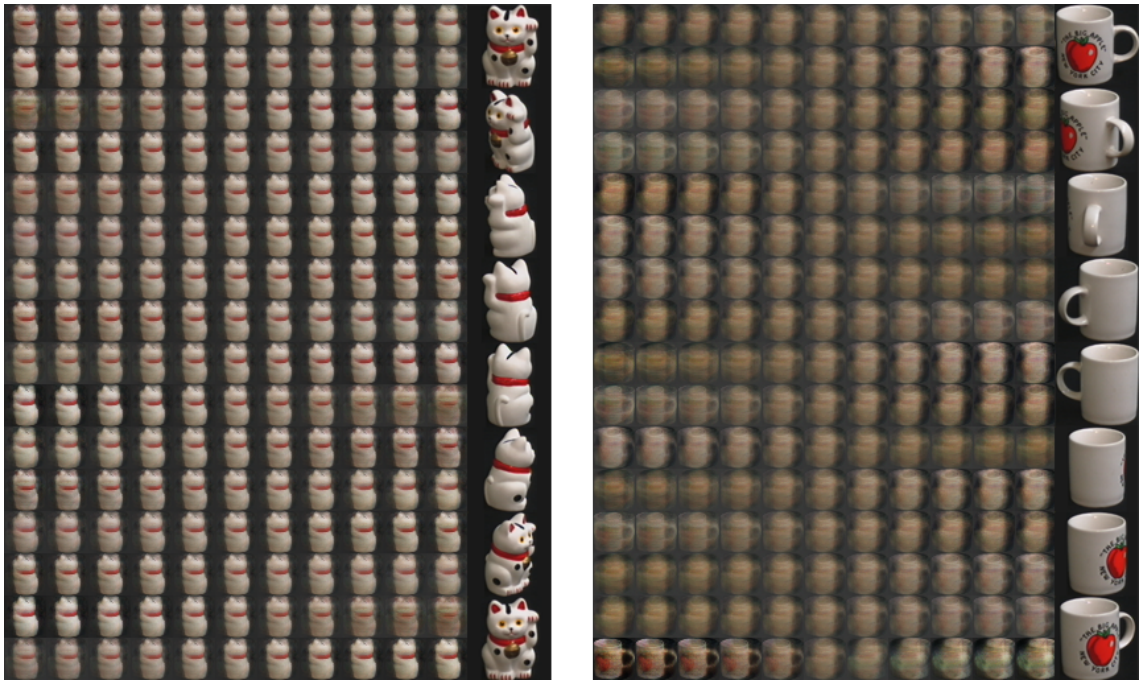

Figure 5.6: CapsNet: Manipulation of output capsules' instantiation parameters.

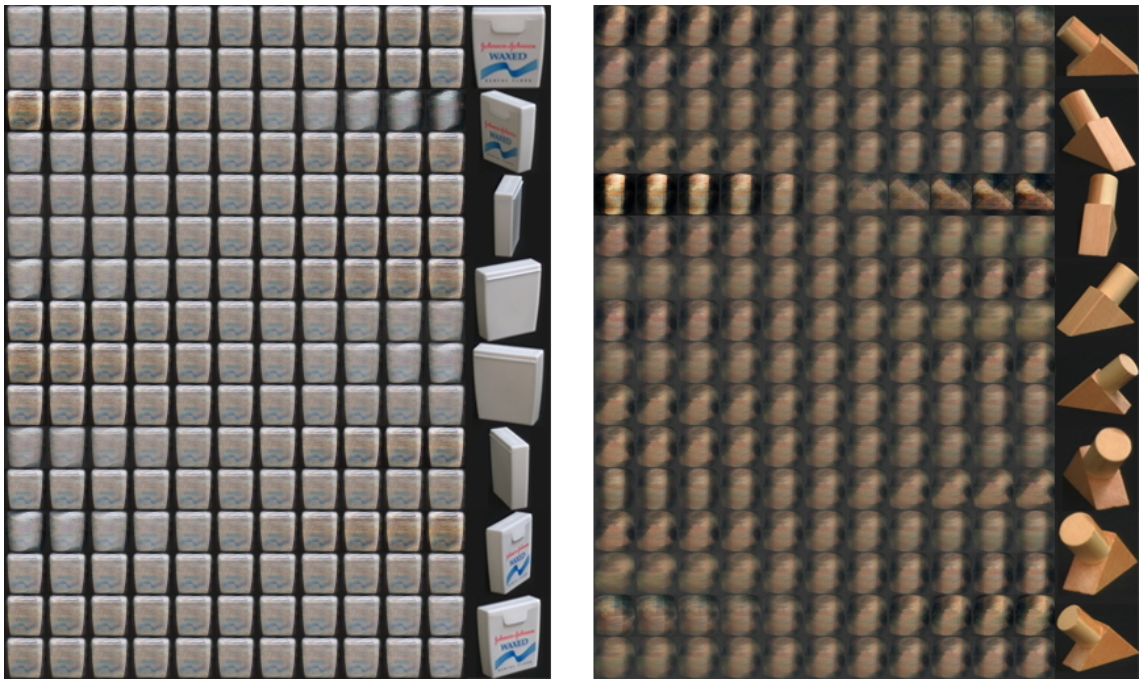

Figure 5.7: CapsNet: Manipulation of output capsules' instantiation parameters.

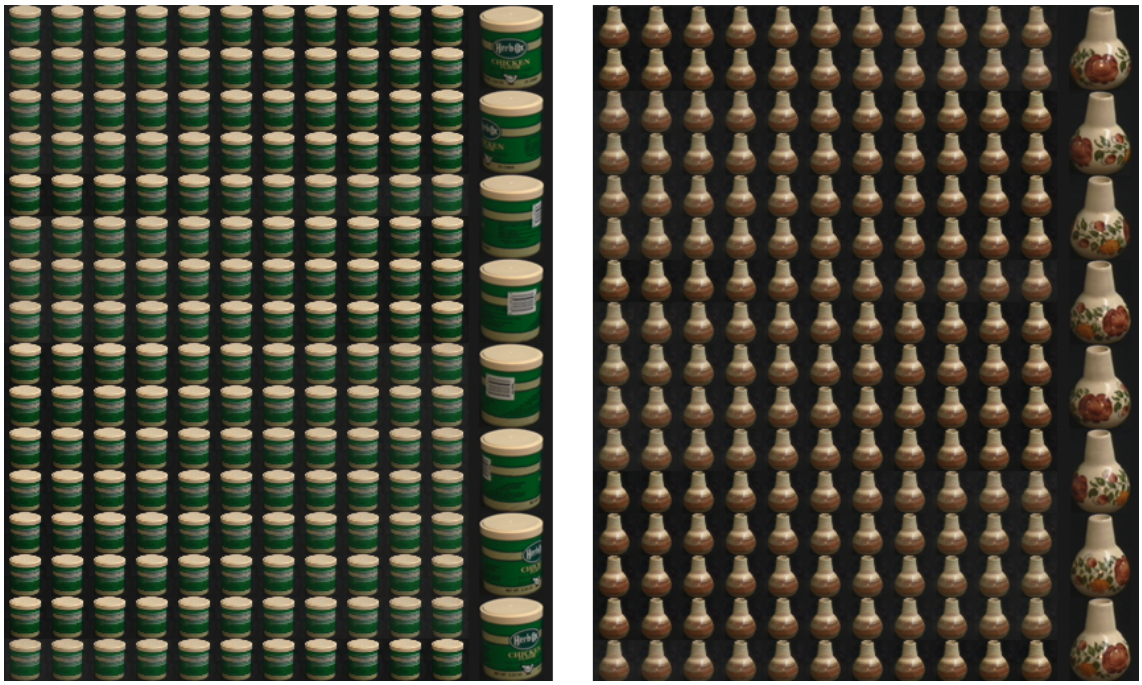

Figure 5.8: CapsNet: Manipulation of output capsules' instantiation parameters.

