DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Evaluation and Generalization of Capsule Networks in Neurorobotics

Jean Amadeus Elsner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Evaluation and Generalization of Capsule Networks in Neurorobotics

Evaluation und Generalisierung von Kapsel Netzwerken in der Neurorobotik

Author:	Jean Amadeus Elsner
Supervisor:	Prof. Dr. Alois C. Knoll
Advisor:	Alexander Kuhn
Submission Date:	TBD

I confirm that this master's thesis in robotics, cognition, intelligence is my own work and I have documented all sources and material used.

Munich, TBD Jean Amadeus Elsner

Acknowledgments

Abstract

The last few years have seen great strides being made in the fields of artificial intelligence and robotics. Many of the advancements are powered by the versatility of artificial neural networks. Especially in computer vision, deep learning architectures are particularly successful. For robotic systems, visual sensors are often their primary means to perceive their environment and negotiate it successfully. Thus, computer vision systems form a vital part of the artificial intelligence necessary to allow autonomous operation of robots. There are however several factors limiting the widespread use of such systems in robotics. Namely, the need for massive data sets, the lack of generalization to novel configurations and the high computational cost associated with the training of neural networks. Neurorobotics tries to avoid these limitations by mimicking the behavior of biological systems, e.g. by using biologically inspired spiking neural networks. Recently, capsule networks have been suggested as an alternative biologically plausible computer vision system, albeit on a higher level of abstraction, based on second generation neural networks. Capsules are conceived as groups of neurons, that represent the presence of an entity and its instantiation parameters. It is proposed that by dynamically constructing a parse tree from a network of capsules, viewpoint invariance and better generalization can be achieved. In this thesis, object recognition tasks generated by the Neurorobotics Platform are used to evaluate the performance of capsule networks and quantify their ability to generalize compared to established methods such as convolutional and spiking neural networks.

Contents

1 Introduction

The success of techniques from the field of artificial intelligence (AI) in recent years is quite evident. Not just are there ever more real-world applications of AI found in search engines, security systems, industrial automation and more. But the term *artificial intelligence* itself has become a vogue word, even outside of academic literature. In fact, often it is not immediately apparent what kind of technology is being referred to when the term AI is used. This is true to the point that opposing paradigms claim the mantle of AI. Classical AIs were conceived as systems performing logical inference on a knowledge database in the form of search algorithms (*cognitivist* paradigm) as opposed to the current use of artificial neural networks (an aspect of the *emergent* paradigm). Research into artificial neural networks today has diverged into two distinct branches [1]. On the one hand there are spiking neural networks that are mostly used as tools to study biological nerve cells by closely modeling the physical properties of neurons and synapses. On the other there are multi-layer perceptrons, convolutional neural networks chief among them, that are optimized for machine learning tasks such as object detection in images or speech recognition. Spiking neural network have inherently great potential, as they are directly derived from biological examples. Still they are almost always outperformed by the latter [2, 3]. A point can be made that much of the drive behind AI comes from the successful application of deep artificial neural networks in computer vision tasks [4]. As convolutional neural networks designed for computer vision share similarities with the mammalian visual cortex [5], a similar case can be made for the contribution of neuroscience. The eye is arguably the most intensely studied of the human sensory organs and the visual cortex is one of the best understood parts of the brain. It is therefore only sensible to try and mimic the properties and behavior of brains when developing new vision systems. While cameras already surpass the spatial and temporal resolution of biological eyes, the human visual system remains unmatched in visual-cognitive tasks. Moreover, computer vision solutions currently available often have a high latency, precluding them from being used in real time experiments. In order to be able to validate biologically inspired AI models it is therefore necessary to simulate a realistic environment [6]. The Human Brain Project's *Neurorobotics Platform*[1] was conceived to provide just such a simulation, a closed loop between a brain simulation based on artificial neural networks and a physics-based simulation of robotic bodies embedded in a dynamic environment [7]. In this thesis, sensor data as generated by an experiment within the Neurorobotics Platform was used to create challenging computer vision tasks. These tasks evaluate general object recognition performance

[1] Available at `http://neurorobotics.net/`

and specific generalization and robustness, e.g. generalization to new viewpoints or robustness against occlusion. The use of a high-fidelity simulation platform allows for the generation of big labelled datasets that can be fine-tuned towards the strengths and weaknesses of the architecture at hand. Something that would otherwise require many hours of manual labor.

1.1 Motivation

Capsule networks have been proposed as a more biologically plausible alternative to convolutional neural networks in computer vision. The routing of signals between capsule layers, based on grouping votes from the lower layer, allows capsules to encode the image structure into more sophisticated internal representations. To address the open question of whether this leads to better generalization, capsule network architectures will be explicitly evaluated in this thesis using custom-made datasets created within the Neurorobotics Platform. Networks representing both ends of the biological plausibility spectrum will be used as baseline to see how capsule networks perform in object recognition tasks and whether there is a need for them at all.

1.2 Scope

The thesis starts with a derivation of the state of the art of the artificial neural networks used as baseline in this thesis in chapter 2. The chapter also includes details about the limitations of the discussed architectures and explains classification by example of object recognition. This is followed by an introduction of the motivation and theory behind capsule networks as well as a brief review of their development in chapter 3. How the Neurorobotics Platform was used to create the datasets as well as how exactly each architecture was trained and tested and what metrics were applied is explained in chapter 4. The numerical results from the experiments described in the previous chapter are presented in chapter 5. An interpretation of those results is discussed in the final chapter 6, which also concludes the thesis with an outlook on possible further developments and open questions.

2 State of the Art

This chapter presents an overview of state-of-the-art approaches to object recognition. Focus is put on two families of artificial neural network (ANN) architectures, which are motivated quite differently. Object recognition techniques based on *convolutional neural networks* (CNNs) currently dominate the field, achieving state of the art performance by a large margin over classical methods on many datasets [8, 9]. Even though CNNs were originally inspired by findings from neuroscience, by now they have diverged quite a bit from the computational models used to study the brain. The neurorobotics approach to cognitive systems, based on *spiking neural networks* (SNNs) on the other hand, attempts to mimic the brain by more closely modelling the physical properties and behavior of neurons and therefore results in biologically more plausible models [10]. Generally speaking, CNNs may be regarded as a more engineering-based approach (or top-down), while SNNs are motivated by results from neuroscience and biology (bottom-up approach).

2.1 Artificial Neural Networks for Object Recognition

Recent years have seen a surge of interest in artificial neural networks and deep learning methods, especially in the field of computer vision. While the theory for training such networks has been around for many years, their recent success is mainly due to the availability of large labelled data sets (so called big data) and the proliferation of highly parallel computing powered by GPUs. Object recognition on the other hand refers to the ability to perceive visual instantiation parameters (such as shape, color and texture) of an object and derive semantic attributes from the observed features (e.g. by identifying the object as a chair). This includes integrating the object into the observer's ontology or model of the world, e.g. by associating previous experience with similar objects, understanding its use and relation to other objects. Humans are able to identify and label visual objects quite effectively, even under varying viewpoints, lighting conditions, etc. This is facilitated by the hierarchical structure of the human visual cortex, where information is processed with increasing complexity depending on the depth. With more basic feature extraction happening in the lower level cortical processors of the primary visual cortex and higher-level reasoning associated with regions such as the inferotemporal cortex at the top [11, 12]. It is by no accident that the ANN models most successful in computer vision tasks are also based on hierarchical representation. In machine learning, tasks such as object recognition are referred to as classification problems. The resulting classifiers are part of a broader class of methods called *supervised*

or *discriminative* learning. In supervised learning, the desired output (i.e. the label) has to be provided for all the samples used to train the classifier (cf. figure 2.1). Advances in neural networks may facilitate understanding of how exactly visual object recognition works in humans. Interest is further fueled by the myriad potential applications in technology, from automated driving and image-based diagnosis in medicine to robot vision and many more. Deep learning today constitutes already a great improvement in computer vision over classical methods and still holds much potential for the future.

Figure 2.1: Images from the CIFAR-10 [13] dataset and their corresponding classes. CIFAR-10 consists of 6000 images at 32×32 pixels for each of the 10 classes. Datasets such as this are often used as a benchmark to evaluate the performance of novel ANN architectures in object recognition. This is done by using a subset of the dataset (referred to as the training set) to train the neural network. The remainder of the images (accordingly called the test set) are used to evaluate the classification accuracy.

The significantly better performance of deep neural networks over traditional machine learning methods (i.e. those using handcrafted features) can be explained by:

1. The hierarchical topology of parameterized non-linear processing units in artificial neural networks is a fundamentally better probabilistic model and prior for real world data as captured in images leading to better generalization

2. Kernels are trained to find good features to extract based on the training samples.

2.2 Convolutional Neural Networks

CNN architectures are generally distinguished by their use of specific types of neuronal layers, namely, convolutional, pooling and fully connected layers. While wildly different network topologies may be found in literature, characterized by their use of skip connections, number of layers, number of paths etc., CNNs can always be reduced to these three basic layer types.

Fully Connected Layer

Fully connected layers are actually the simplest type of layer from a computational point of view. They are often used as the top layers of CNNs and perform high-level reasoning based on the features found by lower level layers. Each neuron in a fully connected layer is connected to all the neurons in the lower layer. The activation of a single neuron (cf. figure 2.2) is calculated by applying a nonlinearity to the weighted sum of its inputs plus a bias.

$$h = g\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

With the nonlinear function g, the learnable weights w_i, the input activations x_i and the learnable bias b. The numerical value h computed by the output function g can be interpreted as representing the spiking rate of that neuron (cf. section 2.3).

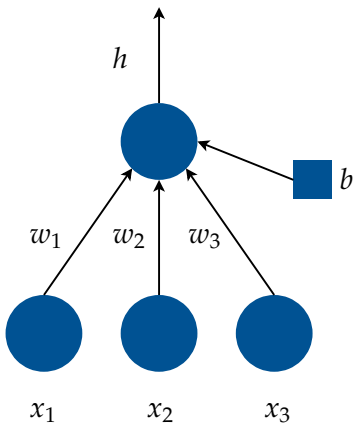


Figure 2.2: Illustration of an artificial neuron with three input connections. Artificial neurons constitute the basic nonlinear computational units of artificial neural networks. The neuron receives the activations of three lower level neurons weighted by the trainable parameters w_i as well as a learnable bias b. After applying the nonlinearity (also known as activation function), the neuron outputs its own activation h as seen in equation 2.1

Compared to linear neurons, the output function g, also called *transfer function*, allows the network to compute nontrivial problems. A popular choice for g is the so-called Rectified Linear Unit (ReLU).

$$g(x) = \max(0, x) \tag{2.2}$$

ReLUs have several advantages over the sigmoidal activation functions that were common in ANNs before. They allowed deep neural networks for the first time to be trained directly without the need for prior unsupervised training [14]. In the case of a fully connected layer, the activations can be computed using matrix multiplication. In tensor notation this may be written as:

$$\mathbf{h_l} = g_l\left(\mathbf{W_l^T h_{l-1}} + \mathbf{b_l}\right). \tag{2.3}$$

With N_l denoting the number of neurons in layer l, $\mathbf{W_l}$ is an $N_{l-1} \times N_l$ dimensional weight matrix, b_l an N_l dimensional vector and g_l the N_l dimensional vectorized activation function of layer l.

$$g_l(\mathbf{x}) = (g_l(x_1), ..., g_l(x_{N_l}))^T \tag{2.4}$$

The computational power of neural networks is shown by the universal approximation theorem. The theorem states that networks with a single hidden layer (cf. figure 2.3) containing a finite number of neurons can approximate arbitrary continuous functions on compact subsets of $\mathbb{R}^n$ [15, 16].

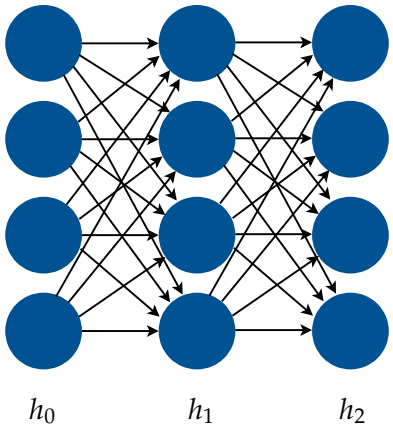


Figure 2.3: Illustration of a fully connected neural network as a directed graph with input layer h_0 and output layer h_2. Layers between the input and output layer are usually referred to as *hidden* layers. Note that compared to figure 2.2 the biases are not explicitly shown.

The CNNs discussed in this section are strictly feed-forward. Feed-forward networks allow propagation of signals in only one direction and do not contain any cycles. While there are other successful architectures like recurrent neural networks (RNNs), restricted Boltzmann machines (RBMs) or deep belief networks (DBNs), they do not play an important role in computer vision tasks compared to CNNs.

Convolutional Layer

In a convolutional layer, the activities of the input layer are convolved with a number of trainable kernels so as to create the same number of feature maps. In computer vision it is common to view the layer's neurons as two-dimensional grids of neurons arranged in channels (cf. figure 2.4).

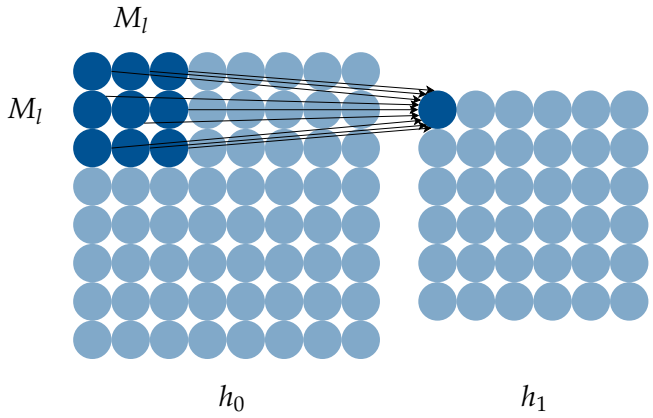


Figure 2.4: Illustration of a 2D convolution with a kernel of size $M_l \times M_l$. As the resulting feature map h_1 has a lower resolution than h_0, applying a convolutional layer is sometimes also referred to as downsampling. For an input layer with $N \times N$ neurons, the feature map's size is computed as $(N - M_l + 1) \times (N - M_l + 1)$. The area comprised of the input neurons that are used to calculate the activity of a feature map neuron (highlighted neurons in layer h_0) are known as that neuron's *receptive field*.

These grids correspond to pixels and color channels in the case of the input layer or activities (feature maps) resulting from convolution with different kernels in the case of intermediate convolutional layers. For a feature kernel $F^l_{m,n}$ of size $M_l \times M_l$ and an input layer $l-1$ with $N_{l-1} \times N_{l-1}$ neurons, the corresponding feature map activities of layer l are computed as

$$h^l_{m,n} = g_l\left(\sum_{m'}^{M_l}\sum_{n'}^{M_l} F^l_{m',n'} h^l_{m+m',n+n'} + b_l\right). \tag{2.5}$$

This is the same as a two-dimensional discrete cross correlation.

$$(f \star g)[n] \overset{\text{def}}{=} \sum_{m=-\infty}^{\infty} f^*[m]\; g[m+n] \tag{2.6}$$

Where f^* is the complex conjugate of the discrete function f. Strictly speaking, the use of the term *convolution* in neural network literature is therefore a misnomer, as either the filter or the image (or feature map) would have to be flipped before the operation. However, as the weights of the kernel are actually learned by the network, the result

will be the same and any flipping would be redundant. For a 2D single channel input layer of size $N_0 \times N_0$, the number of trainable parameters for a convolutional layer with a single $M \times M$ filter is $M^2 + 1$ for the kernel weights and bias respectively, compared to $N_0^2 N_1^2 + N_1^2$ for a fully connected layer of size $N_1 \times N_1$. The sparsity in learnable parameters in convolutional layers compared to fully connected layers (cf. figure 2.5) is often referred to as *weight sharing* (an entire channel *shares* the weights of a single kernel at all positions).

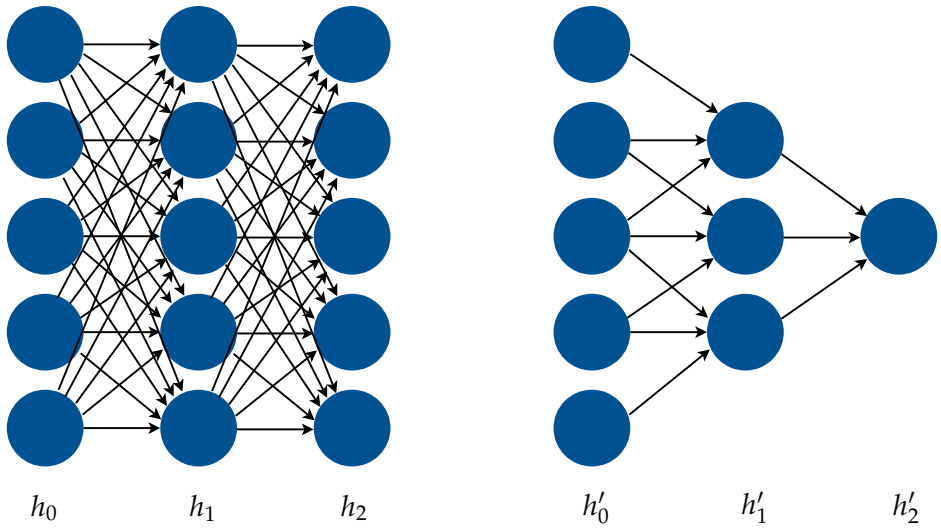


Figure 2.5: Illustration of fully connected layers compared to convolutional layers, making apparent the sparsity of the latter relative to the former. The convolutional layers h'_1 and h'_2 perform a 1D convolution with a filter of size 3. Note that due to weight sharing the number of weight parameters between h'_0 and h'_1 is actually only 3. Also note that convolution reduces the number of neurons (or pixel resolution in the case of an image). For an input layer with N neurons and a filter kernel of size M the number of neurons in the convolutional layer computes as $N - M + 1$.

In the case of multiple input channels, the kernels actually extend throughout the whole depth of the input layer's volume. Equation 2.5 can be extended to accommodate multiple input channels and feature kernels. The activities of feature map k in layer l are then computed as

$$h^l_{k,m,n} = g_l\left(\sum_{k'}^{K_{l-1}} \sum_{m'}^{M_l} \sum_{n'}^{M_l} F^l_{k,k',m',n'} h^l_{k',m+m',n+n'} + b_{l,k}\right). \tag{2.7}$$

With K_l the number of channels in layer l. Most deep learning frameworks offer additional hyperparameters for convolutional layers, like stride and zero padding. Zero padding adds additional rows and columns of zero-activities around the input layer channels, effectively allowing the output feature maps to have the same size (i.e. height

and width, depth is determined by the number of kernels) as the unpadded input layer. Stride on the other hand defines the integer value by which the kernels are moved during convolution. This can be used to keep the receptive fields from overlapping too much. Equation 2.7 is easily extended to take the stride S_l of layer l into consideration.

$$h^l_{k,m,n} = g_l\left(\sum_{k'}^{K_{l-1}}\sum_{m'}^{M_l}\sum_{n'}^{M_l} F^l_{k,k',m',n'} h^l_{k',S_l m+m',S_l n+n'} + b_{l,k}\right) \tag{2.8}$$

The spatial dimension of this feature map can be computed as a function of the input layer size $N_{l-1} \times N_{l-1}$ and the amount of zero padding P_l.

$$N_l = \frac{N_{l-1} - M_l + 2P_l}{S_l} + 1 \tag{2.9}$$

Pooling Layer

Pooling is a form of sub- or downsampling using a sliding window similar to convolution. Most often the stride is set equal to the window size - that way the windows don't overlap. An operator is applied to each window, which selects a single neuron (cf. figure 2.6). Common operators are maximum, average or L_2-Norm. The pooling is applied to each channel and leads to a reduction in the spatial dimensions. This reduction corresponds to a loss of information or reduction in degrees of freedom and therefore reduces the amount of computation required as well as the possibility of overfitting. The choice of pooling operator has a profound effect on the generalization and speed of convergence of CNNs and recently maximum pooling has proven to be the most successful method [17, 18].

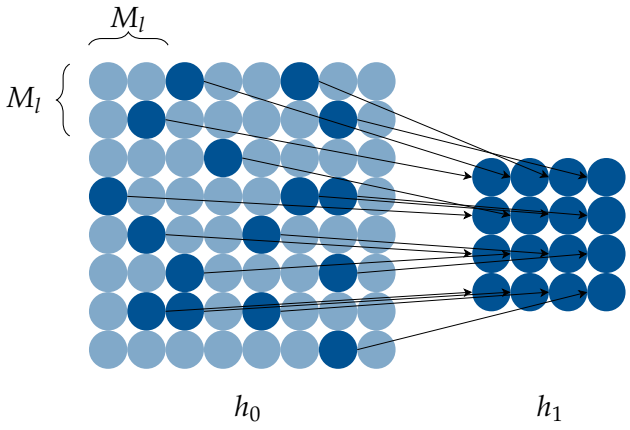


Figure 2.6: Illustration of a pooling layer. The pooling uses a 2×2 window with stride 2 to segment the input layer into non-overlapping tiles. An operator (e.g. maximum) is applied to each 2×2 segment to select a single neuron (the highlighted neurons in layer h_0). The activities of the selected neurons are arranged in a new layer h_1 of size $\frac{N_{l-1}}{M_l} \times \frac{N_{l-1}}{M_l}$ with $M_l \times M_l$ and $N_{l-1} \times N_{l-1}$ the size of the window and the input layer respectively.

