An investigation of recent Deep Learning Techniques applied to blood cell image analysis

Andreas Forslöw

Magistratsvägen 55 K 1001 • 0739 55 39 03 • andreas.forslow@gmail.com

August 20, 2018

LUNDS UNIVERSITET

Abstract

This project has investigated the performances of Capsule Networks in comparison to Convolutional Neural Networks (CNNs) on white blood cell image classification at CellaVision. The Capsule Network models that were investigated are EM Routing Capsule Networks (EMCNs) [Hinton et al., 2018] and Dynamic Routing Capsule Networks (DCNs) [Hinton et al., 2017]. The models were compared with regards to convergence rate, speed, and accuracy performance on datasets with varying size and complexity. The results show that DCNs outperform the other models on small datasets with regards to accuracy and convergence rate, whereas the CNNs outperform the other models on bigger datasets with higher complexity. With regards to speed, CNNs outperform the other models on both CPU and GPU, with DCNs being very slow. EMCNs, meanwhile, give an indication of learning spatial concepts better, as they are more invariant to spatial- and noise transformations of the underlying test datasets.

Acknowledgements

Special thanks to Mattias Nilsson and Anders Heyden, who have been my mentors throughout this project and who have helped correcting this report. Mattias Nilsson has provided a great deal of guidance on how to construct the tests, as well as how to integrate CellaVision's data to the project, and understand the algorithms that were developed. Anders has provided much guidance with regards to analysing the data and how to proceed with regards to the project plan and data available. Furthermore, I would like to thank Johan Appelros for helping me out with keeping an eye on my running tests while I was away. Lastly, I would like to thank CellaVision for providing me with the opportunity to work with this project, as well as for providing a great and friendly work environment.

Contents

1 Nomenclature

Before dealing with the theory of the project, it is necessary to know the notational framework in which this text is written, as well as knowing some definitions of words that will be used. Thus, when reading the text, be aware of the following conventions:

- *Tensor*: In machine learning context, a tensor is simply an n-dimensional array. Thus, a 1-dimensional tensor is a vector, whereas a 2-dimensional tensor is a matrix. $[m_1, m_2, ..., m_n]$-notation for tensors will be used, which is equivalent to denoting the tensor dimensionality as $(\mathbb{R}^{m_1 \times m_2 \times ... \times m_n})$. For example, a [3]-tensor is an $\mathbb{R}^3$-vector, whereas a [3,4]-tensor is a $\mathbb{R}^{3\times4}$-matrix.
- *Image*: An image is a $[m_1, m_2, m_3]$-tensor denoted as $[W, H, C]$, where W is the width, H is the height, and C is the number of channels in the image. Each value in the tensor is a pixel value. In the case of gray-scale images, $C = 1$, and in the case of RGB-images, $C = 3$, yielding the tensors $[W, H, 1]$ and $[W, H, 3]$, respectively.
- $\mathcal{D}$: A dataset containing (image, label)-pairs. This may be used for training classification models, or testing these.
- $\mathcal{C}$: The set containing all possible classes for a dataset $\mathcal{D}$. If $\mathcal{C}$ contains images of digits 1, 2 and 3, $\mathcal{C} = \{1, 2, 3\}$.
- θ: Parameters used in statistical models. Most commonly denotes the parameters of the neural networks in the text.

2 Background

In recent years, deep learning has seen many advancements within the field of image analysis. Meanwhile, the field of hematology is heavily dependent on image analysis, as image data is an important component in diagnosing blood diseases. As such, CellaVision focuses on applying machine learning to blood cell image analysis in order to automate the diagnosing process. Deducing from this, there exists a great incentive in improving the underlying machine learning algorithms, as any such improvements lead to better diagnoses. With this in mind, this project will analyse the performance of some of the most recent techniques within machine learning when applied to blood cell image classification.

2.1 Objectives

The objective of this report is to analyse the performance of the capsule neural networks used in Hinton et al. [2017] and Hinton et al. [2018] when these are applied to white blood cell image analysis. The analysis will focus on the networks' accuracy and convergence rates on labeled white blood cell (WBC) image datasets with varying size and complexity, as well as their memory usage and execution times, and compare these with Convolutional Neural Networks (CNNs) - a technique which CellaVision currently uses. Furthermore, the analysis will explore changes to the Capsule Network algorithms for performance improvements.

2.2 Image classification

Simply defined, image classification is the task of classifying a set of images through an algorithm. Each image is paired with its correct class, its *label*, in order to measure and improve the performance of the algorithm. An example of a classification task is the MNIST classification (discussed in further detail in Section 2.5, in which images of digits 0-9 are to be classified as their proper digit classes 0-9. For example, given an image of 1 as shown in Figure 1 below, the algorithm should output the digit class 1.

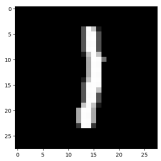


Figure 1: An image of the digit 1 from the MNIST data set.

The image classification task can be seen as a statistical problem, in which a model $\mathcal{M}$ should optimize its statistical parameters θ, such that it maximises the number of correct classifications on a training set $\mathcal{D}_{train}$ of (image, label)-pairs $\{(\mathbf{x}_1, l_1), (\mathbf{x}_2, l_2), ...\} \in \mathcal{D}_{train}$. By introducing a so-called loss function $\mathcal{J}$, which measures how much $\mathcal{M}$'s classifications deviate from the correct labeling, the optimization task is to find the parameters θ_{opt} which minimize $\mathcal{J}(\mathcal{M}|\mathcal{D}_{train})$. When θ_{opt} is found, $\mathcal{M}$ is tested against a test set $\mathcal{D}_{test}$ in order to check how well $\mathcal{M}$ performs on new data.

2.3 Deep learning

In deep learning, the model $\mathcal{M}$ - as described in Section 2.2 - can be described as a directed, weighted graph, which performs the mapping function $\mathbf{f} : \mathbf{x} \rightarrow \mathbf{a}$. Here, $\mathbf{x}$ is the input image, and $\mathbf{a}$ is $\mathcal{M}$'s

classification of $\mathbf{x}$; i.e. a vector with $|\mathcal{C}|$ entries - each entry c representing $\mathcal{M}$'s confidence in $\mathbf{x}$ belonging to class $c \in \mathcal{C}$ ($\mathcal{C}$ being the set of possible classes). As $\mathcal{M}$ can be described by a graph that may contain multiple layers of nodes, $\mathbf{f}$ may be a nested function which transforms $\mathbf{x}$ multiple times before producing the final output $\mathbf{a}$. Further intuition behind this will be given in Section 3, where the simplest model of deep learning - the fully connected neural network - will be described.

Moving forward, the graphs in deep learning are called artificial neural networks, and their nodes are called neurons, while their weighted edges are called weights. Furthermore, the output from one neuron may be transformed through multiple functions before producing the output of its consecutive neuron. Lastly, the optimization task described in Section 2.2 is divided into two parts; one forward pass where the model predicts the probability distribution $\mathbf{a}$ given an input image $\mathbf{x}$, and one backward pass where the model optimizes its parameters given $\mathbf{a}$ and the correct label c for $\mathbf{x}$. When performing this second step, the graph's direction is reversed. More intuition on this will be given in Section 3.

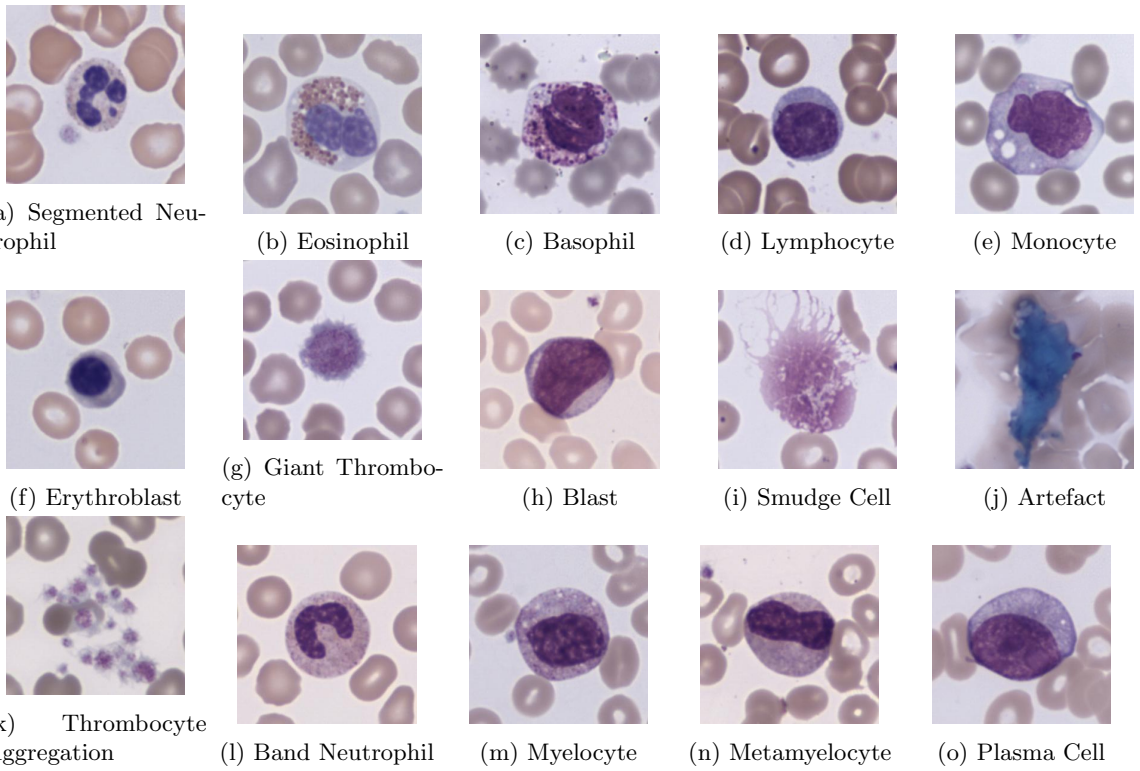
Furthermore, there exists many different neural network models. The models this report will focus on are *fully connected neural networks*, *convolutional neural networks*, *Dynamic Routing Capsule Networks (DCNs)* and *EM Routing Capsule Networks (EMCNs)*.

Lastly, there exists many different optimization problems apart from image classification. Another optimization problem involved in this project is the auto-encoder, which optimizes two neural networks - the encoder and the decoder - to encode an input image and decode an input image $\mathbf{x}$. A detailed explanation of this optimization task will be given in Section 5.

2.4 Hematology - White blood cell (WBC) analysis

As briefly stated in Section 2.1 above, this project's focus is on researching Capsule Networks' performances on CellaVision's labeled white blood cell image datasets, in order to conclude whether CellaVision should continue using Convolutional Neural Networks, or if they should switch to Capsule Networks for image classification. Thus, CellaVision's labeled white blood cell datasets will serve as the benchmark for the Capsule- and Convolutional Neural Networks' performances in this report.

Having explained the purpose of these datasets, their contents are shown in Figure 2 below. This figure shows samples of all the different white blood cell classes that will be used in this report. It is worth knowing that CellaVision's datasets contain more white blood cell classes than these, but that these classes were recommended for benchmarking by Mattias Nilsson and were thence chosen for this project.



(a) Segmented Neutrophil (b) Eosinophil (c) Basophil (d) Lymphocyte (e) Monocyte

(f) Erythroblast (g) Giant Thrombocyte (h) Blast (i) Smudge Cell (j) Artefact

(k) Thrombocyte Aggregation (l) Band Neutrophil (m) Myelocyte (n) Metamyelocyte (o) Plasma Cell

Figure 2: Example WBC images of the different cell types that will be used in the experiments conducted in this report. The images show thin, colorized smears of human blood, and are microscopy images showing an area of approximately 25×25 μm. The WBCs in the images are the cells with a purple color, and they consist of a dark purple/blue nucleus, grainy brown/purple granules outside of the nucleus, as well as a semi-transparent purple cytoplasm. Red blood cells are also present in the images as brown blobs with an often semi-transparent core. Artefacts are various objects that are neither white blood cells nor red blood cells - in the case of Figure 2j above, a kind of blue accumulation.

As can be seen, there exists many different kinds of white blood cells; all of which can be separated into granulocytes (also known as polymorphonuclear leucocytes) and mononuclear cells. Both cell types have granules and may contain a singular nucleus [Bain, 2004]. However, the granulocytes' granules are more prominent, and a mononuclear cell's nucleus is less variable (see Figure 3 below for further understanding). When doing WBC analyses, there also exists non-WBC cellular entities one has to take into consideration.

Granulocytes can further be classified into Neutrophils, Eosinophils and Basophils. Neutrophils, in turn, can be separated into **Segmented Neutrophils** - where the nucleus is divided into two to five segments connected with a thin strand of nuclear material - and **Banded Neutrophils**, where the nucleus is curved but not segmented. **Eosinophils** have a bilobed nucleus and a pale blue cytoplasm packed with large orange-red granules [Bain, 2004]. **Basophils** have a lobulated nucleus and a very pale blue cytoplasm packed by large purple-staining granules. These cell types can be seen in Figure 2, which consists of images from all the cell types that will be used in this report.

Moving on, mononuclear cells can be further divided into **Lymphocytes** and **Monocytes**. Lym-

phocytes have a pale blue and clear cytoplasm and a round or somewhat irregular outline [Bain, 2004]. A **Plasma cell** is a large type of lymphocyte with a high nucleus-to-cytoplasm ratio. Monocytes are the largest normal blood cells, and have a voluminous greyish-blue cytoplasm, a lobulated nucleus, and may be vacuolated or contain fine granules.

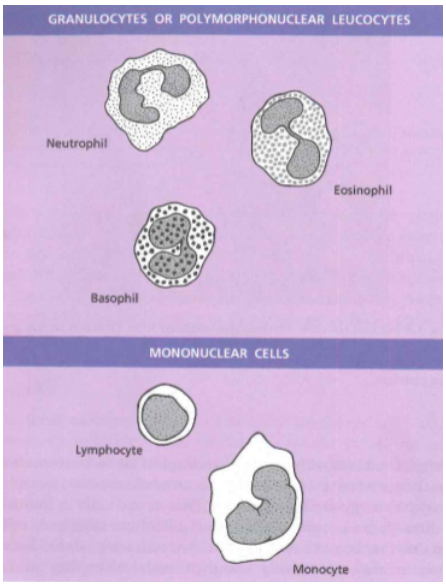


Figure 3: Image showing the two white blood cell types Granulocytes and Mononuclear cells, along with their sub-cell classes. Source: [Bain, 2004].

The cells mentioned above may also exist in different precursory states, as shown in Figure 4. Here, it is shown that **Myelocytes**, **Metamyelocytes** and Band Neutrophils are precursory states to Segmented Neutrophils, and that **Erythroblasts** are nucleated cells with a WBC-like structure, but precursory to red blood cells. Going further back in the chain, we find unipotent cells called **Blast cells**, which have large, comformable nuclei and few or no granules.

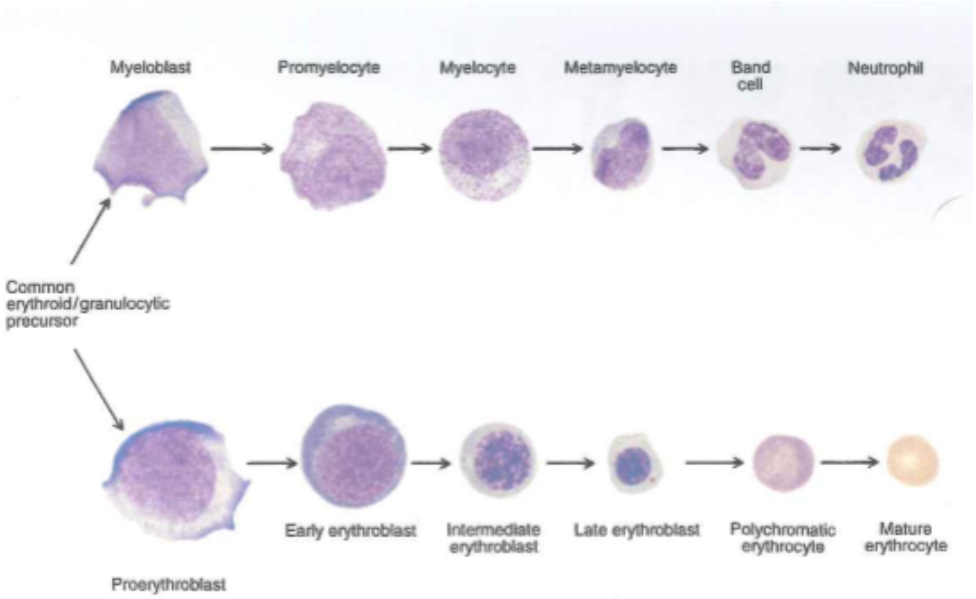


Figure 4: Image showing the precursory states of Neutrophils at the top and Erythrocytes (red blood cells) at the bottom. Source: [Bain, 2004].

Apart from Erythroblasts, there exists other cellular entities of interest in WBC analysis, as mentioned previously. Thrombocytes are fragments of the cytoplasm of cells which have no nucleus, and these initiate blood clots in order to stop bleeding [Bain, 2004]. This report will involve two Thrombocyte types: **Giant Thrombocytes** and **Thrombocyte Aggregations**, both of which are self-explanatory. Furthermore, **Smudge cells** are broken cells, and lastly **Artefacts** are, in this report, other types of visual entities that show up in blood cell tests and cannot be classified.

Owing to this information about blood cell analysis, it is easy to understand that image classification in WBC analysis may be hard, and differentiating between cells in consecutive states, such as Band Neutrophils and Segmented Neutrophil may be a task with which even experts struggle, due to continuity. As such, image classification algorithms may not be able to theoretically achieve 100% classification accuracy on WBC classification tasks.

2.5 MNIST Dataset

The MNIST dataset plays a minor role in this report, but plays a major role in Hinton et al. [2017], which is the Dynamic Routing Capsule Network article that serves part of the fundament to this report. Knowing what the MNIST dataset is may thus be an important component in understanding the Dynamic Routing Capsule Network (e.g. the shown network models in Dynamic Routing Capsule Network Section, Section 6 below). However, they are not used for benchmarking in this report, and are thus only important for conceptual purposes.

Moving on, the MNIST dataset used in this paper consists of $[28, 28, 1]$-images with labels of digits 0-9. This dataset is divided into a training set, $\mathcal{D}_{train}$, a validation set $\mathcal{D}_{validation}$ and a test set $\mathcal{D}_{test}$, containing 55,000, 5,000 and 10,000 (image, label)-pairs, respectively. An example image of this set is shown in Figure 5 below.

Figure 5: Example images of the MNIST dataset. Each image is a $[28, 28, 1]$-tensor and a corresponding label.

3 Basics of artificial neural networks (ANNs)

3.1 Fully connected neural networks

The simplest kind of neural network is the fully connected neural network, as seen in Figure 6. In general, a neural network is a directed, weighted graph that receives an input vector, applies a series of transformations on this according to the graph structure, and outputs a vector. In the case of image classification, the input is often a vector representation of all pixel intensities, and the output is a vectorized probability distribution between all possible classes that the graph classifies between.

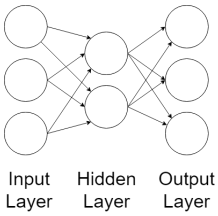


Figure 6: A fully connected neural network. This is a directed, weighted graph with an input vector length 3 and an output length 3. The weights are not shown, due to lack of space. See Figure 7 for more info on how the network works between layers.

A node in a neural network is called a neuron, and its purpose is to apply an activation function to the sum of its weighted input and send this result to the next neuron, which will follow the same procedure. Passing data through the graph in this routing procedure is called *forward propagation*.

3.2 Routing procedure/Forward propagation

Figure 7 describes how forward propagation works between two layers of a graph. The function f is the activation function, and will be described in Section 3.3. As described in Figure 7, each neuron a_i^L in layer L transforms its input data according to (1) and forwards this value to the next neurons in the graph's direction. This routing procedure continues until the graph outputs a vector $\mathbf{a^{out}}$. Here, i and j represent neuron indices in layer L and $L+1$, respectively, and $\mathbf{a^{out}}$ represents the output from the final layer of the graph.

$$a_j^L = f(\sum\nolimits_{a_i^{L-1} \in a^{L-1}} (w_{ij} a_i^{L-1}) + b_j) \tag{1}$$

Following the definitions in Section 2.2, if the range of classes is denoted $\mathcal{C}$, and the set of input data points is denoted $\mathcal{D}$, $\mathbf{a^{out}}$ represents a probability distribution between graph observations of $\mathcal{C}$, given $\mathcal{D}$. Expanding upon the concept of the neuron and forward propagation, one can build a neural network with an arbitrary number of layers, input data length and output data length.