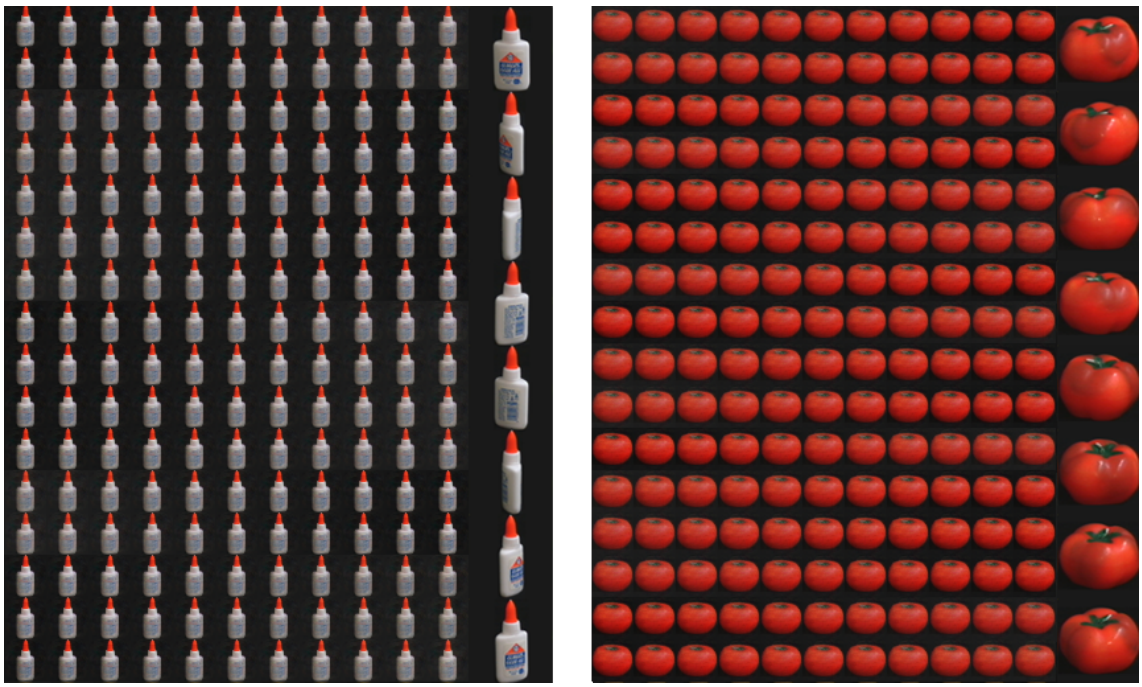

Figure 5.9: CapsNet: Manipulation of output capsules' instantiation parameters.

5.2 Image segmentation

5.2.1 SegCaps

The usage of Capsule Networks for image segmentation has been analyzed by R. LaLonde and U. Bagci, resulting in the architecture named **SegCaps**[10]. It follows the same design of U-Net[8] and FCN[7], consisting in a encoder-decoder CNNs model. Their work includes also a revision of the architecture of CapsNets to deal with the explosion of parameters caused by a deep Capsule Network, utilizing some traits derived from Convolutional Neural Networks, such as a *local connectivity* and *parameter sharing*. In the paper is proposed a new locally-constrained version of the Dynamic Routing algorithm, in which children capsules are routed to parents only within a local window. Moreover, the transformation matrices are shared among each member of the grid, but only for capsules of the same type. These improvements allow SegCaps to handle large images of 512 × 512 pixels, while reducing the number of parameters.

This architecture has been tested on a dataset depicting human lungs, achieving performances on par with U-Net and Tiramisu[25], while using few parameters (figure 5.11).

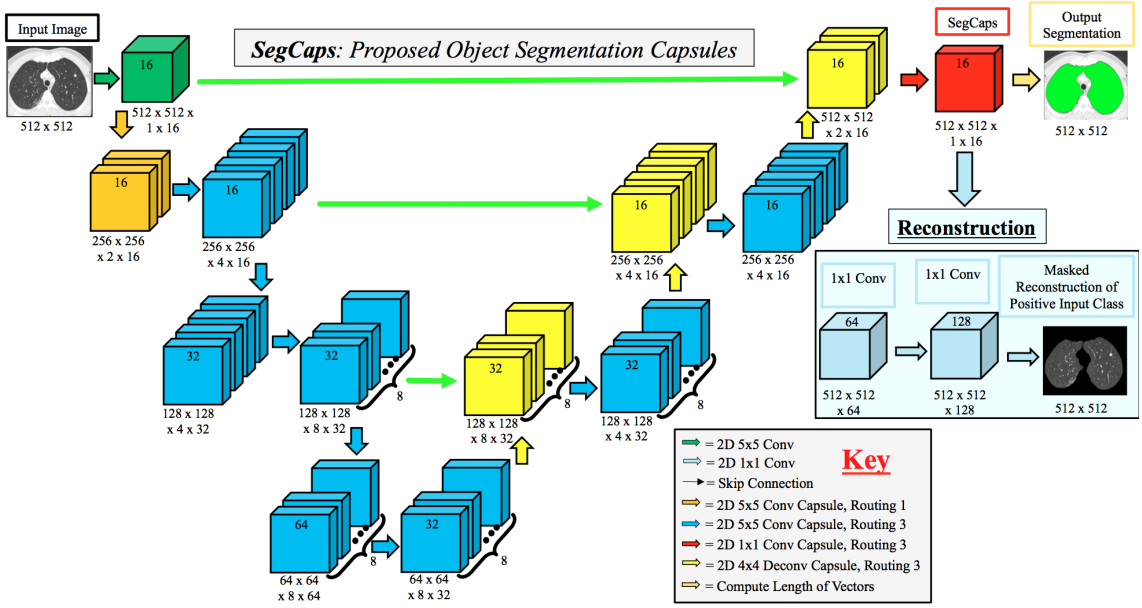

Figure 5.10: SegCaps architecture.

For our evaluation of this architecture we used the **UFBA UESC DENTAL IMAGES** dataset[26], consisting of X-ray images of teeths and their correspondant

Method	Parameters	Split-0 (%)	Split-1 (%)	Split-2 (%)	Split-3 (%)	Average (%)
U-Net	31.0 M	98.353	98.432	98.476	**98.510**	98.449
Tiramisu	2.3 M	98.394	98.358	**98.543**	98.339	98.410
Baseline Caps	1.7 M	82.287	79.939	95.121	83.608	83.424
SegCaps (R1)	**1.4 M**	98.471	98.444	98.401	98.362	98.419
SegCaps	**1.4 M**	**98.499**	**98.523**	98.455	98.474	**98.479**

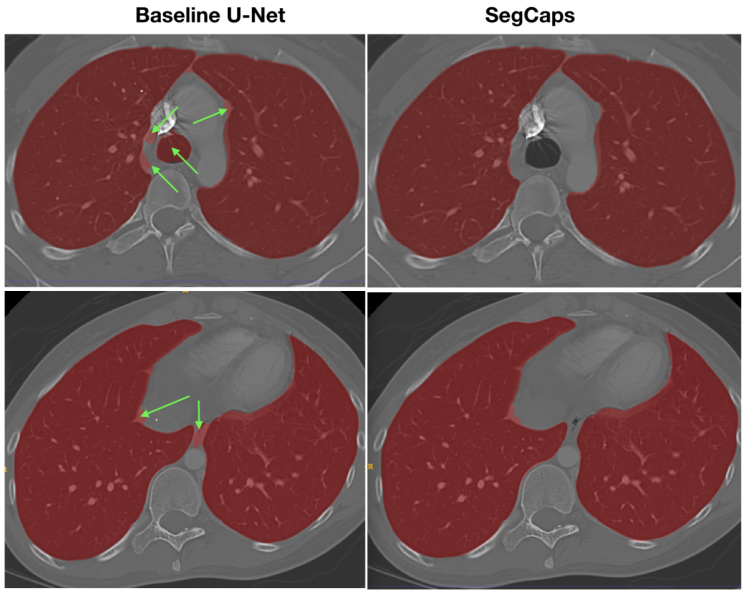

Figure 5.11: SegCaps segmentation results.

teeth masks (figure 5.12). This dataset contains 1500 image of resolution 1991 × 1127. Just like for the dataset of lungs, the goal of segmentation for this dataset is to classify the single pixel to classes teeth or background. The entire dataset has been divided into training and test set using three quarters of the samples for training, and then a validation set has been obtained using one tenth of the training set. Final sizes of those sets are 1013 for the training set, 112 for the validation set and 375 for the test set.