CNN Architecture

The key idea behind the combination of these three types of layers is that essential visual features such as edges and corners within a convolutional layer's receptive field are combined to form higher level features such as shapes by subsequent convolutional layers. In between these convolutions, pooling operations select the most salient features and reduce the computational size of the network. The convolutional kernels are effectively trainable feature detectors and due to weight sharing and pooling naturally incorporate a measure of translational invariance. This hierarchical organization of receptive fields is similar in structure to the mammalian visual cortex [19] and sometimes CNNs are seen to be directly derived from it [20, 21]. More often than not however, the use of convolutional layers with weight sharing is motivated as a means to achieve translational equivariance and faster computation compared to fully connected layers [22, 23, 24]. Finally, fully connected layers are placed on top of the network in order to perform high level inference (cf. figure 2.7). A CNN's architecture is therefore largely defined by its topology: the number and types of layers as well their neurons and the connections between them. The number of stacked layers is referred to as the network's depth. Current networks often employ multiple paths and skip connections (i.e. a layer's output not only flows to its direct descendent but skips several layers) allowing for topologies several hundred layers deep, hence *deep learning* [25, 26, 27, 28].

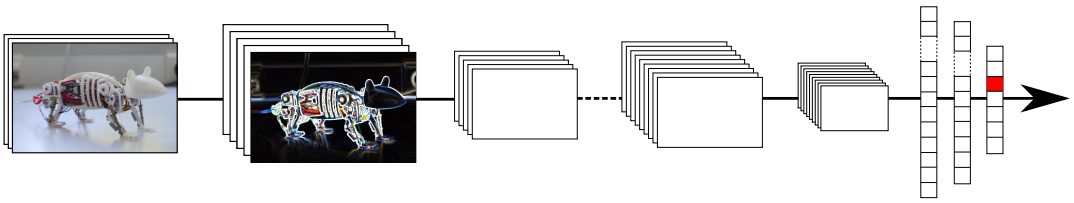


Figure 2.7: Illustration of a typical CNN architecture for object recognition. Note that the feature maps of the first convolutional layer extracted elementary features like edges. The last layer's neurons correspond to the trained classes, while the output neuron with the highest probability is picked as the network's prediction during inference (highlighted in red).

Training

For object recognition, the output of the final layers are the class probabilities. This is achieved by applying a normalizing activation function to the output of the last layer, such as the *softmax* function. In case of the softmax function, the network's output $y_k(\mathbf{x})$ for the class k given input sample $\mathbf{x}$ thus becomes

$$y_k(\mathbf{x}) = \frac{\exp(a_k)}{\sum_{k'=1}^{K} \exp(a_{k'})}. \tag{2.10}$$

With $\mathbf{a} = (a_1, ..., a_K)^T$ the linear activities of the last layer's neurons (in machine learning literature often referred to as *logits*) and K the number of classes. Using one-hot coding for the target class

$$\mathbf{t} = (t_1, ..., t_k, .., t_K)^T = (0, ..., 1, ..., 0)^T, \tag{2.11}$$

the probabilistic model can be defined as a function of the neural network.

$$P(\mathbf{t} \mid \mathbf{x}, \boldsymbol{\theta}) = \prod_{k=1}^{K} y_k(\mathbf{x})^{t_k} \tag{2.12}$$

With $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$ the set of the network's trainable parameters. The model's likelihood is then given by

$$P(\mathbf{T} \mid \mathbf{X}, \boldsymbol{\theta}) = \prod_{n} P(\mathbf{t_n} \mid \mathbf{x_n}, \boldsymbol{\theta}). \tag{2.13}$$

Where the index n denotes nth training sample and target class (also known as label). The negative logarithm of the likelihood, called the negative *log-likelihood* defines the classifier's error function E_{ML}.

$$E_{\mathrm{ML}}(\boldsymbol{\theta}) = -\log(P(\mathbf{T} \mid \mathbf{X}, \boldsymbol{\theta})) = -\sum_{n} \log P(\mathbf{t_n} \mid \mathbf{x_n}, \boldsymbol{\theta}) \tag{2.14}$$

$$= -\sum_{n} \sum_{k=1}^{K} t_{n,k} \log(y_k(\mathbf{x_n})) \tag{2.15}$$

The resulting error function is referred to as the *cross entropy* (equation 2.15). In information theory, the cross entropy between two probability distributions is a measure for the discrepancy between the minimum encoding size (given by the true labels in this case) and an estimate of that distribution (the network's prediction). Cross-entropy decreases as the prediction gets more accurate and becomes zero if the prediction is perfect. As such, the cross-entropy can be used as a loss function to train a classifier. The network's parameters may be trained by minimizing its classification error w.r.t. $\boldsymbol{\theta}$. As the logarithm is a convex function and the error function is the *negative* log-likelihood, this is equivalent to a maximum likelihood estimate.

$$\arg\min_{\boldsymbol{\theta}} E_{\mathrm{ML}}(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} P(\mathbf{T} \mid \mathbf{X}, \boldsymbol{\theta}) \tag{2.16}$$

The gradient of the error function can be computed using the derivative chain rule. This method is known as *backpropagation* and forms the computational backbone of CNN architectures.

$$\frac{\partial E_{\mathrm{ML}}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta_{l-1}}} = \frac{\partial E_{\mathrm{ML}}(\boldsymbol{\theta})}{\partial h_{l-1}} \frac{\partial h_{l-1}}{\partial \boldsymbol{\theta_{l-1}}} = \frac{\partial E_{\mathrm{ML}}(\boldsymbol{\theta})}{\partial h_l} \frac{\partial h_l}{\partial h_{l-1}} \frac{\partial h_{l-1}}{\partial \boldsymbol{\theta_{l-1}}} \tag{2.17}$$

In deep learning frameworks this process is often fully automated. A computational graph is created based on the neural network's topology and all the operations performed on it. The gradient is then computed by applying the chain rule to the graph's nodes (cf. figure 2.8).

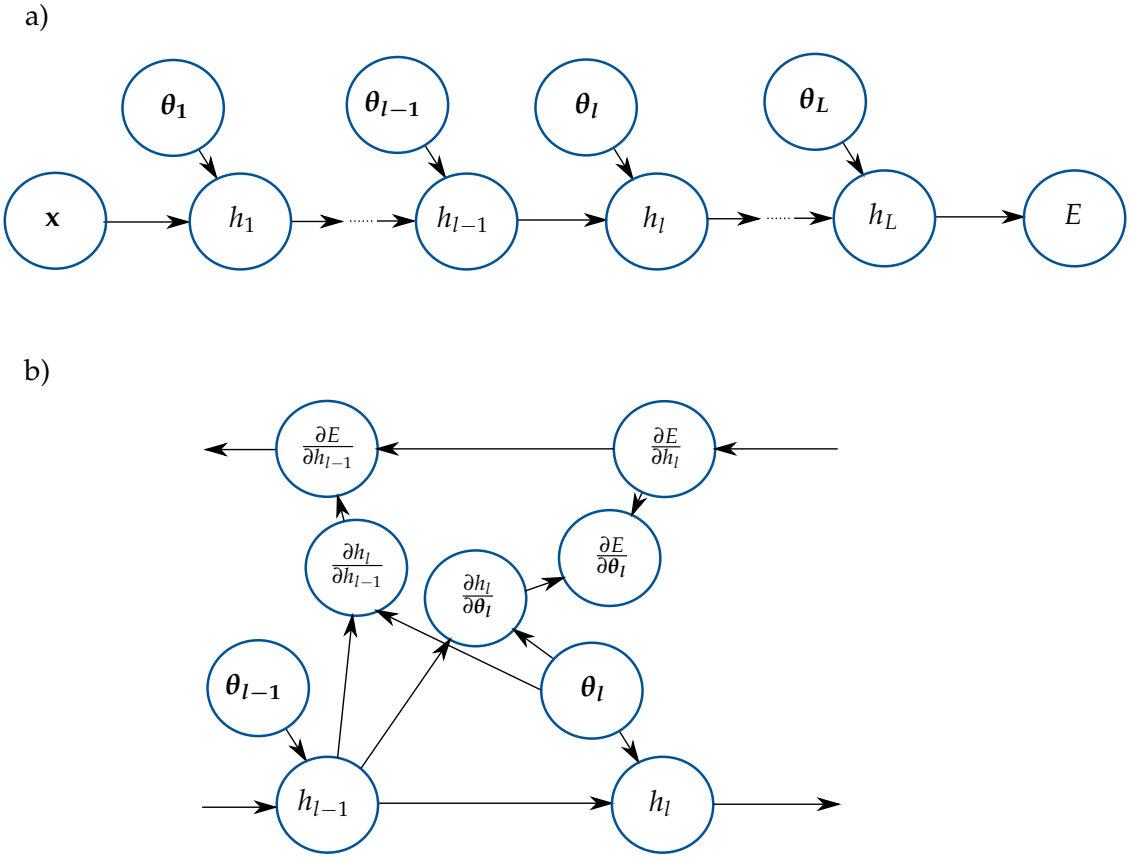


Figure 2.8: Computational graph for error propagation within a neural network. a) shows the dependency of the layers on both the lower layer's output as well as the parameters $\boldsymbol{\theta_l}$ during inference (also known as forward pass). In b) the computational nodes for the error gradient as needed for error backpropagation were added (known as the backward pass).

The gradient calculated based on the training samples is then used to update the network's parameters.

$$\boldsymbol{\theta}^{(s+1)} = \boldsymbol{\theta}^{(s)} - \eta \nabla E_{\mathrm{ML}}\left(\boldsymbol{\theta}^{(s)}\right) \tag{2.18}$$

This process is repeated several times to incrementally improve the performance of the network. Each iteration is referred to as an epoch (denoted by s). The number of epochs necessary to achieve the best classification performance depends on the network's architecture as well as on the training data and can range from a few to several hundred.

η is a hyperparameter known as the *learning rate*, that effectively describes the size of the step taken in direction of the gradient (cf. figure 2.9). In practice, the gradient is often calculated as an average based on a subset of randomly selected training samples.

$$\boldsymbol{\theta}^{(s+1)} = \boldsymbol{\theta}^{(s)} - \eta \frac{1}{M} \sum_{n=1}^{M} \nabla E_n\left(\boldsymbol{\theta}^{(s)}\right) \tag{2.19}$$

The $\frac{N}{M}$ sets of training samples are called (mini-) *batches* and are trained sequently until all samples have been seen, completing an epoch. Training algorithms based on this optimization scheme are known as *stochastic gradient descent* (SGD). It is at this point that vectorization libraries are used to leverage the power of modern GPUs by processing an entire mini batch in parallel.

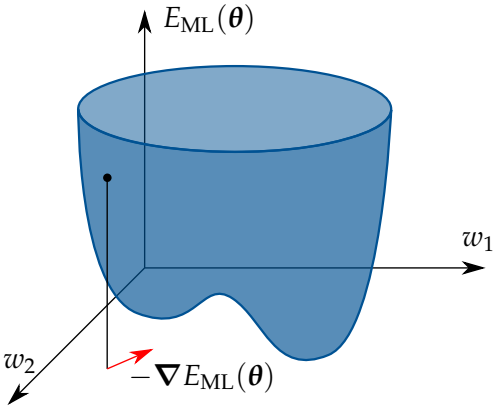


Figure 2.9: Illustration of the surface of a neural network's error function in parameter space. The function is non-convex and contains a local minimum. Note that the resulting minimum depends on both the learning rate and the initialization of the parameters. In this case, following the negative gradient (illustrated as a red arrow) using small steps will converge on a non-optimal local minimum.

State of the art CNN architectures expand upon the building blocks introduced in this chapter in various ways depending on the application. Often it is necessary to regularize the network's weights in order to prevent overfitting. This can be achieved by adding penalties for large weights to the loss function (known as weight decay) or randomly setting some weights to zero during training (so-called dropout) [29, 30, 31, 32]. This can be complemented with adaptive learning schemes in order to more reliably train neural networks, speed up convergence and increase accuracy. Adaptive extensions to SGD try to adjust learning hyperparameters like the learning rate based on performance on training data. It is for example quite common to decrease the learning rate after each training epoch that didn't increase accuracy or use such a scheme with independent learning rates for each weight [33, 34, 35, 36, 37, 38, 39]. To make the process of training a neural network even more robust and prevent the gradients from becoming either

too large or too small, it has proven useful to normalize a layer's input. In addition to stabilizing training, batch normalization can also speed up the convergence [40, 41, 42]. Finally, both the network's topology and the loss function may be chosen in such a way as to encourage the learning of internal representations to fit the task (e.g. learn the pose of a 3D object) [43, 44, 45, 46]. Even though CNNs have been applied successfully in many fields, there is not yet a theory to derive the topology and hyperparameters best suited for a given problem or predict the network's performance.

2.3 Neurorobotics and Spiking Neural Networks

Neurorobotics is an exciting new field that aims to study intelligence by way of an interdisciplinary approach combining computational neuroscience, robotics and artificial intelligence. Taking inspiration from biology is clearly a worthwhile effort as traditional engineering approaches like model-based control systems for robotics or physical symbol systems for artificial intelligence have so far failed to replicate the intelligent behavior of even the simplest of animals. The mathematical models of the nervous system used in neurorobotics are based on results from neuroscience, which is concerned with the study of intelligent biological systems on all levels, from the fast signal processing of single neurons to emergent long-term behavior like memory, perception and consciousness [47, 48, 49, 50, 51]. Artificial implementations or simulations of these models are then used to control a robotic body. This idea of an *embodied* intelligence controlled by brain inspired algorithms, which is in turn *embedded* in an environment it can perceive and interact with is central to neurorobotics. Brain, body and environment in a neurorobotic experiment form a closed perception-action loop where the brain receives percepts from the body's sensors, based on which it will produce signals to move the body, causing a change in the perception of the environment etc. Observing the interactions between the robot and its environment as well as its response to specific stimuli will in turn allow scientists to draw conclusions about the biological plausibility of the used brain model and further their understanding of how the brain works in conjunction with the body. A deeper understanding again leads to better models and more realistic simulations – this way neurorobotics inherently amplifies the transfer of knowledge and feedback between the involved disciplines [52].

Spiking neural networks (*SNN*s) are the method of choice for brain simulations in neurorobotics. They use more elaborate neuronal models than those discussed in the previous section and therefore are able to mimic the behavior of biological networks of neurons more plausibly (although they are still only approximations of biological neurons). Beyond biological plausibility, there are also potential engineering benefits to be gained by mimicking brains. Due to sparse encoding of information, the human brain maintains high versatility and throughput while only consuming about 20 W of power on average [53]. The ability to adapt to a dynamic environment and react in real-time under constrained resources is essential for the deployment of mobile intelligent agents like robots. In SNNs dynamics are added to the states of the neurons

and synapses (in CNNs these are the activities and weights respectively). Biological neurons are surrounded by cell membranes that act as insulators, which can be charged by other neurons in the network with short electrical impulses over the synapses. As the membrane is not a perfect insulator, this charge degrades over time. If a neuron model represents the incoming signals as discrete events (this is a reasonable simplification, as the shape is roughly the same for all of a neuron's impulses) it is referred to as *integrate-and-fire*. Models that additionally include a leakage (i.e. the neuron's membrane potential degrades over time) are by far the most popular and known as *leaky integrate-and-fire* (LIF) [54, 55]. A LIF neuron can be modelled as an RC circuit (cf. figure 2.10) that fires off an impulse, once the potential reaches a threshold.

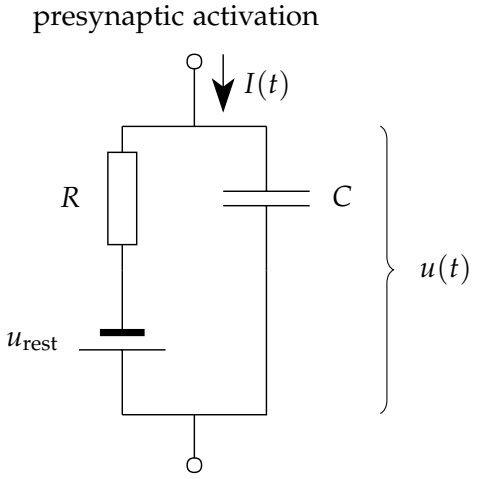


Figure 2.10: Electrical circuit of a leaky integrate and fire neuron. The unit saves incoming charges $I(t)$ called the presynaptic activation into the capacitor C, essentially summing them up or *integrating* them. Once the charge reaches a threshold value, the neuron discharges or *fires* (postsynaptic activation). The dissipation of ions through the cell's membrane is modelled by allowing the charge to *leak* through the resistor R.

Using Kirchhoff's current law, the incoming current $I(t)$ can be written as the sum of the resistive current and the current charging the capacitor.

$$I(t) = I_R + I_C = \frac{u(t) - u_{\text{rest}}}{R} + C\frac{\mathrm{d}u}{\mathrm{d}t} \tag{2.20}$$

Where in the second step Ohm's law was used for the linear resistor R as well as the definition of capacity $C = \frac{q}{u}$ and current $I_C = \frac{\mathrm{d}q}{\mathrm{d}t} = C\frac{\mathrm{d}u}{\mathrm{d}t}$ for the capacitor C.

Multiplying by the resistance R this can be written as

$$RI(t) = u(t) - u_{\text{rest}} + \underbrace{RC}_{\tau} \frac{\mathrm{d}u}{\mathrm{d}t} \tag{2.21}$$

$$\tau \frac{\mathrm{d}u}{\mathrm{d}t} = -(u(t) - u_{\text{rest}}) + RI(t) \tag{2.22}$$

$$\Updownarrow \tag{2.23}$$

$$\tau s U(s) - u\left(0^+\right) = -U(s) + \frac{u_{\text{rest}}}{s} + RI(s) \tag{2.24}$$

$$U(s) = u_{\text{rest}} \frac{\frac{1}{\tau}}{s\left(s + \frac{1}{\tau}\right)} + u\left(0^+\right) \frac{1}{s + \frac{1}{\tau}} + \frac{R}{\tau} I(s) \frac{1}{s + \frac{1}{\tau}} \tag{2.25}$$

$$\Updownarrow \tag{2.26}$$

$$u(t) = u_{\text{rest}} \left(1 - e^{-\frac{t}{\tau}}\right) + u\left(0^+\right) e^{-\frac{t}{\tau}} + \frac{R}{\tau} \int_0^\infty e^{-\frac{t'}{\tau}} I(t - t')\, \mathrm{d}t' \tag{2.27}$$

$$u(t) = u_{\text{rest}} + \frac{R}{\tau} \int_0^\infty e^{-\frac{t'}{\tau}} I(t - t')\, \mathrm{d}t' \,. \tag{2.28}$$

Where the Laplace transform was used to find the solution of the linear differential equation and the initial membrane potential was assumed to be at resting potential $u(0^+) = u_{\text{rest}}$. In the case of LIF models, the input current will consist of discrete events represented by delta distributions.

$$I_{\text{pre}}(t) = q \sum_i \delta\left(t - t_{i,\text{pre}}^{(f)}\right) \tag{2.29}$$

This is known as a *spike train*, with $t_{i,\text{pre}}^{(f)}$ the firing times of the presynaptic neurons and q the charge delivered with each spike. In addition to the presynaptic spike train, each neuron generates its own postsynaptic spike train. The firing times $t_{k,\text{post}}^{(f)}$ are determined by the threshold ϑ.

$$u(t_{k,\text{post}}^{(f)}) = \vartheta \tag{2.30}$$

Once the threshold is reached, the potential is reset to $u_{\text{reset}} < \vartheta$, firing the postsynaptic charge $C(\vartheta - u_{\text{reset}})$. Putting both the pre- and postsynaptic activations $I(t) = I_{\text{pre}}(t) + I_{\text{post}}(t)$ into equation 2.28 leads to the momentary voltage of a LIF neuron.

$$u(t) - u_{\text{rest}} = \frac{R}{\tau} \left(q \sum_i \exp\left(-\frac{t - t_{i,\text{pre}}^{(f)}}{\tau} \right) - C(\vartheta - u_{\text{rest}}) \sum_k \exp\left(-\frac{t - t_{k,\text{post}}^{(f)}}{\tau} \right) \right) \tag{2.31}$$

As the impulses are extremely sparse in time, LIF neurons generally allow for much more efficient computation compared to the neurons found in CNNs. Alternatively, this can be thought of as higher information density: the frequency of rate-based codes corresponds to averaging a temporal window and therefore a loss of information over

the pulse-based code. While it is possible to use a code based on the firing rate, the full potential of spiking neurons is only levelled, when information is also encoded in the timing of the firing [56, 57, 58]. Networks based on the dynamics of spiking neurons are generally known as third generation neural networks, while the first generation of neurons or neural networks refers to networks consisting of linear neurons (called *perceptrons*), and the second generation is understood to refer to architectures using artificial neurons with nonlinear activation functions, such as CNNs as discussed in section 2.2. A single spiking neuron, encoding information in the rate and temporal patterns of its spike train, can replace hundreds of second-generation neurons [59, 60, 61]. The synapses interconnecting the spiking neurons in a SNN are scaled by learnable weights, the same as CNNs. Many of the ideas developed for CNNs, like perceptive fields, different types of layers, batch normalization etc., can be directly applied to SNNs. There are however two major differences:

1. Input data must be encoded into action potential spikes.
2. The synaptic weights must be learned based on the dynamic behavior of the spiking neurons.