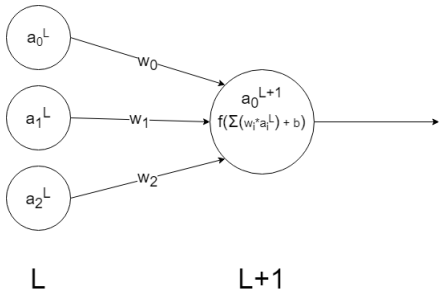


Figure 7: Image describing basic structure of a simple artificial neural network. Data is passed from neurons a_0^L, a_1^L, a_2^L at layer L to a_0^{L+1} at layer L+1 through the weighted edges, and at the neuron a_0^{L+1}, the sum of the weighted inputs, $f(\sum (w_i \cdot a_i^L) + b)$ is calculated, where f is the activation function and b is a bias term that is independent from the input.

3.3 Activation function

The purpose of the activation function is to scale down the received input information - often to a value between values 0 and 1 - and send this scaled data to the next neuron. One classic activation function is the sigmoid function described in (2), which squashes the input sum to a value between 0 and 1. Another commonly used activation function is the ReLU (Rectified Linear Unit) activation function described in (3), which sets a lower bound for the output of the neuron.

$$\text{sigmoid}(x) = \frac{e^{-x}}{1 + e^{-x}} \tag{2}$$

$$\text{ReLU}(x) = \max(0, x) \tag{3}$$

3.4 Loss function

When the above mentioned probability distribution has been calculated, it is sent to a loss function $\mathcal{J}$. $\mathcal{J}$ calculates the differences between predictions and correct values across the set of data points N, and summarises these differences in a scalar value. Thus, $\mathcal{J}$ is a measure of network prediction error, and the optimization task becomes finding the θ for which (4) is minimized, where $\boldsymbol{x}$ is the input to the graph, y is the target, θ is the network parameters and f is the network seen as a function.

$$\mathcal{J} = \min_{\theta} \mathcal{J}(f(\theta, \mathbf{x}), y) \tag{4}$$

One example of a loss function is the cross entropy loss function, which is explained by (5) below [Nielsen, 2015, ch. 3]. Here, N denotes the total number of items in the training data, $\mathcal{C}$ denotes the number of classes in the class domain $\mathcal{C}$, $a_{i,n}^{out} \in [0, 1]$ and $y_{i,n} \in [0, 1]$ describe the activation and correct label, respectively, for output neuron i at data point n. The purpose of this loss function becomes obvious when setting $y_{i,n} = 0$ and $y_{i,n} = 1$, as either case cancels out one of the terms and causes the loss function to punish any $a_{i,n}^{out}$ that deviates from the desired output $y_{i,n}$.

$$\mathcal{J} = \frac{1}{N} \sum_{n=1}^{N} \sum_{i=0}^{|\mathcal{C}|} \left[-y_{i,n} \ln(a_{i,n}^{out}) - (1 - y_{i,n}) \ln(1 - a_{i,n}^{out}) \right] \tag{5}$$

Another example of a loss function is the sum of square differences (6), which punishes squared distances between prediction $a_{i,n}^{out}$ and desired output $y_{i,n}$.

$$\mathcal{J} = \frac{1}{N}\sum_{n=1}^{N}\sum_{i=0}^{|\mathcal{C}|}(a_{i,n}^{out} - y_{i,n})^2 \tag{6}$$

3.5 Back propagation/Optimization

Following calculation of $\mathcal{J}$ comes an optimization process called backpropagation. In brief, back propagation differentiates the loss function with respect to the network parameters, and adjusts the network's parameters according to this differentiated loss. As the negative gradient direction points towards a local minimum of the loss function, taking a step in this direction decreases the loss and increases the accuracy of the model.

One commonly used backpropagation optimization algorithm is gradient descent, as described by (7). θ are the network parameters and α is the *learning rate*. The learning rate adjusts the step length for back propagation, and thus sets the algorithm's convergence rate. If the step length is too big, the algorithm diverges. If the step length is too small, the algorithm converges very slowly.

$$\theta \leftarrow \theta - \alpha\nabla_\theta\mathcal{J} \tag{7}$$

There exists many different optimization algorithms, and due to its computational efficiency and good performance this report will use the Adam optimizer algorithm described in Kingma and Ba [2015]. Furthermore, this paper will at instances use an exponential decay of the learning rate α, which means that α is multiplied by a factor (e.g. 0.99) every n'th step (e.g. every 100'th step) during training. This approach decreases the gradient steps as learning proceeds, which means that the network parameters are fine-tuned as learning proceeds.

As an example of how back propagation works, consider the case where a network should optimize a loss function over a single input sample $n \in N$. If the graph in Figure 7 is used, $a_0^{L+1} = a_0^{out}$, and $a_i^L = a_i^{in}$, a system of equations as shown in (8) and (9) is created. Here, $a_{i,n}^{in}$ and $a_{i,n}^{out}$ is the i:th neuron for the data point n in the input and output layer, respectively.

$$a_{0,n}^{out} = f(\sum_{i=1}^{M}(w_i a_{i,n}^{in}) + b) = f(w_0 a_{0,n}^{in} + w_1 a_{1,n}^{in} + w_2 a_{2,n}^{in} + b) \tag{8}$$

$$\mathcal{J} = \sum_{i=0}^{|\mathcal{C}|}(a_{i,n}^{out} - y_{i,n})^2 \tag{9}$$

The parameters of the network, θ, consist of the weights w and the bias b. The differentiated loss $\frac{\partial L}{\partial w_i}$ can be obtained through the chain rule: $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a_0^{out}}\frac{\partial a_0^{out}}{\partial w_i}$. The components of this equation are given in (12), (10) and (11) respectively. Finally, the weight w_i is updated according to (13). $\frac{\partial L}{\partial b}$ is calculated and updated in the same manner as $\frac{\partial L}{\partial w_i}$. For the single input sample $n \in N$, we get:

$$\frac{\partial \mathcal{J}}{\partial a_o^{out}} = 2(a_0^{out} - y) \tag{10}$$

$$\frac{\partial a_0^{out}}{\partial w_i} = f'(a_0^{in}w_0 + a_1^{in}w_1 + a_2^{in}w_2 + b)\cdot a_i^{in} \tag{11}$$

$$\frac{\partial \mathcal{J}}{\partial w_i} = 2(a_0^{out} - y) f'(a_0^{in} w_0 + a_1^{in} w_1 + a_2^{in} w_2 + b) \cdot a_i^{in} \tag{12}$$

$$w_i \leftarrow w_i - \alpha \frac{\partial \mathcal{J}}{\partial w_i^n} \tag{13}$$

This above procedure can be generalized to multiple neural layers between input and output. In a real setting, one would also optimize the graph over all input samples N. If all points N are used for calculating $\frac{\partial \mathcal{J}}{\partial a_0^{out}}$, it is called a **deterministic gradient** method, which is very slow due to the fact that it evaluates the model on every data point in the data point in each step [Goodfellow et al., 2016, ch. 8.1.3]. A more commonly used gradient method is the **minibatch gradient** method, where a minibatch of m samples $\{\mathbf{x}_1, ..., \mathbf{x}_m\}$ and corresponding labels y_i are sampled for each step, resulting in (14). Here, $\hat{\mathbf{g}}$ is the calculated loss for the next batch of m samples.

$$\begin{aligned} \hat{\mathbf{g}} &= \frac{1}{m} \nabla_\theta \sum_i^m \mathcal{J}(f(\mathbf{x}_i, \theta), y_i) \\ \theta &\leftarrow \theta - \alpha \hat{\mathbf{g}} \end{aligned} \tag{14}$$

3.6 Generalization

A desired and important ability of an ANN is its ability to generalize to new input data. A problem that may arise when training ANNs is overfitting, which happens when the model has too many parameters; making it too flexible as it fits very well with the available data, but fails to generalize to new data points [Nielsen, 2015, ch. 3]. In order to test overfitting, datasets are often divided into *train-*, *validation-* and *test* datasets; $\mathcal{D}_{train}$, $\mathcal{D}_{val}$ and $\mathcal{D}_{test}$. $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$ are used to train and test a specific model θ, whereas $\mathcal{D}_{test}$ is used to test the performance of different models $\{\theta_1, \theta_2, ...\}$. This is analogous to the classification process described in Section 2.2, with the difference that multiple models θ are now to be trained and evaluated.

3.6.1 Regularization

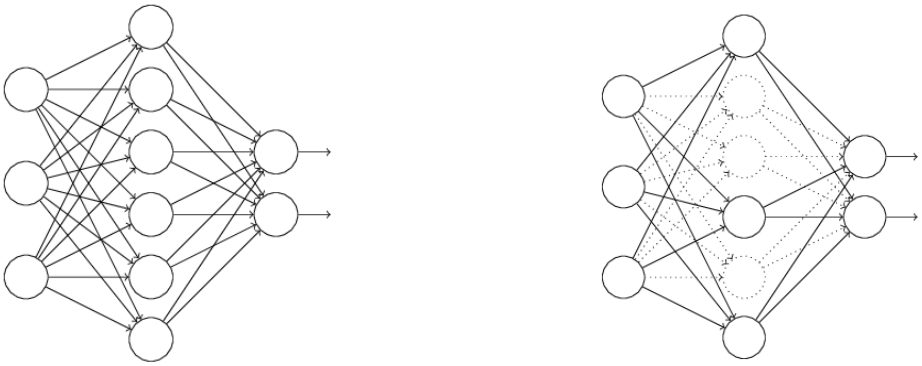
One way to prevent overfitting is to use regularization, where a regularization term is added to the loss function, penalizing parameters if they get too specialised. A commonly used regularization method is the L^2 regularization (also called Ridge Regression) described by (15). Here, $\mathcal{J}$ is the regular loss, λ is a weight parameter determining the magnitude of the regularization loss and θ is the set of network parameters. Here, weights are penalized for growing large; preventing overfitting by preventing any single weight from overenhancing signals between neurons.

$$\mathcal{J}_{reg} = \mathcal{J} + \lambda \sum_{\theta_k \in \theta} \theta_k^2 \tag{15}$$

3.6.2 Dropout

Another way to prevent overfitting is to use dropout. This algorithm is similar to the random forest algorithm, which uses multiple decision trees in order to vote for an output. Here, instead of decision trees, multiple sub-ANNs are used to train a common ANN. This prevents overfitting, as training of the common ANN is averaged between the sub-ANNs.

Dropout takes a common ANN θ_c (see Figure 8a) and randomly divides this into subgraphs $\{\theta_1, \theta_2, ...\}$ by randomly and temporarily deleting some fraction of the hidden weights of the network (see Figure 8b). Often, the fraction is half of the network, which means that when running the whole network, twice as many neurons will be active, which means that the outgoing weights of the subgraphs need to be halved.



(a) The common network, θ_c, that is to be trained.

(b) Subnetwork of θ_c described in Figure 8a, constructed through the dropout procedure.

Figure 8: Networks describing the graph construction in dropout. Source: [Nielsen, 2015, ch. 3]

3.6.3 Data augmentation

Yet another way to prevent overfitting is to increase the amount of available data for training the network, as the parameters θ will have more data to represent. As such, it is often beneficial to augment the available data in some way, as this expands the amount of available data to very little cost. Examples of augmentations include **vertical**- and **horizontal** flips, **rotation**, **translation** and **cropping** of the images, as well as noise addition such as **gaussian noise**. Rotation is easiest done by rotating a big image and cropping this down to the input size of the network. Translation is easiest done by cropping down a big image in different locations.

3.7 Weight initialization

An important part of constructing the neural network is weight initialization; i.e. choosing the initial weight values before training starts. Too small and too large weights can lead to vanishing and exploding signals, respectively, as a signal is propagated through the network. Initializing all the weights to zero will cause the weight updates to be identical. The result of poor weight initialization will be failure of, or slow, convergence [Khan et al., 2018, ch. 5.1]. Thus, there exist several different approaches to weight initialization; most of which are random initializations.

One such approach is the *Gaussian random initialization*. In this approach, the weights are sampled from a gaussian distribution with $\mu = 0$ and a small σ (e.g. 0.1 and 0.01) [Khan et al., 2018, ch. 5.1.1]. A version of this approach is the *Truncated normal initialization*, in which the weights are sampled from a truncated normal distribution.

Another approach is the *Xavier initialization*, in which the weights are sampled from a probability distribution with $\mu = 0$ and variance according to (16). Here, w are the network weights and n_{f-in} and n_{f-out} are the number incoming and outgoing connections from a neuron. This initialization makes

the neurons invariant to their number of incoming and outgoing connections; creating a robustness for network size changes. Although this approach is statistically inaccurate when the network contains nonlinearities, Xavier initialization works quite well in practice and leads to better convergence rates [Khan et al., 2018, ch. 5.1.5].

$$var(w) = \frac{2}{n_{f-in} + n_{f-out}} \tag{16}$$

3.8 Batch Normalization

When training an ANN, the inputs $\mathbf{x}$ to each layer is affected by the parameters of all the preceding layers, which means that small changes to the network parameters amplify as the network grows deeper. In order to prevent this, the input to the network layers is normalized through a two-step process called *batch normalization*.

The first step (17) applies a function $\mathbf{f} : \mathbf{x} \rightarrow \hat{\mathbf{x}}$, such that $\hat{\mathbf{x}}$ has mean 0 and variance 1. This normalization step makes the network more robust to parameter changes, but also results in representational loss, as e.g. the identity transformation $\mathbf{f} : \mathbf{x} \rightarrow \mathbf{x}$ is unattainable.

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mathbb{E}(\mathbf{x})}{\sqrt{\text{Var}[\mathbf{x}]}} \tag{17}$$

To eliminate this representational loss, Ioffe and Szegedy [2015] introduces a second step in batch normalization, (18). Here, γ and β are learnable parameters, $\odot$ is the Hadamard product, and $\mathbf{y}$ is the batch normalized input to the ANN layer. By introducing γ and β, $\mathbf{f} : \mathbf{x} \rightarrow \mathbf{x}$ may be achieved by setting $\gamma = \sqrt{\text{Var}\mathbf{x}}$ and $\beta = \mathbb{E}[\mathbf{x}]$. Thus, this normalization process greatly enhances representational flexibility.

$$\mathbf{y} = \gamma \odot \hat{\mathbf{x}} + \beta \tag{18}$$

4 Convolutional Neural Networks (CNNs)

In a fully connected neural network, weights are statically connected between each neuron in layer L and each neuron in layer $L+1$. Each layer is a 1D-grid of neurons and the routing procedure between each layer is a simple scalar product between each weight and input neuron. In a convolutional neural network, however, the connection between layer L and layer $L+1$ is a 3D-kernel of weights that is convolved with layer L in order to create the input to layer $L+1$. In short, each layer is now a 3D-grid of neurons and the routing procedure is a 2D convolution between layer L and the kernel. An example of a convolutional neural network can be seen in Figure 9. In addition to above described convolutional layers, this network also involves pooling layers, which will be described later in this section.

4.1 Convolution layers in CNNs

Further intuition behind the routing procedure of a convolutional network can be seen in Figure 10 and 11 [Nielsen, 2015]. Here, the input neuron layer is a $[W_1, H_1, 1]$-grid, and the kernel is a $[5, 5, 1]$-grid of weights. As can be seen, these weights are convolved with the input image in order to obtain the $[W_2, H_2, 1]$ first hidden layer grid.

The output of a single neuron in the hidden layer above, a_{x_2,y_2}, can be described by (19). This generalizes to how convolution works for an entire convolutional neural network and it is analogous to

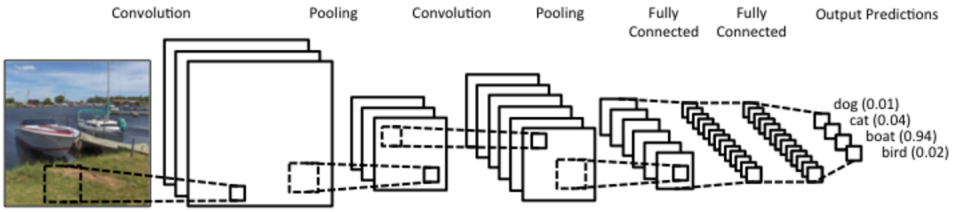


Figure 9: A Convolutional Neural Network. An input image is sent to the network, where a kernel is convoluted over the image to produce a hidden layer of 2D-grid neurons. Each layer can contain multiple 2D-grids of neurons. The core routing procedure between each layer is a convoluting kernel of weights. There exists other procedures such as max pooling, that will be explained later, as well. Source: [Karn, 2016].

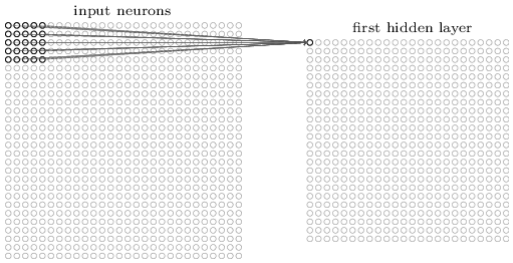


Figure 10: Routing procedure, showing a convolution step between kernel and input grid. Source: [Nielsen, 2015].

a neuron's output in a fully connected neural network. Here, f is the activation function as described in Section 3, a is the input neuron, w is the kernel weight, b is a bias, k_1 and k_2 are kernel indices, and x_1, y_1, x_2, y_2 are grid indices in the first and second layer, respectively.

$$a_{x_2,y_2} = f\left(b + \sum_{k_1=0}^{4}\sum_{k_2=0}^{4} w_{k_1,k_2} a_{x_1+k_1,y_1+k_2}\right) \tag{19}$$

As described in Figure 9, however, each layer in a CNN may be 3-dimensional; i.e. an input layer may be $[n_1, n_2, C_1]$-dimensional. In this case, the kernel becomes $[k_1, k_2, C_1]$-dimensional; its depth (number of channels) always matches the input layer's depth. In order to create C_2 channels at the hidden layer, C_2 number of kernels are applied to the input layer. Often, this is denoted as using a $[k_1, k_2, C_2]$-kernel (C_1 is omitted as it is not an adjustable parameter), and through the rules of 2-dimensional convolution, the hidden layer's dimensionality now becomes $[W_2, H_2, C_2]$.

In the example shown in Figure 10 and 11, the kernel size is $[5, 5, 1] = [k_1, k_2, 1]$, or $[k_1, k_2]$ in short. The kernel is strided over the input grid with a step length of 1 in the column direction and 1 in the row direction; i.e. its *stride* is $[1, 1]$. If a layer L with grid size $[x_1, y_1, C_1]$ is convoluted with C_2 kernels of size $[K_1, K_2, C_1]$ and a stride of $[S_1, S_2]$, the layer *L+1* will have the dimensionality

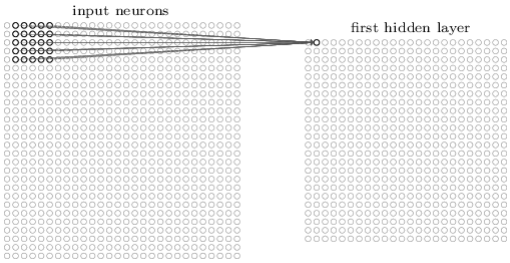


Figure 11: Routing procedure, showing another convolution step between kernel and input grid. Source: [Nielsen, 2015].

$[W_2, H_2, C_2]$ according to (20).

$$(W_2, H_2, C_2) = \left(\left\lfloor \frac{W_1 - K_1}{s_1} \right\rfloor + 1, \left\lfloor \frac{H_1 - K_2}{s_2} \right\rfloor + 1, C_2 \right) \tag{20}$$

Having explained the basics of convolution between layers in a CNN, the general formula for convolution is shown in (21). Here, $[S_1, S_2]$ is the stride, $[K_1, K_2]$ is the kernel size, $[x_2, x_2]$ is the second layer's grid coordinates, C_1 is the number of channels in the first layer, C_2 is the number of channels in the second layer, and $a^{(1)}$, $a^{(2)}$ is the output of the first and second layer, respectively.

$$a^{(2)}_{x_2,y_2,C_2} = f\left(b_{C_2} + \sum_{c_1 \in C_1} \sum_{l=0}^{K_1-1} \sum_{m=0}^{K_2-1} w_{(c_2,l,m,c_1)} a^{(1)}_{(s_1 x_2 + l, s_2 y_2 + m, c)} \right) \tag{21}$$

4.2 Padding

As can be seen from (20) and Figure 10, layer *L+1*'s 2D-grid is a fixed size smaller or equal to layer *L*'s. This grid size can, however, be adjusted by padding the input grid. The way this is most often done is through zero padding, where 0-values are added to the edges of the input grid according to Figure 12. A common procedure is to zero pad such that the dimensions of layer L are preserved in layer *L+1*. In this text, this type of padding is labeled as *padding: same*, and no padding is labeled as *padding: valid*. Another commonly used padding method is replicating the values on the border; a method which, however, will not be used in this project.