5.2.2 Training

In order for the input images to be feedable to the 3 different models, they have been rescaled and cropped into images of 512x256. The size of the input and of the networks required a batch size of 2 for U-Net and Tiramisu and of 1 for the SegCaps case, in order to fit inside the 8GB of memory of the used GPU. Dice score has been used as a evaluation metric during training. All models have been trained for 200

epochs using Early Stopping with a patience of 25 epochs, monitoring the validation dice score. As optimizer Adam has been used with a learning rate of 0.0001, that is reduced by a factor 0.05 if the validation dice score does not improve after 5 epochs. The loss used is the weighted binary cross-entropy, that weights the ratio between pixels of the two classes. Data augmentation has also been performed, using rotations, shifts, shear, zoom, image flips and noises.

Figures 5.13, 5.14, 5.15 show results of training and validation for loss and dice score. Both U-Net and Tiramisu models stopped at 54 epochs, while SegCaps took 118 epochs, even though the improvements stopped after around 50 epochs.

5.2.3 Results

Test scores displayed in table 5.3 show the Tiramisu model as the winner among the three, obtaining top performances in Dice and Jaccard scores. SegCaps model was the worse performer of the three, although not by a big margin.

Analyzing the segmentation results, most of the times U-Net and Tiramisu output a clearer segmentation mask compared to the one given by SegCaps. Figures 5.16 and 5.17 show those cases, where the Dice score for SegCaps still gives a pretty high score compared to the Tiramisu and U-Net models, but the Jaccard punishes more this situation. Figure 5.18 shows a more visible case of this phenomenon, where the mask of SegCaps goes completely off in the top-left corner.

However SegCaps were much more robust in particularly complex situations, such as the ones in figures 5.19 and 5.20. In those cases both U-Net and Tiramisu missed entire portions of teeths, while SegCaps performed regularly. Those samples were badly judged especially by the Jaccard index, restoring the balance among the models in average scores and bringing SegCaps closer to the other two.

Model	**Average Dice Coefficient**	**Average Jaccard Index**
U-Net	0.8815	0.8015
Tiramisu	0.8892	0.8103
SegCaps	0.8783	0.7892

Table 5.3: Segmentation models average test scores

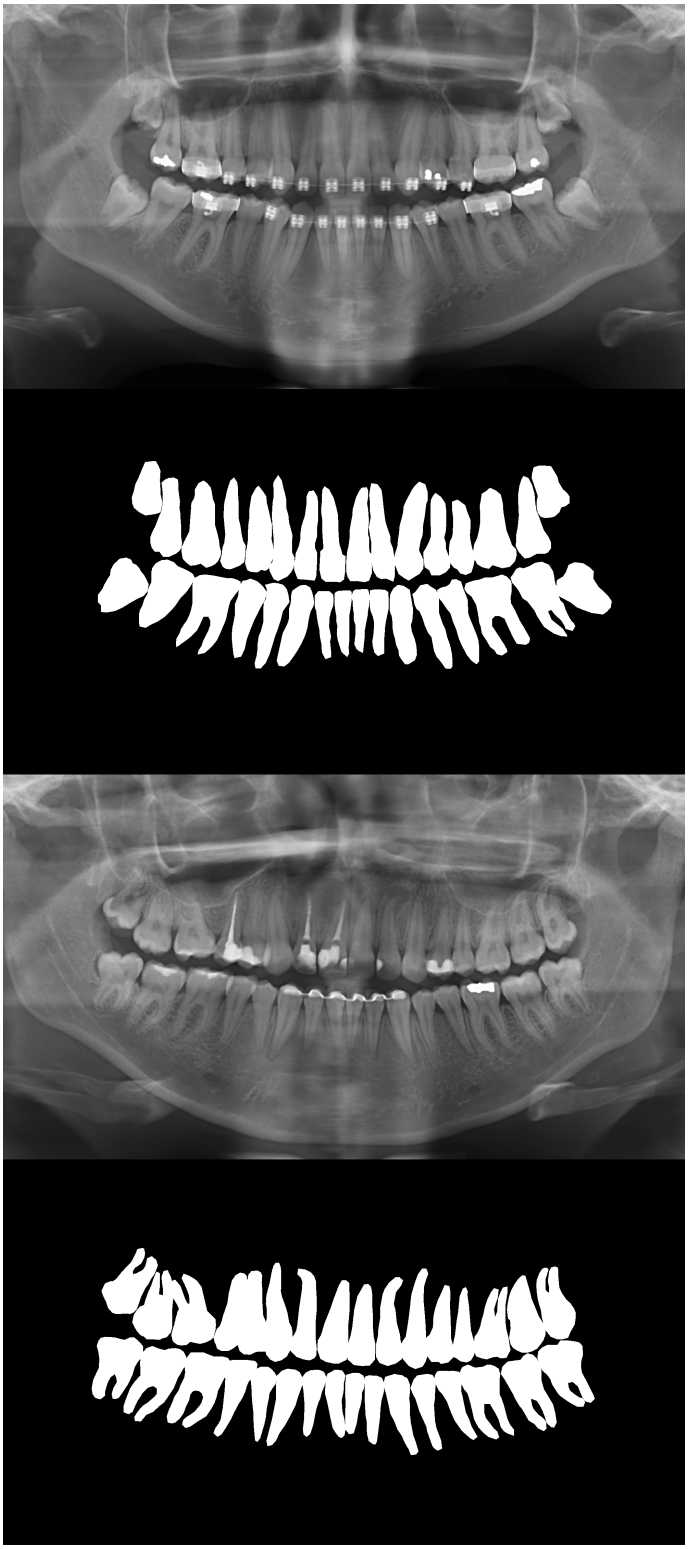

Figure 5.12: UFBA dental dataset samples.

Figure 5.13: U-Net training. First row represents training data, second row validation data. First column is loss, second is dice score.