In computer vision experiments it is quite common to use rate based stochastic encoding schemes. The spike trains are generated by sampling from a stochastic distribution (e.g. Poisson) with firing rates proportional to the intensity of the pixels [62]. SNNs may be trained using biologically plausible algorithms. Learning rules which strengthen synapses based on the timing of spikes are referred to as *Hebbian* learning and often colloquially summarized by the phrase "Cells that fire together, wire together" [63]. Mathematically, the change in synaptic weight Δw_{ij} between two neurons i and j may be expressed as

$$\Delta w_{ij} \propto v_i v_j. \tag{2.32}$$

with v_i the activity of neuron i. Because there is no labelled data, or any training signal involved, Hebbian learning rules are inherently unsupervised. Methods that consider the precise timing and order of pre- and postsynaptic spikes are known as *spike-timing-dependent plasticity* (STDP). In STDP the firing of a postsynaptic spike increases the strength of presynaptic weights that fired shortly before. This is referred to as *long term potentiation* (LTP). The reverse order leads to a decrease in synaptic strength and is conversely known as *long term depression* (LTD). The change in weight Δw_i for a synapse from a presynaptic neuron i according to STDP can also be expressed mathematically.

$$\Delta w_i = \sum_{f=1}^{N} \sum_{n=1}^{N} F\left(t_n^{(j)} - t_f^{(i)}\right) \tag{2.33}$$

Where $t_n^{(j)}$ and $t_f^{(i)}$ denote the times of the firing of spikes by the postsynaptic neuron j and the presynaptic neuron i respectively. The STDP function F or *learning window*

largely determines the specific behavior of the STDP model. A popular choice, that also corresponds well with experimental findings [64] is given by

$$F(\Delta t) = \begin{cases} A_+ \exp\left(\frac{\Delta t}{\tau_+}\right) & \text{for} \quad \Delta t < 0 \\ -A_- \exp\left(-\frac{\Delta t}{\tau_-}\right) & \text{for} \quad \Delta t \geq 0. \end{cases} \tag{2.34}$$

With $A_{+/-}$ linear coefficients and $\tau_{+/-}$ a kind of decay defining the size of the window (cf. figure 2.11).

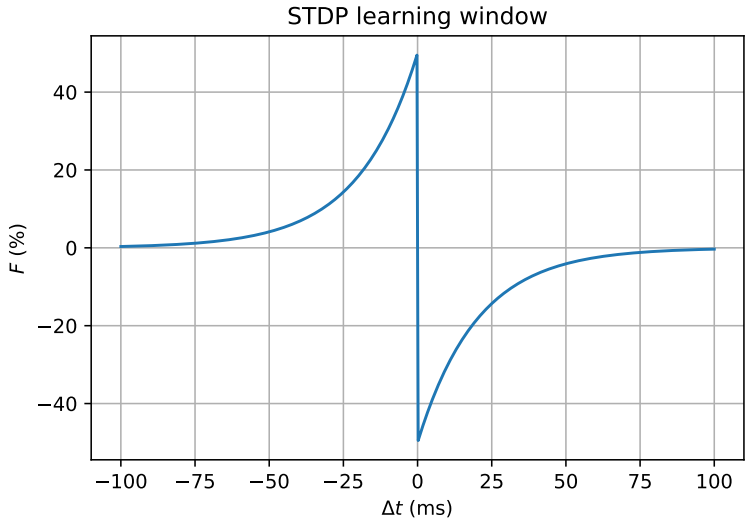


Figure 2.11: Typical STDP window function. The change in conductance for a synapse due to a single pre- and postsynaptic spike pair in STDP is proportional to $F(\Delta t)$ with Δt the time of the presynaptic spike minus the time of the postsynaptic spike. In this figure, F was expressed as a percentage. Note that in this case, action potentials that are more than 100 ms apart have virtually no influence, while rapid spikes induce up to 50 % LTP/LTD.

Gradient descent is generally regarded as not a realistic option for how the brain learns. This is because it requires a teaching signal in the form of labelled data and calculates a gradient specific to each neuron based on that signal. While research into biologically plausible learning algorithms is ongoing [65, 66], they are generally outperformed by engineering methods such as gradient descent in tasks that allow for supervised training (e.g. classification and regression) [62]. For the supervised training of spiking feed-forward networks using backpropagation, there are generally two established approaches.

1. Rate-based training of a second-generation network (e.g. a CNN) with established methods followed by conversion into a SNN for inference.

2. Direct optimization of an objective function using an approximation of spatio-temporal gradient descent similar to backpropagation in CNNs.

However, training a network of spiking neurons discriminatively using gradient descent with backpropagation is difficult, because the LIF neuron's activity is non-differentiable at the time of spikes (cf. figure 2.12).

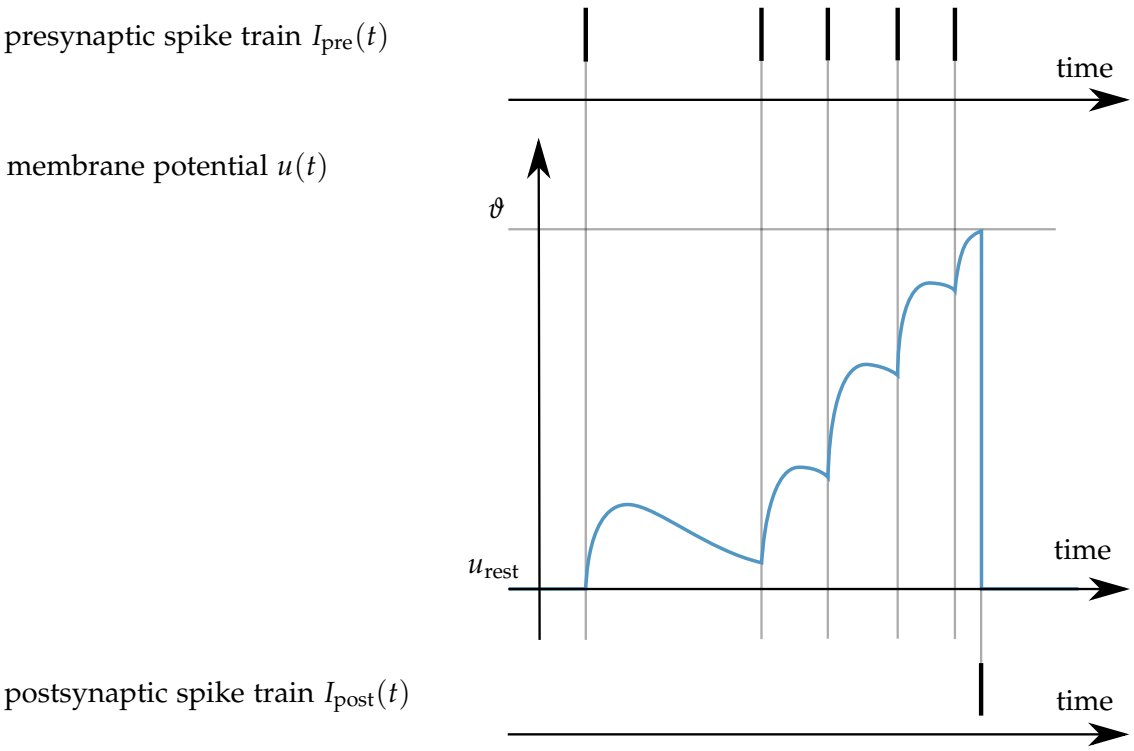


Figure 2.12: Presynaptic spike train, membrane potential and postsynaptic spike train of a LIF neuron. The three plots are synchronous in time. A presynaptic spike adds a charge to the membrane potential that decays over time. Once the membrane potential reaches the threshold ϑ, the neuron discharges, generating a postsynaptic spike and resetting to resting potential u_{rest}. Note the discontinuity whenever pre- or postsynaptic neurons fire.

Recently, deep spiking convolutional networks have been trained successfully using backpropagation by treating the discontinuities at spike times as noise i.e. approximating a smooth signal [67, 68]. While all approaches to SNNs currently do not perform quite as well as CNNs on computer vision tasks, they are inherently computationally less expensive and could at the very least lead to power efficient hardware implementation in the form of neuromorphic systems (i.e. hardware implementations of neural circuitry in silico that allow for fast and direct computation on neural networks.) [69].

2.4 Limitations of Artificial Neural Networks

CNNs use translated replicas of learned feature detectors. The rationale behind this is that knowledge about good features in one part of the image may be useful in other regions as well. While this has proved quite a powerful semantic and grants CNNs a measure of invariance to translations, they do not generalize well to other transformations. This requires the networks to be trained using a large number of training images, covering all desired variations in viewpoint, scale, lighting, color etc. It is common to artificially extend the training set by applying image transformations. This process is known as *data augmentation*. [70, 71]. Even so, creating datasets for neural networks remains quite time consuming and expensive. Even if labelled data is available, training is computationally expensive and can take several days [72, 73]. The same can be said for inference, which often requires great computational resources and still suffers from relatively high latency [74]. These drawbacks along with the fact that high computational costs come with large energy requirements, make real time and mobile applications under constrained resources difficult. Another shortcoming of CNNs is due to their use of pooling layers. The bottle-neck architecture of CNNs allows them to use a growing number of feature maps with large receptive fields, representing high-level features and shapes. With every pooling operation however, information on the location of features is lost. This means that CNNs cannot learn the precise spatial relations between a whole and its parts. For tasks like facial recognition however, part-whole relations play an important role [75]. SNNs on the other hand try to avoid the limitations imposed by the high computational costs and energy requirements of CNNs by mimicking the sparse encoding of information in spikes as observed in brains. SNNs running on neuromorphic hardware are extremely energy efficient and are able to operate in real time by design, independent of the size and topology of the network. Still, the computation on current neuromorphic circuity is only a qualitative approximation of digitally simulated spiking neurons and even those do currently not approach state of the art CNN accuracy in computer vision tasks [76, 62]. Biological systems, especially the visual cortex of the primate brain, are not subject to these limitations [77, 78]. It therefore stands to reason that all of these challenges can be met and that findings from biology and neuroscience may hold a clue as to how to engineer solutions.

Recently an ANN architecture consisting of groups of neurons called *capsules* has been proposed as a possibly biologically plausible alternative for CNNs in computer vision systems. Capsule networks can be thought of as a visual attention mechanism, that encodes poses between features. Additionally, the activation functions and pooling operations found in CNNs are replaced by a sophisticated routing organism, that retains positional information. In theory this grants capsule networks a measure of viewpoint invariance as well as access to precise part-whole relations throughout the network. As capsule networks are still rate-coded and trained by supervised SGD with backpropagation, they fall somewhere in-between CNNs and SNNs in regard to biological plausibility but are possibly more expressive than CNNs. The theory behind capsules is explored in detail in the next chapter.

3 Capsule Network Architectures

A capsule is conceived as a group of neurons whose outputs encode different properties of an entity such as an object or part of an object in an image. The properties could represent instantiation parameters on the object's appearance manifold like the precise pose, lighting and deformation or the probability that the entity is present. Layers within a capsule network consist of several capsules. Routing between the layers is determined by the votes from the capsules in the lower layer. Capsules can make a guess as to what they expect the higher layer's capsules instantiation parameters to be and cast a vote. Votes are calculated by applying discriminatively learned transformations to a capsule's neuron's outputs. The routing algorithm will then determine, based on the lower capsules' votes, which higher layer capsules to activate. The goal of capsules is to achieve invariance in the presence probability of the object they represent regarding the object's instantiation parameters. A specific capsule architecture is largely defined by the nature of its routing algorithm and the dimension of its output as well as the order of the tensors used to compute the votes.

3.1 Transforming Auto-encoders

The first publication on capsules came from Hinton et al. in the form of transforming auto-encoders in 2011 [79]. These capsules encode the instantiation parameters of their internal representation together with the probability that the learned entity is present into a small output vector. Each capsule would learn to recognize a single visual entity over a limited subdomain of its appearance manifold. Ideally, the probability would be locally invariant as long as the instantiation parameters remain within the trained submanifold. This is in contrast to the instantiation parameters, which are equivariant. Meaning as the viewing conditions change, the parameters encoding the appearance change by an equal amount. The transforming auto-encoders introduced by Hinton et al. in 2011 were already motivated by a desire to retain precise part-whole relationships, which are lost by the pooling operations in convolutional neural networks (cf. section 2.4). If a capsule can be trained to encode the pose of the visual entity it represents in its output vector, it is a simple matter to test whether two capsules A and B agree about a higher-level capsule C. To see this, suppose that the matrix $\mathbf{T}_{A/B}$ is the pose between the canonical entity as learned by the capsule A or B respectively and the current instantiation. Multiplying this with the part-whole transformation matrix $\mathbf{T}_{A/B\to C}$ will result in a prediction for the pose of capsule C's entity. Now, if

$$\mathbf{T}_A\mathbf{T}_{A\to C} \approx \mathbf{T}_B\mathbf{T}_{B\to C}, \tag{3.1}$$

then the capsules are a close match and activate capsule C. Hinton et al. argue that this naturally leads to precise part-whole relationships. If the entities of capsules A and B e.g. correspond to a mouth and a nose respectively, then they have to be in right spatial relationship to form a face if they activate the higher capsule C representing the whole face. The pose of capsule C can then be determined by the average of the lower capsules' votes.

$$\mathbf{T_C} = \sum_{i \in \{A,B\}} \mathbf{T}_i \mathbf{T}_{i \to C} \tag{3.2}$$

This means that except for the first layer of capsules, only the viewpoint invariant part-whole transformations have to be learned, while the entity poses are encoded in the neural activities at run time. The rest of the paper focuses on how to extract the poses from the pixel intensities for the first layer. This is done by training capsules on transformed image pairs and providing the capsules with non-visual access to the transformation. Hinton et al. justified this by pointing out that the human visual system is also provided with information about the eye-movement while saccades correspond to a translational transformation of the retinal image. These first capsule networks actually consist of only one layer of capsules and do not include the routing mechanism of later implementations. Instead, they include two internal layers of logistic neurons, termed recognition and generation units respectively (cf. figure 3.1). The recognition units are trained to extract pose parameters and the probability that the visual entity is present directly from the provided image.

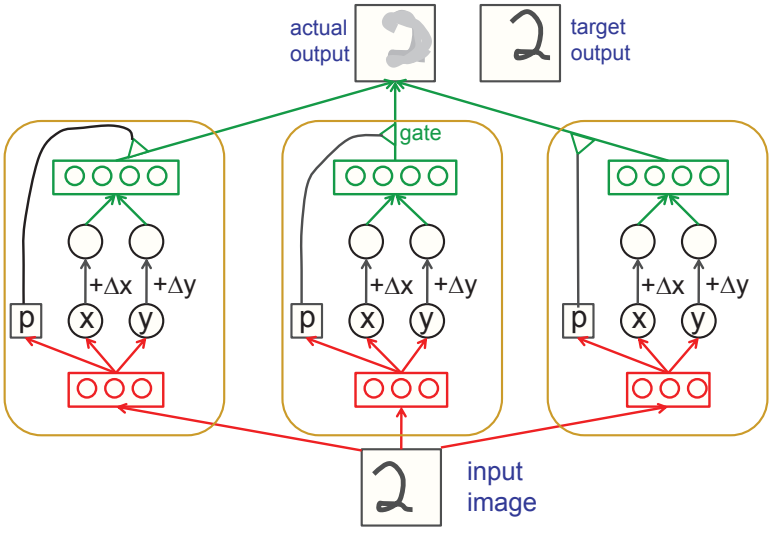


Figure 3.1: Capsules in a transforming auto-encoder [79]. The network consists of a single layer of capsules. Each capsule includes hidden recognition (in red) and generation units (in green). During training transformation parameters (in this case the translations Δx and Δy) as well as the transformed image are provided to the network. A capsule's output vector is further multiplied with its probability, thus muting capsules with a low probability.

A capsule's input can be constrained to a fixed receptive field with the capsules aligned in a grid across the image. The capsules then apply a transformation to the pose and provide the generation units with this information. The generation units on the other hand are trained to output the transformed part of the input image centered at the capsule's receptive field. All the neurons within a capsule are trained discriminatively using gradient descent with backpropagation. The training signal is provided by a form of reconstruction loss where the capsules have access to the transformation and the input and reconstruction target are the original and transformed images respectively. Transforming auto-encoders are able to learn poses in the form of $3x3$ matrices and can be used to successfully apply affine transformations to input images once trained. However, they are limited to one layer of capsules and cannot be scaled effectively, as the capsules are fixed to a single position, compared to the shared weights of moving filters in convolutional layers. Also, they are only able to learn properties, that can be controlled and provided in explicit non-visual form to the network.

3.2 Dynamic Routing Between Capsules

In 2017 Sabour et al. extended upon the transforming auto-encoders by suggesting a dynamic routing algorithm for capsules [80]. While the benefit of dynamic routing between capsule layers in regard to part-whole relationships was already elaborated upon by Hinton et al., Sabour et al. also motivated the mechanism biologically. They pointed out that human vision ignores irrelevant details by focusing attention on salient features in a sequence of fixations and only ever processes a fraction of the optic array at the highest resolution. They assumed that with a single fixation, groups of activated neurons build a tree structure within the fixed multi-layer visual system. The paper suggests implementing the nodes of this image parse tree as active capsules and have each capsule choose their parent node based on the routing algorithm. As with the capsules of transforming auto encoders, the capsule's neurons should learn to represent various instantiation parameters such as pose, deformation, velocity, albedo, hue, texture, etc. However, the probability of the instance's presence is encoded in the length of a vector of instantiation parameters. A nonlinearity is applied to the output vector in order to keep the length from exceeding 1, while keeping the orientation unchanged.

$$\mathbf{u}_j = \frac{\left\|\mathbf{s}_j\right\|^2}{1 + \left\|\mathbf{s}_j\right\|^2} \frac{\mathbf{s}_j}{\left\|\mathbf{s}_j\right\|} \tag{3.3}$$

With $\mathbf{u}_j$ the output vector of capsule j and $\mathbf{s}_j$ the sum of its inputs. This *squashing* function will reduce short vectors to almost length zero and squash long vectors to a length slightly below 1. For every capsule j that is not a first layer capsule, the input $\mathbf{s}_j$ is computed from the votes of the lower level capsules.

$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i} \tag{3.4}$$

Where $\hat{\mathbf{u}}_{j|i}$ is the lower capsule i's guess for the higher capsule j's instantiation vector. Votes are calculated by applying a transformation matrix $\mathbf{W}_{ij}$ on a capsule's output vector, as was suggested by Hinton et al. for transforming auto-encoders.

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i \tag{3.5}$$

In this scheme, so far only the part-whole transformation matrices $\mathbf{W}_{ij}$ have to be learned. The coupling coefficients c_{ij} for a lower capsule i sum to 1. They describe the influence a vote from capsule i has over capsule j and are iteratively fine-tuned by the routing algorithm during the network's forward pass. For the first iteration, the coupling coefficients are computed as a softmax function over the log prior coupling logits b_{ij}.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})} \tag{3.6}$$

The b_{ij} can be trained discriminatively along with the other parameters. But it was found that the priors have little influence over the routing algorithm. Using a vector as capsule output allows the use of a powerful dynamic routing-by-agreement algorithm. The agreement between capsule i and each possible parent j is calculated using the scalar product of the lower capsule's vote and the higher capsule's current output.

$$a_{ij} = \mathbf{u}_j^T \hat{\mathbf{u}}_{j|i} \tag{3.7}$$

If the scalar product is large, it induces top-down feedback, increasing the coupling coefficient between the two capsules by adding the agreement to the coupling logit b_{ij} (cf. algorithm 1).

Algorithm 1 Dynamic routing-by-agreement. Agreement between a lower capsule's vote and the higher capsule's current output is measured by their scalar product. The current output is a weighted average of the lower capsules' votes and updated each iteration.

procedure ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
  for all $i \in \Omega_l$ **do** ▷ Ω_l is the set of all capsules in layer l
    for all $j \in \Omega_{l+1}$ **do**
      $b_{ij} \leftarrow 0$
  for r iterations **do**
    for all $i \in \Omega_l$ **do**
      $\mathbf{c}_i \leftarrow \text{softmax}(\mathbf{b}_i)$ ▷ computes equation 3.6
    for all $j \in \Omega_{l+1}$ **do**
      $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
      $\mathbf{u}_j \leftarrow \text{squash}(\mathbf{s}_j)$ ▷ computes equation 3.3
    for all $i \in \Omega_l$ **do**
      for all $j \in \Omega_{l+1}$ **do**
        $b_{ij} \leftarrow b_{ij} + \mathbf{u}_j^T \hat{\mathbf{u}}_{j|i}$
  return $\mathbf{u}_j$

As the lower capsule's coupling coefficients are computed as a softmax of the logits, this will simultaneously decrease all the other couplings. Seeing how the internal representation, the votes and the agreement are all vectors, this type of architecture will be referred to as *vector capsules* for the remainder of this thesis. Sabour et al. further improved upon transforming auto-encoders by developing convolutional capsule layers. This allows them to use learned features all across the image without losing whole-part relationships. They achieve this by having each capsule of a convolutional layer put out a local grid of vectors for each type of capsule in the higher level, where every position on the grid and type of layer is assigned a different part-whole transformation matrix. A capsule's type within a convolutional layer is analogous to the use of different kernels or filters in CNNs (cf. section 2.2). The resulting map of capsule activities i.e. the lengths of the output vectors, can be thought of as a feature map. While this is not strictly convolution, the net effect is the same. Capsules in a convolutional layer possess a receptive field that gets wider the higher up the layer is in the network's topology. For low level capsules, positional information is encoded in the capsule's position within the feature map. The higher up a convolutional capsule is, the smaller the feature map becomes and information on position gets encoded more and more in the instantiation parameter vector. The primary capsules i.e. the first capsule layer computes its input vectors from the reshaped output of a standard convolutional layer. For the next capsule layers, the routing algorithm is applied. This is yet another improvement over transforming auto-encoders, as the network can be trained on images only without needing access to the applied transformations. The final capsule layer is fully connected and includes one capsule per class (cf. figure 3.2).