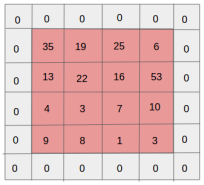


Figure 12: Zero padding of a CNN layer. Source: [Saxena]

4.3 Pooling layers in CNNs

In addition to convolution layers, CNNs can contain pooling layers. These layers condenses the output from the previous layer according to the pooling function, as can be seen in Figure 13, [Nielsen, 2015]. In this figure, the pooling function is the max function, which means that the function computes the maximum of a region in the input layer, and outputs this value to the next layer. This type of pooling layer is called a max pooling layer.

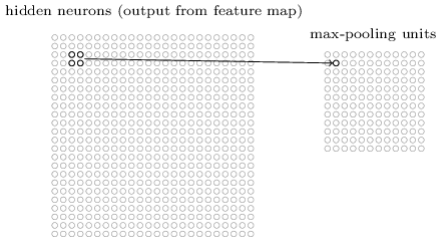


Figure 13: Max pooling layer, where the maximum of a region in layer L is sent to the region's corresponding neuron in layer *L+1*. A feature map is simply the output from a neural layer. Source: Nielsen [2015].

Another type of pooling layer that will be used in this report is the average pooling layer, which replaces the maximum function above with an average function that calculates the average of above mentioned region.

5 Autoencoders

An autoencoder is a neural network that tries to copy its input to its output [Goodfellow et al., 2016]; it has as many input parameters as output parameters. The autoencoder consists of two parts - the *encoder* and the *decoder*. The encoder transforms the input $\boldsymbol{x}$ to a hidden layer $\mathbf{h} = f(\mathbf{x})$, which is an encoding for the entity represented in the input. The decoder transforms $\boldsymbol{h}$ to a reconstruction $\mathbf{r} = g(\mathbf{h})$ layer, and its purpose is reconstruct $\boldsymbol{x}$ given the encoding $\boldsymbol{h}$. Mathematically, the autoencoder objective can be expressed as minimizing a loss function $L = L(\mathbf{x}, g(f(\mathbf{x})))$ - for example the mean of squares difference described in (22) below. A visual representation of an autoencoder can be seen in Figure 14.

$$L(\mathbf{x}, g(f(\mathbf{x}))) = \frac{1}{|\mathbf{x}|} \sum_{x \in \mathbf{x}} \left(x - g(f(x))\right)^2 \tag{22}$$

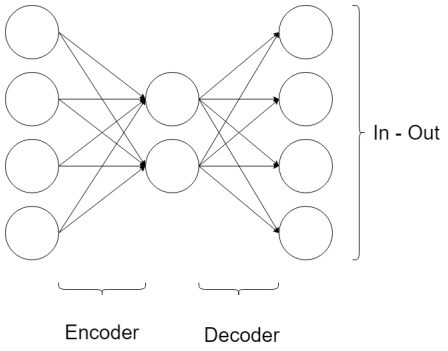


Figure 14: A simple autoencoder using a fully connected neural network. The input and output vectors have the same dimension; i.e. they are both 4-dimensional vectors. The network is divided into an encoder network, where the 4D input vector is scaled down to 2D, and a decoder network, where this 2D vector is rescaled to 4D. The network tries to minimize the difference between the input and output vector.

A common architecture for autoencoders is to keep $\boldsymbol{h}$ small (smaller to much smaller than the input dimension) in order to prevent **undercomplete** (learning too few parameters, only capturing the most salient features of the training data) and **overcomplete** (learning too many parameters, making the system too specific and failing to generalize over the data distribution) autoencoders [Goodfellow et al., 2016]. Both fully connected neural networks and CNNs can be used in autoencoders; both in the encoder and the decoder. This fact is easily grasped when dealing with fully connected neural networks, as these can scale input data arbitrarily.

In the case of CNNs, however, traditional convolutional- and max pool layers cannot upsample data. To solve this problem, upsampling may be done using an interpolaton method such as nearest neighbor or bilinear interpolation, followed by a CNN layer. As nearest neighbor interpolation followed by a traditional convolution is the recommended approach [Odena et al., 2016], it will be used in this report. For more information about nearest neighbor and bilinear interpolation, refer to [Wu et al., 2008, p. 53].

6 Capsule networks - Dynamic Routing

In short, a Dynamic Routing Capsule network introduces two new concepts to a neural network:

- It expands the neuron from a 1-dimensional entity to an n-dimensional vector called a capsule.
- It creates an iterative election process in which capsules from a layer Ω_L have to determine a concensus for the capsule activations of layer Ω_{L+1}.

A CNN layer could be described as a $[W, H]$-grid (H rows, W columns) of scalar values, set across C channels. A dense way of describing this structure is through a $[W, H, 1, C]$-tensor, consisting of C channels of $[W, H, 1]$-dimensional grids.

Structurally, a CNN represented in this way makes it obvious that a CNN structure is a special case of a Dynamic Capsule Network (DCN) structure, which may be described as a $[W, H, n, C]$-tensor when the neuron becomes an n-vector (i.e. capsule). The reason for creating these n-dimensional capsules, is that they encode more information than a 1-dimensional neuron. Each capsule is activated if a

particular entity/feature (that it encodes) is present in its input field, and an n-dimensional vector may encode many different variations ("instantiations"), e.g. orientation, width, position, brightness. For clarity purposes, the capsule dimension is called thickness, which means that the above mentioned capsule has thickness n.

The channel structure remains in a DCN, which means that a DCN consists of C channels (called Capsule Channels further down for differentiability purposes) of $[W, H, n]$-dimensional cubes. Thus, a DCN structure may be reduced to a CNN structure by making the capsules 1-dimensional; something which is visualized in figure 15 below.

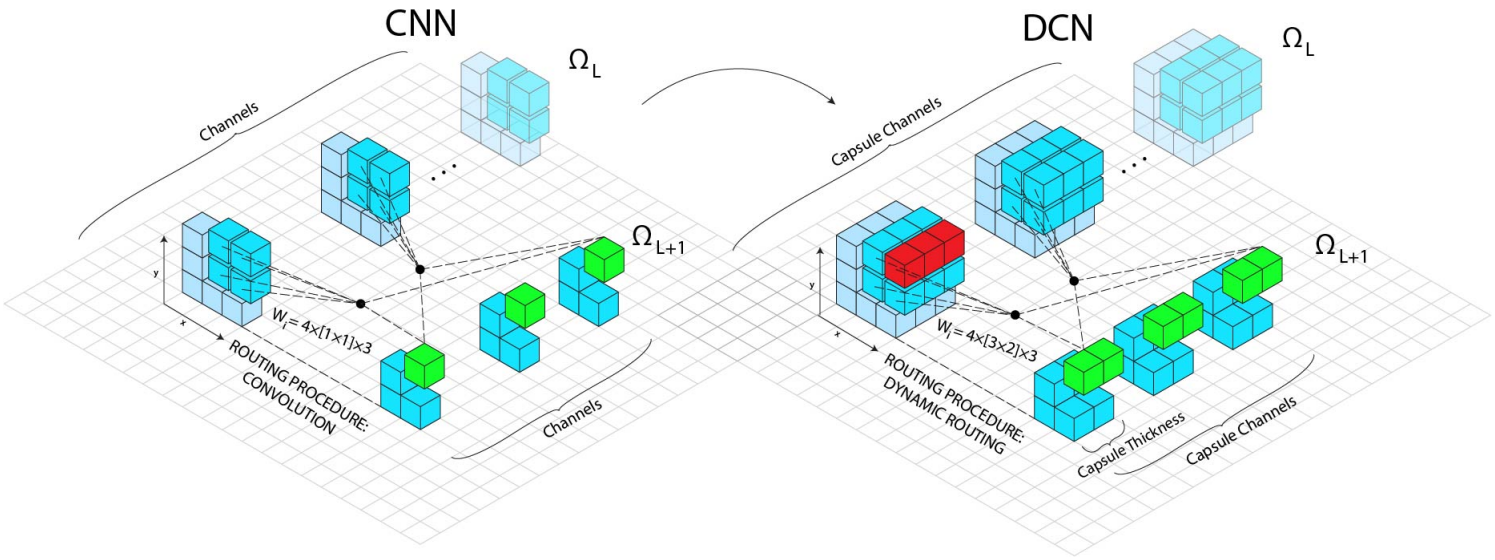


Figure 15: Visualization of how a CNN is, structurally, a special case of a DCN. Convolutional layers are marked in yellow, and capsule layers are marked in blue. As indicated by the figure on the left, a convolutional layer is a capsule layer with thickness 1. An individual capsule in the capsule layers is marked in red. In both of these networks, convolution with kernel $[2, 2]$ and stride $[1, 1]$ is used to route Ω_L to Ω_{L+1}. In the DCN, however, Dynamic Routing is also added to the convolution. In the DCN, the kernel consists of $[3, 2]$ weights to perform $\mathbb{R}^3 \to \mathbb{R}^2$ mapping between capsules in Ω_L to Ω_{L+1}.

Functionality-wise, however, DCNs add the Dynamic Routing procedure on top of the convolutional routing in CNNs. This **routing procedure** is quite complicated and will be dedicated its own section in Section 6.5. Furthermore, a DCN's **activation**- and loss functions are very different from a CNN, due to the fact that multi-dimensional neurons (i.e. capsules) lead to different rules. One such rule is that the capsule length codes for its activation. Further explanation for this will be provided in Section 6.3 and 6.4.

6.1 Network structure

Unfortunately, the structure of a capsule network may actually be a little more complicated than described in the introduction of this chapter. The network structure may actually consist of two parts, described by Figure 16 and 17 below. If this second part of the network is used, the complete

network can be seen as an autoencoder - meaning that the above mentioned network parts are the encoder and the decoder.

6.1.1 Encoder

The first part can be seen as an encoder, and it consists of the capsule layer structure described in the introduction of this chapter. It can be visualised by Figure 16 below, where a capsule network is applied to the MNIST set.

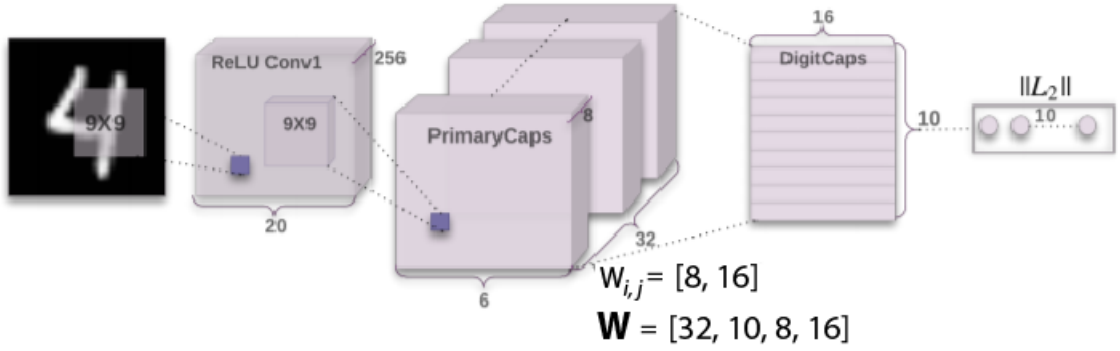


Figure 16: A Capsule network for the dynamic routing algorithm as described by Hinton et al. [2017]. The network starts with a CNN convolution layer with kernel size $[9, 9]$, stride $[1, 1]$, and 256 output channels. This layer is further convolved with a second CNN convolution layer with kernel size $[9, 9]$, stride $[2, 2]$ and 256 output channels. The 256 output channels is then divided into 32 capsule channels, which creates the PrimaryCaps structure with $[6, 6]$-grids of 8D-capsules, set across 32 channels. The PrimaryCaps layer is further routed to the DigitCaps layer according to the routing procedure described in procedure 1. The DigitCaps layer consists of a $[1, 1]$-grid of 16D-capsules set across 10 channels. Lastly, a margin loss is calculated from the DigitCaps layer's output according to (27). The $\mathbf{W}$ tensor above is the collection of all $W_{i,j}$ weights used between PrimaryCaps and DigitCaps. Source: [Hinton et al., 2017].

In detail, this network consists of the following parts:

- An arbitrary number of regular CNN layers applied to an image of size $[x_{in}, y_{in}, C_{in}]$ (C_{in} is 1 or 3 depending on image type). These layers exist to downscale and resize the data from the input image, in order to produce a tractable input for the capsule layers, as these are computationally very demanding. The final output from this part is a $[x_{CNN}, y_{CNN}, C_{CNN}]$-tensor, where x_{CNN} is the number of rows, y_{CNN} is the number of columns, and C_{CNN} is the number of channels from the CNN output.
- A Primary Capsule layer. This layer reshapes the aforementioned $[x_{CNN}, y_{CNN}, C_{CNN}]$-tensor to a capsule network layer, described by the tensor $[x_{CNN}, y_{CNN}, C_{PC}, D_{PC}]$. Here, C_{PC} is the number of capsule channels in the layer (represented by a cube in Figure 16), and D_{PC} is the depth of each capsule.
- An arbitrary amount of capsule layers. These have equivalent structure to the primary capsule layer. Between each capsule layer, the routing procedure described in Section 6.5 below is used.

- A Digit Capsule layer. This is the output layer of the encoder and if the set of classes is denoted $\mathcal{C}$, this layer consists of $|\mathcal{C}|$ number of D_{DC}-dimensional capsules. In tensor notation, this is a $[|\mathcal{C}|, D_{DC}]$-tensor.

In Hinton et al. [2017], this encoder is described by Figure 16 above. The input images are $[28, 28, 1]$-tensors, and the domain has 10 different classes. These are followed by a CNN convolution layer with kernel size $[9, 9]$, stride 1, 256 channels and ReLU activation function. This, in turn is followed by another CNN convolution layer with kernel size $[9, 9]$, stride 2, 256 channels and ReLU activation function. The resulting CNN layer can be described by the tensor $[6, 6, 256]$, which is reshaped to a Primary Capsule layer with 32 capsule channels and a capsule depth of 8; in other terms a $[6, 6, 8, 32]$-tensor. Lastly, this Primary Capsule Layer is multiplied by weights $\mathbf{W}_{ij}$, followed by the Dynamic Routing Procedure (see Section 6.5) to produce the output layer - the Digit Capsule layer - which consists of 10 16-dimensional capsules. The weight multiplication applies a $\mathbb{R}^{t_1} \rightarrow \mathbb{R}^{t_2}$ transformation to the capsules in layer L, where t_1 is the capsule thickness in layer L and t_2 is the capsule thickness in layer *L+1*. Each Capsule channel is associated with only one (!) weight $\mathbf{W}_{ij}$ connecting all its capsules to the next layer. In the case of DigitCaps coding for 10 classes, this means that each capsule channel is associated with 10 weights; one for each DigitCaps capsule. This may be seen as a special type of convolution applied to capsule layers, i.e. a capsule convolution, which will be explained in Section 6.2 below.

6.1.2 Decoder

The optional second part of the network structure is the decoder. In Hinton et al. [2017], this is described by Figure 17, and its purpose is to reconstruct images from the output capsules in Figure 16. When an image has been forward propagated through the aforementioned encoder network, the output capsule corresponding to the correct output label (symbolized by the orange color in Figure 17) is selected and forward propagated through the network in Figure 17. This network is a fully connected, three-layer network, where the first two layers have the ReLU activation function, and the last layer has the sigmoid activation function. The last layer consists of 784 output neurons in order to match the dimension of the input image.

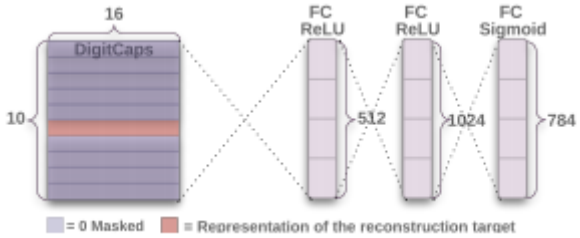


Figure 17: A decoder network which takes as input a single 16-dimensional capsule and outputs a 784-dimensional vector corresponding to a reconstructed image of the image input in Figure 16. This network is a 3-layer fully connected network, where the first layer has 512 neurons with a ReLU activation function, the second layer has 1024 neurons with a ReLU activation function, and the output layer has 784 neurons with a sigmoid activation function. Source: [Hinton et al., 2017].

6.2 Capsule Convolution

As briefly described in the introduction of this section, the capsules in Ω_L are convolved with a kernel $\mathbf{K}$ of weight-matrices $\mathbf{W}$ before being sent as input to the Dynamic Routing Procedure:

$$\mathbf{K} = \begin{bmatrix} \mathbf{W}_1 & \mathbf{W}_2 \\ \mathbf{W}_3 & \mathbf{W}_4 \end{bmatrix} \tag{23}$$

Here, $\mathbf{W}$ are $[m, n]$-matrices performing the transformation $\mathbb{R}^m \rightarrow \mathbb{R}^n$ on the capsules $\mathbf{u}_i \in \Omega_L$ ($\mathbf{u}_i \in \mathbb{R}^m$); producing so called "prediction vectors" or "votes" $\hat{\mathbf{u}}_{j|i}$ for the activations $\mathbf{v}_j \in \Omega_{L+1}$ ($\mathbf{v}_j \in \mathbb{R}^n$) as described by (24). These prediction vectors may be seen as votes for the values of $\mathbf{v}_j \in \Omega_{L+1}$.

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i \tag{24}$$

Here, $\mathbf{W}_{ij}$ is the weight in $\mathbf{K}$ that connects capsule $i \in \Omega_L$ to capsule $j \in \Omega_{L+1}$. This type of convolution will further down be called *Capsule Convolution*. It is applied in different ways depending on whether the convolution leads into the output layer or not:

- If the convolution leads to the output layer, the kernel of weights has the same width W and height H as its input layer, and $\mathbf{W}_1 = \mathbf{W}_2 = ... = \mathbf{W}_{W \cdot H}$ in the capsule kernel. In essence, this means that there is only one weight matrix $\mathbf{W}$ per Capsule Channel and output capsule. This procedure is referred to as the *special case of capsule convolution* below.
- Else, the kernel of weights follows regular procedure ($1 \leq K_W \leq W$, $1 \leq K_H \leq H$, where K_W and K_H are the capsule kernel width and height, respectively, and the capsule kernel weights $\mathbf{W}$ may take any value).

6.3 Activation function

In a dynamic routing capsule network, a capsule's length, $||\mathbf{u}|| \in [0, 1]$, is proportional to the probability of the entity represented by the capsule having been observed. The activation function is described by (25). Here, $\mathbf{v}_j$ is a vector representing the activation of a capsule $j \in \Omega_{L+1}$, and $\mathbf{s}_j$ is a weighted mean over the prediction vectors $\hat{\mathbf{u}}_{j|i}$, given by (26). c_{ij} here is called a *coupling coefficient*, and determines how much impact each $\hat{\mathbf{u}}_{j|i}$ has on $\mathbf{s}_j$. More information on how c_{ij} are calculated will be given in Section 7.4 below.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||} \tag{25}$$

$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i} \tag{26}$$

6.4 Loss function

The loss function Hinton et al. [2017] use can be divided into two parts - the Margin loss $\mathcal{J}^{ml}$ and the Reconstruction loss $\mathcal{J}^{rl}$. $\mathcal{J}^{ml}$ optimizes classification in the decoder network, whereas $\mathcal{J}^{rl}$ optimizes image reconstruction for the entire network.