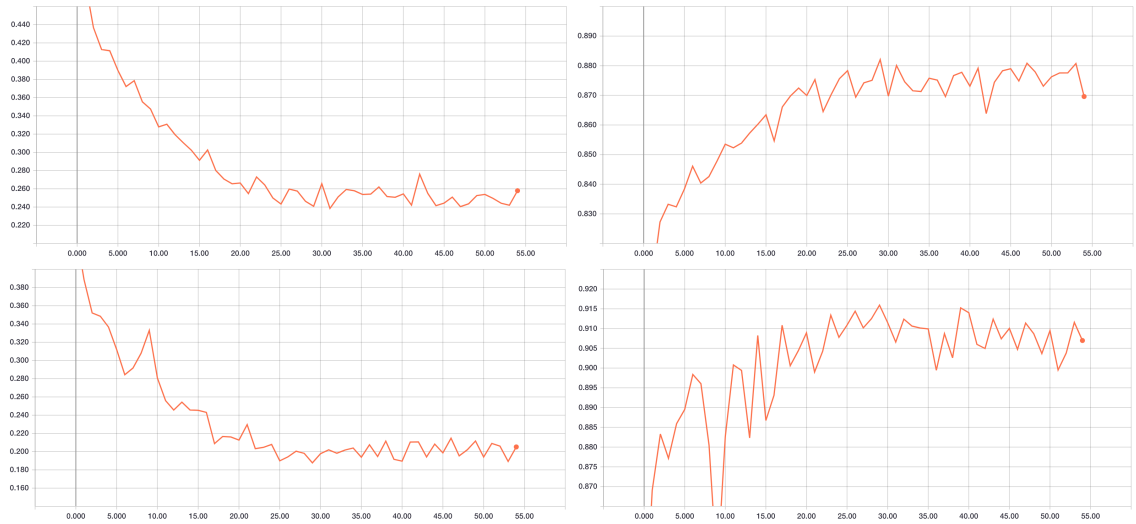

Figure 5.14: Tiramisu training. First row represents training data, second row validation data. First column is loss, second is dice score.

Figure 5.15: SegCaps training. First row represents training data, second row validation data. First column is loss, second is dice score.

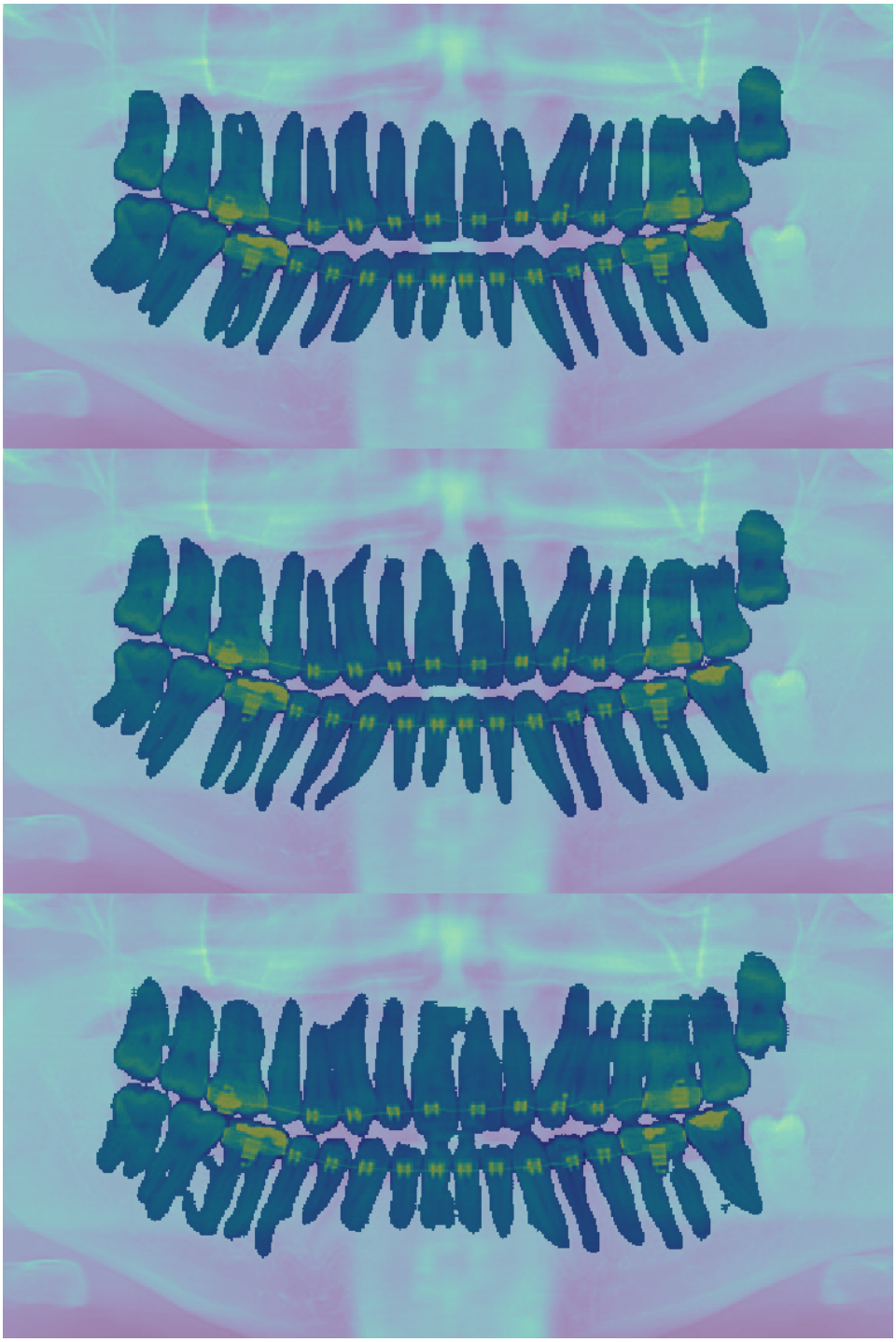

Figure 5.16:
1: **U-Net** Dice: 0.897, Jaccard: 0.813;
2: **Tiramisu** Dice: 0.901, Jaccard: 0.820;
3: **SegCaps** Dice: 0.877, Jaccard: 0.781