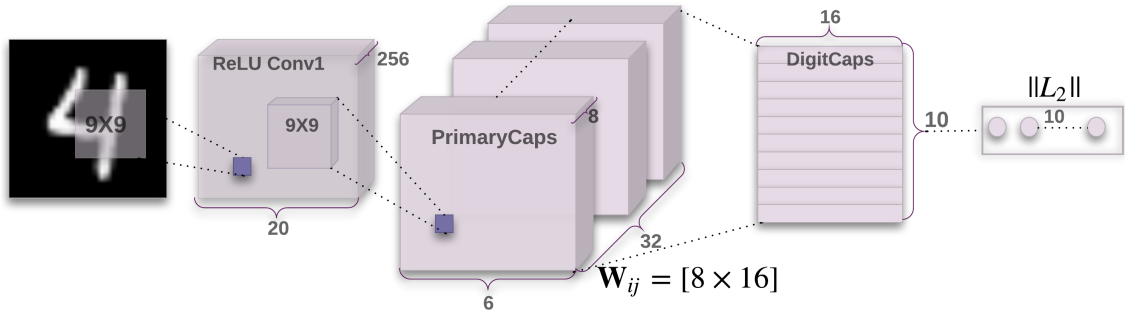


Figure 3.2: Vector capsule network with 3 layers [80]. Conv1 is a standard convolutional layer consisting of 256 9×9 kernels that learn basic features from pixel intensities, that are converted to 8D vector representations by the primary capsule layer. The primary capsule layer is the only convolutional capsule layer in the network and has 32 types of capsules of which each has a receptive field of the size 9×9 and a stride of 2. The class capsule layer (here referred to as DigitCaps) uses 16D vectors and 10 fully connected capsules. Note that the routing algorithm is applied only between the primary and the class capsule layer.

Margin loss is used to compute the error signal based on the class capsules' output vector length. For each class a loss L_k is introduced so as to allow multiple object instances of different classes. This is done to make it possible to train the network to segment highly overlapping objects.

$$L_k = T_k \max\left(0, m^+ - \|\mathbf{v}_k\|\right)^2 + \lambda(1 - T_k)\max\left(0, \|\mathbf{v}_k\| - m^-\right)^2 \tag{3.8}$$

Where $\mathbf{T} = (0, ..., 1, ...0)^T$ is a one-hot encoded vector of the true class labels and $\lambda = 0.5$ is used to down-weigh the loss for absent classes so that the vectors don't become too short during initial training. In addition, Sabour et al. used a reconstruction loss not unlike with transforming auto-encoders in order to encourage the class capsules to encode useful instantiation parameters for each class. This is done by masking out all but the correct class capsule during training and feeding its output vector to a 3-layer decoder (cf. figure 3.3). The reconstruction loss is calculated as the Euclidean distance between the decoder's output and the input image and scaled down so as not to dominate the margin loss.

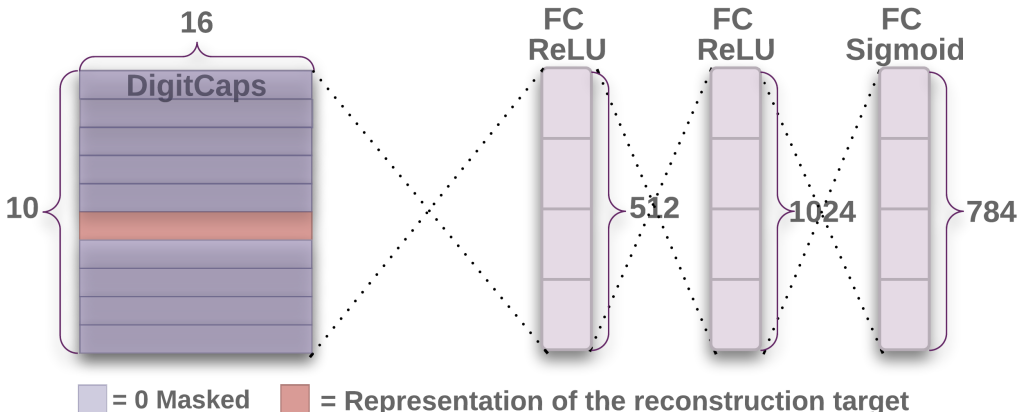


Figure 3.3: Decoder on top of a class vector capsule layer [80]. The $16D$ output vector of the correct class capsule is used as input for a 3-layer fully connected decoder while the output of the other capsules is masked. By reconstructing the input image based on the output vector the class capsules are encouraged to encode useful instantiation parameters.

After the network has been trained, the decoder can be used to inspect the instantiation parameters learned by the class capsules. To do this, for a single input image, each dimension of the correct class capsule's output vector is varied and the effect on the reconstructed image is observed.

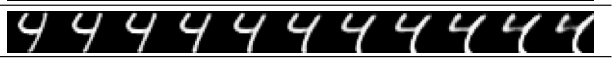
Scale and thickness	
Localized part	
Stroke thickness	
Localized skew	
Width and translation	
Localized part	

Figure 3.4: Instantiation parameters learned by a vector capsule network on MNIST [80]. The decoder of a trained vector capsule network is used to reconstruct an input image. By perturbing each dimension of the class capsule, sensible instantiation parameters are found to be encoded within the capsule. In this example the shown dimensions where incremented by 0.05 on the interval $[-0.25, 0.25]$ around the base values. Note the continuous distribution of the parameters. This is similar to the latent space of variational auto-encoders [81]

Sabour et al. were able to demonstrate that a shallow network with only two capsule layers can already achieve good results in object recognition tasks and that the class capsules encode sensible instantiation parameters like the thickness of strokes in case of the MNIST[1] dataset (cf. figure 3.4). The biological plausibility of vector capsules was further demonstrated by their ability to effectively explain-away regions of overlap so as to successfully segment highly overlapping objects.

3.3 Matrix Capsules with EM Routing

The most recent update on capsules came from Hinton et al. in 2018. They pointed out several shortcomings of vector capsules, that they intended to overcome, including [82]:

1. While routing based on agreement is motivated by an assumed parse tree like structure of grouped neurons in the human visual system [83], the choice to encode the probability that an object is present in the length of a capsule's output vector requires an unprincipled nonlinearity (cf. equation 3.3) that precludes the dynamic routing-by-agreement algorithm from minimizing a sensible objective function. This in turn reduces the overall biological plausibility of the vector capsules introduced by Sabour et al. in 2017.

[1] Available at `http://yann.lecun.com/exdb/mnist/`

2. Dynamic routing-by-agreement has difficulty differentiating between good and very good clusters of votes as the scalar product used to determine the agreement effectively computes the cosine of two angles – a function that saturates around 1.

3. Using an n-dimensional vector as the internal representation rather than a matrix with n elements requires a part-whole transformation matrix with n^2 weights compared to n for matrix-representation.

Compared to vector capsules, the internal representation of matrix capsules is separated into a 4×4 pose matrix $\boldsymbol{\mu}$ and the object presence probability a. This means that the part-whole transformation matrices between all capsule layers are also of size 4×4. The vote $\mathbf{V}_{ij}$ between a lower capsule i and a higher capsule j is computed as a matrix multiply.

$$\mathbf{V}_{ij} = \boldsymbol{\mu}_i \mathbf{W}_{ij} \tag{3.9}$$

Primary capsules again receive their activity and instantiation parameters from a standard convolutional layer (cf. section 3.2). The activities of all other consecutive capsule layers are determined by the routing algorithm. Routing between matrix-capsules is based on an Expectation-Maximization (EM) algorithm fitting a gaussian mixture model, where the higher capsules are interpreted as gaussian distributions while the vectorized pose matrices of the lower capsules constitute the data points. EM algorithms are commonly used in statistics and machine learning in order to estimate latent parameters of distributions. They alternate between iteratively computing a log-likelihood function using the current estimation of the parameters in the expectation step and computing parameters maximizing the same log-likelihood function in the following maximization step [84]. The use of an EM algorithm as a routing mechanism can be thought of as a clustering of lower capsules' votes – each cluster activating a higher capsule. A fixed cost of $-\beta_u$ must be paid per data point assigned to a capsule if that capsule is not activated. Formally, this cost is the negative log probability density of seeing a data point under an improper uniform prior. On the other hand, if a capsule gets activated, a fixed cost $-\beta_a$ must the paid for coding its mean and variance as well as additional costs for seeing each assigned data point under the distribution given by the activated capsule. Technically, the active capsule's prediction of seeing a data point should be computed in the lower capsule's pose space using the inverse of the part-whole transformation matrix. Hinton et al. use the probability of the data point's vote under the assigned capsule as a computationally low-cost estimate instead. These costs are motivated by a concept from information theory called the minimum description length principle. It states that the best hypothesis for a given dataset is the one with the best coding i.e. the highest compression of the data [85]. The difference in cost (penalizing lengthy description) between activating a capsule and not activating it is used to calculate the higher capsules' activation probabilities.

The probability of seeing data point i under active capsule j's gaussian with mean $\boldsymbol{\mu}_j$ and axis aligned covariance matrix can be computed for each component h of i's vote $\mathbf{V}_{ij}$.

$$P_{i|j}^h = \frac{1}{\sqrt{2\pi\left(\sigma_j^h\right)^2}} \exp\left(-\frac{\left(V_{ij}^h - \mu_j^h\right)^2}{2\left(\sigma_j^h\right)^2}\right) \tag{3.10}$$

And its negative log-likelihood can be written as

$$-\ln\left(P_{i|j}^h\right) = \frac{ln(2\pi)}{2} + \ln\left(\sigma_j^h\right) + \frac{\left(V_{ij}^h - \mu_j^h\right)^2}{2\left(\sigma_j^h\right)^2}. \tag{3.11}$$

The total cost of explaining the assigned data points of an activated capsule j per dimension h can then be computed as the sum over all the assigned lower capsules' negative log-likelihoods weighed by the coupling coefficients or assignment probabilities R_{ij}.

$$cost_j^h = -\sum_i R_{ij} \ln\left(P_{i|j}^h\right) \tag{3.12}$$

$$= \left(\frac{ln(2\pi)}{2} + \ln\left(\sigma_j^h\right)\right) \sum_i R_{ij} + \sum_i \frac{R_{ij}\left(V_{ij}^h - \mu_j^h\right)^2}{2\left(\sigma_j^h\right)^2} \tag{3.13}$$

$$= \left(\frac{\ln(2\pi)}{2} + \ln\left(\sigma_j^h\right) + \frac{1}{2}\right) \sum_i R_{ij} \tag{3.14}$$

Where in the last step the definition of covariance $\left(\sigma_j\right)^2$ was used with $p_{i\in\Omega_l}$ the prior probability density over the lower capsules' assignment to higher capsule j.

$$\left(\sigma_j^h\right)^2 = \sum_i p_i \left(V_{ij}^h - \mu_j^h\right)^2 = \sum_i \frac{R_{ij}\left(V_{ij}^h - \mu_j^h\right)^2}{\sum_i R_{ij}} \tag{3.15}$$

Finally the activation probability of the higher capsule j can be computed as the logistic function applied to the difference of the activation costs.

$$a_j = \text{logistic}\left(\lambda\left(\beta_a - \beta_u \sum_i R_{ij} - \sum_h cost_j^h\right)\right) \tag{3.16}$$

The fixed costs (over the course of a forward pass) β_a and β_u can be learned discriminatively. Hinton et al. suggested using a fixed schedule for the inverse temperature parameter λ. λ is called an inverse temperature based on the fact that the logistic function has the same "S" shape as the Fermi-Dirac (FD) distribution used in quantum

statistics to describe the distribution of energy states in a fermionic ensemble. In a FD distribution, the inverse of the absolute temperature defines the steepness of the curve. Here, the use of λ in the logistic function is analogous to the inverse of the temperature. Using an EM algorithm for routing and the logistic function to compute the activation probabilities can be motivated by interpreting the network as a physical system with the description cost as the system's energy. In thermodynamics a quantity called *free energy* is defined as the difference between the expected energy and the entropy of a system. Minimizing free energy is then achieved by finding the optimal trade-off between minimizing the expected energy while maximizing the entropy. In the context of this analogy, the logistic function then computes the distribution over the difference in energies for activating a capsule, thereby minimizing the system's free energy. In the EM algorithm's M-step the expected energy (i.e. the part of the activation cost paid for explaining the data points) is minimized by estimating the distribution's mean and variance using the fixed assignment probabilities (thereby leaving the entropy unchanged). During the following E-step the assignment probabilities are computed using the softmax function, again minimizing the free energy. This can be seen by treating the logits as negative energies E_i. The softmax function is proportional to $\exp(-E_i)$, which in physics is known as the Boltzmann distribution. The Boltzmann distribution can be shown to minimize free energy. The objective function minimized by the EM routing algorithm (cf. algorithm 2) is then the sum of the activation cost, negative entropy of the activations and assignment probabilities during the E-step and the expected energy.

$$L(\mathbf{a}, \mathbf{R}) = \sum_{j \in \Omega_{l+1}} a_j(-\beta_a) + a_j \ln\big(a_j\big) + (1 - a_j) \ln\big(1 - a_j\big) \tag{3.17}$$

$$+ \beta_u \sum_{i \in \Omega_l} R_{ij} + \sum_{i \in \Omega_l} a_i R_{ij} \ln\big(R_{ij}\big) \tag{3.18}$$

$$+ \sum_h cost_j^h \tag{3.19}$$

Overall, all three steps can be shown to minimize the same free energy, greatly supporting the biological plausibility of the proposed system [86]. Another argument can be made for the plausibility of using a softmax in the routing procedure. Computing the assignment probabilities as a softmax over the higher-level capsules effectively means that the top-down feedback from very active capsules can inhibit other nearby capsules whose receptive fields overlap. This is not unlike the competition of neurons in a winner-take-all circuit, designed to model decision-making and attention mechanisms in the brain [87]. Matrix capsule networks include convolutional capsule layers. This is implemented in the EM routing algorithm, simply by restricting assignments between lower and higher-level capsules to the higher-level capsule's receptive field. The learned whole-part transformation matrices for a capsule type are then shared across all positions (like weight-sharing in CNNs). This makes it possible to use different kernel-sizes and strides, allowing for more complex and fine-tuned network topologies.

Algorithm 2 Expectation-Maximization routing for matrix capsules. The routing procedure acts between two capsule layers, where the lower level's activities and pose matrices have already been determined. The algorithm can be interpreted as a form of high-dimensional coincidence filtering. A higher-level capsule is activated by the agreement of the votes from a lower level cluster of capsules transformed into its pose space.

procedure EM-ROUTING($\mathbf{a}, \mathbf{V}, r, l$)
  for all $i \in \Omega_l$ **do**
    for all $j \in \Omega_{l+1}$ **do**
      $R_{ij} \leftarrow \frac{1}{|\Omega_{l+1}|}$
  for r iterations **do**
    for all $j \in \Omega_{l+1}$ **do**
      M-STEP($\mathbf{a}.\mathbf{R}, \mathbf{V}, j$)
    for all $i \in \Omega_l$ **do**
      E-STEP($\boldsymbol{\mu}, \boldsymbol{\sigma}, \mathbf{a}, \mathbf{V}, i$)
  return $\mathbf{a}, \boldsymbol{\mu}$
procedure M-STEP($\mathbf{a}, \mathbf{R}, \mathbf{V}, j$)
  for all $i \in \Omega_l$ **do**
    $R_{ij} \leftarrow R_{ij} a_i$
  for $h \leftarrow 1$ **to** H **do**
    $\mu_j^h \leftarrow \frac{\sum_i R_{ij} V_{ij} h}{\sum_i R_{ij}}$
    $\left(\sigma_j^h\right)^2 \leftarrow \frac{\sum_i R_{ij} \left(V_{ij}^h - \mu_j^h\right)^2}{\sum_i R_{ij}}$
    $cost^h \leftarrow \left(\beta_u + \log\left(\sigma_j^h\right)\right) \sum_i R_{ij}$
  $a_j \leftarrow \text{logistic}\left(\lambda\left(\beta_a - \sum_h cost^h\right)\right)$
  return $\boldsymbol{\mu}_j, \boldsymbol{\sigma}_j$
procedure E-STEP($\boldsymbol{\mu}, \boldsymbol{\sigma}, \mathbf{a}, \mathbf{V}, i$)
  for all $j \in \Omega_{l+1}$ **do**
    $p_j \leftarrow \frac{1}{\prod_h^H 2\pi\left(\sigma_j^h\right)^2} \exp\left(-\sum_h^H \frac{\left(V_{ij}^h - \mu_j^h\right)^2}{2\left(\sigma_j^h\right)^2}\right)$
    $R_{ij} \leftarrow \frac{\exp\left(a_j p_j\right)}{\sum_{k \in \Omega_{l+1}} \exp(a_k p_k)}$
    return $\mathbf{R}$

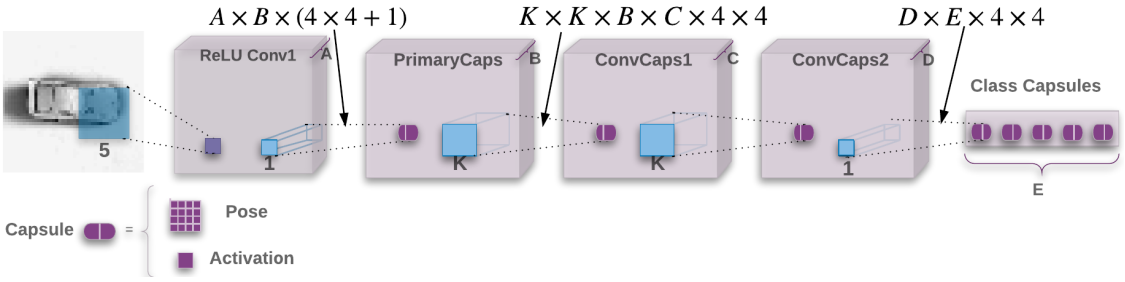


Figure 3.5: Multi-layer matrix capsule network [82]. The standard convolutional layer has A filters, while the 4 capsule layers consist of B, C, D and E types of capsules respectively. Between two capsule layers, the number of weights is given by the product of the size of the receptive field ($K \times K$ for the convolutional capsules, 1×1 for the class capsules), the number of capsule types of both layers and the size of the pose matrix (4×4). This means in this network for example, the routing mechanism has to determine $K \times K \times B \times C$ activation probabilities and pose matrices between the primary and the first convolutional capsule layer.

For the final layer, there is again one fully connected capsule per class, like for the vector capsule networks (cf. figure 3.5). However, the class capsules are connected differently. So as not to lose positional information, the coordinates of the center of the active capsule's receptive field on the grid are added to its vote matrix. By doing this, all the capsules of a type in the second to last layer share the same whole-part transformation matrix and at the same time pass information on their position to the final layer of class capsules. This technique is referred to as *Coordinate Addition* by Hinton et al. A new loss function called *spread loss* is introduced to train directly on the class capsules' activities, without any reconstruction.

$$L_i = (\max(0, m - (a_t - a_i)))^2 \tag{3.20}$$

With a_t the correct class activity. Compared to earlier versions of capsule networks, no effort is made to encourage the final layer of capsules to encode instantiation parameters useful for reconstructing a transformed version of the input image. Instead they are left to learn whatever increases their classification accuracy. Spread loss effectively penalizes activities that are further than the *margin* from the target activity. The margin can be increased on a linear schedule during training so as to avoid the capsules from dying early. Hinton et al. were able to demonstrate that their matrix capsules could achieve similar performance to CNNs with fewer learnable weights and perform much better than vector capsules on more complex shape-recognition datasets like smallNORB [88]. They are also more robust against adversarial attacks of the *fast gradient sign method* than baseline CNNs and possibly generalize better to novel viewpoints, although the latter could not be conclusively shown.

4 Experimental Setup

This chapter explains the experiments in detail. How they are set up, where the data comes from and what exactly is measured and why. All the experiments in this thesis are designed in the Python programming language and evaluated on the same hard- and software configuration (cf. table 4.1).

Processor	6 core Intel Core i7-6850K CPU @ 3.60GHz
Memory	32GB
Graphics Processor	2× Geforce GTX 1080 8GB linked via NVLink
Operating System	Ubuntu 16.04 LTS
CUDA	9.0
Python	3.6.6
PyTorch	0.4.1
BindsNET	0.1.8

Table 4.1: Hardware and software components used to run all the experiments.

4.1 Neural networks

The experiments are run on 4 different types of neural networks, that follow different paradigms and vary in their degree of biological plausibility. Namely they are:

1. A classical 4-layer convolutional neural network.
2. Vector capsules with dynamic routing in a 3-layer neural network.
3. Convolutional matrix capsule layers as part of a 5-layer network.
4. A shallow spiking neural network with a single hidden layer.

The CNN, vector and matrix capsules are all implemented within the PyTorch [89] framework. This allows them to be trained using the same established optimization schemes. Spiking neural networks however require additional dynamics, that are not usually part of the typical tensor-based neural network software libraries. Therefore, the SNN used in the experiments is implemented on top of the recently released BindsNET [90], a python package that extends PyTorch with the neural dynamics necessary for spiking neural networks and biological learning algorithms like STDP. The respective networks' topologies are determined largely by a mix of experience, published results

on similar data as well as trial and error. However, due to the relative simplicity and low resolution of the used data, they are all shallow architectures, which makes it easier to draw conclusions about the used technology w.r.t their performance. Note that the following networks all include an input layer which is not explicitly listed. The input layer has always the same dimension as the input image.

CNN

As CNNs are generally well established a good network topology for the experiments could easily be found and comes down to a variation of a classical LeNet [5]. The network consists of 4 layers, with max pooling operations after each convolutional layer. For the exact network topology, see table 4.2.