6.4.1 Margin loss

The margin loss is described by (27). Here, $T_k = 1$ iff class k is present, else $T_k = 0$. $\mathbf{v}_k$ is the output capsule activation coding for class k (corresponding to a_k^{out} in previous network architectures), and $||\mathbf{v}_k|| \in [0, 1]$ is the capsule's length. $m^+ = 0.9$ and $m^- = 0.1$ are margin constants, i.e. values that $||\mathbf{v}_k||$ should aim for in case of present or absent class k. In case of a present class, only the first term in (27) contributes to the loss, and this loss increases the further away from $m^+ = 0.9$ that $||\mathbf{v}_k||$ is. In case of an absent class, only the second term in (27) contributes to the loss, and this loss increases the further away from $m^- = 0.1$ that $||\mathbf{v}_k||$ is. This means that $||\mathbf{v}_k||$ encodes for the probability of class k having been observed. Lastly, $\lambda = 0.5$ and is used to stop loss for absent classes from shrinking the lengths of all output capsules during initial learning.

$$\mathcal{J}_k^{ml} = T_k \max(0, m^+ - ||\mathbf{v_k}||)^2 + \lambda(1 - T_k)\max(0, ||\mathbf{v_k}|| - m^-)^2 \tag{27}$$

6.4.2 Reconstruction loss

The reconstruction loss is described by (28). Here, im_{in} is the image input to Figure 16, im_{out} is the output from the decoder in Figure 17 (corresponding to a^{out} in a classic autoencoder), I is the set of pixels in the images, and $|I|$ is the number of pixels in the images. This loss is derived by calculating the mean of square differences between the reconstructed image and the original image.

$$\mathcal{J}^{rl} = \frac{1}{|I|} \sum\nolimits_{i \in I} (im_{in}(i) - im_{out}(i))^2 \tag{28}$$

6.4.3 Total loss

Combining the margin loss and the reconstruction loss yields the total loss described by (29). Here, $\mathcal{J}_k^{ml}$ is the margin loss for class k, $\mathcal{J}^{rl}$ is the reconstruction loss, and γ is a parameter used to determine the proportions with which the losses should be reflected in the total loss. In Hinton et al. [2017], $\gamma = 0.392$.

$$\mathcal{J} = \gamma \cdot \mathcal{J}^{rl} + \sum_k \mathcal{J}_k^{ml} \tag{29}$$

6.5 Dynamic Routing procedure

When the prediction vectors $\hat{\mathbf{u}}_{j|i}$ have been calculated, they are sent through the Dynamic Routing procedure in order to calculate the activations $\mathbf{v}_j \in \Omega_{L+1}$. $\hat{\mathbf{u}}_{j|i}$ may be seen as votes, whereas the Dynamic Routing procedure is an iterative election that determines the values of $\mathbf{v}_j$. The Dynamic Routing procedure is described in Algorithm 1 below.

Before the loop starts, an unnormalized coupling coefficient b_{ij} is set to 0 in step 2, in order to create equal impact for all votes $\hat{\mathbf{u}}_{j|i}$.

Going into the loop, b_{ij} is normalized through the softmax function (see (30) below) in step 4, which results in the normalized coupling coefficients c_{ij}. c_{ij} may be seen as a probability distribution over i, which means that $\mathbf{s}_j$ is a weighted average of all $\hat{\mathbf{u}}_{j|i}$ voting for capsule $j \in \Omega_{L+1}$. In step 6, this weighted mean is sent through the activation function (25) in order to produce the first consensus $\mathbf{v}_j^{(1)}$. This consensus is then compared against the votes $\hat{\mathbf{u}}_{j|i}$ through the dot product $\hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$, in order to determine how much the vote "agrees" with the consensus. In step 7, this agreemment value is used to update the unnormalized coupling coefficients b_{ij}. Looping this process, this means that

Algorithm 1 Dynamic Routing. The input is $\hat{\mathbf{u}}_{j|i}$, calculated according to (26), the number of iterations r, as well as the layer index l that symbolises the input capsule layer to the routing procedure. The output of the routing procedure is the set of capsule activations $\mathbf{v}_j$ belonging to capsule layer *l+1*.

procedure ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
  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: $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$ ▷ softmax computes (30)
    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 $(l+1)$: $\mathbf{v}_j \leftarrow \text{squash}(\mathbf{s}_j)$ ▷ squash computes (25)
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$
  return $\mathbf{v_j}$

the "election" is a two-step process where consensus $\mathbf{v}_j^{(1)}$ is first created from the votes, and votes who highly agree on this consensus gain high influence on the next consensus $\mathbf{v}_j^{(2)}$; repeating until r iterations have run - outputting the final activation value $\mathbf{v}_j$.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})} \tag{30}$$

The above procedure is illustrated by Figure 18 below.

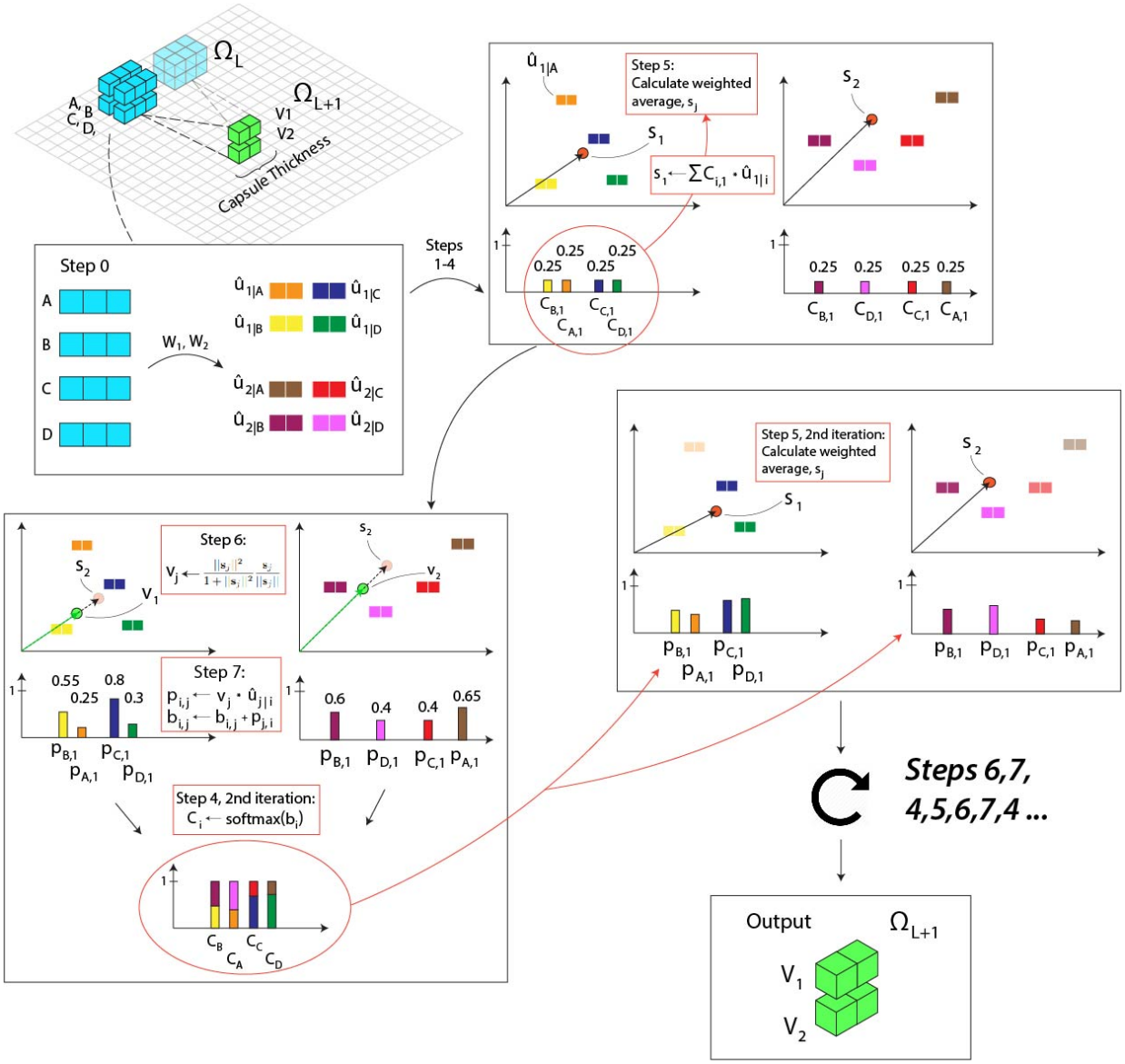


Figure 18: The Dynamic Routing procedure visualized. The capsule activations in layer Ω_L are first transformed by weight matrices in step 0. Thereafter, the procedure is initialized by steps 1-4 in Algorithm 1 above. After this, the routing procedure continues its loop of steps 4-7 for n iterations ($n = 3$, as this produces the best results [Hinton et al., 2017], until it finally exits the loop to produce the capsule activations $\mathbf{v}_j$ for layer Ω_{L+1}. Conceptually, this procedure may be seen as an election system, in which votes $\hat{\mathbf{u}}_{j|i}$ from layer Ω_L come up with a consensus for the activations $\mathbf{v}_j$ of the capsules in layer Ω_{L+1}.

7 Capsule networks - EM Routing

An EM Capsule Network (EMCN) expands upon the concepts of the DCNs. In dynamic routing, each capsule $\mathbf{u}$ was a vector of arbitrary dimension. This vector was multiplied by a weight matrix $\mathbf{W}_{ij}$ (see (26)) to produce a vote $\hat{\mathbf{u}}_{j|i}$ for the dynamic routing algorithm.

In EM Routing, each capsule is an (M, a)-pair, where M is a $[4, 4]$ pose matrix and a is a scalar activation value. The pose M is, in a similar fashion to dynamic routing, multiplied by a $[4, 4]$ weight matrix $\mathbf{W}_{ij}$ to produce a $[4, 4]$ vote matrix $\mathbf{V}_{ij}$, that - together with the activation a - is sent to the EM Routing algorithm (shown in Algorithm 2 in Section 7.4 below). The intuition behind using $[4, 4]$-matrices in the routing procedure comes from computer graphics, and will be explained in detail in Section 7.5 below.

7.1 Network structure

The structure of the EM Routing network is described by Figure 19 below. The structure starts off with a $[5, 5]$ convolutional layer (called ReLU Conv1) with 32 channels ($A = 32$), stride $[2, 2]$ and a ReLU activation function. ReLu Conv1 is thereafter convolved with 2 $[1, 1]$ convolutional layers - one with $16 \times B$ ($B = 32$) output channels and no activation function and one with B output channels and a sigmoid activation function to create the (pose, activation)-pairs (with size $([4, 4], 1)$) that make up the PrimaryCaps layer.

When the PrimaryCaps layer output has been created, it is convolutionally routed through EM Routing to the next capsule layer, called ConvCaps1. This means, similarly to a DCN convolution, that a kernel of size $[K_1, K_2, A]$ is convolved with the PrimaryCaps layer. Each element in the EM kernel is a $[4, 4]$-dimensional weight matrix $\mathbf{W}$ - in contrast to the DCN kernel where each element was an $[m, n]$-matrix. Each $[4, 4]$-matrix is multiplied with its input capsule's pose matrix according to (34) to create a vote V as input to Algorithm 2.

Having established this logic, the PrimaryCaps Layer is followed by two convolutional capsule layers (CCLs) with kernel size $[3, 3]$, 32 channels ($C = D = 32$, and the channels are called capsule types in Hinton et al. [2018]), and strides 2 and 1, respectively. The last CCL, ConvCaps2, is fully connected to the class capsule layer, and the set of capsules in each channel in ConvCaps2 shares the same $[4, 4]$-weights (one $[4, 4]$-weight per output class).

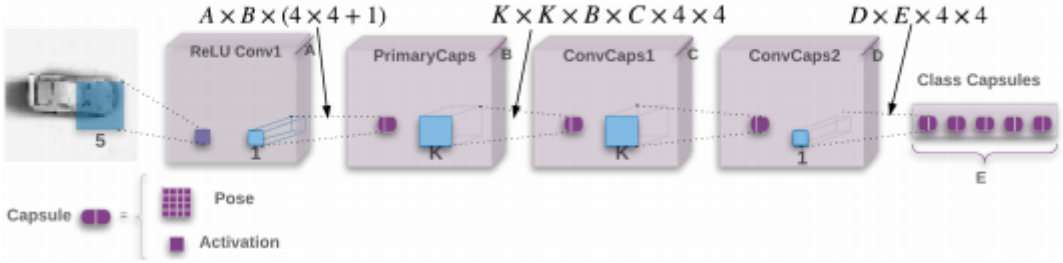


Figure 19: The network structure used for the EM Routing algorithm. The network starts off with a ReLU convolutional layer, followed by a primary capsule layer, two convolutional capsule layers and lastly a class capsule layer. Source: [Hinton et al., 2018].

Between each capsule layer, the EM Routing algorithm is used. This algorithm will be further

explained in Section 7.4, but it has the same intuition behind it as Dynamic Routing - it works as an iterative election system trying to find consensus capsule values (M, a_j) for Ω_{L+1} based on the votes $\mathbf{V}$ and activations a_i from Ω_L. Like Dynamic Routing, this algorithm also uses coupling coefficients to determine vote impact on the consensus, but these are denoted R_{ij} instead of c_{ij}.

7.1.1 Coordinate Addition

As a final step in the network structure, Hinton et al. [2018] adds an operation called Coordinate Addition between ConvCaps2 and Class Capsules in Figure 19. This means that for each capsule i in ConvCaps2, the 2D-coordinates of the center of its receptive field (with respect to the original image coordinates) is scaled and added to the first two elements of the capsule's rightmost column. The reason for this is to keep information about the capsule's location in order to imitate the computer graphics mathematical models described in Section 7.5; in particular the **b**-vector in (37) and 38.

7.2 Activation function

In dynamic routing, the activation function for each capsule was essentially determined by the capsule's length. In EM Routing, however, the activation function for a capsule $j \in \Omega_{L+1}$ is determined by:

1. How much its incoming votes agree with each other. The agreement is measured as a weighted standard deviation between the votes, and is denoted σ_j^h, where j is a capsule in Ω_{L+1} and h is an entry in the $[4, 4]$ vote matrix.

2. How much impact the incoming votes has in the election. The impact is determined by the coupling coefficient R_{ij}, where i indexes the incoming vote $\mathbf{V}$ from Ω_L and j indexes the capsule in Ω_{L+1} for which consensus should be found.

If the incoming votes are highly clustered and have low impact on the election, the activation will be high, whereas if the incoming votes are loosely distributed and have high influence. Conceptually, this means that highly agreeing and quiet voters are more trustworthy than highly disagreeing and loud voters. The activation function is explained by (31) and (32). Here β_u and β_a are learnable biases, and λ is an inverse temperature parameter which is used to adjust the entropy of the distribution of votes; making it easier to compare extreme votes against the rest of the distribution. During the EM Routing procedure, λ is increased iteratively. More information about this activation function may be found in Hinton et al. [2018].

$$a_j = logistic\left(\lambda\left(\beta_a - \sum_h cost_j^h\right)\right) \tag{31}$$

$$cost_j^h = \left(\beta_u + \ln(\sigma_j^h)\right)\sum_i R_{ij} \tag{32}$$

7.3 Loss function

The loss function Hinton et al. [2018] uses is called spread loss, and is described by (33). Here, a_i is the activation from an (activation, pose)-pair of the output capsule layer (in the case of 10 classes, $i \in [1, 2, ..., 10]$). a_t is the activation from the target output capsule (i.e. the output capsule corresponding to the correct label for the input image), and m is a constant, called margin, which

increases from 0.2 to 0.9 during training. If the activation of a wrong class, a_i, is closer to the margin m than a_t, it is penalized by the squared distance to the margin.

$$\mathcal{J}_i = (\max(0, m - (a_t - a_i)))^2, \quad \mathcal{J} = \sum_{i \neq t} \mathcal{J}_i \tag{33}$$

7.4 Routing procedure

7.4.1 Introduction

In Dynamic Routing, each capsule sent a vote $\hat{\mathbf{u}}_{j|i}$ to the routing procedure which determined the coupling coefficients c_{ij} between capsules $i \in \Omega_L$ and $j \in \Omega_{L+1}$. These coupling coefficients determined each vote's impact and forward propagated the vote signals accordingly, in order to determine the activation for capsule j.

EM Routing works in a similar way to dynamic routing, but is based on the EM algorithm; hence the name EM Routing. First, each pose matrix M_i in layer L is multiplied by a weight matrix W_{ij} in order to produce a vote matrix V_{ij} according to (34).

$$V_{ij} = M_i W_{ij} \tag{34}$$

Secondly, the votes V_{ij} are sent through the routing procedure described in Algorithm 2, where the coupling coefficients R_{ij} (analogous to c_{ij} in Dynamic Routing) are calculated. These, in turn, determine the strength of the vote signal that should be sent from capsule $i \in \Omega_L$ to capsule $j \in \Omega_{L+1}$ - or, conceptually, the vote's impact on the election.

R_{ij} is normalized along the capsule channel dimension of Ω_{L+1}. This means that if all capsules $j \in \Omega_{L+1}$ located at (x, y) are denoted $\Omega_{L+1}^{(x,y)}$, then:

$$\sum_{j \in \Omega_{L+1}^{(x,y)}} R_{ij} = 1$$

The output of the election is a pose matrix M_j and an activation a_j belonging to a capsule $j \in \Omega_{L+1}$.

7.4.2 Detailed algorithm description

The EM Routing algorithm is described in detail by Algorithm 2.

Firstly, each vote matrix V_{ij} is vectorized, such that it is now a 16-dimensional vector indexed by the coordinate h ($V_{ij} = V_{ij}^{(h_1,\ldots,h_{16})} \in \mathbb{R}^{16}$). The pose matrix M returned by the EM Routing procedure is μ_j^h in step 9 of Algorithm 2, re-matricized to a $[4, 4]$ matrix. ϵ is a small constant that is added for numerical stability, $\Omega_{L+1}^{(x,y)}$ is the set of capsules on an (x, y)-point along the channel dimension, $|\Omega_{L+1}^{(x,y)}|$ the number of channels in Ω_{L+1}, and $\Omega_{L+1}^{(j_x,j_y)}$ is the set of capsules that share the same (x, y)-grid location as capsule $j \in \Omega_{L+1}$; (j_x, j_y).

The algorithm works by first setting $R_{ij} \leftarrow 1/|\Omega_{L+1}^{(x,y)}|$, which means that each vote initially gets equal impact on the election. Thereafter, the algorithm enters a loop where the E-step and M-step follow each other for t iterations. When this loop has finished, the activations a_j and the poses M_j of the capsules in Ω_{L+1} have been calculated and are returned.

The purpose of the M-step is firstly to find the gaussian distribution parameters μ_j^h and σ_j^h that best explain the votes, secondly to find the activations a_j of capsules $j \in \Omega_{L+1}$. In step 7, R_{ij} is first weighted against $a_i \in \Omega_L$ in order to let the activations in Ω_L have impact on the election. Step 8

Algorithm 2 EM Routing. Returns the **activation** and **pose** of the capsules in layer *L+1* given the **activations** and **votes** of capsules in layer L. V_{ij}^h is the h^{th} dimension of the vote from capsule i with activation a_i in layer L to capsule j in layer *L+1*. β_a, β_u are learned discriminatively and the inverse temperature λ increases at each iteration with a fixed schedule. This algorithm is a modification of the algorithm shown by Hinton et al. [2018], to make the algorithm numerically stable.

1: **procedure** EM ROUTING($\mathbf{a}, V$)
2: $R_{ij} \leftarrow 1/|\Omega_{L+1}^{(x,y)}|$ ▷ [Hui, 2017]
3: **for** t iterations **do**
4: $\forall j \in \Omega_{L+1}$: M-step($\mathbf{a}, R, V, j$)
5: $\forall i \in \Omega_L$: E-step($\mu, \sigma, \mathbf{a}, V, i$)
return $\mathbf{a}, M$
6: **procedure** M-STEP($\mathbf{a}, R, V, j$)
7: $\forall i \in \Omega_L : R_{ij} \leftarrow R_{ij} \cdot \mathbf{a}_i$
8: $\forall h : \mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij}^h}{\sum_i (R_{ij}) + \epsilon}$
9: $\forall h : (\sigma_j^h)^2 \leftarrow \frac{\sum_i R_{ij} (V_{ij}^h - \mu_j^h)^2}{\sum_i (R_{ij}) + \epsilon} + \epsilon$
10: $\text{cost}_j^h \leftarrow \left(\beta_u + \ln(\sigma_j^h)\right) \sum_i R_{ij}$
11: $a_j \leftarrow \text{logistic}(\lambda \cdot \text{Batch\_Norm}(\sum_h \text{cost}_j^h)))$
12: **procedure** E-STEP($\mathbf{a}, R, V, i$)
13: $\forall j \in \Omega_{L+1} : \ln(\mathbf{p}_j) \leftarrow \sum_h^H \left(-\frac{1}{2} \ln(2\pi(\sigma_j^h)^2) - \frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2} \right)$
14: $\forall j \in \Omega_{L+1} : \mathbf{R}_{ij} \leftarrow \frac{\mathbf{a}_j \exp(\ln(\mathbf{p}_j))}{\sum_{k \in \Omega_{L+1}^{(j_x, j_y)}} \mathbf{a}_k \exp(\ln(\mathbf{p}_k))}$ ▷ [Hui, 2017]
15: **procedure** BATCH_NORM(cost_j)
16: $\mu_{\text{cost}\,j} \leftarrow \frac{\text{cost}_j}{\sum_{j' \in \Omega_{L+1}^{(j_x, j_y)}} \text{cost}_{j'}}$
17: $(\sigma_{\text{cost}\,j})^2 \leftarrow \frac{\sum_{j' \in \Omega_{L+1}^{(j_x, j_y)}} (\text{cost}_{j'} - \mu_{\text{cost}\,j'})^2}{|\Omega_{L+1}^{(x,y)}|}$
18: **return** $\beta_a - \gamma \cdot \frac{\mu_{\text{cost}\,j} - \text{cost}_j}{\sigma_{\text{cost}\,j} + \epsilon}$

and 9 calculate the consensus parameters μ_j^h and σ_j^h weighted against the coupling coefficients. Step 10 and 11 then calculate the activation a_j for capsule $j \in \Omega_{L+1}$, as discussed in Section 7.2 above. A batch normalization on the $\sum_h cost_j^h$ parameter is added to the calculation of a_j for numerical stability. The output of the M-step is now the gaussian distribution parameters μ_j^h and σ_j^h, as well as the activations a_j of capsules $j \in \Omega_{L+1}$.