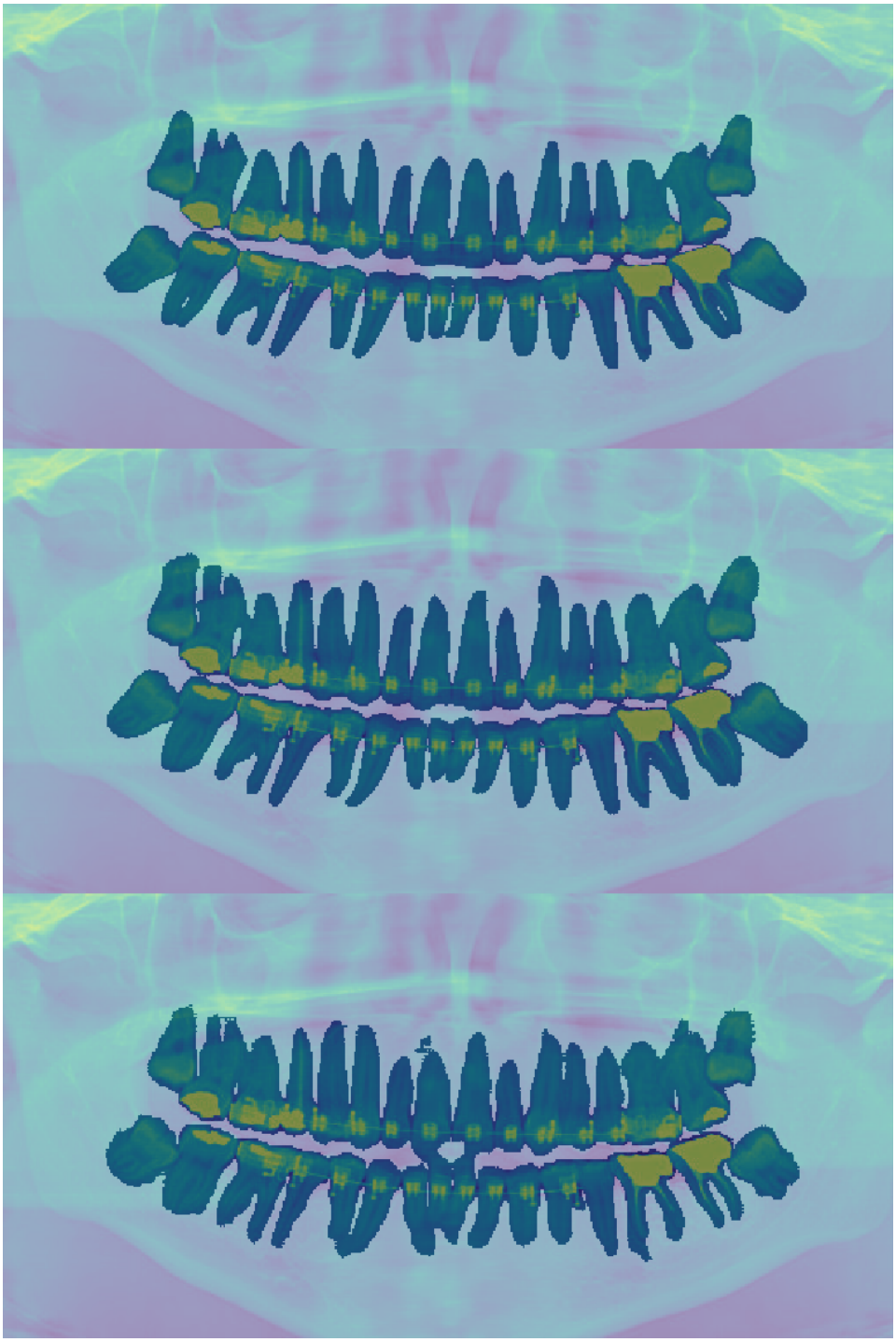

Figure 5.17:
1: **U-Net** Dice: 0.923, Jaccard: 0.856;
2: **Tiramisu** Dice: 0.918, Jaccard: 0.849;
3: **SegCaps** Dice: 0.899, Jaccard: 0.816

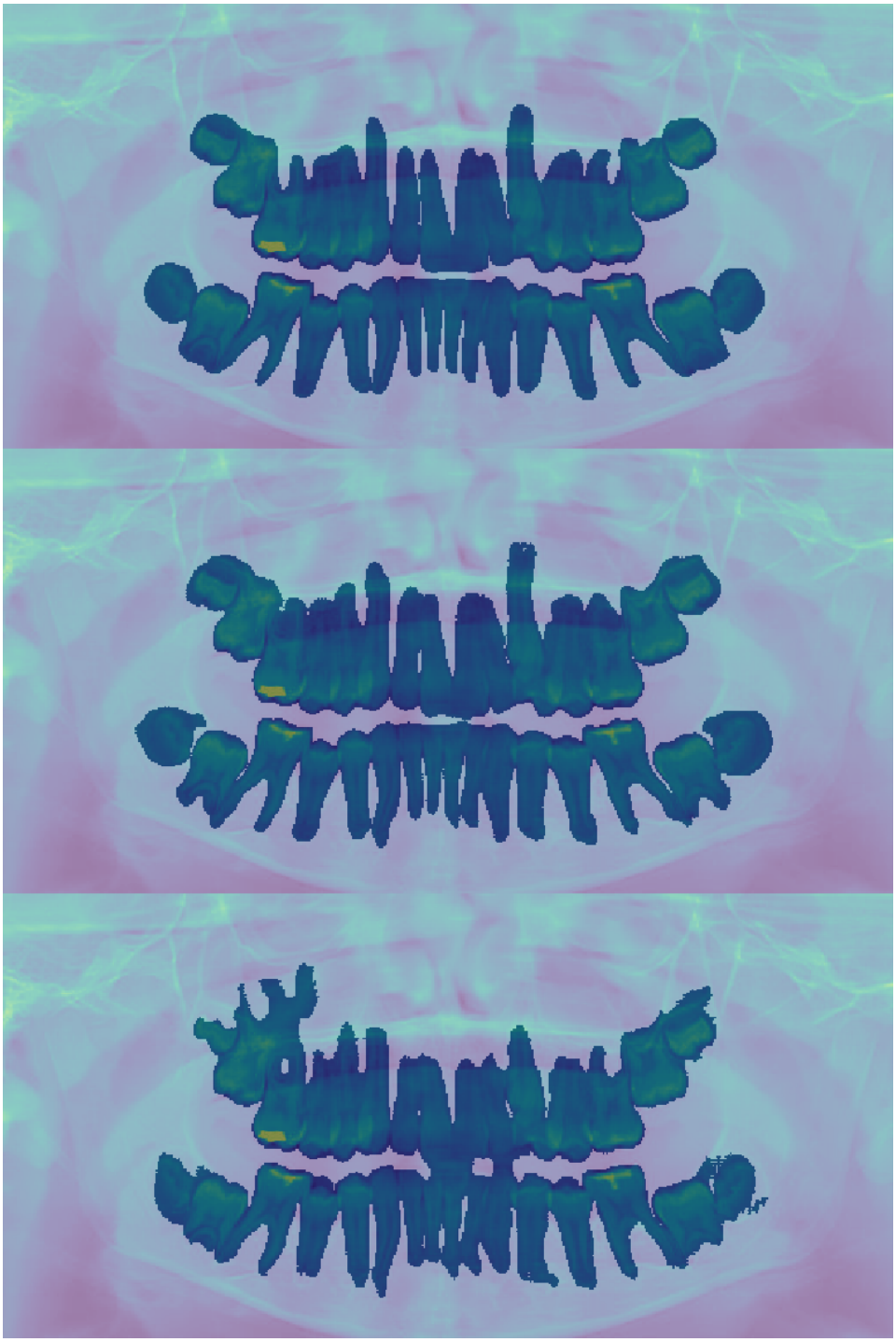

Figure 5.18:
1: **U-Net** Dice: 0.927, Jaccard: 0.864;
2: **Tiramisu** Dice: 0.908, Jaccard: 0.831;
3: **SegCaps** Dice: 0.888, Jaccard: 0.799