Layer	Configuration	
Convolutional layer 1	Number of kernels	32
	Kernel size	5×5
	Stride	1
	Activation function	ReLU
Max pooling	Window size	2×2
Convolutional layer 2	Number of kernels	64
	Kernel size	5×5
	Stride	1
	Activation function	ReLU
Max pooling	Window size	2×2
Fully connected layer 1	Number of neurons	1024
	Activation function	ReLU
Fully connected layer 2	Number of neurons	5
	Activation function	Softmax

Table 4.2: Layers of the convolutional neural network and their configuration, defining the entire network's topology. Note that the number of weights between the second convolutional layer and the first fully connected layer computes as the product of the number of total neurons in each layer, i.e. $(64 \times 5 \times 5) \times 1024 = 1\,638\,400$

Vector Capsules

The vector capsule network is composed of 3 layers in total. This composition is the minimal configuration to run a capsule network and similar to the one used by Sabour et al. in 2017. Namely it is made up of a convolutional layer that is used to initialize the poses and activations of the primary capsule layer and the class capsule layer on top. However, other than the network used by Sabour et al., there is no decoder on top of

the class capsule layer and no reconstruction loss is used to train the network. Instead, the vector capsule network is directly trained using the length of the output vector of the class capsules, similarly to the matrix capsule network by Hinton et al. This has the added benefit that the vector capsules can be trained using the exact same mechanism as the convolutional and matrix capsule networks. Also, the network is not forced to encode features, that are useful for reconstruction but rather encouraged to learn features, that are useful for correct classification. Additionally, compared to Sabour et al. the width of the convolutional layer, the vector output dimensionality as well as the kernel size of the primary capsules are reduced, without losing any accuracy. Together with the removal of the decoder, this greatly reduces the number of parameters to train (cf. table 4.3). Routing-by-agreement (cf. algorithm 1) is used between the capsule layers. The routing algorithm is implemented using PyTorch tensors and operators – this makes it possible to efficiently run the network on GPUs and use the automatic gradients to train the part-whole matrices.

Layer	Configuration	
Convolutional layer	Number of kernels	32
	Kernel size	5×5
	Stride	1
	Activation function	ReLU
Primary capsules	Number of capsule types	32
	Kernel size	1
	Stride	2
	Dimensionality	14
	Activation function	Squash (cf. equation 3.3)
Class capsules	Number of types	5
	Input dimension	14
	Output dimension	8

Table 4.3: Layers of the vector capsule network and their configuration. The network consists of 3 layers, only 2 of which are capsule layers. This represents the bare minimum for a capsule network, as one layer is required to initialize the primary capsules and routing is only possible between two consecutive capsule layers.

Matrix Capsules

There is only little research published on matrix capsules to draw from. For this reason, the network topology is exactly the same as the "small" configuration used by Hinton et al., which they used for image data of the same size as in these experiments. Nevertheless, training matrix capsules is quite challenging, as the gradient will often either vanish or the capsules won't learn at all. To prevent this from happening, capsule-

aware versions of dropout and batch normalization are implemented (cf. section 2.2). They differ from the regular implementation by considering the internal representation of matrix capsules (4×4 pose matrices and $2D$ feature activity maps). By respectively deactivating or normalizing highly correlated groups of neurons, the network can be regularized during training [91]. All of the capsule layers are convolutional and comprised of matrix capsules (cf. table 4.4).

Layer	Configuration	
Convolutional layer	Number of kernels	64
	Kernel size	5×5
	Stride	2
	Padding	2
	Activation function	ReLU
Batch normalization		
Dropout		
Primary capsules	Number of capsule types	8
	Kernel size	1×1
	Stride	1
	Dimensionality	4×4
	Activation function	Sigmoid (only activities)
Dropout		
Convolutional capsules 1	Number of capsule types	16
	Kernel size	3×3
	Stride	2
	Dimensionality	4×4
Dropout		
Convolutional capsules 2	Number of capsule types	16
	Kernel size	3×3
	Stride	1
	Dimensionality	4×4
Dropout		
Class capsules	Number of types	5
	Kernel size	1×1
	Stride	1
	Dimensionality	4×4

Table 4.4: Layers of the matrix capsule network. Note that the matrix capsules include a 4×4 pose matrix as well as a logit representing the activation probability. As such they are comprised of 17 neurons each. Further note that only the size of the primary capsule kernels is given in pixels, while the others are given in whole capsules.

Expectation-Maximization routing (cf. algorithm 2) is employed between all adjacent capsule layers. As with vector capsules, automatic generation of gradients for the operations used inside the routing algorithm is utilized to train the part-whole matrices and description costs discriminatively.

SNN

In order to achieve a similar performance and allow comparison with the other networks, the SNN is designed to facilitate training using stochastic gradient descent with backpropagation. Input images are rate-encoded by setting the firing rate of the spiking neurons in the input layer proportional to the pixel-intensities. The error signal during training is then calculated by applying a softmax to the average firing rates of the second fully connected layer (cf. table 4.5). Although this is only an approximation as the firing rate is not differentiable w.r.t the synaptic weights, this approach achieves better classification performance than STDP-based learning with the same network.

Layer	Configuration	
Fully connected layer 1	Number of neurons	100
	Neuron model	LIF
Fully connected layer 2	Number of neurons	5
	Neuron model	LIF
	Activation function	Softmax

Table 4.5: Layers of the spiking neural network. All of the neurons are modelled as leaky integrate-and-fire neurons. Only a single hidden layer is used due to the expressiveness of spiking neurons. The softmax in the final layer is only applied to the average firing rates during exposure to the input and assumes the role of a loss function.

Thanks to the BindsNET software package, spiking neural networks can be run on graphical processing units, same as the other network architectures. However, as there is no support for the automatic computation of gradients for networks of spiking neurons yet – in BindsNET or any other framework – the gradients have to be computed by hand and implemented directly. This somewhat limits the complexity of the available network's topology.

Hyperparameters

Because they have a huge influence over the respective network's performance, all the relevant hyperparameters are optimized in an exhaustive search. To this end, parameter candidates are arranged on a heuristically determined hypergrid with each axis corresponding to a hyperparameter. Each point on the hypergrid is trained and evaluated on a full object recognition dataset to find a good tuple of parameters (i.e. one

that achieves high accuracy on a test set). The hyperparameters included in the search for the CNN, vector capsule and matrix capsule networks are explained in table 4.6 as they use the same optimization scheme. For the SNN hyperparameters, see table 4.7.

Parameter	Description
batch size	Number of samples processed during a pass
learning rate	Size of the step taken during gradient descent
epochs	Number of training epochs
routing iterations	Iterations of dynamic routing, only used in capsule networks

Table 4.6: Hyperparameter definitions of the convolutional, vector capsule and matrix capsule networks. Note that not all the hyperparameters are used by all the network architectures.

Parameter	Description
hidden neurons	The number of neurons in the hidden layer
time	Length of time in milliseconds input is generated during training
learning rate	Size of the step taken during gradient descent
decay rate	Rate the learning rate is decayed by the scheduler
patience	Number of intervals without progress before the scheduler decays the learning rate
interval	Size of an interval in number of samples processed
epochs	Number of training epochs

Table 4.7: Hyperparameter definitions of the spiking neural network. Note that compared to common CNN schedulers, the learning rate can be decayed during an epoch.

Training

All of the networks are trained discriminatively by stochastic gradient descent (or a variation thereof) with backpropagation over several epochs. The CNN, vector and matrix capsules all employ the same optimization scheme. The scalar output of their final layer is used to compute spread loss (cf. equation 3.20), which is minimized by PyTorch's implementation of the Adam algorithm [35]. A scheduler is used, that decreases the learning rate by a factor of 10 after each epoch of training, which doesn't increase accuracy on the training data. Additionally, the inverse temperature of the logistic function (cf. equation 3.16) used in EM routing for the matrix capsules is adjusted on a linear schedule. The net result of this is that the sigmoidal shape of the logistic function becomes steeper over the epochs, effectively encouraging the routing to

become more concise. The SNN on the other hand uses SGD computed based on the average firing rates of the output neurons with softmax as an error function. The inputs are generated by sampling for a fixed period of time from a Poisson distribution with the firing rate proportional to the pixels' intensities. Because of the asynchronous nature of spiking neurons, only a single training sample can be viewed at a time, compared to the other networks, that process a batch of input samples in parallel. A similar scheduler is implemented to decrease the learning rate, but greater care has to be taken w.r.t choosing the scheduler's hyperparameters as the training is more susceptible to changes in learning rate than the Adam algorithm.

4.2 Datasets

A virtual experiment for the Neurorobotics Platform (NRP) is designed to create object recognition data. Experiments for the NRP consist of a virtual environment, one or more robots and a brain based on spiking neural networks that can be connected to the robot's sensors and actuators via so-called transfer functions [52]. The environment in this experiment is fairly simple, it consists of a plane, several objects and a number of different lights. A digital clone of an iCub robot [92] is placed into the environment and presented each object (cf. figure 4.1).

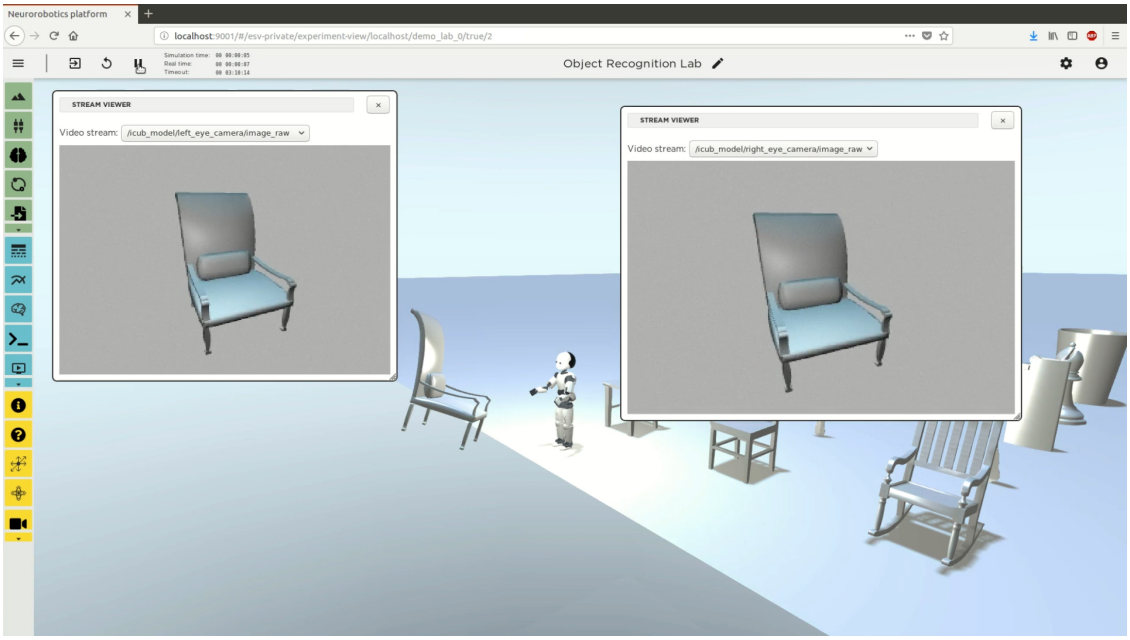


Figure 4.1: Screenshot of the Neurorobotics Experiment creating object recognition data. The Experiment is viewed from inside a browser. Note that each of the two windows is a live view of one of the iCub robot's video cameras placed in its eyes. Thus, realistic and to scale stereo images can be generated.

The objects are presented from different viewpoints and under varying lighting conditions, while the NRP handles physical simulation and rendering of the scene. This includes simulation of the robot's noisy sensors, which can be queried for their data. In this case, the iCub's video cameras are used to capture stereo images from the iCub's point of view. The captured images can then be grouped into datasets and trained on the different neural network architectures. Each image includes labels describing the class the viewed object belongs to, the lighting conditions as well as the viewpoint. In total, there are 8 different objects of 5 classes each. They are designed to represent everyday household items, such as future robots may need to recognize (cf. figure 4.2).

Figure 4.2: Samples from the object recognition dataset. The 5 classes are chairs, chess figures, cutlery, cups and lighters. Each class consists of 8 objects, that are viewed from 162 different viewpoints under 4 distinct lighting-setups. Note that while the objects obviously vary in shape they also vary in topology, e.g. there are cups with and without a handle or chairs standing on 4 legs compared to rocking chairs.

As matrix capsules were specifically conceived by Hinton et al. as shape recognition units to learn part-whole relationships, the datasets are designed with an emphasis on shape recognition. As such they do not feature any textures or colors. However, they are still very challenging object recognition tasks. Outliers that vary greatly in shape as well as topology were deliberately included. The rationale behind this is, that while there is some variance within a class, the classes still differ enough from each other,

such that a good recognition system should be able to rule out false positives. While the lighting can be varied continuously, 4 different setups are chosen, that vary in the number, direction and intensity of lights in the scene. The different viewpoints are taken from a solid angle of size 2π sr, i.e. the upper half-dome with the object at the center, discretely sampled at 9 elevations and 18 azimuths for a total of 162 points. However, rather than moving the entire robot, only the object is rotated. Specifically, the lights are not rotated with the object. Compared to similar datasets, this results in complex interactions between the lights and the object (cf. figure 4.3).

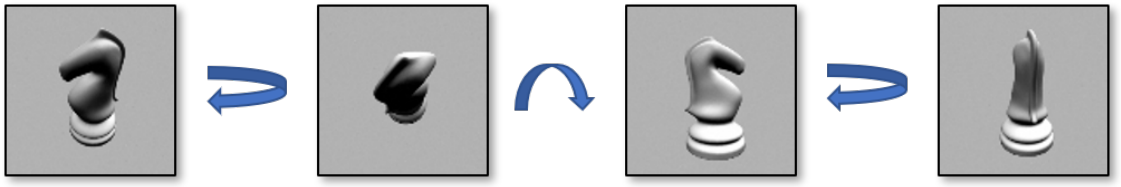


Figure 4.3: Samples of different viewpoints from the object recognition dataset. The same object from the class of chess figures is viewed from different azimuths and elevations under the same lighting. Note that while the shape of the figure is symmetrical across one axis, the two sides are lit-up differently under different viewpoints.

Based on the basis dataset (consisting of all the objects, viewpoints and 4 different lighting setups) different augmented versions are created. These augmentations can be used as additional test sets for networks trained on the basis dataset to evaluate their capabilities further. To investigate how proficient the networks are at explaining-away perturbations in the form of occlusions, the datasets are incrementally augmented. Small geometric primitives that vary in shape, color and size are randomly placed on top of the objects until they cover a fixed percentage of the image (cf. figure 4.4).

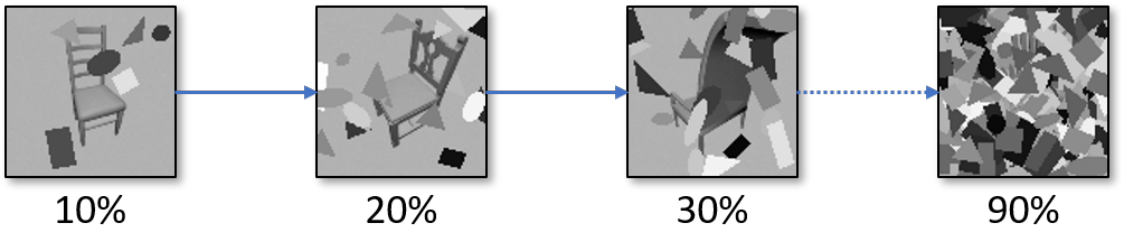


Figure 4.4: Augmentation of the object recognition dataset to test for robustness against occlusion. Ellipses, rectangles and triangles of varying shape and color are randomly placed on the images to occlude the object. An entire dataset is created for each increment in the percentage of coverage.

Another augmentation of the datasets is intended to test for robustness of part-whole relationships. Capsule networks specifically are only supposed to activate the capsules in higher levels, if the capsules in the lower layers agree about the higher capsule's pose based on the pose of their own features. Classifiers that don't learn part-whole relationships could for example still activate as long as the parts are present, even if their pose is not consistent with the whole. By dividing each image into square tiles and randomly switching them, the location of features is effectively permuted on an even grid (cf. figure 4.5). An argument can be made that the difficulty of testing for part-whole relationships increases with the number of tiles, because for fewer (i.e. bigger) tiles there are both bigger features and locally consistent submanifolds of features that agree.

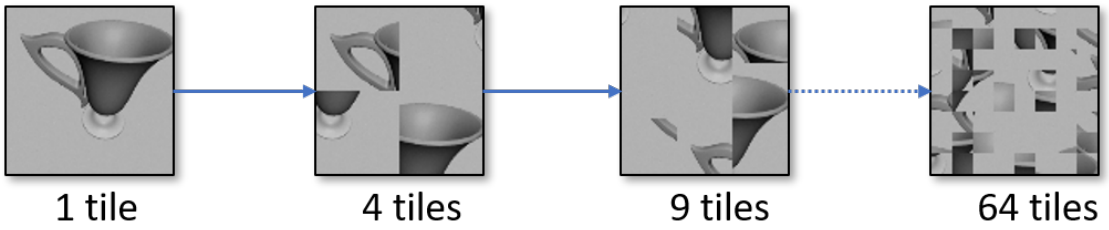


Figure 4.5: Augmentation of the object recognition dataset to test the robustness of part-whole relationships. The images are divided into an increasing number of squares that are randomly permuted. For each number of tiles, an entire dataset is created.

4.3 Metrics

The metrics used to evaluate the neural networks can be divided into runtime and classification metrics. The former deals with resource requirements needed to run the networks while the latter looks at their statistical performance as classifiers in object recognition tasks.

Runtime Metrics

To allow comparison of the runtime metrics, all the networks are trained for one epoch on the basic version of the object recognition dataset using the same batch size (i.e. practically normalizing time and memory measurements over batch size). This means training only one sample at a time, because the spiking neural network is unable to process mini batches. During training the memory allocated by the network and its computation graph is measured as well as the number of parameters that need to be trained and the time required to train the entire epoch. Latency of the trained networks is then measured by simply timing inference on the test set. Also, during inference, the

memory requirements are measured again. Finally, to quantify the benefits of parallel processing on GPUs, the networks are again timed during training and inference using mini batches of the sizes determined by the hyperparameter grid search.

Classification Metrics

The most obvious classification metric is accuracy, i.e. the fraction of correct predictions during inference over the test set.

$$\text{accuracy}\,(\mathbf{Y},\mathbf{T}) = \frac{1}{|T|}\sum_{i=1}^{|T|}\mathbb{1}(y_i = t_i) \tag{4.1}$$

With $\mathbf{Y} = (y_1, y_2, ...)^T$ and $\mathbf{T} = (t_1, t_2, ...)^T$ containing the classifier's estimate of the training samples and the true class labels respectively and $\mathbb{1}$ the indicator function. Information about all of the network's predictions in relation to the true labels can be neatly expressed within the so-called *confusion matrix*. In a confusion matrix each row corresponds to the true class labels and each column to the prediction. Each entry is then the number of samples of class i predicted to be class j (cf. figure 4.6).

$$M_{i,j} = \sum_{k=1}^{|T|}\mathbb{1}(t_k = i)\mathbb{1}(y_k = j) \tag{4.2}$$

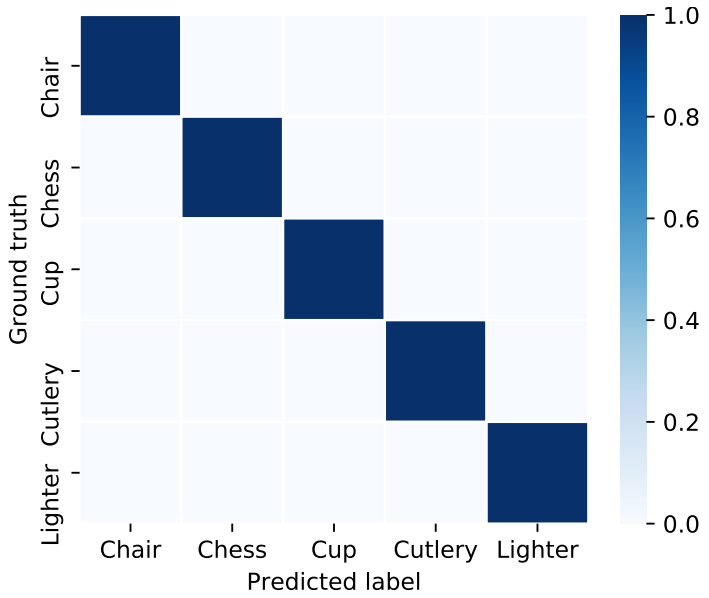


Figure 4.6: Example of a confusion matrix by means of a perfect classifier. The confusion matrix is plotted in the form of a heatmap for easy inspection of the accuracy across different classes. Note that a perfect classifier results in a diagonal matrix as the off-diagonal elements correspond to misclassifications.

For binary classifiers it is interesting to know the *true positive rate* as well as the *false positive rate*, i.e. the fraction of positive predictions that were correctly classified and the fraction of negative samples that were falsely classified as positive respectively. This information can be visualized by plotting the true positive rate against the false positive rate at various classification thresholds. The resulting curve is known as a *receiver operating characteristic* (ROC) curve. It can be interpreted as depicting the trade-off between reward (true positives) and cost (false positives). The ROC space is divided by a diagonal into points representing good classifications above and predictions worse than random below (cf. figure 4.7). The curve information can be compiled into a single scalar value representing expected performance by computing the area under the curve (AUC). The ROC AUC of a classifier is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance [93].