The purpose of the E-step is to find the coupling coefficients R_{ij} that deals impact proportional to how much a vote agrees with the gaussian distribution $G(\mu_j^h, \sigma_j^h)$. In step 13, each vote V_{ij}^h is measured against $G(\mu_j^h, \sigma_j^h)$ in order to determine how much the vote agrees with the consensus. These measurements are stored in the $\mathbf{p}_j$ variable, which may be seen as an unnormalized probability distribution for the votes $\mathbf{V_{ij}}$ correctly predicting the pose M_j of capsule $j \in \Omega_{L+1}^{(j_x, j_y)}$. In step 14, R_{ij} is calculated by weighting $\mathbf{p}_j$ is against the activations a_j and normalizing along the Capsule Channel dimension in Ω_{L+1}. Thus, the E-step outputs R_{ij}, which determines how the vote impact should be distributed along the Capsule Channel dimension of Ω_{L+1}.

The calculations used in Algorithm 2 above use the logarithm of (35) to calculate $\ln(\mathbf{p}_j)$, as using this is numerically more stable.

$$\mathbf{p}_j = \prod_{h=1}^{16} \frac{1}{\sqrt{2\pi(\sigma_j^h)^2}} \exp\left(-\frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) = \frac{1}{\sqrt{\prod_{h=1}^{16} 2\pi(\sigma_j^h)^2}} \exp\left(-\sum_{h=1}^{16} \frac{(V_{ij}^h - \mu_j^h)^2}{2(\sigma_j^h)^2}\right) \tag{35}$$

A visualization of the algorithm above is provided in Figure 20 on next page.

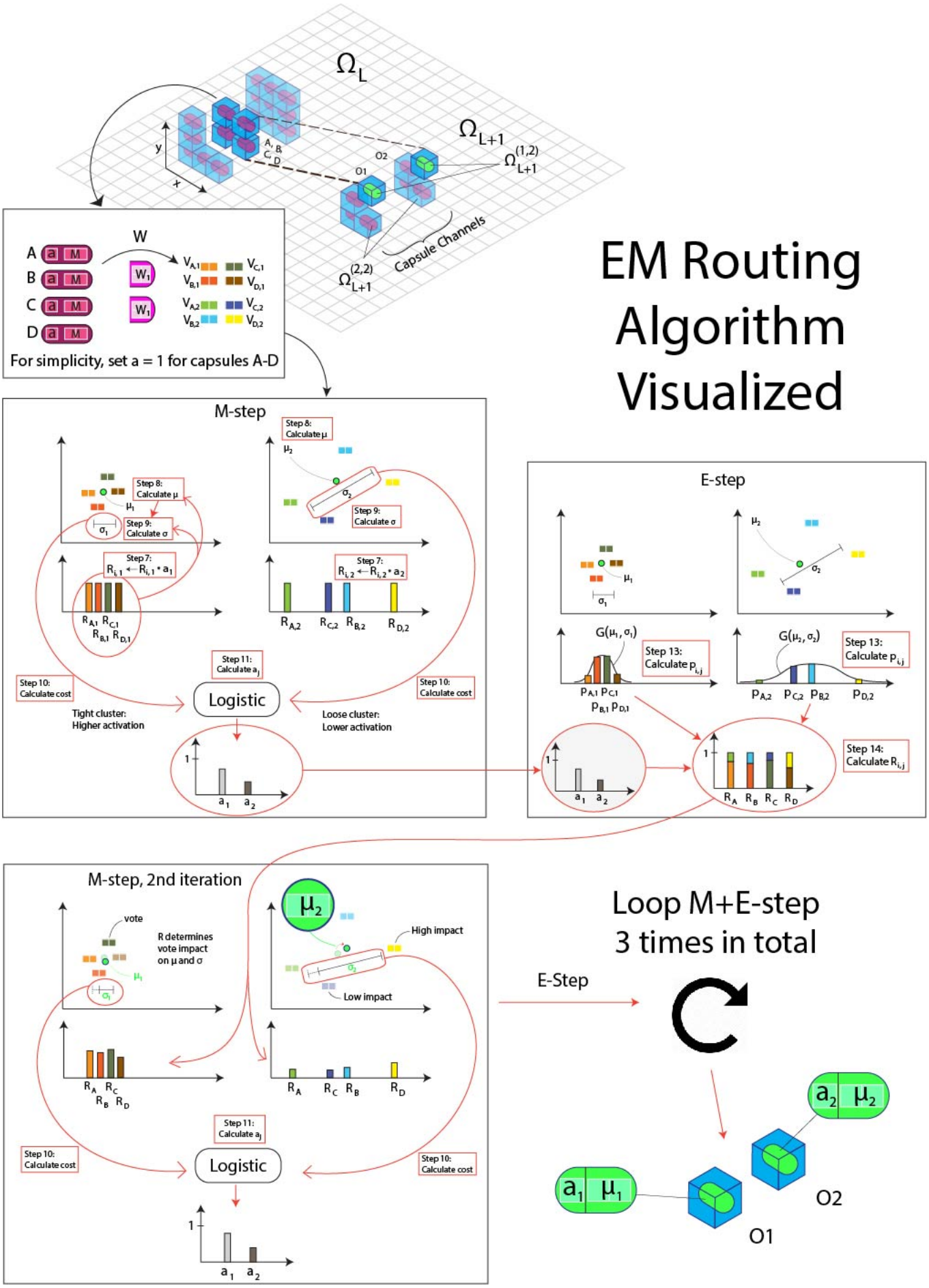


Figure 20: EM Routing Algorithm visualized. The figure focuses on four capsules routing to two capsules O_1 and O_2, which means that there are four votes V per O-capsule. In order to simplify the visualization, the pose- and vote matrices M and V are $\mathbb{R}^2$-vectors. For conceptual simplicity, $a_A = a_B = a_C = a_D = 1$ for capsules A-D. In the M-step, each vote $V_{i,j}$'s color contrast represents its $R_{i,j}$ value; i.e. its impact on μ and σ. The output capsules, O_1 and O_2, consist of the calculated means μ_1 and μ_2, and the calculated activations a_1 and a_2 in the M-step. The calculated means μ_1 and μ_2 are represented as dots in the M- and E-steps in order to save space.

7.5 Computer graphics and intuition behind the EM Routing Capsule Networks

A linear transformation maps vectors from one vector space to another, and can be described by (36), where $\mathbf{A}$ is an $[m, n]$-matrix, $\mathbf{b}$ is an m-vector, and $\mathbf{x}$ is an n-vector representing point coordinates to be transformed. The $\mathbf{b}$-vector translates the point $\mathbf{x}$, whereas the $\mathbf{A}$-matrix scales and rotates it.

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b} \tag{36}$$

Computer graphics perform nearly equivalent tranformations between affine spaces; so called affine transformations that can also be represented by (36). In computer graphics, however, these transformations are often represented in block matrix form as described by (37). Here, $\mathbf{0}^T$ is an n-vector, 1 is a scalar, and the rest of the parameters' dimensionalities are retained [Van Verth and Bishop, 2016, p. 116].

$$\mathbf{y} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} = \mathbf{M}\mathbf{v} \tag{37}$$

Affine transformations preserve two properties of geometry: collinearity and relative proportions. Thus, points on a line will remain collinear and ratios of distances will remain constant.

As a complement to affine transformations, computer graphics also involve projective transformations, which are of the form shown in (38). Here, $\mathbf{c}$ is an n-vector and d a scalar, making $\mathbf{c}\mathbf{x} + d$ a scalar value. Because matrix multiplication with affine coordinates cannot capture denominators, the projective transformation is represented by the block matrix equation shown in (39). This equation is performed in the homogenous coordinates $[\mathbf{x}, w]$, where the denominator in (38) is stored in w [Goldman, 2009, p. 190-194].

$$\mathbf{y} = \frac{\mathbf{A}\mathbf{x} + \mathbf{b}}{\mathbf{c}\mathbf{x} + d} \tag{38}$$

$$\mathbf{y} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{c} & d \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ w \end{bmatrix} = \mathbf{M}\mathbf{v} \tag{39}$$

As can be seen in (37) and 38, the component block matrices $\mathbf{M}$ and $\mathbf{v}$ have the same dimensionalities in either transformation. If the points $\mathbf{x}$ are 3-dimensional and $\mathbf{A} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, $\mathbf{M}$ is a $[4, 4]$-dimensional matrix, and $\mathbf{v}$ is a 4-dimensional vector. Multiple $\mathbf{M}$-matrices can be multiplied together to create a single $\mathbf{M}$-matrix symbolizing multiple affine and projective transformations. With this functionality in mind, the $[4, 4]$ W_{ij} matrices in EM Routing can be seen as learned affine and projective transformations applied to objects in the image; objects represented by neuron outputs in the hidden layers of the grid. In EM Routing, however, $\mathbf{v}$ is a $[4, 4]$ pose matrix, representing the entity's pose.

Moving on with this line of reasoning, a capsule's pose matrix $M \in \Omega_L$ is thus transformed by a matrix W_{ij}, and the result is used in the routing procedure to determine another capsule's pose matrix M in the consecutive layer Ω_{L+1}. $M \in \Omega_L$ are transformed through W_{ij} to cast votes V_{ij} for the values of $M \in \Omega_{L+1}$. The routing procedure finds clusters of high agreement and proportionally propagate their signals in calculation of $M \in \Omega_{L+1}$.

As a side note, the pose matrices used in Hinton et al. [2018] assume that the domain in which they are applied is 3-dimensional, due to the fact that they are $[4, 4]$-matrices. If, however, the domain is 2-dimensional, $[3, 3]$-matrices may be used, as transformations need only be applied in 2 dimensions.

8 Methodology

8.1 Datasets used

Six different train sets were used together with one validation- and one test set in order to determine classification performance on varying number of classes and variable set size. The datasets used are displayed in table 1. Here, $\mathcal{D}_{hard}$, $\mathcal{D}_{medium}$ and $\mathcal{D}_{basic}$ is the same dataset, $\mathcal{D}_{hard}$, divided into three different subsets with a different number of classes; with each added set of classes adding complexity to the classification. The following three datasets, $\mathcal{D}_{minisize}$, $\mathcal{D}_{smallsize}$ and $\mathcal{D}_{mediumsize}$, are subsets of $\mathcal{D}_{hard}$, where the data amount in each class is small; testing how well the algorithm performs with varying amounts of data. The last two datasets, $\mathcal{D}_{val}$ and $\mathcal{D}_{test}$, are the validation- and test sets, respectively. Examples of images used in the datasets have already been shown in Figure 2 in Section 2.4 above.

Cell types	$\mathcal{D}_{hard}$	$\mathcal{D}_{medium}$	$\mathcal{D}_{basic}$	$\mathcal{D}_{minisize}$	$\mathcal{D}_{smallsize}$	$\mathcal{D}_{mediumsize}$	$\mathcal{D}_{val}$	$\mathcal{D}_{test}$
Segmented Neutrophil	2113	2113	2113	100	300	900	235	669
Eosinophil	2112	2112	2112	100	300	900	235	687
Basophil	2101	2101	2101	100	300	900	233	857
Lymphocyte	2119	2119	2119	100	300	900	235	718
Monocyte	2110	2110	2110	100	300	900	235	737
Erythroblast	2098	2098	-	100	300	900	233	437
Giant Thrombocyte	901	901	-	100	300	900	97	175
Blast	2093	2093	-	100	300	900	233	855
Smudge Cell	2047	2047	-	100	300	900	229	743
Artefact	1752	1752	-	100	300	900	205	697
Thrombocyte Aggregation	1092	-	-	100	300	900	124	432
Band Neutrophil	2106	-	-	100	300	900	234	763
Myelocyte	2099	-	-	100	300	900	234	822
Metamyelocyte	1885	-	-	100	300	900	209	452
Plasma Cell	363	-	-	100	300	363	39	260

Table 1: Table displaying the set sizes used for testing image classification performance. All datasets $\mathcal{D}_{hard}$-$\mathcal{D}_{mediumsize}$ are subsets of $\mathcal{D}_{hard}$. $\mathcal{D}_{val}$ is a validation set, and $\mathcal{D}_{test}$ is a test set; both of which are used in conjunction with all datasets $\mathcal{D}_{hard}$-$\mathcal{D}_{mediumsize}$.

8.2 Network Structures used

All tested networks consist of two parts. The first part is a downsampling CNN, where a $[224, 224, 3]$ input image is downsampled to a $[20, 20, 128]$-tensor through a CNN. The second part is a network with one to two layers of the specific network algorithm (i.e. CNN, DCN or EMCN) that is tested. This enables good comparability between the different algorithms structurally but not with regards to the number of parameters used in the different networks.

The downsampling network structures tested are shown in Figure 21 and 22. Figure 21 shows a downsampling network structure with max pooling layers, and Figure 22 shows a downsampling network structure without max pooling layers. The second part of the network is different between the different algorithms and will be shown in the following sections.

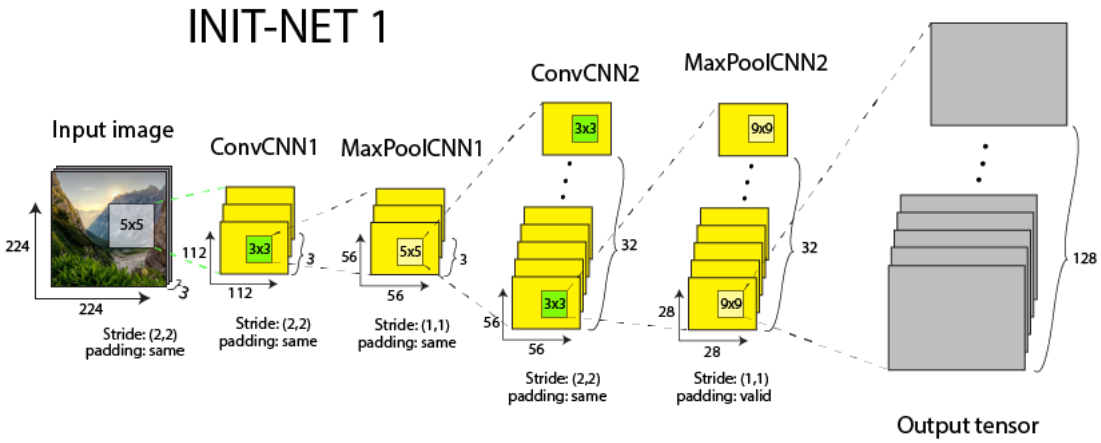


Figure 21: Initial network structure with max pooling layers. The input image is a $[224, 224, 3]$-tensor, convolutional layers are yellow, max pooling layers are green, and the output is a $[20, 20, 128]$-tensor marked in grey. The total number of parameters in this network is $\sim 334\ 000$.

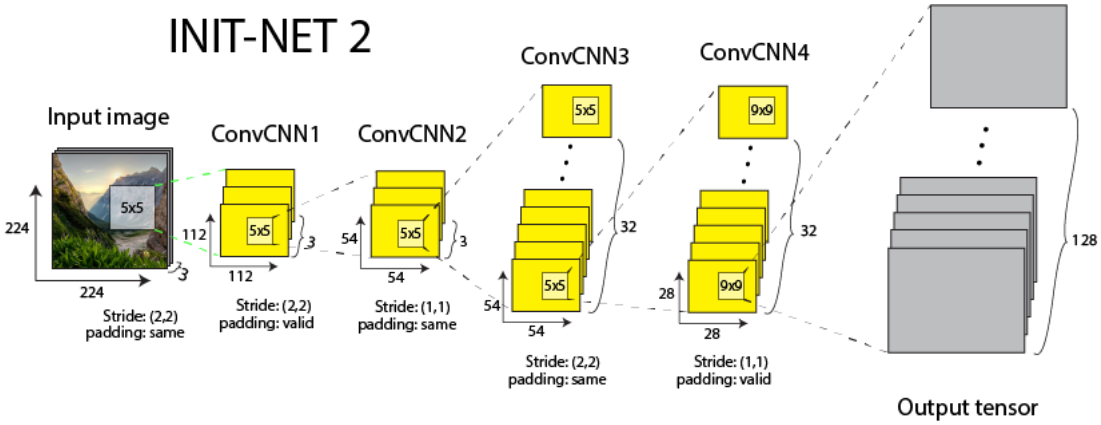


Figure 22: Initial network structure without max pooling layers. The input image is a $[224, 224, 3]$-tensor, convolutional layers are marked in yellow, and the output is a $[20, 20, 128]$-tensor marked in grey. The total number of parameters used in this network is $\sim 360\ 000$.

8.2.1 CNN

Three different CNN structures following the initial CNN structure in Figures 21 and 22 were tested against the datasets in Table 1. The three different structures are displayed in Figures 23-25. The CNN parts of Figures 24 and 25 are designed to resemble the tested DCN structures as closely as possible; making the number of parameters in these CNN structures equal to, or greater than, the number of parameters used in the DCNs. The ANN in Figure 23 may be seen as a reference structure, to get a base-line for the performance of the two above mentioned algorithms.

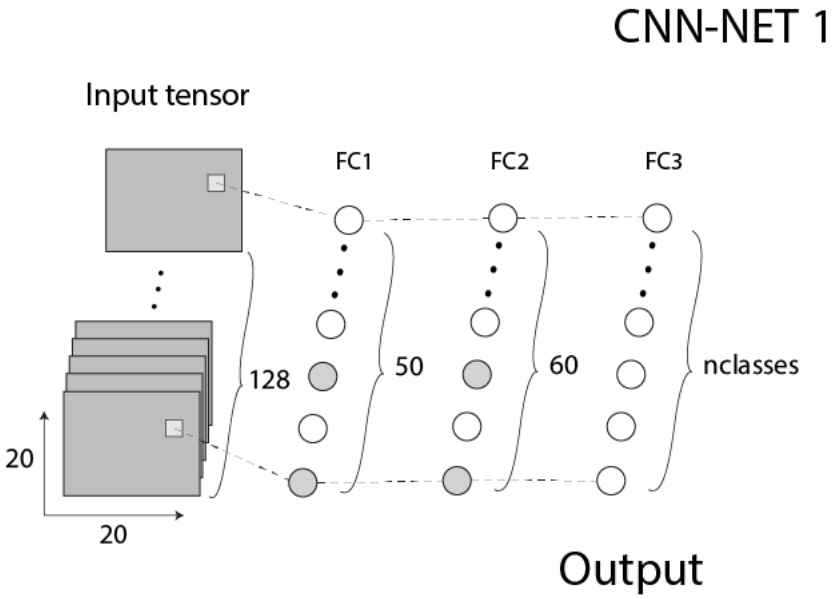


Figure 23: **CNN-NET 1**. A CNN structure used as a reference structure for the performance of the structures in Figures 24 and 25. The input tensor is a $[20, 20, 128]$-tensor marked in grey, and no convolutional layers are added; only fully connected layers, represented by white circles. Fully connected layers with dropout are represented with grey and white circles in succession. The total number of parameters used is $\sim 2\ 560\ 000$.