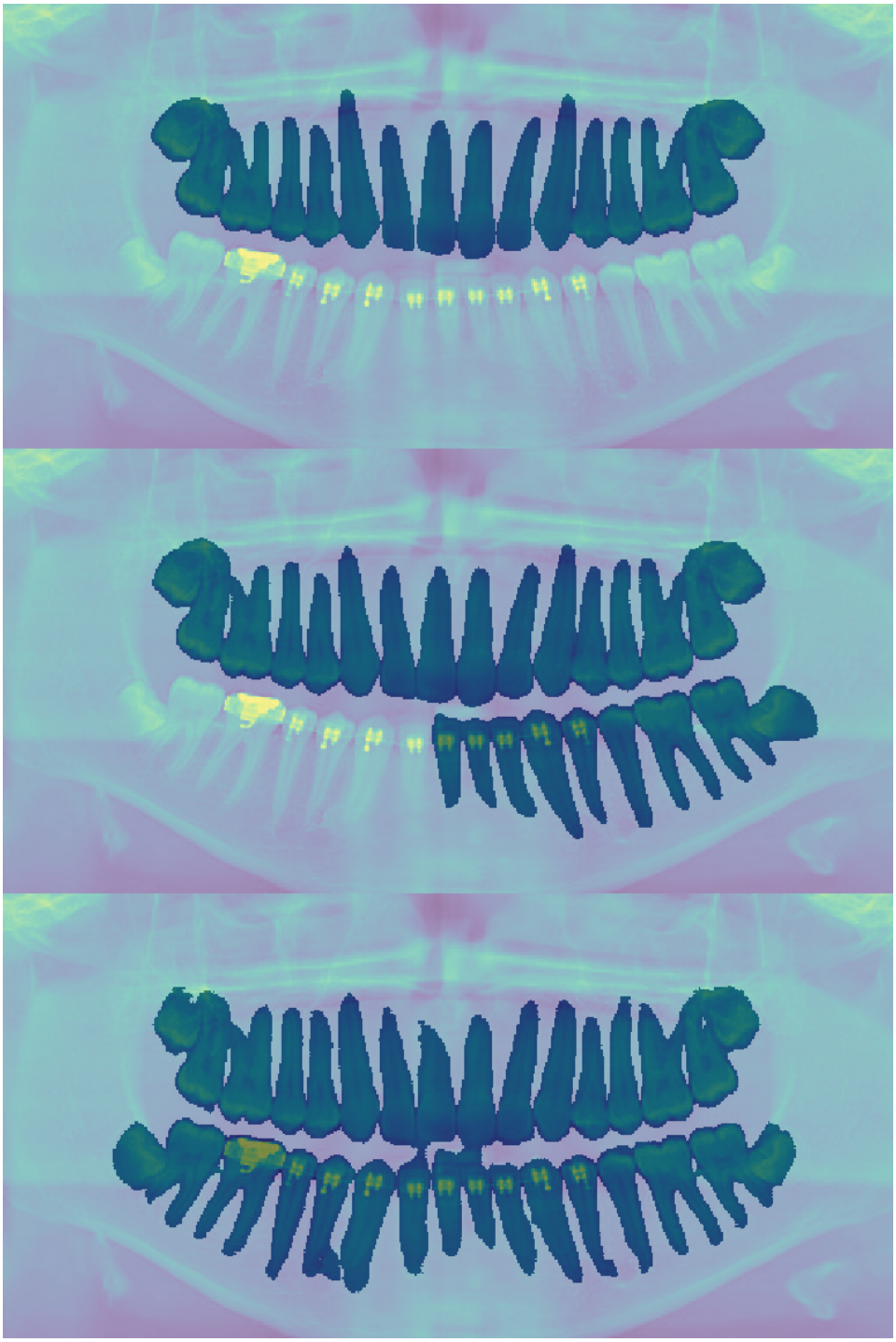

Figure 5.19:
1: **U-Net** Dice: 0.650, Jaccard: 0.482;
2: **Tiramisu** Dice: 0.812, Jaccard: 0.683;
3: **SegCaps** Dice: 0.900, Jaccard: 0.819

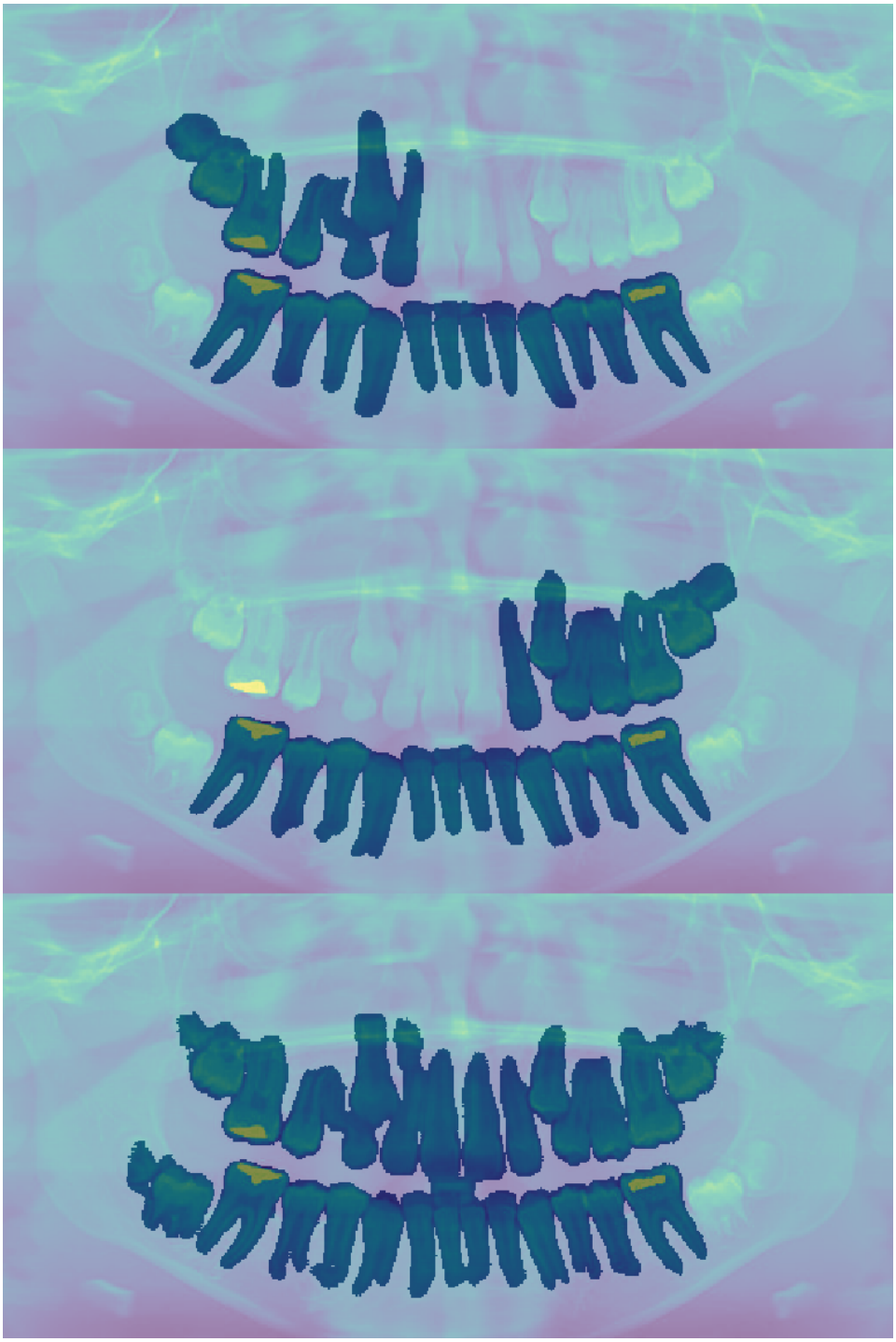

Figure 5.20:
1: **U-Net** Dice: 0.694, Jaccard: 0.532;
2: **Tiramisu** Dice: 0.699, Jaccard: 0.538;
3: **SegCaps** Dice: 0.8669, Jaccard: 0.763

5.3 Generative Adversarial Networks

For the evaluation of CapsNets inside Generative Adversarial Networks models we will use them as the Discriminator *D*. The general architecture tested follows the best practises of DCGANs[14]. In particular in this chapter we compare an almost vanilla DCGAN against a GAN created with the same generator as the DCGAN, but using a CapsNet discriminator. The goal of this test is to generate images from the **COIL-20** dataset[27], consisting of 20 objects depicted in 72 different viewpoints. This dataset is a subset of the COIL-100 dataset used in section 5.1, however the samples included in COIL-20 are grayscale images. The choice of this dataset instead of the full COIL-100 is dictated by the renowned difficulty of training of GANs models. Moreover the grayscale input images match the MNIST dataset on which the original CapsNet architecture has been defined. The original COIL-20 images of $128 \times 128 \times 1$ pixels are rescaled to $32 \times 32 \times 1$ for our tests involving GANs. Just like image classification, this dataset has been chosen for the presence of different viewpoints, which should highlight the characteristics of the CapsNets. One particular constraint is the number of training examples, only 1440, which can cause overfit.