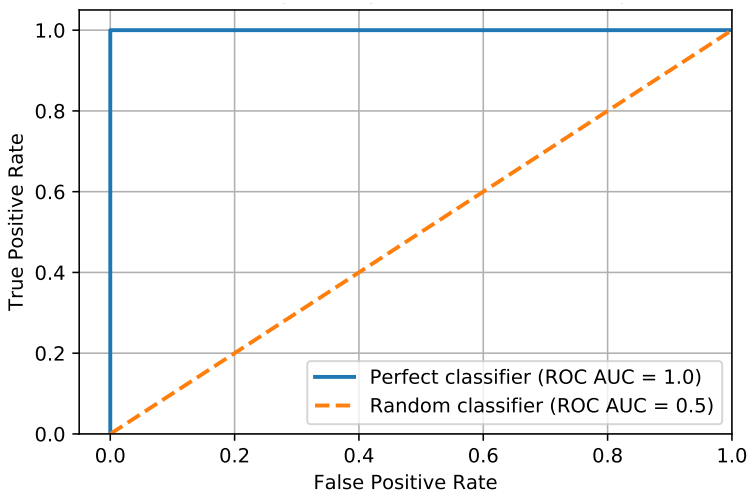


Figure 4.7: Example of a ROC curve by means of a perfect classifier. The curve of a perfect classifier always yields a true positive rate of 100 %, while a random guess produces as many true as false positives and therefore corresponds to a diagonal. A real classifier will lie somewhere between those curves, i.e. the closer the ROC curve to the upper left corner of ROC space, the better the classifier.

To evaluate the neural networks used in these experiments, the ROC is extended to multi-label classifiers. This is done by binarizing the output (one label is interpreted as positive while all others are negative, i.e. one-vs-all classification) and generating one ROC curve per class. All the ROC curves are then averaged to generate a single curve as well as AUC score [94].

Finally, a metric is introduced that fully leverages the versatility of the generated data from the Neurorobotics Platform to evaluate a network's ability to generalize. To test generalization, the networks are trained on a basic version of the data and tested on a perturbed version. The perturbed versions include the augmented data (occlusions and tiles respectively, cf. section 4.2) but also viewpoints that are not explicitly trained. However, simply measuring the established metrics on a perturbed dataset only yields a single data point or point estimate of the network's ability to generalize. Instead, the "extent" of the perturbation across perturbation space can be parameterized to introduce a latent variable.

$$P_{\text{Perturb}}(\mathbf{t} \mid \mathbf{x}, \boldsymbol{\theta}) = \int_{\Omega} P(\mathbf{t} \mid \mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\tau}) P(\boldsymbol{\tau}) \, \mathrm{d}\boldsymbol{\tau} \tag{4.3}$$

Where the sum and product rules of probability are used along with $\boldsymbol{\tau}$ the perturbation parameter and Ω the perturbation space. Note that each possible value of $\boldsymbol{\tau}$ corresponds to an entire dataset perturbed to the extent as described by $\boldsymbol{\tau}$. By sampling $\boldsymbol{\tau}$ from the prior distribution $P(\boldsymbol{\tau})$ the integral can be computed using Monte Carlo integration.

$$\int_{\Omega} P(\mathbf{t} \mid \mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\tau}) P(\boldsymbol{\tau}) \, \mathrm{d}\boldsymbol{\tau} \approx \frac{1}{T} \sum_{t=1}^{T} P(\mathbf{t} \mid \mathbf{x}, \boldsymbol{\theta}, \boldsymbol{\tau}_t) \tag{4.4}$$

Using this formalism, the generalization is then expressed by the difference between the metrics of the unperturbed and the perturbed networks. The latter is computed by stochastically sampling from the parameterized perturbation space and computing the average of the metric evaluated on all drawn samples. This is an approximation and not generally possible but with the metrics used in these experiments the sum over the samples from the test set can be switched with the sum over the perturbation parameter samples. Use of Monte Carlo integration is justified as both the prior distribution of the latent variable is known (its uniform as it's generated in even increments) and the micro perturbations are generated stochastically. The augmented datasets are already parameterized. Another parameterized dataset can be created by incrementally removing random viewpoints from the training set and adding them to the test set. The generalization metrics determined this way should then be more informative than a single evaluation.

5 Results - Work in Progress

All experimental results are compiled into this chapter, starting with the optimization of the hyperparameters followed by the runtime and classification metrics of the object recognition tasks and finally the generalization experiments.

5.1 Hyperparameters

The 3 tuples of hyperparameters found for each neural network architecture that achieved the highest accuracy on the basic, non-perturbed version of the dataset are listed in the tables 5.1 and 5.2. They are found by refining the hypergrid around the most promising candidates after an initial extensive search. The best hyperparameter tuple is used in training and inference for the remainder of the experiments, unless explicitly mentioned otherwise.

model	batch size	learning rate	routing iterations	accuracy
matrix capsule network	21	9×10^{-3}	2	88.96%
	19	1×10^{-2}	2	84.72%
	21	3×10^{-2}	2	79.25%
vector capsule network	1	2×10^{-4}	2	90.82%
	3	3×10^{-4}	2	87.24%
	2	3×10^{-4}	2	83.69%
CNN	130	1×10^{-5}	-	85.23%
	128	1×10^{-5}	-	84.25%
	131	1×10^{-5}	-	83.90%

Table 5.1: Top 3 results of the hyperparameter optimization for the matrix capsule, vector capsule and convolutional neural networks. The highest classification accuracy on the test set is reached by a vector capsule network, closely followed by the best result of the matrix capsule network. Most of the top results of the vector capsules use very small mini batches or process single samples even, whereas the matrix capsules and CNN prefer mini batches one or two orders of magnitude larger respectively. No large variation ($\ll \times 10$) in the range of preferred learning rate can be observed for any of the 3 networks. Note that routing does not apply to CNN training, vector capsules perform best at 2 iterations and matrix capsules only converge for a maximum of 2 iterations.

hidden neurons	time	learning rate	decay rate	patience	interval	accuracy
100	15	5×10^{-3}	0.7	10	250	79.77%
100	5	5×10^{-3}	0.7	10	250	78.04%
250	15	7×10^{-3}	0.5	10	250	77.72%

Table 5.2: Top 3 results of the hyperparameter optimization for the spiking neural network. The preferred tuples show only little variance and are confined to a small subspace of the hypercube. A smaller number of neurons in the hidden layer (search range from 25–1000) is clearly preferred by the network.

To visualize the distribution of the hyperparameters, each parameter's correlation with the accuracy is determined by computing the mutual information (MI) between the two distributions [95]. The accuracy is plotted over the two parameters with the highest MI scores in figure 5.1.

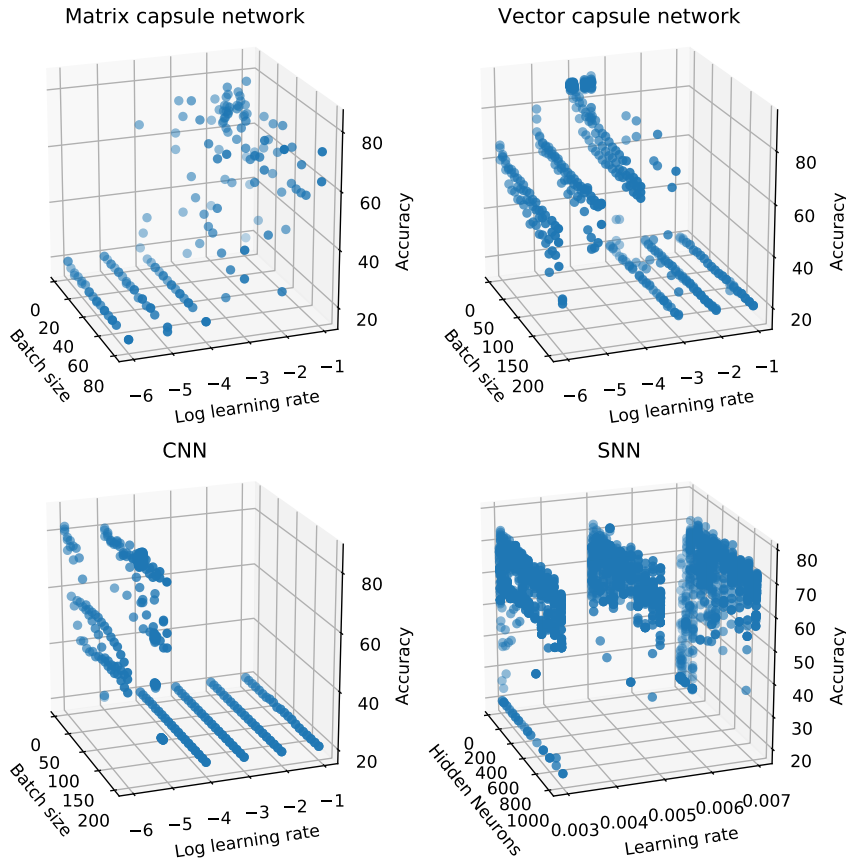


Figure 5.1: Accuracy over the two most relevant hyperparameters for each network. The CNN and especially the vector capsule network highly correlate with batch size, while the SNN and the matrix capsule network in particular show considerable variance of accuracy along the axes.

The temporal progression of training is investigated by plotting accuracy on the test set after each training epoch for the top 3 hyperparameter tuples of each network architecture (cf. figure 5.2).

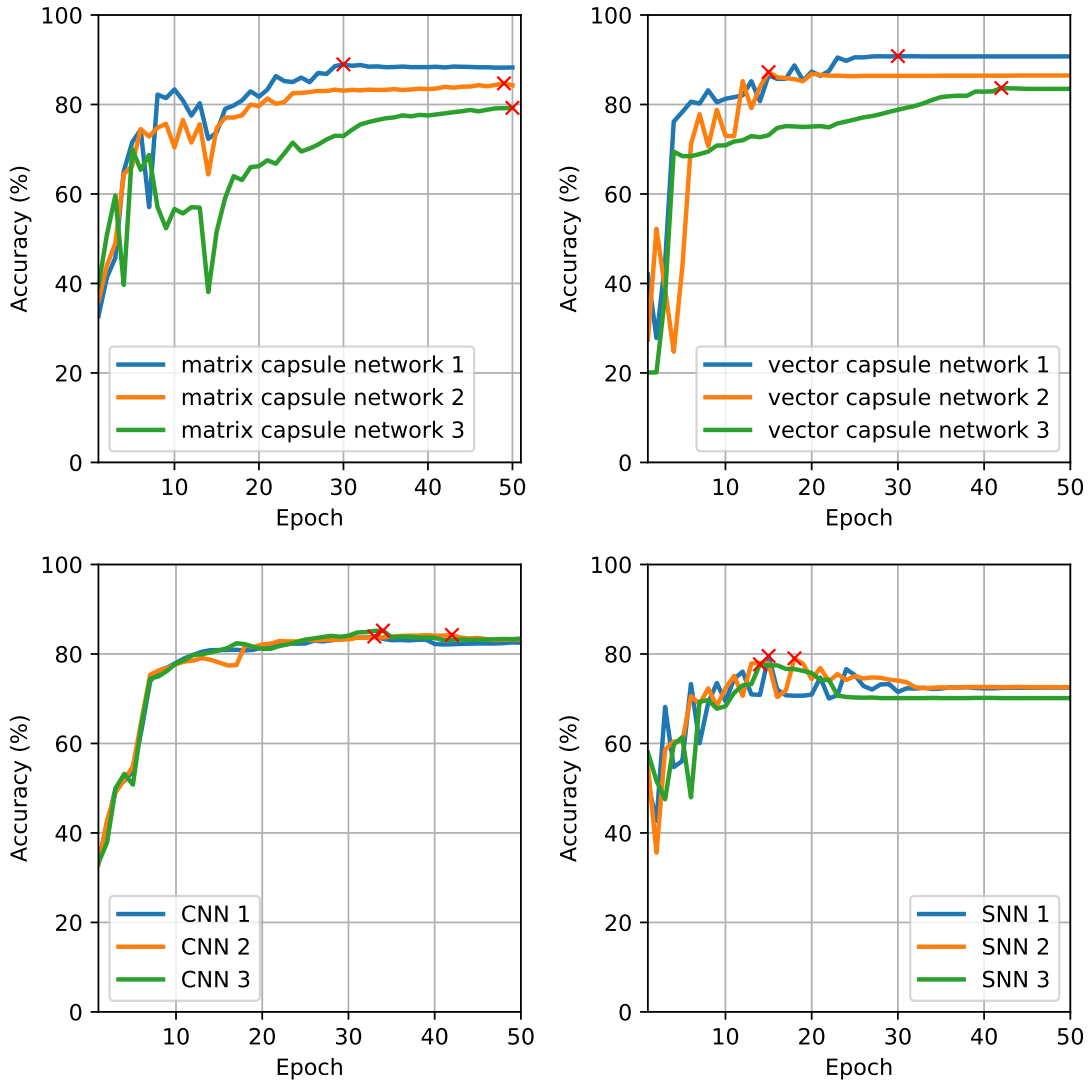


Figure 5.2: Accuracy on the test set during training over 50 epochs for the top 3 hyperparameter tuples of all network architectures. The red cross marks the epoch with the highest accuracy for each configuration. The capsule networks only start learning after several epochs and keep zigzagging for several more. Of all the architectures, CNNs show the most stable behavior and converge straightforwardly while the SNNs show somewhat unstable behavior before they converge. Additionally, the SNNs reach their maximums before convergence and lose a few percentage points before they converge.

5.2 Runtime

Figure 5.3 shows the memory usage of each network required to train and test a single input image at 32 × 32 pixels. This is contrasted with the number of trainable parameters. There is a huge difference in the required memory on GPU. The networks based on capsules can easily take up several GB of memory, whereas the CNNs and SNNs use less than 100 MB. The exact numerical results are also available in table 5.3.

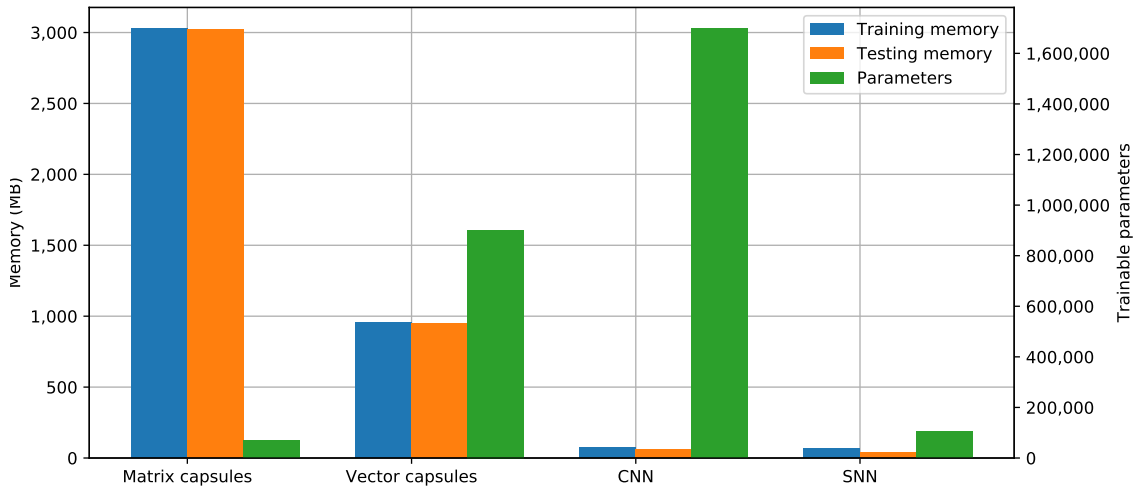


Figure 5.3: Memory requirements and number of trainable parameters for all the network architectures. The matrix capsules require the most memory by far but they also need to train the fewest parameters to learn the data. They are closely followed by the SNN regarding number of parameters while the CNN and vector capsule network need many more parameters. SNNs are the only architecture that require both little memory as well as few parameters.

	Matrix capsules	Vector capsules	CNN	SNN
Training memory (MB)	3025.47	956.16	76.94	64.29
Testing memory (MB)	3023.48	945.76	57.52	38.46
Trainable Parameters	67 442	901 536	1 696 645	103 005

Table 5.3: Numerical results of the memory requirements and number of trainable parameters for all the networks. Note that the difference in memory requirements between training and inference is largest for the SNN, in both relative and absolute terms.

For the final runtime metric, the actual time it takes to run through a training and inference epoch respectively is measured. Because of the uneven train/test set split, these measurements are additionally normalized over sample size. The measurements given in figure 5.4 therefore represent the average time it takes to process a single sample.

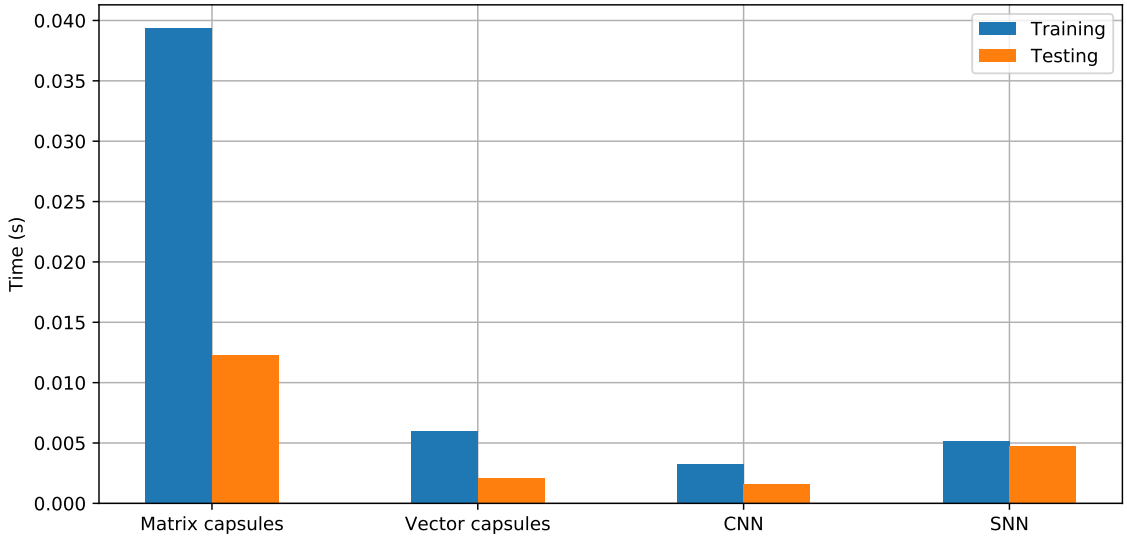


Figure 5.4: Training and inference runtime normalized over sample size. All the network architectures with the exception of SNNs show a marked decrease in runtime for inference compared to training. This is particularly true for the matrix capsule network, which also has the longest absolute runtimes by a large margin.

To consider the advantage of highly parallel computing on GPUs all the runtime metrics are again measured during inference with the maximum possible mini batch size (cf. table 5.4).

	Matrix capsule network	Vector capsule network	CNN
Time	0.0023 s	0.000 25 s	0.000 13 s
Batch size	465	1897	2406
Memory footprint	5492 MB	6985 MB	486 MB

Table 5.4: Runtime metrics for inference on GPU with maximally sized mini batches. The given time is the average per processed sample, whereas the memory footprint is for the entire mini batch. Note that the CNN can process the entire dataset in a single step without running into memory limitations.

5.3 Object Recognition

The object recognition task in these experiments consists of classifying images of 5 different classes. Each class is represented by 8 different instances viewed from roughly 160 different viewpoints. The networks are trained on 5 object instances of each class and then tested on the remaining 3. This means that an accuracy of 20 % is the worst possible score for a classifier – a random guess without any learned knowledge. The classification metrics are compiled into confusion matrices and receiver operating characteristic curves for all the network architectures. The diagonal elements of the confusion matrix count the number of correct classifications for each object class, while the off-diagonal elements all correspond to misclassifications. ROC curves on the other hand allow a quick estimation at what cost (in the sense of false positives) classification accuracy is achieved. Summaries of the results for each network can be found in table 5.5 as well as figures 5.5, 5.6, 5.7, 5.8 and 5.9 on the following pages.

Model	Vector capsule network	Matrix capsule network	CNN	SNN
Accuracy	90.82 %	88.96 %	85.23 %	79.77 %

Table 5.5: Ranking of the neural networks by their performance in object recognition.

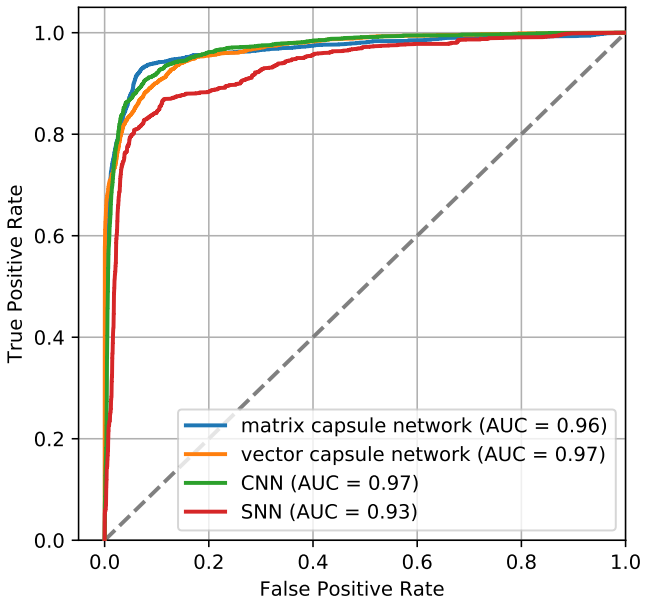


Figure 5.5: Averaged ROC curves and AUCs for all the networks. The average curves are computed form the binary single-class ROC curves of each class. Note that all the networks offer a similar trade-off between cost and precision, except for the SNN, which cannot reach the same level of accuracy.