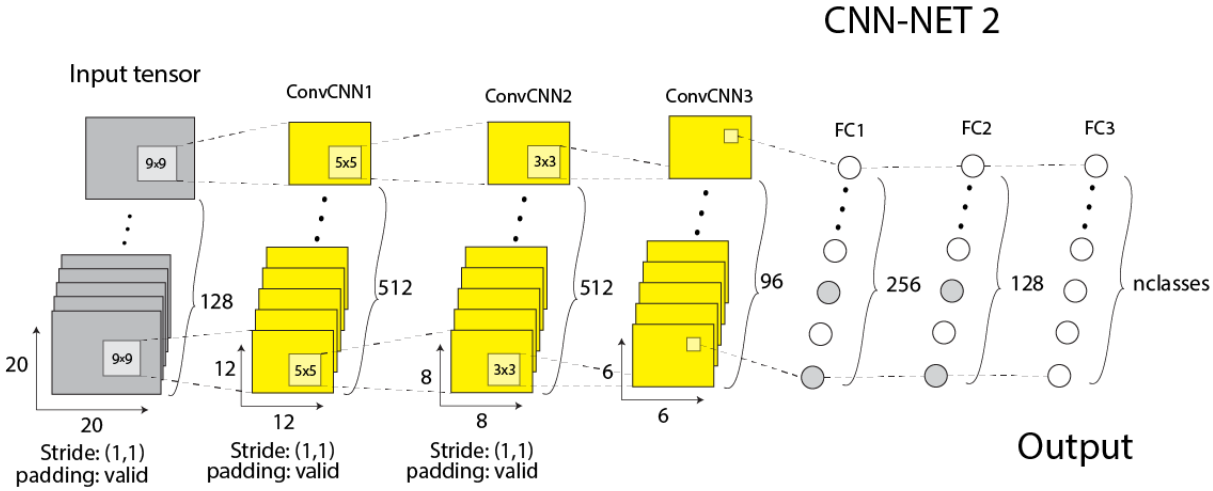


Figure 24: **CNN-NET 2**. A CNN structure with the purpose of closely resembling tested capsule networks with two capsule layers at the end. The input tensor is a $[20, 20, 128]$-tensor marked in grey, convolutional layers are marked in yellow, and fully connected layers with dropout are represented by grey and white circles in succession. The total number of parameters used is $\sim 13\ 220\ 000$.

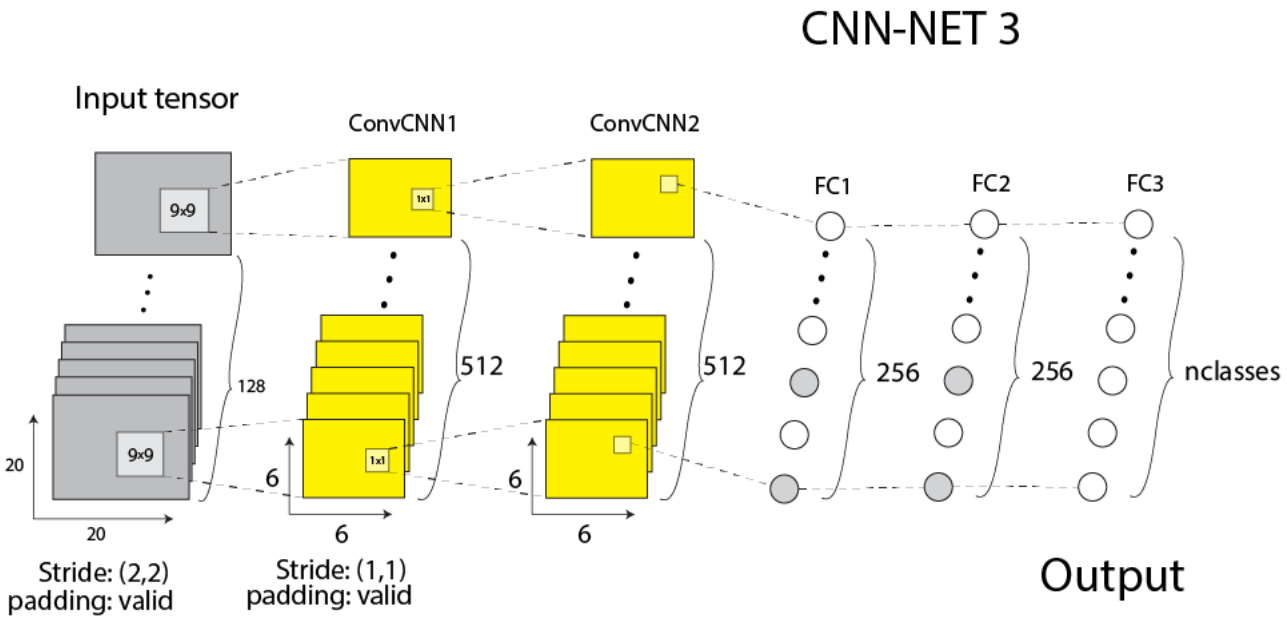


Figure 25: **CNN-NET 3**. A CNN structure with the purpose of closely resembling tested capsule networks with only one capsule layer at the end. The input tensor is a $[20, 20, 128]$-tensor marked in grey, convolutional layers are marked in yellow, and fully connected layers with dropout are represented by grey and white circles in succession. The total number of parameters used is $\sim 10\ 360\ 000$.

8.2.2 DCN

The DCN structures tested are similar to the CNN structures previously mentioned. They contain as many or fewer parameters as the yellow marked layers in the Figures shown in Section 8.2.1 above. They also consist of one CNN Convolution layer (ConvCNN) plus one to two capsule layers (ConvCaps); analogous to the convolutional layers in Figures 23-25. The DCN structures used are shown in Figure 26 and 27 below. Here, the PrimaryCaps structure is simply the output from the ConvCNN layer, represented by capsules to show where the capsule layers begin. Following the PrimaryCaps layer are ConvCaps layers, which follow the same procedures described previously.

The reason for using a capsule thickness of 96 for the ClassCaps capsules is somewhat arbitrary, but based on the capsule thickness used in Hinton et al. [2017], along with memory limitations. Firstly, Hinton et al. [2017] used a ClassCaps capsule thickness of 16 on gray-scaled images, and as the white blood cell images used in this paper are RGB-images, this thickness was multiplied by a factor of 3. Moreover, the white blood cell image dataset is necessarily more complex than the MNIST dataset, which is why this product is multiplied by another factor of 2. Memory limitations made it impossible to increase this factor by a higher integer value. Lastly, the reason for making the ClassCaps capsules thick, is that these serve as the decoder input when the input image is reconstructed; meaning that higher dimensionality here should lead to more details in the reconstructed images.

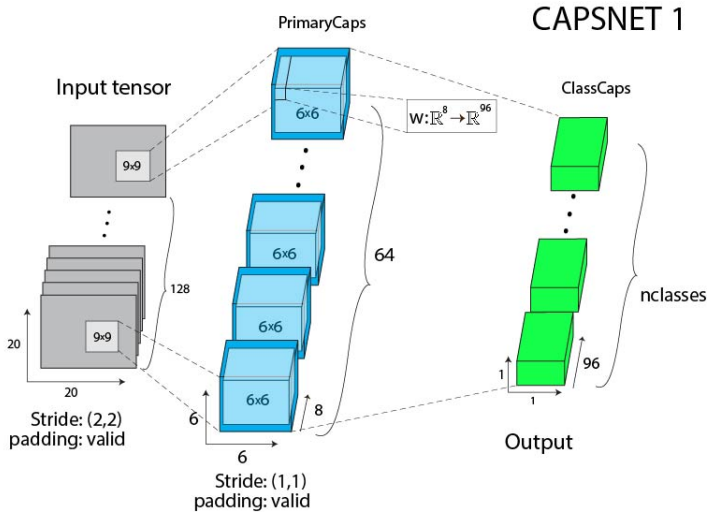


Figure 26: **CAPSNET 1**. A DCN structure which consists of a CNN Convolution - producing the PrimaryCaps layer, which is followed by the ClassCaps output layer. Between the PrimaryCaps and ClassCaps layers, the previously discussed special case of Capsule Convolution together with Dynamic Routing is used. This net uses $\sim 6\ 000\ 000$ parameters.

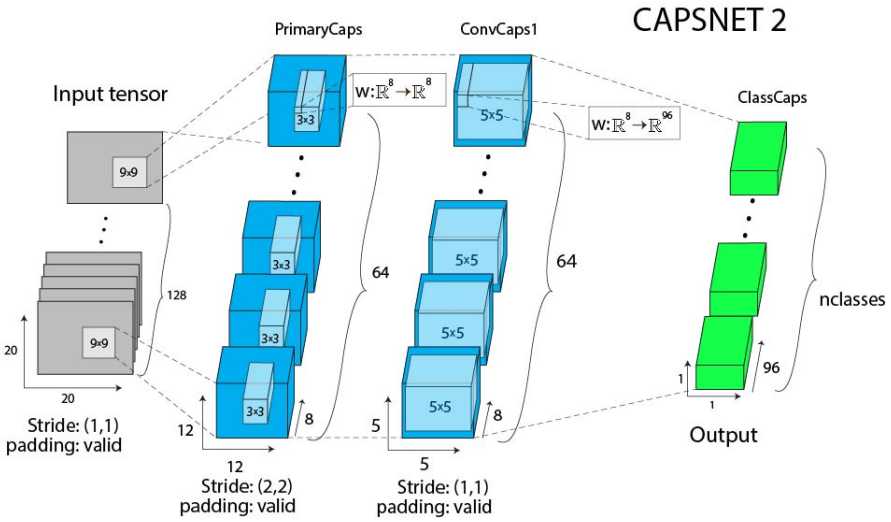


Figure 27: **CAPSNET 2**. A DCN structure with similar structure to Figure 26, but with an added ConvCaps layer between the PrimaryCaps and ClassCaps layer. Between each capsule layer, Dynamic Routing is used. Between PrimaryCaps and ConvCaps1, Capsule Convolution is used. Between ConvCaps1 and ClassCaps, the special case of Capsule Convolution is used. This net uses $\sim 8\ 000\ 000$ parameters.

8.2.3 EMCN

The EM Routing Capsule networks that were tested are shown in Figures 28 and 29 below. Both of these used $[4, 4]$ pose- and weight matrices and tested the performance when adding one and two Convolutional EM Capsule layers, respectively. The loss shown in Equation 33 has its margin m increased with 0.1 every 3000 steps, and starts from 0.2 and stops increasing at 0.9. Between each capsule layer, EM Routing is used, and between the last capsule layer and the ClassCaps layer, the special case of Capsule Convolution is used, together with coordinate addition.

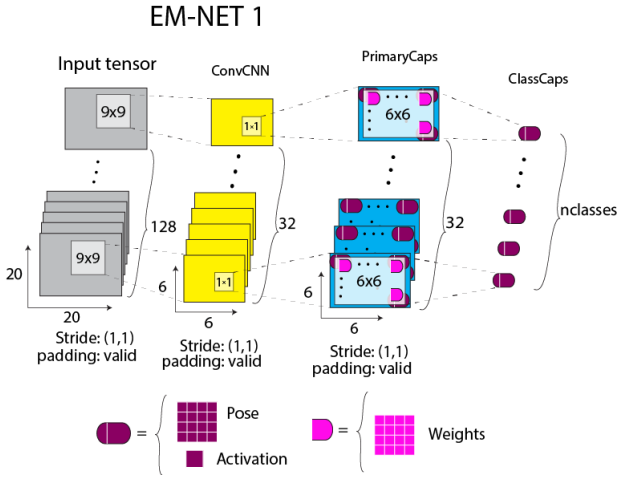


Figure 28: **EM-NET 1**. An EMCN structure with one layer of EM Capsules. The ConvCNN is a CNN Convolution layer, the PrimaryCaps layer is the first EMCN layer, and the ClassCaps layer is the output layer consisting of EM Capsules. The capsules are dark purple, and the weights are pink. In PrimaryCaps, the kernel is a grid of 6×6 weight matrices of dimension $[4, 4]$. The total number of parameters used is $\sim 350\ 000$.

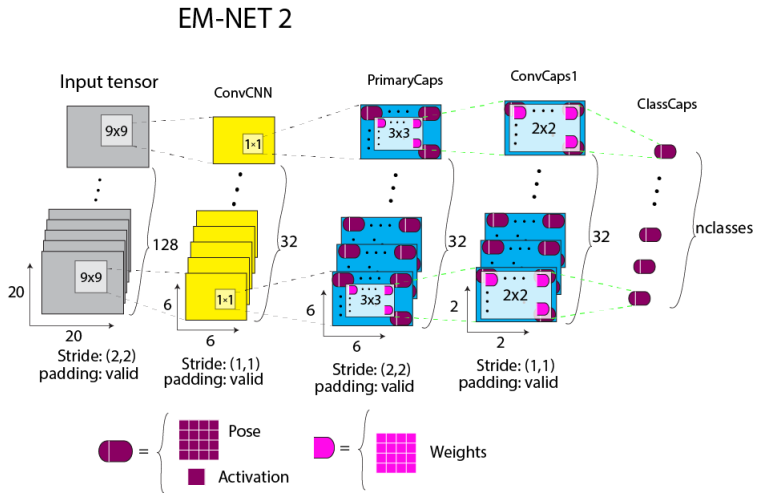


Figure 29: **EM-NET 2**. An EMCN structure with two layers of EM Capsules. The layers are represented in the same ways as in Figure 28, with an addition of the ConvCaps layer, which is a Convolutional Capsule layer. The total number of parameters used is $\sim 500\ 000$.

8.2.4 Decoder net

In both the EMCNs and the DCNs, a decoder network was sometimes also connected. The decoder net, called *DECODER NET*, is shown in Figure 30 below.

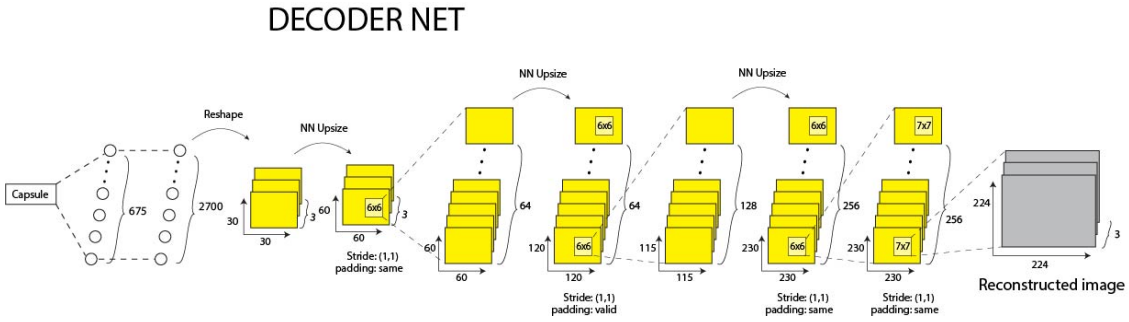


Figure 30: The decoder network that was sometimes applied to the DCNs and the EMCNs. The capsule box at the left represents one capsule at the output layer of the capsule network, and is thus a vector in case of a DCN and a pose matrix in case of an EMCN. *NN Upsize* means *Nearest Neighbour Upsize*.

8.2.5 Number of parameters used

Table 2 below lists the number of parameters used in each network shown in previous sections. It is worth noting that the EMCNs only have $\sim 300,000 - 500,000$ parameters, whereas the DCN and CNN have $\sim 5,000,000 - 10,000,000$ and $\sim 2,500,000 - 13,000,000$ parameters, respectively. The CNN-NETs are chosen to resemble the CAPSNETs and EM-NETs, which make them slightly less representative of typical CNN networks, which may be shallower, but it is worth noting that each added CNN layer adds many parameters to the whole model.

Network	Layer						Total
	1	**2**	**3**	**4**	**5**	**6**	
INIT-NET 1	$5 \cdot 5 \cdot$ $3 \cdot 3$	0	$5 \cdot 5 \cdot$ $3 \cdot 32$	0	$9 \cdot 9 \cdot$ $32 \cdot 128$	-	$\sim 334\ 401$
INIT-NET 2	$5 \cdot 5 \cdot$ $3 \cdot 3$	$5 \cdot 5 \cdot$ $3 \cdot 3$	$5 \cdot 5 \cdot$ $3 \cdot 32$	$5 \cdot 5 \cdot$ $32 \cdot 32$	$9 \cdot 9 \cdot$ $32 \cdot 128$	-	$\sim 360\ 226$
CNN-NET 1	$20 \cdot 20 \cdot$ $128 \cdot 50$	$50 \cdot 60$	$60 \cdot \lvert\mathcal{C}\rvert$	-	-	-	$\sim 2\ 563\ 000$ $+ 60 \cdot \lvert\mathcal{C}\rvert$
CNN-NET 2	$9 \cdot 9 \cdot$ $128 \cdot 512$	$5 \cdot 5 \cdot$ $512 \cdot 512$	$3 \cdot 3 \cdot$ $512 \cdot 96$	$6 \cdot 6 \cdot$ $96 \cdot 256$	$256 \cdot 128$	$128 \cdot \lvert\mathcal{C}\rvert$	$\sim 13\ 221\ 888$ $+ 128 \cdot \lvert\mathcal{C}\rvert$
CNN-NET 3	$9 \cdot 9 \cdot$ $128 \cdot 512$	$1 \cdot 1 \cdot$ $512 \cdot 512$	$6 \cdot 6 \cdot$ $512 \cdot 256$	$256 \cdot 256$	$256 \cdot \lvert\mathcal{C}\rvert$	-	$\sim 10\ 354\ 688$ $+ 256 \cdot \lvert\mathcal{C}\rvert$
CAPSNET 1	$9 \cdot 9 \cdot 128 \cdot$ $(8 \cdot 64)$	$8 \cdot 64 \cdot$ $96 \cdot \lvert\mathcal{C}\rvert$	-	-	-	-	$\sim 5\ 308\ 416$ $+ 49\ 152 \cdot \lvert\mathcal{C}\rvert$
CAPSNET 2	$9 \cdot 9 \cdot 128 \cdot$ $(8 \cdot 64)$	$3 \cdot 3 \cdot$ $8 \cdot 64 \cdot$ $8 \cdot 64$	$8 \cdot 64 \cdot$ $96 \cdot \lvert\mathcal{C}\rvert$	-	-	-	$\sim 7\ 667\ 712+$ $49\ 152 \cdot \lvert\mathcal{C}\rvert$
EM-NET 1	$9 \cdot 9 \cdot$ $128 \cdot 32$	$1 \cdot 1 \cdot 32 \cdot$ $(16 + 1) \cdot 32$	$4 \cdot 4 \cdot$ $32 \cdot \lvert\mathcal{C}\rvert$	-	-	-	$\sim 349\ 184+$ $512 \cdot \lvert\mathcal{C}\rvert$
EM-NET 2	$9 \cdot 9 \cdot$ $128 \cdot 32$	$1 \cdot 1 \cdot 32 \cdot$ $(16 + 1) \cdot 32$	$3 \cdot 3 \cdot 4 \cdot$ $4 \cdot 32 \cdot 32$	$4 \cdot 4 \cdot$ $32 \cdot \lvert\mathcal{C}\rvert$	-	-	$\sim 496\ 640+$ $512 \cdot \lvert\mathcal{C}\rvert$
DECODER NET	$675 \cdot CT$	$675 \cdot 2700$	$6 \cdot 6$ $3 \cdot 64$	$6 \cdot 6$ $64 \cdot 128$	$6 \cdot 6$ $256 \cdot 256$	$7 \cdot 7$ $256 \cdot 3$	$\sim 4\ 521\ 252+$ $675 \cdot CT$

Table 2: Table of the number of parameters used in each of the networks. Here, $|\mathcal{C}|$ is the number of output classes that the net has, and CT is the capsule thickness, which is 96 in case of *CAPSNET 1* and *CAPSNET 2*, and 16 in case of *EMNET 1* and *EMNET 2*. The calculations do not consider biases, which is why the total number of parameters shown is a close approximation. Some layers have 0 parameters, which means that these are maxpool layers.

8.3 Evaluating generalization capabilities

Section 3.6.3 briefly introduced different types of data augmentations used to increase the amount of available data in the train set. In this project, however, the data augmentations will be solely applied to the test set in order to check how well the algorithms generalize to transformations of the data. The different augmentations that will be checked are: *Horizontal- and Vertical flips together with Rotation*, *Translation*, *Color addition* and *Gaussian noise*. This means that there were four test cases; each tested separately by adding its augmentation at random to the test set after the model had been trained.

8.4 Evaluating convergence rate, speed and accuracy

The performance of the investigated algorithms will be evaluated with regards to their convergence rate, speed and accuracy. The accuracy will be measured by calculating the percentage of correct classifications on the test set $\mathcal{D}_8$ above.

The convergence rate will be a qualitative analysis on the plots produced by measuring each algorithm's accuracy on the validation set $\mathcal{D}_7$ during training. The plots will show the models' accuracy on the y-axis, and the number of iterations the model has trained for on the x-axis. Each

iteration here means that the network is trained on a batch of 5 images from its training set. When the model has run through the training set, it is run through the validation set, $\mathcal{D}_{val}$; producing an accuracy value for the validation set at training iteration t.

The speed will be four measures of the forward- and backward pass time for a batch of size 5 for each algorithm; two on a CPU and two on a GPU. The CPU is an *Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz (CPUs)*, and the GPU is an *NVIDIA GeForce GTX 1050 Ti*.