The architecture of *G* uses a projection into a space of $16 \times 16 \times 128$, followed by a series of Convolution layers with 256 5×5 kernels, with in-between a Transposed Convolution layers using 256 4×4 kernels, stride 2, for upsampling to 32×32 pixels. For each of these layers the padding is set to *same*, and the activation is a *LeakyReLU*. The last layers consists of a convolution with 1 7×7 kernel, using *same* padding and *tanh* activation (see table 5.4).

Layer	**Description**
Input	latent dimension input shape
Fully connected	$128 \times 16 \times 16$
Reshape	$16 \times 16 \times 128$
Convolution 1	256 filters, 5×5 kernel, LeakyReLU
Transposed convolution 1	256 filters, 4×4 kernel, stride 2, LeakyReLU
Convolution 2	256 filters, 5×5 kernel, LeakyReLU
Transposed convolution 2	256 filters, 4×4 kernel, stride 2, LeakyReLU
Convolution 3	256 filters, 5×5 kernel, LeakyReLU
Convolution 4	1 filter, 7×7 kernel, tanh

Table 5.4: DCGAN generator

The architecture of the CNN discriminator follows the best practises of DCGANs, using only convolutional layers and LeakyReLU activations. A dropout of 0.6 is added in order to facilitate training and prevent the discriminator to overcome the generator (see table 5.5).

Layer	**Description**
Input	$32 \times 32 \times 3$ input shape
Convolution 1	128 filters, 3×3 kernel, LeakyReLU
Convolution 2	128 filters, 4×4 kernel, stride 2, LeakyReLU
Convolution 3	128 filters, 4×4 kernel, stride 2, LeakyReLU
Convolution 4	128 filters, 4×4 kernel, stride 2, LeakyReLU
Dropout	0.6 rate
Output	1 neuron, sigmoid

Table 5.5: CNN discriminator

The CapsNet discriminator uses the exact same architecture used for MNIST training in the original paper about dynamic routing, making use of a decoder for regularization (see table 5.6).

Layer	**Description**
Input	$32 \times 32 \times 1$ input shape
Convolution	256 filters, 9×9 kernel, stride 1
Primary Capsules	8-dimensional capsules, 32 channels, 9×9 kernel, stride 2
Output Capsules	20 16-dimensional capsules, 3 routing iterations
Decoder dense 1	512 neurons
Decoder dense 2	1024 neurons
Reshape	$32 \times 32 \times 1$ output shape

Table 5.6: Capsule Network discriminator

5.3.1 Training

In order to achieve a stable training for a GAN architecture, several hyperparameters must be tuned. All models have been trained with Adam optimizer, using $\beta_0 = 1$ and gradient clipping at 1.0.

The first thing that has been tested is the latent space dimension in input to the generator. After a series of tests, a latent dimension of 200 was chosen as a

compromise. In fact models trained with dimensions less than 200 resulted in poorer image quality, while increasing the size over 200 did not result in improvements.

Batch size was tested in values 16, 32 and 64. Regarding discriminator and generator learning rate, a preliminary step has highlighted the ranges of values for CNN and CapsNet. For the generator the values 0.00005, 0.0001 and 0.0005.

As far as the discriminator is concerned, different models required different learning rates. The CNN discriminator allowed a stable training with learning rates of 0.00001, 0.00002, 0.0001, 0.0005, while the CapsNet one preferred lower values of 0.00001, 0.00002, 0.0001.

All those different models have been trained for 10000 steps. During those steps the stable models were able to maintain in equilibrium both losses at around 0.7. During training CapsNets models have proven to be more stable to train, being able to maintain the equilibrium in the losses for all the time of training, while the losses of most of DCGANs models diverged after 6000 to 7000 steps, leading to the automatic discard of those models.

5.3.2 Results

For each possible combinations of batch size, discriminator and generator learning rate, each model has been evaluated after a fixed number of steps, but only for steps larger than 6000, based on the quality of generated images. Due to their variability in generated image quality, CapsNets models have been evaluated after 150 steps, compared to the 200 steps for CNNs models. Lots of models have been discarded because of completely black or white generated images, caused by the divergence of the generator and discriminator losses.

For the evaluation of the selected models, Inception Score and Fréchet Inception Distance have been used. These two metrics however use an InceptionV3 network, which has been conceived for large RGB images. The calculation of Inception Score has been possible using the same concept, but with the usage of a specific CNN trained on this specific dataset, having the same structure of the network described in table 5.2. The calculation of FID has been done replicating the single input channel and scaling the resolution to achieve the need input shape of $299 \times 299 \times 3$.

The entire process of evalaution selected 6 CapsNets models and 8 CNNs models. The results are available in tables 5.7 and 5.8.

On average, DCGANs performed better both on Inception Score and FID. Analyzing the generated images for the two architectures in figures 5.21 and 5.22, the motivation of those scores should be attributed to the better diversity of generated images deriving from the DCGANs. Images from CapsNets GANs however result in better overall quality.

Model	IS	FID
batch=16 disc_lr=0.0001 gen_lr=0.0001 step=8700	3.9354029	3.1004
batch=32 disc_lr=0.0001 gen_lr=0.0001 step=9000	3.9195573	3.1423
batch=32 disc_lr=0.00002 gen_lr=0.0001 step=9900	5.243708	3.2289
batch=64 disc_lr=0.0001 gen_lr=0.0001 step=8250	4.1814694	3.20337
batch=64 disc_lr=0.0001 gen_lr=0.0001 step=9750	5.4388456	2.8140
batch=64 disc_lr=0.00002 gen_lr=0.0001 step=9900	4.64521	3.16186

Table 5.7: CapsNet discriminator

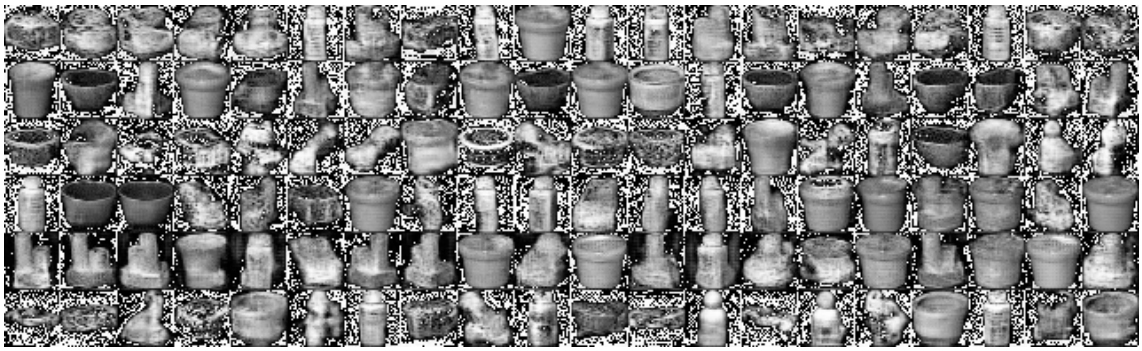

Figure 5.21: CapsNet generated images

Model	IS	FID
batch=16 disc_lr=0.0005 gen_lr=0.0001 step=9800	6.0332246	2.9204
batch=16 disc_lr=0.0005 gen_lr=0.00005 step=10000	5.3966613	3.2121
batch=16 disc_lr=0.00001 gen_lr=0.00005 step=9000	8.058795	3.0304
batch=16 disc_lr=0.00002 gen_lr=0.00005 step=9400	5.051762	3.1601
batch=32 disc_lr=0.0005 gen_lr=0.00005 step=9400	2.8316786	3.3522
batch=32 disc_lr=0.00001 gen_lr=0.00005 step=9800	6.610988	3.1083
batch=32 disc_lr=0.00002 gen_lr=0.00005 step=10000	8.551685	2.1765
batch=64 disc_lr=0.00001 gen_lr=0.00005 step=10000	6.4365435	3.0838