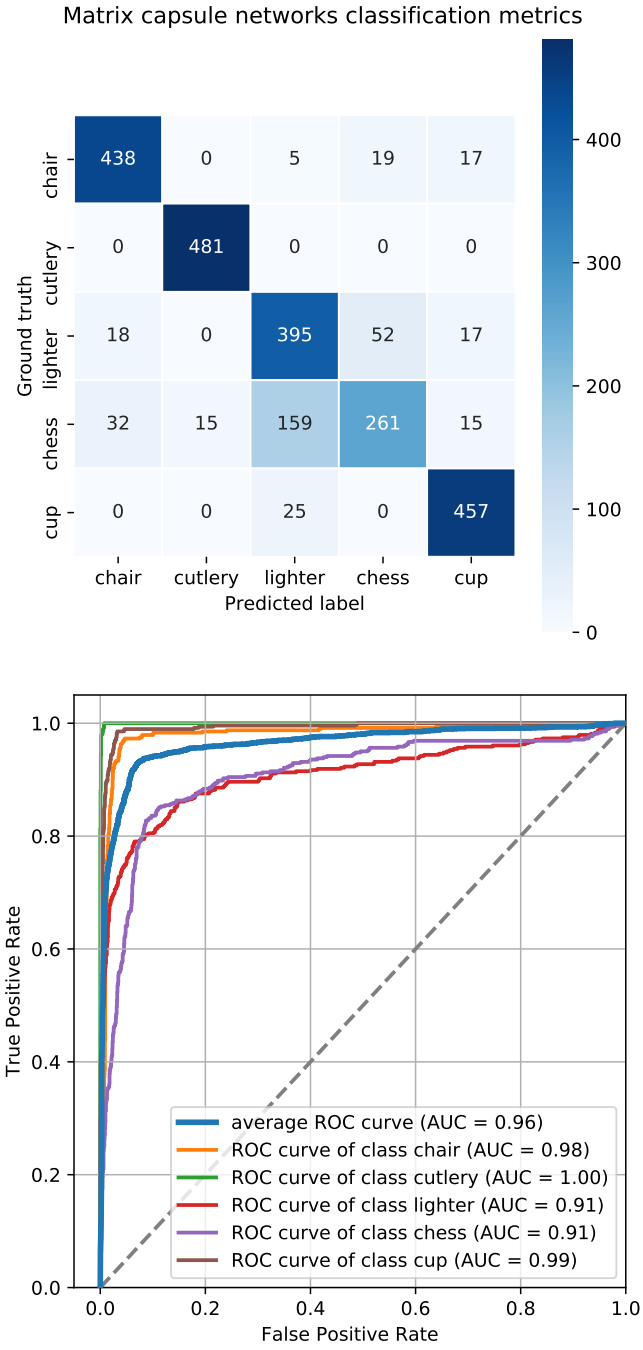


Figure 5.6: The network is best able to classify the object class of cutlery, never mistaking it for a different class. This is followed by the cups, although the network sometimes mistakes them for lighters. Most misclassifications happen for the chess objects, they are very often predicted to be lighters as well. The ROC curves on the right confirm this information. Classes chess and lighter perform below average, whereas the other curves approach a perfect classifier.

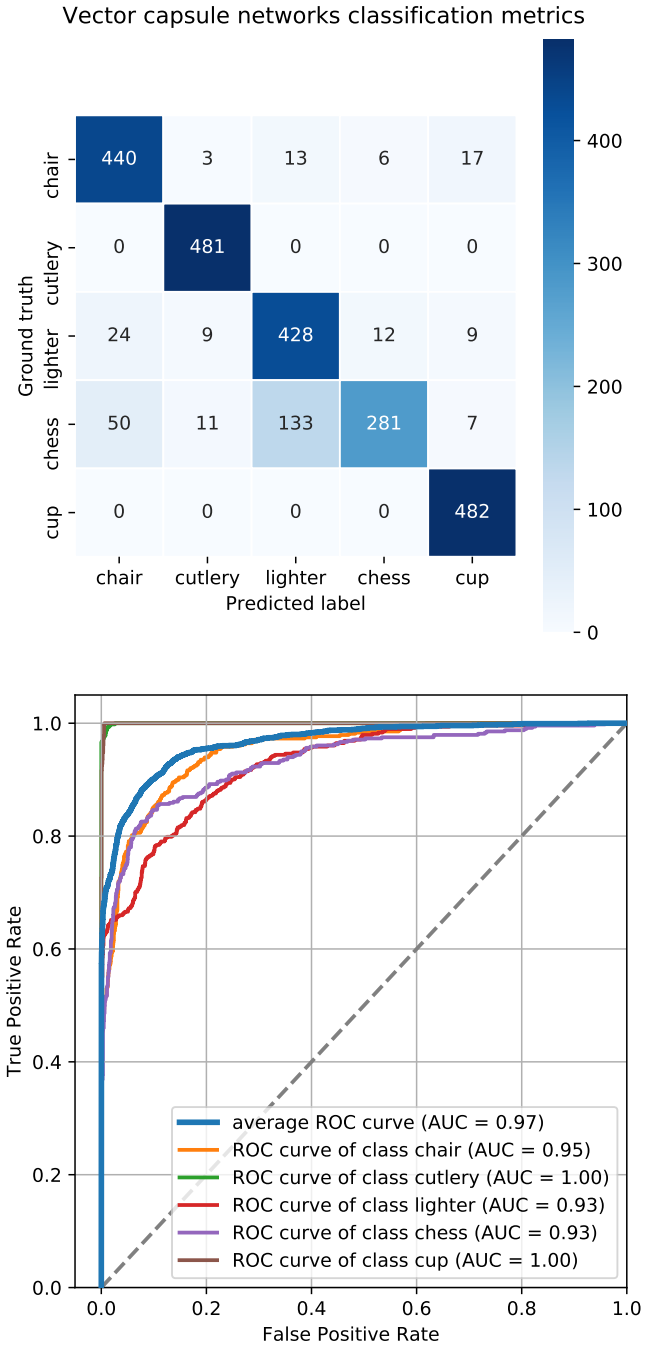


Figure 5.7: The network correctly identifies all the cutlery and cups without too many false positives, resulting in nearly perfect ROC curves for those classes. Like the matrix capsule network, it often mistakes objects of the chess class for lighters, although not quite as often as the matrix capsules do. On average the vector capsules outperform the matrix capsules by lowering the error rates of the lighter and chess classes as can be seen by the AUC. Performance on the other classes is either on par with matrix capsules or slightly reduced as in the case of the chairs (higher false positive rate).

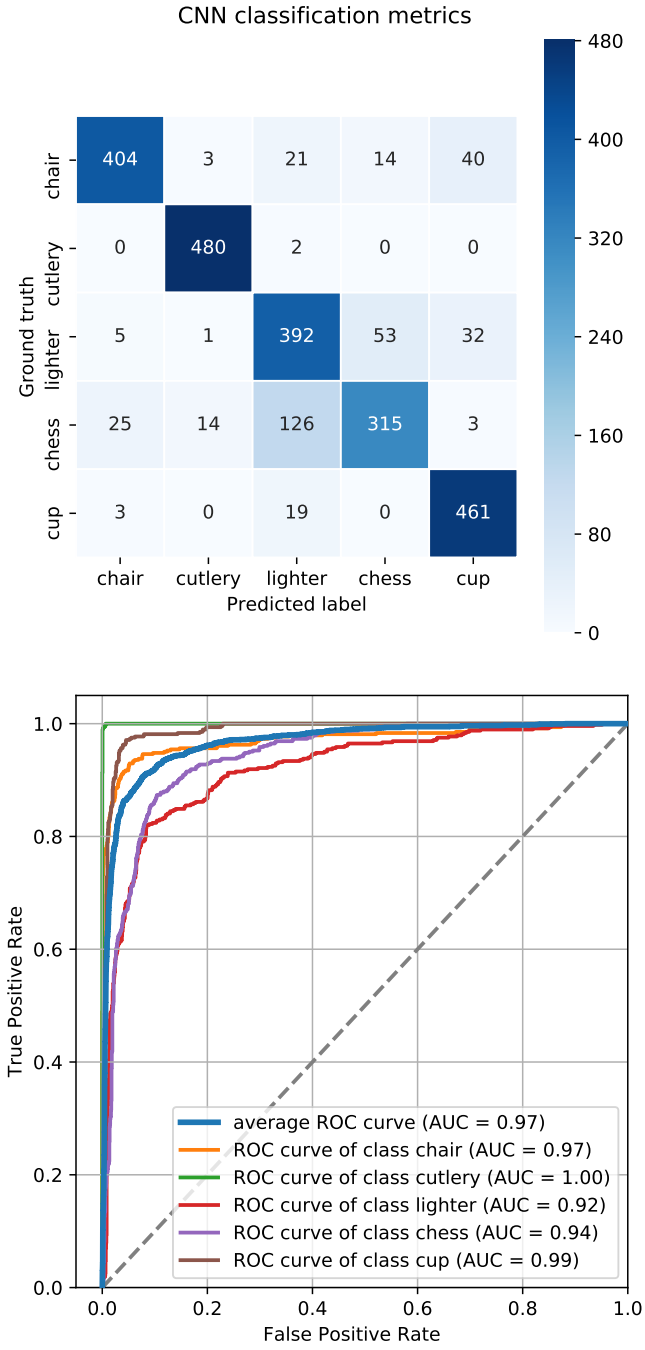


Figure 5.8: Classification metrics for the CNN. Similarly to the capsule networks, the CNN approaches a perfect classifier for the cutlery class and has problems keeping chess pieces and lighters apart. Additionally, the CNN outperforms the other networks when it comes to identifying chess pieces, even if it can't beat the capsule networks in accuracy across all the classes.

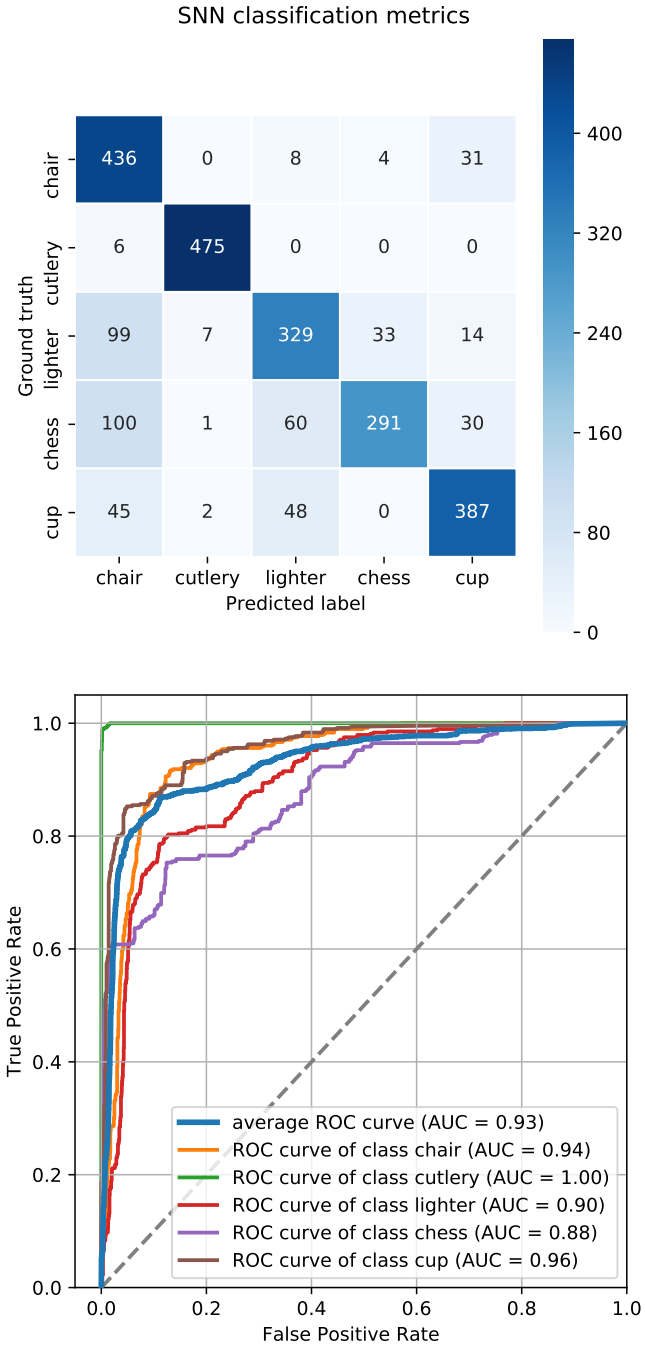


Figure 5.9: Classification metrics for the SNN. The SNN shows the highest variance as can be seen by the off-diagonal elements in the confusion matrix and the distribution of ROC curves. In particular it often predicts a chair, except if it sees cutlery. On the other hand it does not nearly as often confuse chess pieces with lighters as the other networks.

5.4 Generalization

The networks trained on the basic object recognition dataset are tested against new variations in lighting. These variations in lighting include adding additional spot lights, changing intensity and position of the lights etc. Figure 5.10 shows the accuracy of each network during inference relative to the accuracy of the originally trained lighting. This is not the same as the perturbation parameterization used later as there are only three arbitrarily chosen light setups that do not cover the entire perturbation space. But it can be used as a baseline to determine, how useful that approach is.

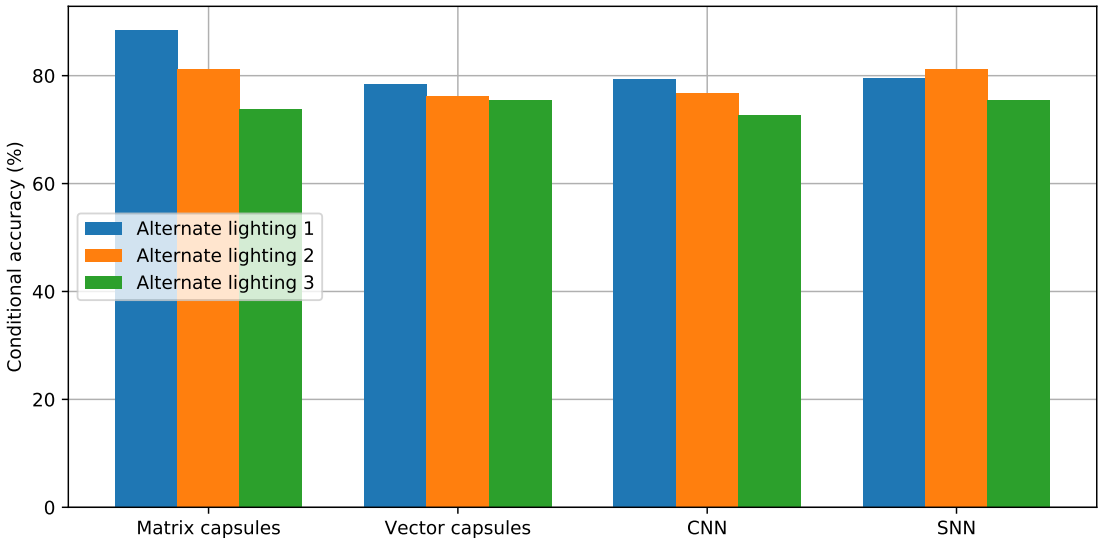


Figure 5.10: Relative accuracy under different lighting. The networks lose around 20 % of accuracy under unknown lighting. However, the matrix capsule network appears to have a small advantage over the other architectures as it reaches 80 % of its unperturbed accuracy on two of the three test sets.

The conditional accuracy used in these plots refers to the conditional probability of classifying an object correctly given perturbations. For the metrics, this can also be expressed as relative or conditional accuracy, i.e. accuracy on the perturbed data normalized over the unperturbed performance. Consequently a conditional accuracy of 100 % means the perturbation has no influence on classification accuracy. Robustness against perturbation or analogously the ability to generalize can then be interpreted as the conditional accuracy across the entire perturbation space (i.e. the area under the curve). The concept of generalization is explored further by measuring the AUC of the conditional accuracy for three kinds of perturbation. This is done by taking the optimized networks that were trained on the basic, unperturbed and unaugmented version of the object recognition dataset and test them on the perturbed versions, where the accuracy on each perturbed dataset corresponds to one measurement point.

The first perturbation is simple occlusion by geometric primitives placed on top of the object. Generating the perturbation in this ways makes it easy to sample from the distribution of the perturbation and compute an accurate curve and AUC (cf. figure 5.11).

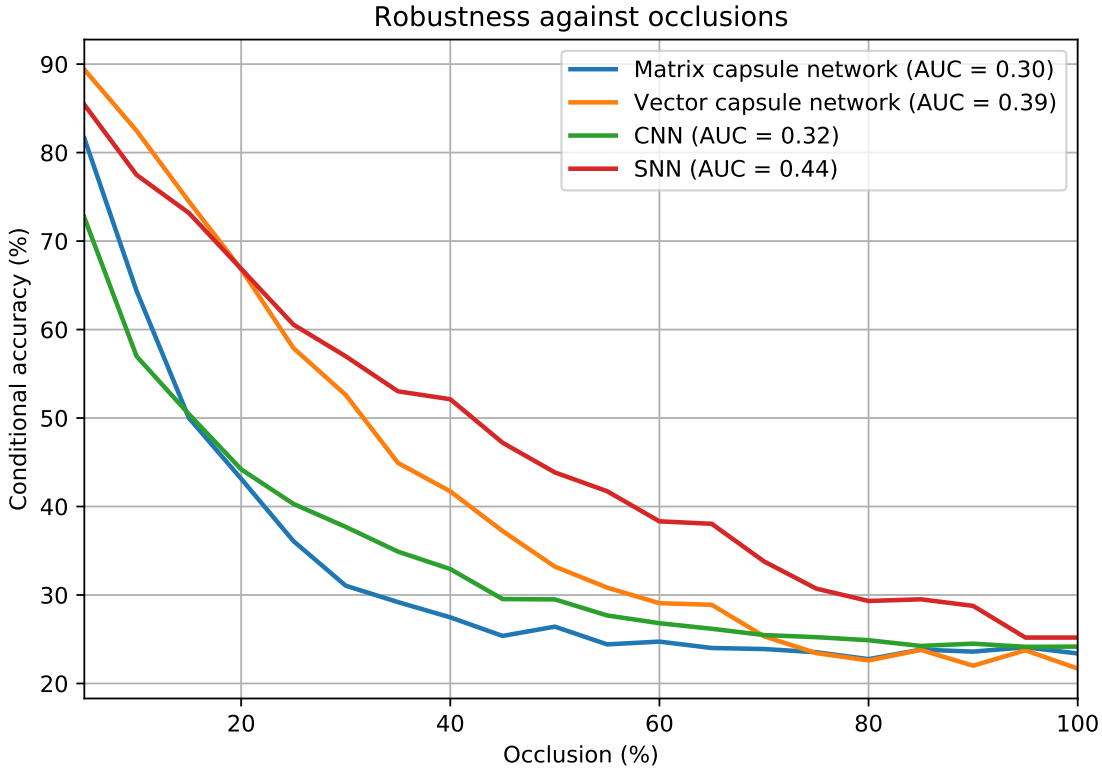


Figure 5.11: Generalization of the networks in the presence of occlusion. With increasing occlusion the networks' ability to classify goes down. Of all the networks, the SNN relinquishes the least accuracy by a large margin while the matrix capsule network shows the worst performance. Eventually, as the entire image is covered, all the classifiers perform randomly. Note that the accuracy curves occasionally cross each other

Another interesting generalization ability is robustness against part-whole decomposition. A classifier that strictly adheres to part-whole relationships should not classify an object where those part-whole relationships have been deconstructed, i.e. they dont exist. The tile augmentation randomly switches image segments and therefore breaks relationships between features in pose space. By definition the robustness against part-whole decomposition is then the probability of the opposite event (i.e. misclassification). A perfectly generalizing classifier would therefore in the face of part-whole deconstruction give a random classification accuracy (20 % in this experiment). The way this ability is defined may appear to measure the opposite of generalization but semantically this can

be thought of as the ability to retain the coherence of internal representation against outside perturbation, i.e. generalization to novel situations. A plot of the curves and the networks' AUC can be found in figure 5.12.

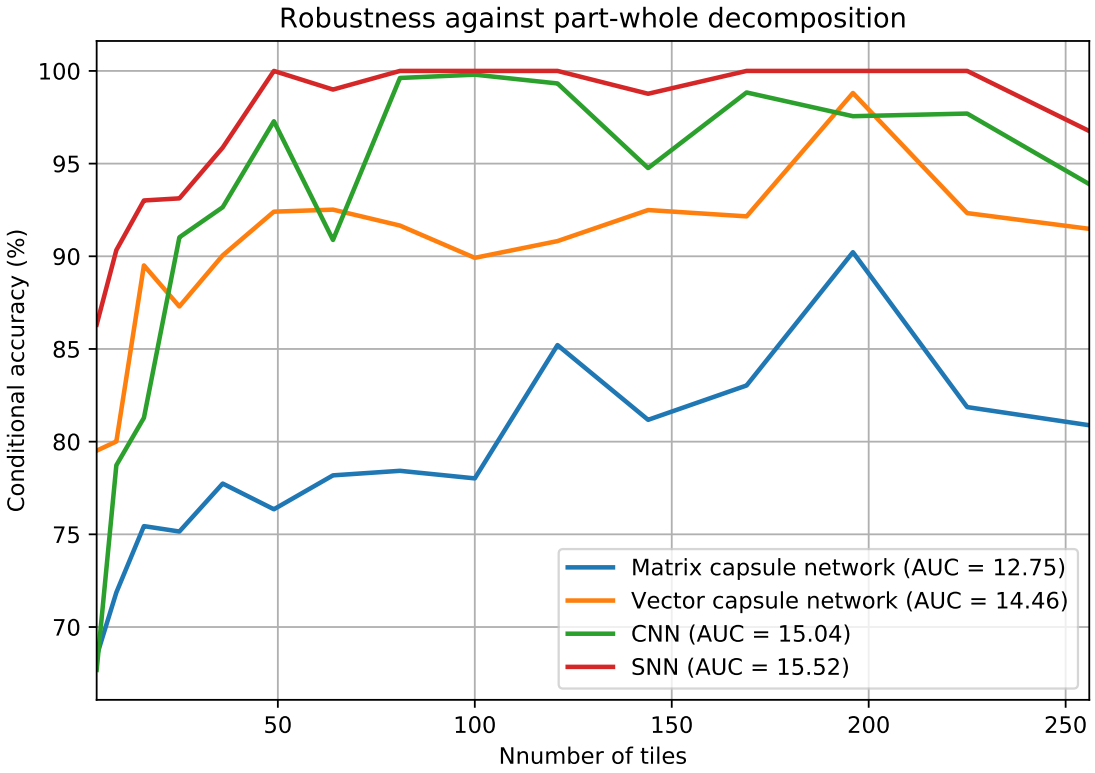


Figure 5.12: Generalization of the networks in the presence of part-whole decomposition. Again the matrix capsules show the least generalization, in thise case even by a large margin, almost across the entire perturbation space. The CNN generalized quite well and the SNN is again on top. Note that the number of tiles doesn't increase beyond 256. This is because the tiles eventually become smaller than 2×2 pixels and can no longer thought to include visually meaningful features.