9 Results

9.1 Test results

The test results are shown in Tables 3 and 4 below. All nets that were introduced in Section 8 above were evaluated on the different training data sets using the test dataset $\mathcal{D}_{test}$. Refer to Section 8.1 for a description of the data sets. The best results from these tests are shown in Table 3, which also lists whether image reconstruction was used or not. It is worth knowing that there have been many other tests performed with different calibrations, but that the results shown in Tables 3 and 4 were the best performing nets. Also, each *CAPSNET* and *EMNET* were tested with and without image reconstruction.

As for Table 4, it shows the models' accuracies when adding translation, rotation and noise to the images in $\mathcal{D}_{test}$, as discussed in Section 8.3 above. Thus, the results here show the nets' generalization capabilities when the images are transformed in different ways. The tests run in Table 4 were made on the same nets whose test results are shown in the $\mathcal{D}_{hard}$ column of Table 3; something which is indicated by the $\mathcal{D}_{hard}$ column in Table 4.

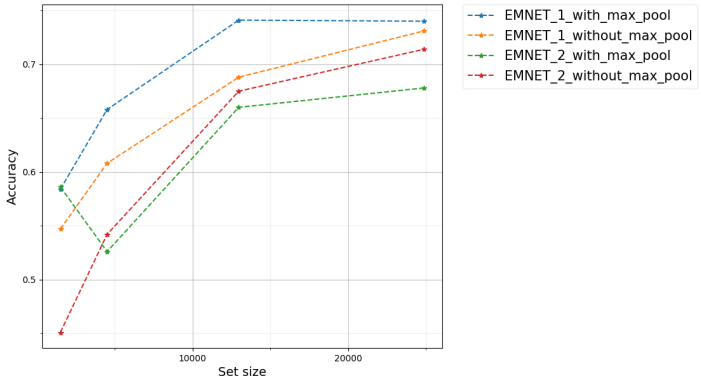
Model net	**INIT-NET**	**Training dataset**					
		$\mathcal{D}_{minisize}$	$\mathcal{D}_{smallsize}$	$\mathcal{D}_{mediumsize}$	$\mathcal{D}_{basic}$	$\mathcal{D}_{medium}$	$\mathcal{D}_{hard}$
CAPSNET 1	1	0.62	(0.713)	0.708	0.884	0.859	0.732
	2	0.515	0.634	0.663	0.892	0.849	0.597
CAPSNET 2	1	(0.636)	0.694	0.627	0.907	0.829	0.674
	2	0.578	0.428	0.566	0.901	0.78	0.639
EMNET 1	1	0.584	0.658	(0.741)	(0.926)	0.861	0.74
	2	0.547	0.608	0.688	0.907	0.855	0.731
EMNET 2	1	0.586	0.526	0.66	0.915	0.809	0.678
	2	0.451	0.542	0.675	0.884	0.812	0.714
CNN-NET 1	1	0.434	0.388	0.358	0.793	0.75	0.253
	2	0.377	0.43	0.304	0.869	0.623	0.4
CNN-NET 2	1	0.57	0.663	0.689	0.911	0.819	0.557
	2	0.509	0.543	0.483	0.809	0.455	0.645
CNN-NET 3	1	0.542	0.316	0.599	0.804	0.856	(0.772)
	2	0.431	0.538	0.455	0.891	(0.865)	0.76

Table 3: This table shows the best test results on the different network models that were trained on the datasets $\mathcal{D}_{minisize}$- $\mathcal{D}_{hard}$. The circled values were the best test performances on each dataset. The gray cells are nets on which a decoder network has been attached; creating a reconstruction image on the input.

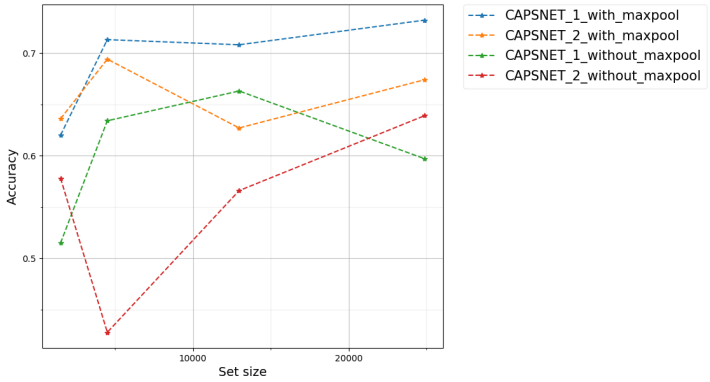
Model net	**INIT-NET**	**Test dataset**			
		$\mathcal{D}_{hard}$	$\mathcal{D}_{tr}$	$\mathcal{D}_{rot}$	$\mathcal{D}_{noise}$
CAPSNET 1	1	0.732	0.282	0.727	0.463
	2	0.597	0.15	0.597	0.264
CAPSNET 2	1	0.674	0.241	0.671	0.511
	2	0.639	0.211	0.633	0.326
EMNET 1	1	0.74	0.504	0.738	0.711
	2	0.731	0.414	0.721	0.709
EMNET 2	1	0.678	0.284	0.674	0.657
	2	0.714	0.271	0.708	0.71
CNN-NET 1	1	0.253	0.1	0.251	0.194
	2	0.4	0.112	0.399	0.397
CNN-NET 2	1	0.557	0.187	0.559	0.512
	2	0.645	0.222	0.633	0.642
CNN-NET 3	1	0.772	0.391	0.762	0.758
	2	0.76	0.3	0.752	0.572

Table 4: Table showing the test results when translations, rotations, and noise were applied to the $\mathcal{D}_{test}$ data set, when testing the models. Circled values are the best of each data set. There are no grey cells, as image reconstruction did not yield the best results on the $\mathcal{D}_{hard}$ data set.

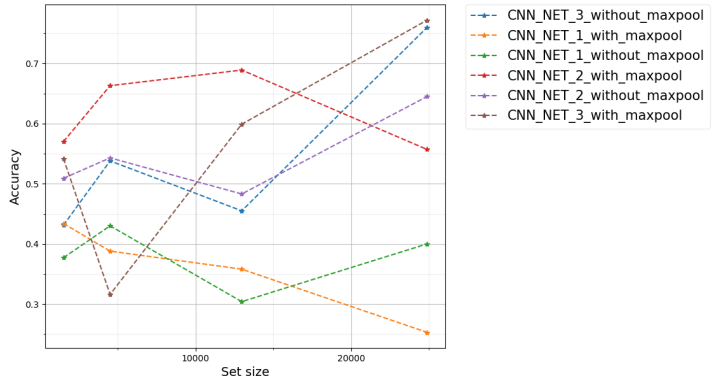
In Figure 31 below, the test results for each model were plotted against the dataset size. The datasets used for measuring this were $\mathcal{D}_{minisize}$, $\mathcal{D}_{smallsize}$, $\mathcal{D}_{mediumsize}$ and $\mathcal{D}_{hard}$, with the total number of images for each dataset being 1500, 4500, 12963 and 24881, respectively. The values in the plots are directly obtained from Table 3 above.



(a) Test results on EMNETs with regards to size of dataset.



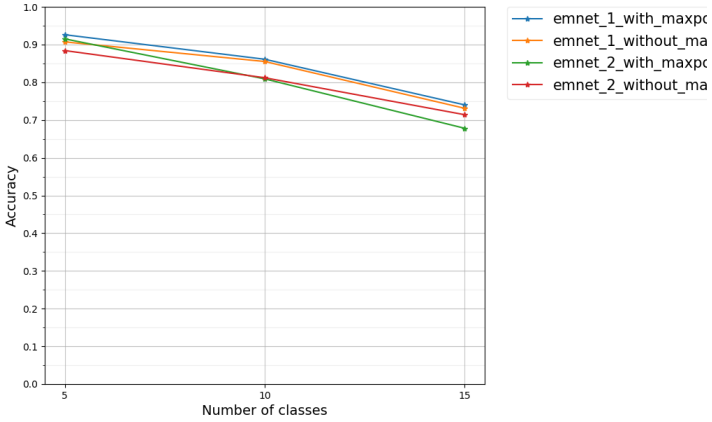
(b) Test results on DCNs with regards to size of dataset.



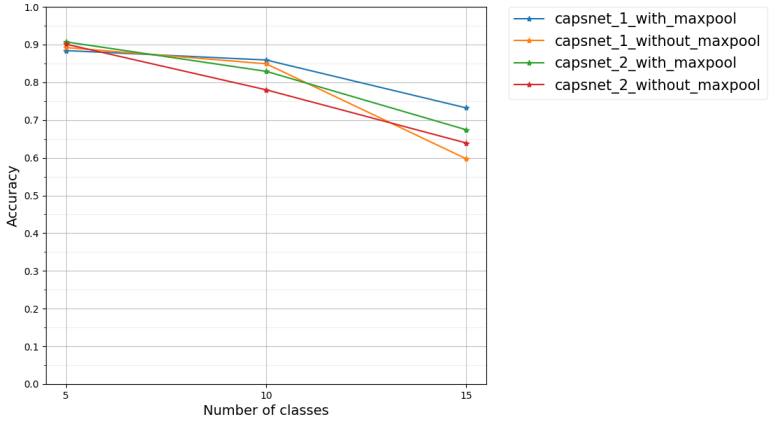
(c) Test results on CNNs with regards to size of dataset.

Figure 31: Test accuracies for each model in Table 3 with regards to the size of the dataset that the model was trained on. The y-axis shows the accuracy of the model, and the x-axis shows the total number of labeled images in the dataset. In the plot legends, *with_maxpool* means that INIT-NET 1 was used, whereas *without_maxpool* means that INIT-NET 2 was used.

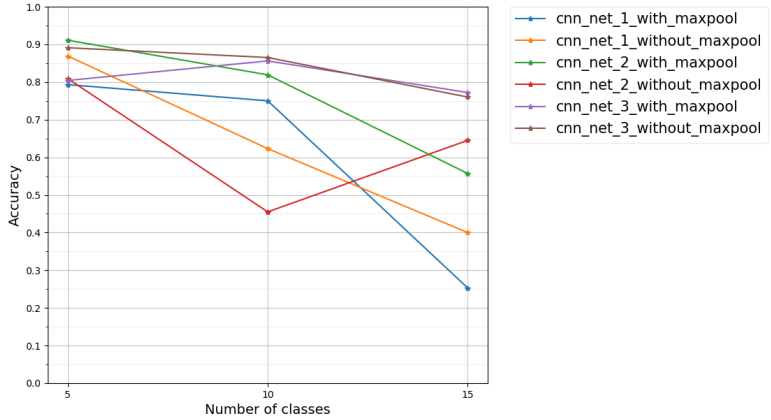
In Figure 32 below, the accuracies of the models were plotted against the number of classes that the model was trained on. In effect, this means that this figure show how the models perform when the complexity of the underlying dataset is changed. The EMCNs and DCNs show very similar performances, where the EMCNs perform slightly better with added complexity. The CNNs have a more varied performance when the complexity of the underlying dataset is changed.



(a) Test results on EMNETs with regards to the number of classes dataset.



(b) Test results on DCNs with regards to the number of classes in dataset.

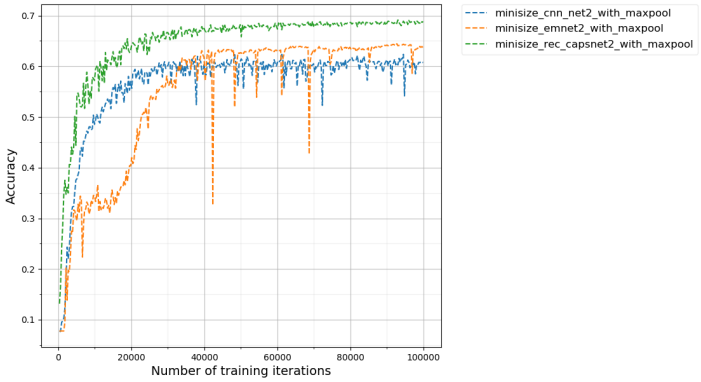


(c) Test results on CNNs with regards to the number of classes in dataset.

Figure 32: Test accuracies for each model in Table 3 with regards to the number of classes (i.e. complexity) in the dataset that the model was trained on. In the plot legends, *with_maxpool* means that INIT-NET 1 was used, whereas *without_maxpool* means that INIT-NET 2 was used.

9.2 Comparison between DCNs', EMCNs' and CNNs' performances.

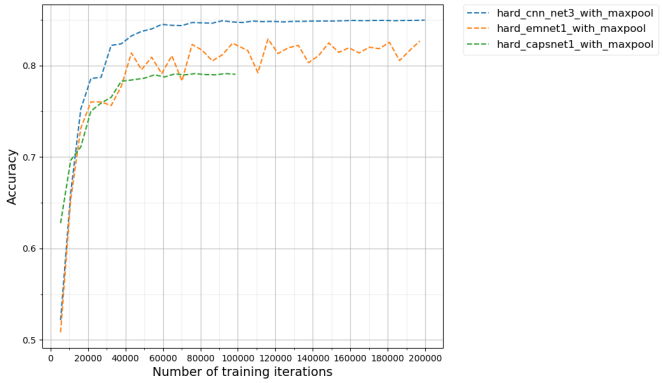
During training of the different models on the different data sets listed in 1, validation accuracies were measured, and the best performing models per data set are shown in Figure 33 below. As can be seen in this figure, the DCNs outperform EMCNs and CNNs when trained using the $\mathcal{D}_{minisize}$ and $\mathcal{D}_{smallsize}$ data sets, whereas when training using the larger datasets $\mathcal{D}_{basic}$ and $\mathcal{D}_{medium}$, all models perform equally. On the hard dataset, the CNN performs best, the EMCN has the second best performance, and the DCN the worst. The models were run for 100,000 iterations on all sets except the hard set, where the EMCN and CNN models were run for 200,000 iterations to check convergence. In the graphs (e.g. 33a and 33b, the EMCN model seems to be unstable in its convergence, as it shows impulses of bad accuracies.



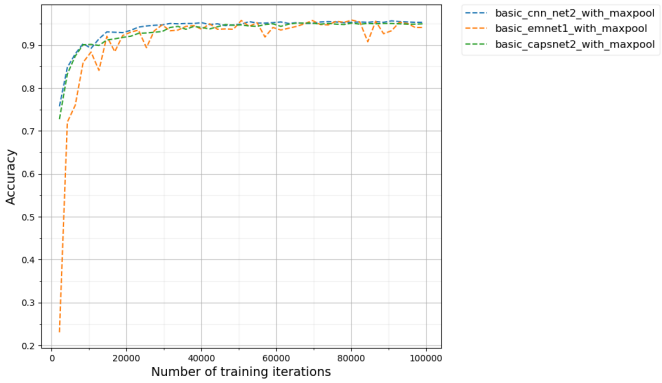
(a) The models' accuracies on the minisize set.

(b) The models' accuracies on the smallsize set.

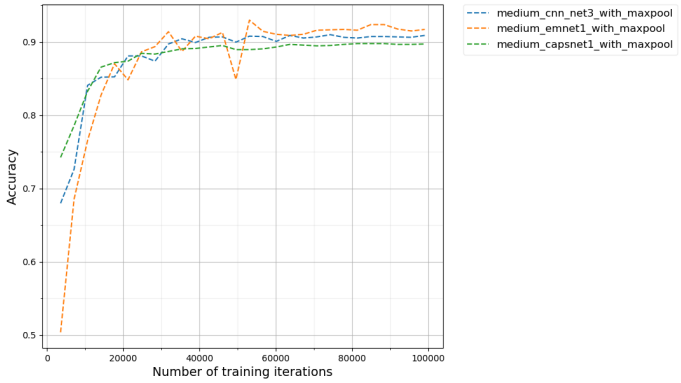
(c) The models' accuracies on the mediumsize set.



(d) The models' accuracies on the hard set.



(e) The models' accuracies on the basic set.



(f) The models' accuracies on the medium set.

Figure 33: Graphs showing the best results of the CNN-, DCN- and EMCN models' validation set accuracies during training. The graphs shown correspond to the best DCN-, EMCN and CNN models of Table 3 above, and the green lines are DCN validation curves, blue lines are CNN validation curves and orange lines are EMCN validation curves. The model settings are indicated by the legend to the right of each graph. *with_maxpool* means that *INIT-NET 1* was used, *no_maxpool* means that *INIT-NET 2* was used, *rec* means that reconstruction of the image was used; i.e. the *DECODER* network was applied. The y-axis corresponds to the accuracy for a validation epoch, and the x-axis corresponds to the number of iterations the plots have been trained for with a batch size of 5 per iteration.

Figure 33 shows that the EMCN has some instabilities in its convergence. In order to test for divergence, the learning rate was shifted for EMNET 1 + INIT-NET 1, and the result is shown in Figure 34 below. These spikes are still present when decreasing the learning rate. Decreasing the learning rate too much makes the model fail to converge.

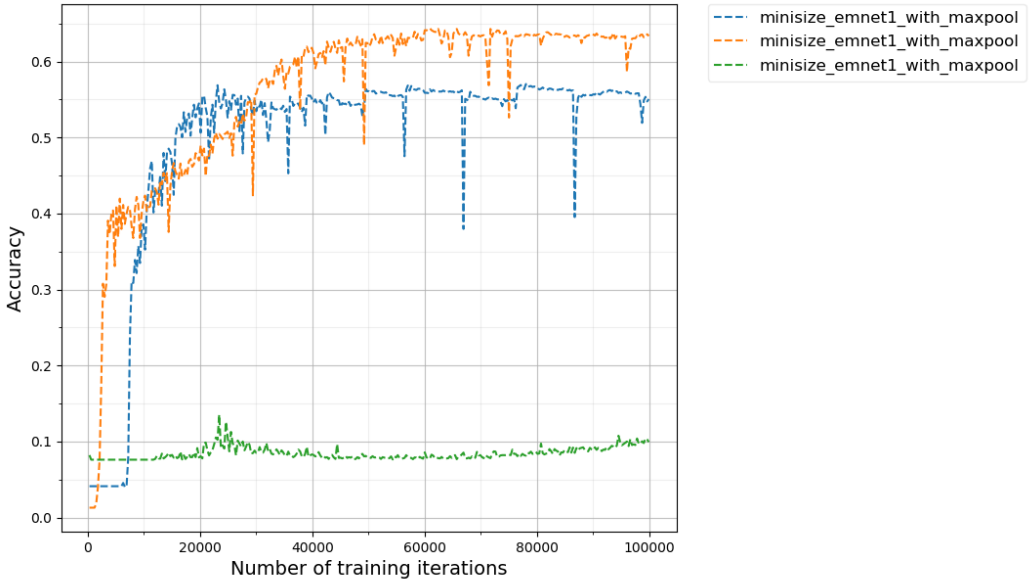


Figure 34: Comparison between different learning rates for EMNET 1 + INIT-NET 1. The blue line shows the EMNET's performance when the learning rate is set to $3 \cdot 10^{-4}$, the orange line when the learning rate is set to $3 \cdot 10^{-5}$ and the green line when the learning rate is set to $3 \cdot 10^{-6}$.

9.3 Reconstruction images

For both the DCN and the EMCN structures, tests were made with applying *DECODER NET* to the end of the network. After having trained these networks, the capsule corresponding to the correct label was perturbated. This means that an image is sent through the network, setting the correct label capsule to values $\mathbf{v}$, and in $\mathbf{v}$, one element v_k at a time gets its value changed with a fixed schedule s (e.g. $s \in \{-0.1, 0, 0.1\}$); setting $v_k \leftarrow v_k + s$. Figure 35 shows this perturbation on the DCN, and Figure 36 shows this perturbation on the EMCN capsule's pose matrix.

In Figure 35, the reconstructed images are blurry, and the perturbations show combinations of rotation, cell types, shapes and colorization. Figure 36 also shows blurry images, but these show much less detail and variation; only the cell nucleus' shape, position and rotation is the varying component.