Table 5.8: CNN discriminator

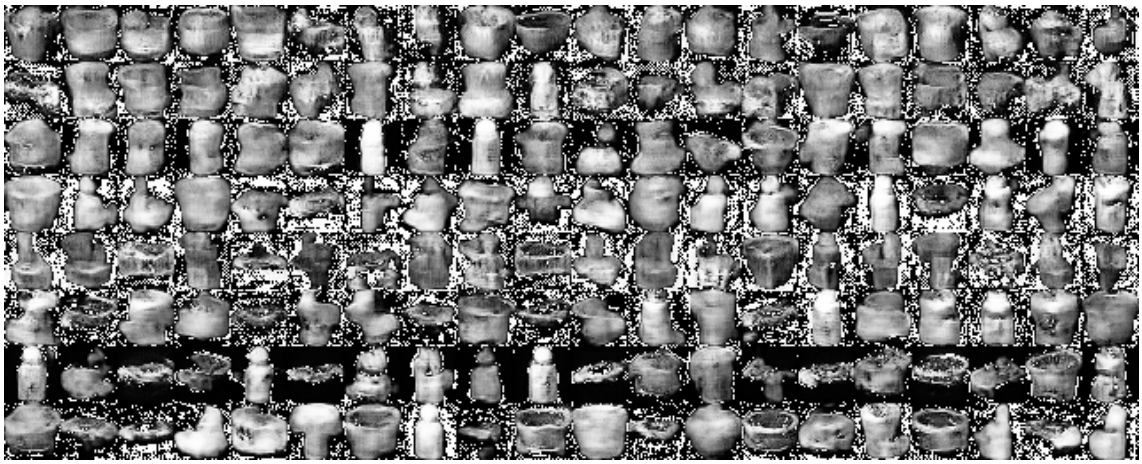

Figure 5.22: DCGAN generated images

Chapter 6

Conclusion and future work

In view of the results presented in the previous chapter, Capsule Networks appear as a promising but still very immature architecture for object recognition. Performance-wise they showed some interesting feature that can be further developed and exploited. During the analysis of image classification, CapsNets proved to be the ideal solution when dealing with dataset involving affine transformations. This scenario also apperead in complex cases for image segmentation, where SegCaps was able to delineate a complete mask for each of the samples. Regarding GANs, the Capsule Networks discriminator allowed the generation of good quality images, but lacking diversity. Digging deeper, the analysis of instantiation parameters of the output capsules confirmed the benefits deriving from using a vector as output instead of a single scalar.

Even though the goal of this thesis was to test Capsule Networks in more practical contexts, differently from the environment on which they have been originally tested, our analysis had to deal with limitations deriving from time, computing power and the architecture itself. The evaluation of each CapsNet model has always to deal with the overall slowness of the Dynamic Routing algorithm: it is in fact an iterative algorithm, which adds consistent time during training and inference phases, leading to the need of a lot of computing power in order to tune the hyperparameters of the networks. Current implementations of Capsule Networks do not run directly on GPU hardware but are implemented on high level programming languages, reducing the performances of these models. Moreover, this situation led to the impossibility of use of large images inside this architecture, both for time and memory issues. To cope with this problem, SegCaps introduced a method for training large images using layers containing capsules. A similar approach has been proposed by the same authors of the original paper with *Matrix Capsules with EM routing*[28], although its source code is still not available and

current implementations do not fully respect the original architecture presented in this paper.

Tests involving Capsule Networks against adversarial examples[29] proved the framework to be very robust against white box attacks[28]. Further research regarding adversarial attacks has been conducted with the goal to prevent attacks using the decoder derived from a Capsule Network[30].

Future research should be focused on improving the scalability of this architecture, allowing for large input images training and deeper networks. The purpose of an architecture of this kind is to give an artificial neural network a better understanding of the relationships among the elements present inside an image, with the final goal to achieve the best overall accuracy. In this situation training and inference times do not play a crucial role, however performance optimization should be definitively considered, especially regarding situations where hyperparameter tuning is a considerable factor.

In conclusion, deep learning models that use capsules with Dynamic Routing algorithm are not ready to replace Convolutional Neural Network in the current state. The small improvements deriving from the adoption of capsules do not justify the high price in term of performance of the model and limitations to datasets. Still, capsules provide a different and successful approach for dealing with generalization on new viepoints compared to a standard convolutional model. The design of capsules constitutes an important starting point for models able to distinguish objects using insights deriving from the characteristichs of the objects themselves, rather than introducing artificial information and increasing the network size.

Bibliography

[1] I. Goodfellow et al., *Deep Learning*, 2016.

[2] F. Rosenblatt, *The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain*, 1958.

[3] D. Rumelhart, G. Hinton, J. Williams, *Learning representations by back-propagating errors*, 1986.

[4] S. Sabour, N. Frosst, G. Hinton, *Dynamic Routing Between Capsules*, 2017.

[5] G. Hinton, A. Krizhevsky, S. Wang, *Transforming auto-encoders*, 2011.

[6] Y. LeCun, *Generalization and Network Design Strategies*, 1989.

[7] J. Long, E. Shelhamer, T. Darrell, *Fully Convolutional Networks for Semantic Segmentation*, 2014.

[8] O. Ronneberger, P. Fischer, T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015.

[9] , V. Badrinarayanan, A. Kendall, R. Cipolla, *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*, 2015.

[10] R. LaLonde, U. Bagci, *Capsules for Object Segmentation*, 2018.

[11] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, *Gradient-based learning applied to document recognition*, 1998.

[12] J. Deng, W. Dong, R. Socher, L. Li, K. Li, L. Fei-Fei, *ImageNet: A Large-Scale Hierarchical Image Database*, 2009.

[13] I. Goodfellow et al., *Generative Adversarial Nets*, 2014.

[14] A. Radford, L. Metz, S. Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, 2015.

[15] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, *Improved Techniques for Training GANs*, 2016.

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, *Going deeper with convolutions*, 2014.

[17] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, *Rethinking the Inception Architecture for Computer Vision*, 2015.

[18] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, S. Hochreiter, *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, 2017.

[19] G. Hinton *What's wrong with convolutional nets?*, MIT, https://techtv.mit.edu/collections/bcs/videos/30698-what-s-wrong-with-convolutional-nets, 2014.

[20] F. Chollet et al., *Keras*, https://keras.io, 2015.

[21] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng*TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, https://www.tensorflow.org, 2015.

[22] S. A. Nene, S. K. Nayar, H. Murase, *Columbia Object Image Library (COIL-100)*, Technical Report CUCS-005-96, 1996.

[23] D. P. Kingma, J. Ba *Adam: A Method for Stochastic Optimization*, 2014.

[24] X. Guo, *CapsNet-Keras*, https://github.com/XifengGuo/CapsNet-Keras, 2017.

[25] S. Jégou, M. Drozdzal, D. Vazquez, A. Romero, Y. Bengio *The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation*, 2016.

[26] G. Silva; L. Oliveira; M. Pithon, *Automatic segmenting teeth in X-ray images: Trends, a novel data set, benchmarking and future perspectives.*, 2018.

[27] S. A. Nene, S. K. Nayar, H. Murase, *Columbia Object Image Library (COIL-20)*, Technical Report CUCS-005-96, 1996.

[28] G. Hinton, S. Sabour, N. Frosst *Matrix capsules with EM routing*, 2018.

[29] I. Goodfellow, J. Shlens, C. Szegedy *Explaining and Harnessing Adversarial Examples*, 2014.

[30] N. Frosst, S. Sabour, G. Hinton, *DARCCC: Detecting Adversaries by Reconstruction from Class Conditional Capsules*, 2018.