Finally, an attempt is made to determine the networks' ability to handle unknown viewpoints to classify known objects. To this end the networks are trained only on a subset of viewpoints whilst inference is performed on a test set including all the viewpoints. The relative amount of viewpoints to exclude from training is set incrementally for each measurement and randomly removed from the training set. The results of this endeavor may be observed in figure 4.3.

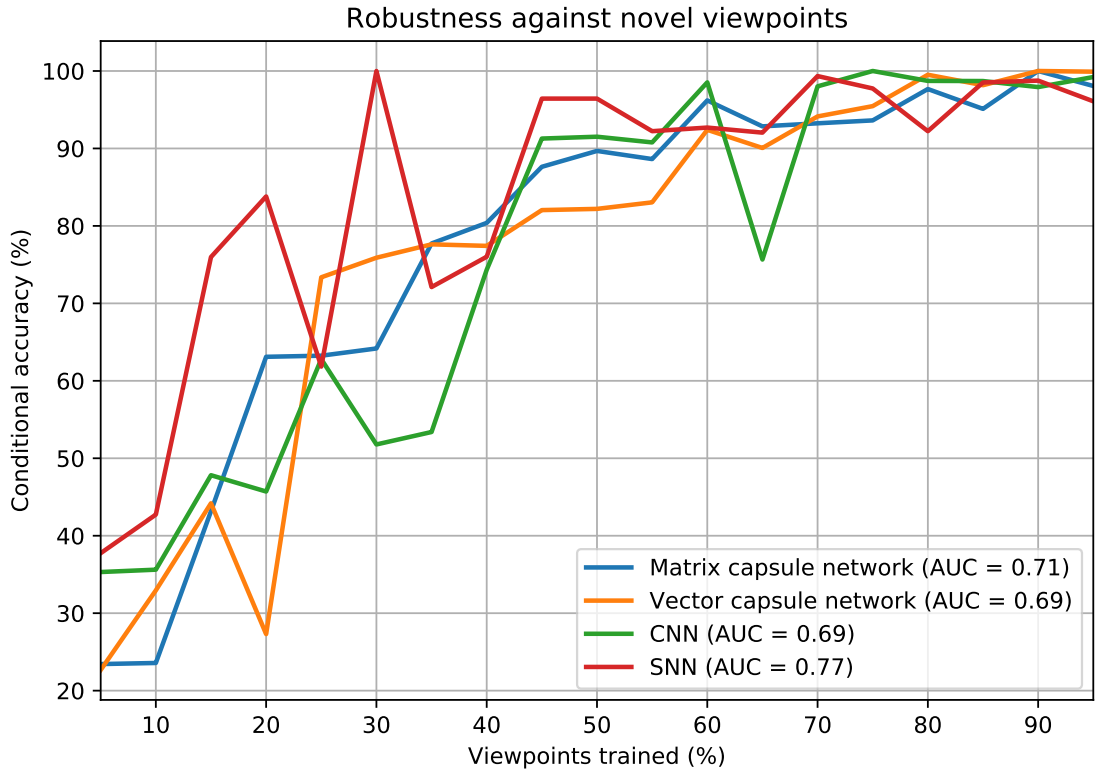


Figure 5.13: Generalization of the networks in the presence of untrained viewpoints. There is quite a bit of fluctuation in the networks' conditional accuracy curve, whereas the matrix capsule's curve seems the most stable. The SNN particularly has a few spikes that give it the highest AUC. But other than that, no network is able to set itself apart. Note that as more viewpoints are included in the training, the more the curves converge to 100 %, i.e. unperturbed performance.

6 Discussion

6.1 Results

6.2 Conclusion

6.3 Outlook

List of Figures

List of Algorithms

List of Tables

Bibliography

[1] J. L. Krichmar. "Neurorobotics—A Thriving Community and a Promising Pathway Toward Intelligent Cognitive Robots." In: *Frontiers in Neurorobotics* 12 (2018).

[2] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[4] L. Paulun, A. Wendt, and N. K. Kasabov. "A retinotopic spiking neural network system for accurate recognition of moving objects using NeuCube and dynamic vision sensors." In: *Frontiers in Computational Neuroscience* 12 (2018), p. 42.

[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[6] E. Falotico, L. Vannucci, A. Ambrosano, U. Albanese, S. Ulbrich, J. C. Vasquez Tieck, G. Hinkel, J. Kaiser, I. Peric, O. Denninger, et al. "Connecting artificial brains to robots in a comprehensive simulation framework: The neurorobotics platform." In: *Frontiers in neurorobotics* 11 (2017), p. 2.

[7] A. Knoll, F. Röhrbein, A. Kuhn, M. Akl, and K. Sharma. "Neurorobotics." In: *Informatik-Spektrum* 40.2 (2017), pp. 161–164.

[8] A. Diba, V. Sharma, A. M. Pazandeh, H. Pirsiavash, and L. V. Gool. "Weakly Supervised Cascaded Convolutional Networks." In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 5131–5139.

[9] W. Ouyang, X. Zeng, X. Wang, S. Qiu, P. Luo, Y. Tian, H. Li, S. Yang, Z. Wang, H. Li, K. Wang, J. Yan, C. C. Loy, and X. Tang. "DeepID-Net: Object Detection with Deformable Part Based Convolutional Neural Networks." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.7 (July 2017), pp. 1320–1334. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2016.2587642.

[10] A. J. Schofield, I. D. Gilchrist, M. Bloj, A. Leonardis, and N. Bellotto. "Understanding images in biological and computer vision." In: *Interface Focus* 8.4 (2018). ISSN: 2042-8898. DOI: 10.1098/rsfs.2018.0027. eprint: http://rsfs.royalsocietypublishing.org/content/8/4/20180027.full.pdf.

[11] D. Marr. "Early processing of visual information." In: *Phil. Trans. R. Soc. Lond. B* 275.942 (1976), pp. 483–519.

[12] M. Bar. "A cortical mechanism for triggering top-down facilitation in visual object recognition." In: *Journal of cognitive neuroscience* 15.4 (2003), pp. 600–609.

[13] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. 2009.

[14] X. Glorot, A. Bordes, and Y. Bengio. "Deep sparse rectifier neural networks." In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[15] G. Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. DOI: https://doi.org/10.1007/BF02551274.

[16] K. Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural Networks* 4.2 (1991), pp. 251–257. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90009-T.

[17] Y.-L. Boureau, J. Ponce, and Y. Lecun. "A Theoretical Analysis of Feature Pooling in Visual Recognition." In: *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*. Nov. 2010, pp. 111–118.

[18] D. Scherer, A. Müller, and S. Behnke. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition." In: *Artificial Neural Networks – ICANN 2010*. Ed. by K. Diamantaras, W. Duch, and L. S. Iliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101. ISBN: 978-3-642-15825-4.

[19] H. D. H. and W. T. N. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex." In: *The Journal of Physiology* 160.1 (), pp. 106–154. DOI: 10.1113/jphysiol.1962.sp006837. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1962.sp006837.

[20] K. Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position." In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251.

[21] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning." In: *nature* 521.7553 (2015), p. 436.

[22] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. Chap. 9. eprint: http://www.deeplearningbook.org/contents/convnets.html.

[23] M. Oquab, L. Bottou, I. Laptev, and J. Sivic. "Is object localization for free? - Weakly-supervised learning with convolutional neural networks." In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 685–694. DOI: 10.1109/CVPR.2015.7298668.

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. "Going deeper with convolutions." In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[25] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks." In: *CVPR*. Vol. 1. 2. July 2017, p. 3.

[26] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. "Inception-v4, inception-resnet and the impact of residual connections on learning." In: *AAAI*. Vol. 4. 2017, p. 12.

[27] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. "Aggregated residual transformations for deep neural networks." In: *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. IEEE. 2017, pp. 5987–5995.

[28] L. Wang, Y. Xiong, Z. Wang, and Y. Qiao. "Towards good practices for very deep two-stream convnets." In: *arXiv preprint arXiv:1507.02159* (2015).

[29] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[30] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. "Regularization of Neural Networks using DropConnect." In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, June 2013, pp. 1058–1066.

[31] A. Krogh and J. A. Hertz. "A simple weight decay can improve generalization." In: *Advances in neural information processing systems*. 1992, pp. 950–957.

[32] N. K. Treadgold and T. D. Gedeon. "Simulated annealing and weight decay in adaptive learning: The SARPROP algorithm." In: *IEEE Transactions on Neural Networks* 9.4 (1998), pp. 662–668.

[33] M. D. Zeiler. "ADADELTA: an adaptive learning rate method." In: *arXiv preprint arXiv:1212.5701* (2012).

[34] J. Duchi, E. Hazan, and Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[35] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[36] B. T. Polyak and A. B. Juditsky. "Acceleration of stochastic approximation by averaging." In: *SIAM Journal on Control and Optimization* 30.4 (1992), pp. 838–855.

[37] A. Graves. "Generating sequences with recurrent neural networks." In: *arXiv preprint arXiv:1308.0850* (2013).

[38] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. "On the importance of initialization and momentum in deep learning." In: *International conference on machine learning*. 2013, pp. 1139–1147.

[39] I. Loshchilov and F. Hutter. "Sgdr: Stochastic gradient descent with warm restarts." In: *arXiv preprint arXiv:1608.03983* (2016).

[40] J. L. Ba, J. R. Kiros, and G. E. Hinton. "Layer normalization." In: *arXiv preprint arXiv:1607.06450* (2016).

[41] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In: *arXiv preprint arXiv:1502.03167* (2015).

[42] T. Salimans and D. P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." In: *Advances in Neural Information Processing Systems*. 2016, pp. 901–909.

[43] D. Worrall and G. Brostow. "CubeNet: Equivariance to 3D Rotation and Translation." In: *arXiv preprint arXiv:1804.04458* (2018).

[44] D. E. Worrall, S. J. Garbin, D. Turmukhambetov, and G. J. Brostow. "Harmonic networks: Deep translation and rotation equivariance." In: *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. Vol. 2. 2017.

[45] U. Schmidt and S. Roth. "Learning rotation-aware features: From invariant priors to equivariant descriptors." In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 2050–2057.

[46] T. Cohen and M. Welling. "Group equivariant convolutional networks." In: *International conference on machine learning*. 2016, pp. 2990–2999.

[47] F. E. Bloom. "Chapter 1 - Fundamentals of Neuroscience." In: *Fundamental Neuroscience (Fourth Edition)*. Ed. by L. R. Squire, D. Berg, F. E. Bloom, S. du Lac, A. Ghosh, and N. C. Spitzer. Fourth Edition. San Diego: Academic Press, 2013, pp. 3–13. ISBN: 978-0-12-385870-2. DOI: https://doi.org/10.1016/B978-0-12-385870-2.00001-9.

[48] A. H. Bell, L. Pessoa, R. B. Tootell, and L. G. Ungerleider. "Chapter 44 - Visual Perception of Objects." In: *Fundamental Neuroscience (Fourth Edition)*. Ed. by L. R. Squire, D. Berg, F. E. Bloom, S. du Lac, A. Ghosh, and N. C. Spitzer. Fourth Edition. San Diego: Academic Press, 2013, pp. 947–968. ISBN: 978-0-12-385870-2. DOI: https://doi.org/10.1016/B978-0-12-385870-2.00044-5.

[49] J. R. Manns and E. A. Buffalo. "Chapter 48 - Learning and Memory: Brain Systems." In: *Fundamental Neuroscience (Fourth Edition)*. Ed. by L. R. Squire, D. Berg, F. E. Bloom, S. du Lac, A. Ghosh, and N. C. Spitzer. Fourth Edition. San Diego: Academic Press, 2013, pp. 1029–1051. ISBN: 978-0-12-385870-2. DOI: https://doi.org/10.1016/B978-0-12-385870-2.00048-2.

[50] C. Koch. "Chapter 51 - The Neuroscience of Consciousness." In: *Fundamental Neuroscience (Fourth Edition)*. Ed. by L. R. Squire, D. Berg, F. E. Bloom, S. du Lac, A. Ghosh, and N. C. Spitzer. Fourth Edition. San Diego: Academic Press, 2013, pp. 1091–1103. ISBN: 978-0-12-385870-2. DOI: https://doi.org/10.1016/B978-0-12-385870-2.00051-2.

[51] J. H. Byrne. "Chapter 47 - Learning and Memory: Basic Mechanisms." In: *Fundamental Neuroscience (Fourth Edition)*. Ed. by L. R. Squire, D. Berg, F. E. Bloom, S. du Lac, A. Ghosh, and N. C. Spitzer. Fourth Edition. San Diego: Academic Press, 2013, pp. 1009–1027. ISBN: 978-0-12-385870-2. DOI: https://doi.org/10.1016/B978-0-12-385870-2.00047-0.

[52] A. Knoll and M.-O. Gewaltig. "Neurorobotics: a strategic pillar of the Human Brain Project." In: *Science Robotics* (2016).

[53] D. Drubach. *The Brain Explained*. Prentice-Hall, 2000.

[54] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve." In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[55] A. N. Burkitt. "A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input." In: *Biological cybernetics* 95.1 (2006), pp. 1–19.

[56] S. M. Bohte, H. La Poutré, and J. N. Kok. "Unsupervised clustering with spiking neurons by sparse temporal coding and multilayer RBF networks." In: *IEEE Transactions on neural networks* 13.2 (2002), pp. 426–435.

[57] S. M. Bohte. "The evidence for neural information processing with precise spike-times: A survey." In: *Natural Computing* 3.2 (2004), pp. 195–206.

[58] J. J. Hopfield. "Pattern recognition computation using action potential timing for stimulus representation." In: *Nature* 376.6535 (1995), p. 33.

[59] W. Gerstner and W. M. Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[60] W. Maass. "Networks of spiking neurons: the third generation of neural network models." In: *Neural networks* 10.9 (1997), pp. 1659–1671.

[61] F. Rieke and D. Warland. *Spikes: exploring the neural code*. MIT press, 1999.

[62] P. U. Diehl and M. Cook. "Unsupervised learning of digit recognition using spike-timing-dependent plasticity." In: *Frontiers in computational neuroscience* 9 (2015), p. 99.

[63] D. Hebbs. "The organization of behavior." In: *Wiely and Sons, New York, NY, USA* (1949).

[64] S. Song, K. D. Miller, and L. F. Abbott. "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity." In: *Nature neuroscience* 3.9 (2000), p. 919.

[65] Z. Bing, C. Meschede, F. Röhrbein, K. Huang, and A. C. Knoll. "A Survey of Robotics Control Based on Learning-Inspired Spiking Neural Networks." In: *Frontiers in neurorobotics* 12 (2018), p. 35.

[66] S. Bartunov, A. Santoro, B. A. Richards, G. E. Hinton, and T. Lillicrap. "Assessing the scalability of biologically-motivated deep learning algorithms and architectures." In: *arXiv preprint arXiv:1807.04587* (2018).

[67] J. H. Lee, T. Delbruck, and M. Pfeiffer. "Training Deep Spiking Neural Networks Using Backpropagation." In: *Frontiers in Neuroscience* 10 (2016), p. 508. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00508.

[68] P. Panda and K. Roy. "Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition." In: *arXiv preprint arXiv:1602.01510* (2016).

[69] M. Hopkins, G. Pineda-García, P. A. Bogdan, and S. B. Furber. "Spiking neural networks for computer vision." In: *Interface Focus* 8.4 (2018), p. 20180007.

[70] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell. "Understanding data augmentation for classification: when to warp?" In: *arXiv preprint arXiv:1609.08764* (2016).

[71] L. Perez and J. Wang. "The effectiveness of data augmentation in image classification using deep learning." In: *arXiv preprint arXiv:1712.04621* (2017).

[72] C.-S. Lee, M.-H. Wang, S.-J. Yen, T.-H. Wei, I.-C. Wu, P.-C. Chou, C.-H. Chou, M.-W. Wang, and T.-H. Yan. "Human vs. Computer Go: Review and Prospect [Discussion Forum]." In: *Comp. Intell. Mag.* 11.3 (Aug. 2016), pp. 67–72. ISSN: 1556-603X. DOI: 10.1109/MCI.2016.2572559.

[73] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. "Mastering the game of Go with deep neural networks and tree search." In: *nature* 529.7587 (2016), p. 484.

[74] Y. Dong, Y. Wang, Z. Lin, and T. Watanabe. "High performance and low latency mapping for neural network into network on chip architecture." In: *ASIC, 2009. ASICON'09. IEEE 8th International Conference on*. IEEE. 2009, pp. 891–894.

[75] J. W. Tanaka and D. Simonyi. "The "parts and wholes" of face recognition: A review of the literature." In: *The Quarterly Journal of Experimental Psychology* 69.10 (2016), pp. 1876–1889.

[76] G. Indiveri, B. Linares-Barranco, T. J. Hamilton, A. Van Schaik, R. Etienne-Cummings, T. Delbruck, S.-C. Liu, P. Dudek, P. Häfliger, S. Renaud, et al. "Neuromorphic silicon neuron circuits." In: *Frontiers in neuroscience* 5 (2011), p. 73.

[77] T. Stone, M. Mangan, A. Wystrach, and B. Webb. "Rotation invariant visual processing for spatial memory in insects." In: *Interface Focus* 8.4 (2018), p. 20180010.

[78] J. B. Isbister, A. Eguchi, N. Ahmad, J. M. Galeazzi, M. J. Buckley, and S. Stringer. "A new approach to solving the feature-binding problem in primate vision." In: *Interface Focus* 8.4 (2018), p. 20180021.

[79] G. E. Hinton, A. Krizhevsky, and S. D. Wang. "Transforming auto-encoders." In: *International Conference on Artificial Neural Networks*. Springer. 2011, pp. 44–51.

[80] S. Sabour, N. Frosst, and G. E. Hinton. "Dynamic routing between capsules." In: *Advances in Neural Information Processing Systems*. 2017, pp. 3856–3866.

[81] D. P. Kingma and M. Welling. "Auto-encoding variational bayes." In: *arXiv preprint arXiv:1312.6114* (2013).

[82] G. E. Hinton, S. Sabour, and N. Frosst. "Matrix capsules with EM routing." In: (2018).

[83] G. E. Hinton, Z. Ghahramani, and Y. W. Teh. "Learning to parse images." In: *Advances in neural information processing systems*. 2000, pp. 463–469.

[84] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm." In: *Journal of the royal statistical society. Series B (methodological)* (1977), pp. 1–38.

[85] J. Rissanen. "Modeling by shortest data description." In: *Automatica* 14.5 (1978), pp. 465–471.

[86] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. Huang. "A tutorial on energy-based learning." In: *Predicting structured data* 1.0 (2006).

[87] D. K. Lee, L. Itti, C. Koch, and J. Braun. "Attention activates winner-take-all competition among visual filters." In: *Nature neuroscience* 2.4 (1999), p. 375.

[88] Y. LeCun, F. J. Huang, and L. Bottou. "Learning methods for generic object recognition with invariance to pose and lighting." In: *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*. Vol. 2. IEEE. 2004, pp. II–104.

[89] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. "Automatic differentiation in PyTorch." In: *NIPS-W*. 2017.

[90] H. Hazan, D. J. Saunders, H. Khan, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma. "BindsNET: A machine learning-oriented spiking neural networks library in Python." In: *ArXiv e-prints* (June 2018). arXiv: 1806.01423.

[91] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors." In: *arXiv preprint arXiv:1207.0580* (2012).

[92] L. Vannucci, A. Ambrosano, N. Cauli, U. Albanese, E. Falotico, S. Ulbrich, L. Pfotzer, G. Hinkel, O. Denninger, D. Peppicelli, et al. "A visual tracking model implemented on the iCub robot as a use case for a novel neurorobotic toolkit integrating brain and physics simulation." In: *Humanoids*. 2015, pp. 1179–1184.

[93] T. Fawcett. "An introduction to ROC analysis." In: *Pattern recognition letters* 27.8 (2006), pp. 861–874.

[94] F. Provost and P. Domingos. "Well-trained PETs: Improving probability estimation trees." In: (2000).

[95] A. Kraskov, H. Stögbauer, and P. Grassberger. "Estimating mutual information." In: *Physical review E* 69.6 (2004), p. 066138.