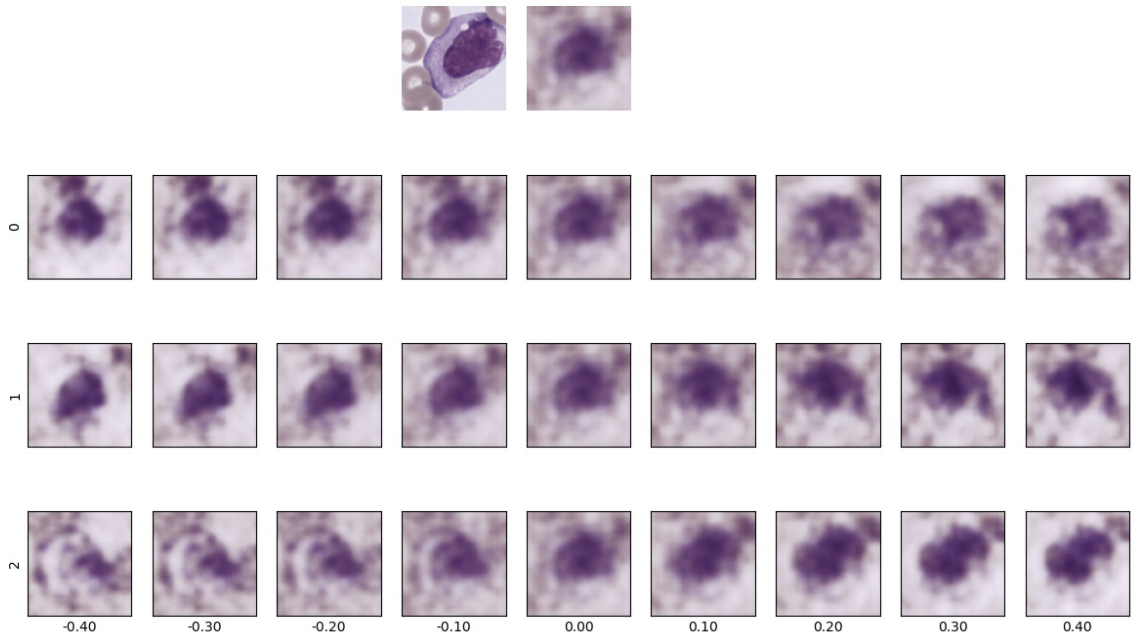


Figure 35: Perturbation of the DCN structure with 3 dimensions shown. As can be seen, the images are blurry. The perturbation schedule ranges from -0.40 to 0.40. The top left image is the input image, and the top right image is the reconstructed image.

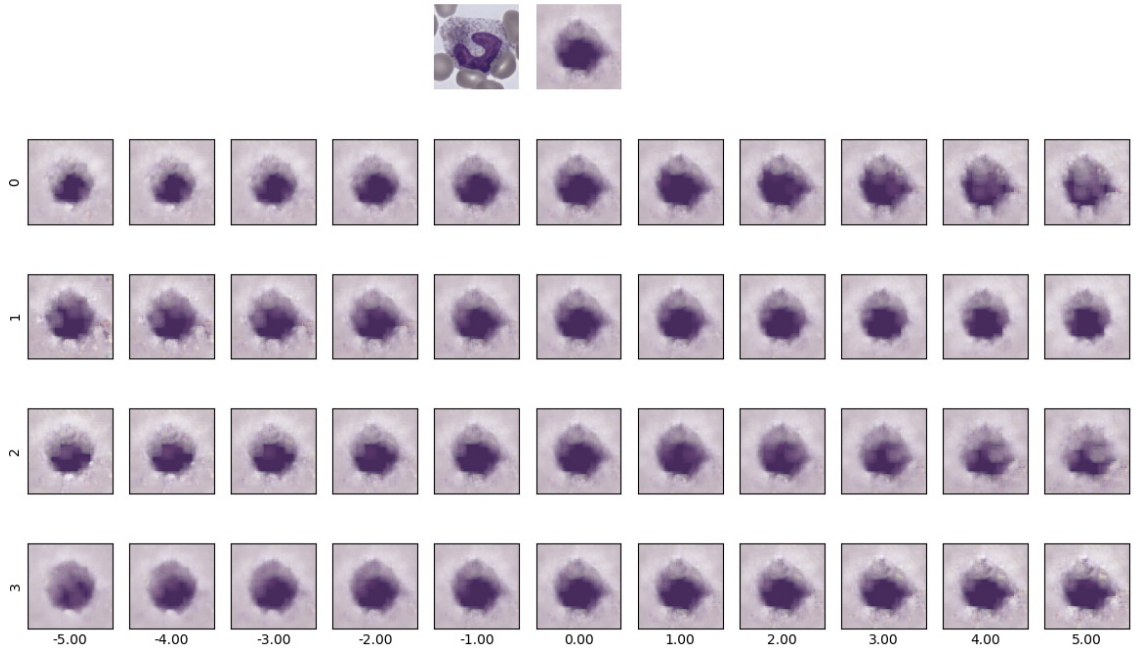


Figure 36: Perturbation of the EMCN structure with 4 dimensions shown. The perturbation schedule ranges from -5.00 to +5.00. The top left image of this figure is the original image, and the top right image is the original reconstruction of the image.

For the DCN, the convergence curve for CAPSNET 1 + INIT-NET 1 was compared for when reconstruction loss was used, compared to when it was not. The result is shown in Figure 37 below. In this case, the performance is clearly improved by the addition of reconstruction loss.

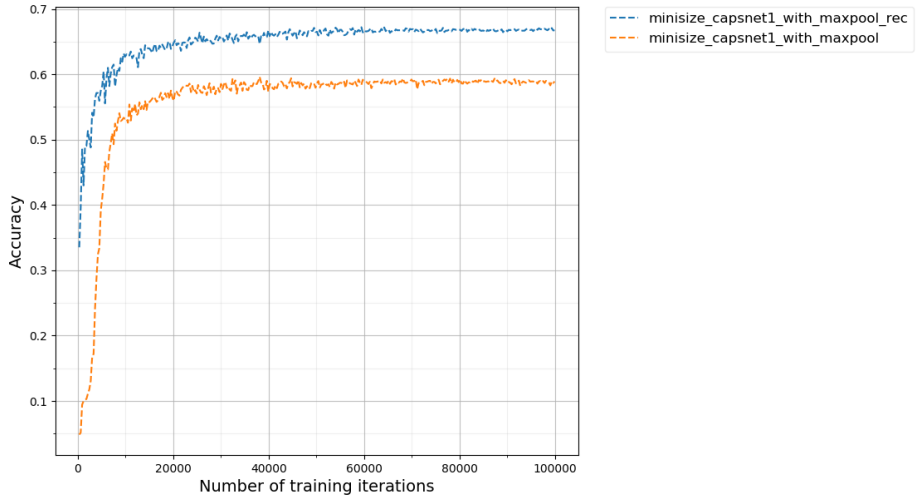


Figure 37: Comparison between the convergence for *CAPSNET 1 + INIT-NET 1* with and without reconstruction loss. Here, *with_maxpool* means that INIT-NET 1 was used, and *rec* means that image reconstruction was applied to the network.

9.4 Execution times

The average execution times for the models are shown in table 5 below. The execution times are measured on both GPU and CPU, and the batch size is 5. As can be seen, CNN-NETs perform the fastest on both CPU and GPU, and EMCNs the second fastest; almost as fast on GPU, whereas DCNs are much slower than the other models.

Model net	**Hardware**	
	CPU	GPU
CAPSNET 1	0.83	0.17
CAPSNET 2	2.62	0.55
CAPSNET 1 + decoder	2.79	0.25
CAPSNET 2 + decoder	4.63	0.67
EMNET 1	0.84	0.12
EMNET 2	2.79	0.12
CNN-NET 1	0.11	0.09
CNN-NET 2	0.16	0.10
CNN-NET 3	0.13	0.09

Table 5: The execution times in seconds for the different models. The models chosen use INIT-NET 1, and are measured when running the algorithms on the dataset $\mathcal{D}_{hard}$.

10 Discussion

10.1 Test results

10.1.1 DCN performance

As can be seen in Table 3 and Figure 33 above, the DCN performs better than the other models on the $\mathcal{D}_{minisize}$ and the $\mathcal{D}_{smallsize}$ datasets, which means that it performs better than the other networks when less data is available. This is especially clear in Figures 33a and 33b, where the DCN both converges faster and better than the other models. From Table 3, the reconstruction loss seems to play a big role in increasing the model's accuracy; something which is further supported in Figure 37, where *CAPSNET 1 + INIT-NET 1 with reconstruction loss* outperforms *CAPSNET 1 + INIT-NET 1 without reconstruction loss*.

10.1.2 EMCN performance

Another thing that is apparent from the results in Figure 33, is that the EMCN gets more unstable the less data that is available, and – like the CNN and unlike the DCN – performs better the more data that is available. This upward trend in stability is also reflected in Figure 31a, where almost all curves have a very steady upward trend. The reason for the instability on smaller datasets is unclear, but it is unlikely that the model diverges, as decreasing the learning rate creates the same instabilities, as shown in Figure 34. It also cannot be that updating the margin loss m by 0.1 every 3000 steps creates this instability, as this updating process is stopped at the 24,000'th step (at $m = 0.9$), and these instabilities still occur after this step. It is also unlikely that the iterative process of EM Routing can cause the model to diverge, as the iterative process of Dynamic Routing does not cause these instabilities. It thus remains unclear what it is that is causing these instabilities.

Moving on, the EMCN performs better than the other models on the $\mathcal{D}_{mediumsize}$ dataset. It is unclear what creates this improvement, as the DCN performs better on smaller datasets and CNNs perform better on bigger datasets. By analyzing the plots in Figure 31, however, the CNN model seems to hit a spot here in which the performance of *CNN-NET 2 + INIT-NET 1* hits a limit, and *CNN-NET 3 + INIT-NET 1* has not yet transitioned into its sweet-spot. As for the DCN, its performance improvements with regards to the amount of data available are too small, which makes the EMNET and DCN outperform this model after a certain set size has been reached.

Furthermore, the EMCN performs better than the other models on the $\mathcal{D}_{basic}$ dataset. This increase in performance, however, is not large in comparison to *CNN-NET 2 + INIT-NET 1* (see Table 3). Indications of the reasons for this performance may be given in Figure 36, however, where the model seems to learn only very basic concepts of the images - which in turn may be due to the fact that the capsules' pose matrices have quite few parameters; having $[4, 4]$-matrices, i.e. 16 parameters, in comparison to the 96 parameters in the tested DCN models. With this indication in mind, the EMCNs might learn basic concepts in the images quite well but fail to learn more complex patterns; making the models perform well on the $\mathcal{D}_{basic}$ dataset but worse on the more complex $\mathcal{D}_{basic}$ dataset.

Lastly, the EMCNs are in general more invariant to transformations of the underlying dataset, as can be seen in Table 4, where the EMCNs - in comparison to the other models - show little variance when transformations are applied to the $\mathcal{D}_{test}$ data set. It even outperforms the other models on the $\mathcal{D}_{tr}$ data set. This gives an indication that EMCNs indeed perform pseudo-computer graphical operations, as they are able to better generalize underlying spatial properties of the images. The reason why all models perform relatively poor on the $\mathcal{D}_{tr}$ dataset, is that all cell images are centered in the training dataset, which means that the nets are not trained on translations of the data.

10.1.3 CNN performance

The CNN model outperforms the other models when more data is available, as can be seen in Figure 33d and in the $\mathcal{D}_{hard}$, $\mathcal{D}_{rot}$ and $\mathcal{D}_{noise}$ columns in Table 3. As previously mentioned, Figure 31 shows that the CNN models *CNN-NET 2 + INIT-NET 1* and *CNN-NET 3 + INIT-NET 1* (together with *CNN-NET 3 + INIT-NET 2*) switch leading positions in terms of accuracy somewhere between 13,000 and 25,000 iterations. It is unclear what is causing *CNN-NET 2 + INIT-NET 1* to perform worse with 25,000 data samples than with 13,000, and why *CNN-NET 1* has a downward accuracy trend with increasing dataset size. The most likely explanation for this decrease in performance might lie in the fact that all CNN-NETs ran for 200,000 iterations on the $\mathcal{D}_{hard}$ dataset (25,000 data points), which might have led to *CNN-NET 2* overfitting the data - especially since it has the most amount of parameters of the CNN-NETs. In case of *CNN-NET 1*, it is most likely a bad model, as its performance is worse or much worse than *CNN-NET 2*'s and *CNN-NET 3*'s on all datasets; which means that its downward trend can most likely be ignored.

10.1.4 Performances with regards to complexity

As can be seen in Figure 32 above, EMCNs and DCNs have very similar performances with regards to changing the complexity of the underlying dataset. DCNs perform slightly worse than EMCNs when complexity is increased, which - in conjunction with the small performance improvements with larger datasets - may also explain why DCNs have the worst performances of all the models on the $\mathcal{D}_{hard}$ dataset.

In comparison to Figure 32c, however, both the EMCN and DCN models seem to have a smaller variation when complexity is varied in the underlying dataset. The reason for this difference might be that *CNN-NET 1* might be a bad model, as was briefly discussed in section 10.1.3 above. Furthermore, the *CNN-NET 2 + INIT-NET 2* model in Figure 32c was trained for 100,000 iterations when trained on 10 classes, and 200,000 iterations when trained on 15 classes, which might mean that the model has not converged to an optimum for the 10-class point at the time of measurement. Another explanation is that the model may overfit the training data at this point; something which is backed up by the fact that this particular data point does not use regularization in comparison to the 5-class point.

10.1.5 Criticism of the run tests

A possible criticism of the tests that have been made, is that some models may overfit the training data, and thus generalize poorly. This has been dealt with in best effort by using *L2 regularization* and different network structures with a highly varying amount of parameters. The CNN-nets were created with this in mind, as these have different sizes in order to account for overfitting and underfitting; if one model overfits, another will fit better. Moreover, these have dropouts, which further reduces overfitting. Furthermore, these nets have been constructed in order to increase the comparability of the models, which means that it is hard to create smaller CNN nets that can still be compared to the EMCN and DCN models in a similar manner as the one used in this paper.

10.2 Image reconstruction perturbations

The reconstructed images in both the DCN model and the EMCN model are blurry, as seen in Figures 35 and 36. In the DCN model, the bluriness can be described by the mean of squares loss, due to the fact that high differences in single pixels contribute to the loss, not more generalized patterns; failing to catch more general features of the cell images. In the EMCN model, the bluriness can be, apart from the mean of squares loss, explained by the fact that its pose matrix only contains 16 elements,

which may be too small to catch details in the images. The DCN model has been run with thicker capsules with similar bluriness in the produced images, which means that the number of elements do not seem to be the limitation in details for the reconstructed images.

Nevertheless, both Figures 35 and 36 give indications as to how these models "see" the input data. The DCN seems to learn combinations of rotation, cell shapes, colorization and background noises. The EMCN, meanwhile, seems to focus on the nucleus, as the cell shape and background noise is almost constant throughout the perturbations, while the nucleus shape, color and position changes. This further indicates that the EMCNs perform pseudo-computer graphical operations, as they focus more heavily on spatial properties than purely graphical ones.

10.3 Execution times

The execution times as shown by Table 5 show that the DCNs are slow on both CPU and GPU, whereas CNNs are fast on both CPU and GPU. The EMCNs are in between, which means that they are slow on CPU, but almost as fast as the tested CNN models when run on GPU. This does not, however, take into consideration that the CNN models have a considerably higher amount of parameters (7-37 times more, to be precise), which - in combination with the fact that GPUs perform parallell computations, which makes the speed non-linear - makes it harder to accurately compare these models' speeds. Also, the EMCNs and DCNs are not optimized, as these have been written in TensorFlow. Tensorflow does not currently have any optimized implementation for these algorithms, which means that their execution times may be improved upon, e.g. by implementing them in C++.

Furthermore, the CNN models tested in this paper have more layers and parameters than the ones CellaVision currently use, which means that the execution times for the CNNs shown in table 5 are slower than the ones currently in use. According to Mattias Nilsson, the execution times need to be less than 50 ms on CPU in order to be shipped with CellaVision's instruments, which neither of these algorithms can do. The CNN nets can achieve this by scaling down the model, but the DCN and (to smaller extent) EMCN models need many layers in order to scale down the input $[224, 224, 3]$-images; otherwise the hardware runs out of memory.

Lastly, the execution times shown in Table 5 are checked when running the models on a batch size of 5. Increasing the batch size would make the execution time per image faster on the GPU, as it is able to process the images in parallell. However, the batch size of 5 is necessary, as the hardware cannot provide the necessary memory for computing with higher batch sizes for the DCNs. Also, CellaVision's systems only use a batch size of 1, which means that increasing the batch size is not necessary for making the algorithms compatible with CellaVision's systems. In conclusion, The DCN and EMCN models are unfit for use in CellaVision's current microscopes, with regards to execution times.

10.4 Memory usage

Owing to Table 2, the EMCN model has clearly got the smallest parameter usage, whereas the CNN and DCN models have more than ten times the parameter usage. However, the CNN model does not necessarily need this many parameters, and mainly uses this many parameters in order to use more or an equal amount of parameters to the DCN. It is, however, still reasonable to assume that a well performing CNN model uses more parameters than a well performing EMCN model, as indicated both by Table 2 and Hinton et al. [2018]. Thus, in this aspect, EMCNs are better than CNNs, which - in turn - are better than DCNs.

10.5 Ethical considerations

All the cell images in this project are anonymous and thus cannot be linked to any person.

10.6 Conclusion

The DCN outperforms the CNN and EMCN models when the datasets are small, and image reconstruction is applied to the DCN. The EMCN model gets unstable when the datasets are small and complex, but seems to have a better translational invariance than the DCN and the CNN, as shown in Table 3. The CNN, meanwhile, outperforms the DCN and the EMCN when the datasets are large and complex, as shown in Figure 33d. With regards to speed, the DCN is very slow in comparison to the CNN and CellaVision's required speed, whereas the EMCN has similar speeds to the CNN on GPU, but is much slower on a CPU. Both of these algorithms, however, can be optimized further to improve their speed.

To conclude, the DCN and EMCN models are both too slow and perform worse than the CNN models on big and complex datasets; which is the type of dataset in which CellaVision seeks performance improvements. The EMCN indicates better generalization for spatial concepts as well as relatively good execution times and little memory usage, but it does not achieve the necessary accuracies and is too unstable to replace CNNs. This model needs improvements, but may prove useful in the future. The DCN may provide better performances with regards to accuracy if any future dataset of interest is very small, and execution times and memory usage is not a limitation. Otherwise, the CNN models should see their continued use in CellaVision's systems.

10.7 Further improvements

As previously mentioned, the EMCN and DCN implementations may be optimized, e.g. by implementing them in C++ or similar hardware-close programming languages. Another improvement that may be done is to combine the EMCN with neural architecture search (NAS), [Zoph and Le, 2017], in order to find optimal architectures; something which has been hard to find in this project, due to many degrees of freedom with many different variables such as lambda schedules, regularization scale, image reconstruction and learning rates. The reason why DCNs are left out of this improvement suggestion, is that these networks easily run into configurations where the hardware runs out of memory; creating a more error-prone search space.

References

Barbara J. Bain. *A Beginner's Guide to Blood Cells*. Blackwell Publishing, 2nd edition, 2004.

Ronald Goldman. *An Integrated Introduction to Computer Graphics and Geometric Modeling*. CRC Press, 2009. URL `https://books.google.se/books?id=Wy3NBQAAQBAJ&printsec=frontcover&hl=sv#v=onepage&q&f=false`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Geoffrey E. Hinton, Sara Sabour, and Nicholas Frosst. Dynamic routing between capsules. 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 2017. Neural Information Processing Systems (NIPS) Foundation. URL `https://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf`.

Geoffrey E. Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with em routing. International Conference on Learning Representations (ICLR) 2018, Vancouver Convention Center, Vancouver CANADA, 2018. URL https://openreview.net/pdf?id=HJWLfGWRb.

Jonathan Hui. Understanding matrix capsules with em routing (based on hinton's capsule networks). Blog post, November 2017. URL https://jhui.github.io/2017/11/14/Matrix-Capsules-with-EM-routing-Capsule-Network/.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. Mar 2015. arXiv: 1502.03167.

Ujjwal Karn. An intuitive explanation of convolutional neural networks. Blog post, August 2016. URL https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, and Mohammed Bennamoun. *A Guide to Convolutional Neural Networks for Computer Vision*. Morgan Claypool Publishers, 2018. URL https://books.google.se/books?id=e0NRDwAAQBAJ&printsec=frontcover&hl=sv#v=onepage&q&f=false.

Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. International Conference on Learning Representations (ICLR) 2015, Vancouver Convention Center, Vancouver CANADA, 2015. URL https://arxiv.org/pdf/1412.6980.pdf.

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. http://neuralnetworksanddeeplearning.com (Last updated: 2017-12-02, Last visited: 2018-07-26).

Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016. doi: 10.23915/distill.00003. URL http://distill.pub/2016/deconv-checkerboard.

Abhineet Saxena. Convolutional neural networks (cnns): An illustrated explanation. URL https://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/. Last updated: 2016-06-29, Last visited: 2018-04-24.

James M. Van Verth and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications*. CRC Press, 3rd edition, 2016. URL https://books.google.se/books?id=_9OOCgAAQBAJ&printsec=frontcover&hl=sv&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false.

Qiang Wu, Fatima Merchant, and Kenneth R. Castleman. *Microscope Image Processing*. Academic Press, 2008. URL https://books.google.se/books?id=uGWmR0f_350C&printsec=frontcover&hl=sv#v=onepage&q&f=false.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. Feb 2017. arXiv: 1611.01578.