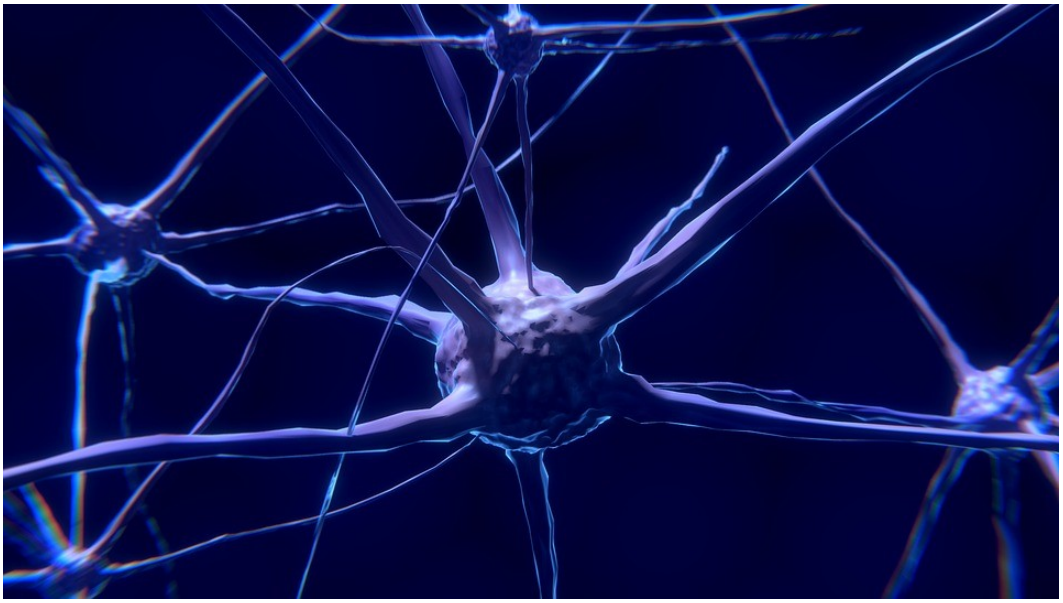

# Supervised Learning with Dynamic Network Architectures

Master's thesis in Computer science - Algorithms, Language and Logic

Herman Carlström

Filip Slottner Seholm

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

# Supervised Learning With Dynamic Network Architectures

HERMAN CARLSTRÖM
FILIP SLOTTNER SEHOLM

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Supervised Learning With Dynamic Network Architectures
HERMAN CARLSTRÖM
FILIP SLOTTNER SEHOLM



Supervisor: Claes Strannegård, Computer Science – algorithms, languages and logic
Examiner: Morteza Haghir Chehreghani, Computer Science – algorithms, languages and logic

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A neuron, representing the building block of the brain and a neural network.

Gothenburg, Sweden 2019

Supervised Learning With Dynamic Network Architectures
HERMAN CARLSTRÖM
FILIP SLOTTNER SEHOLM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

There exists many different techniques for training neural networks, but few are designed to handle the case of lifelong learning. Most models are based on the assumption that there is a training phase with a finite amount of data. This thesis investigates and evaluates a brand new algorithm, namely the *Lifelong Learning starting from zero (LL0)* which can be used for lifelong learning where there is a continuous stream of data. The algorithm stems from biology, logical rules, and machine learning. The algorithm builds a dynamic artificial neural network architecture over time, based on four different concepts; extension, generalization, forgetting and backpropagation. These first three concepts all have their origin in biology, and can be found in animals and humans in the form of neuroplasticity.

The model is evaluated and benchmarked against five other models on different datasets and problems. The obtained results act as a proof of concept for the algorithm. Lastly, the pros and cons of the model are discussed, followed by a discussion on future work. The model proposed outperforms all models on chosen benchmarks, primarily in the area of one-shot learning. The model manages to achieve about 90% accuracy on unseen data after only training on a small portion of the training set on multiple datasets. LL0 also shows promising results in the area of lifelong learning.

Keywords: Machine Learning, Neural Networks, Dynamic Architectures, Supervised Learning, Lifelong Learning, One-Shot Learning, Transfer Learning, Computer Science

# Acknowledgements

We would like to thank our supervisor Claes Strannegård for the possibility to do this thesis, as well as the continous feedback that he has been providing during the entire project. We also want to thank Niklas Engsner and Fredrik Mäkeläinen for the discussions regarding the model, as well as the feedback we have received to update the model.

We further want to thank our examiner Morteza Haghir Chehreghani for his very thorough ideas on how to scope the project and make it doable.

Finally we would like to thank our friends and families for the encouragement and support throughout the process, with an extra big thanks to the study group Freges.

Herman Carlström & Filip Slottner Seholm, Gothenburg, June 2019

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

Humans and animals live in an ever-changing environment. No situation is *exactly* like another, and we must constantly learn and adapt. This is something that has been observed in all kinds of animals [28]. It is quite understandable why there is a necessity for learning new information, but the capability to *forget* information has proven useful by removing outdated memories [36].

In later years there has been a great breakthrough in the area of Artificial Intelligence (AI). AI has beaten humans in games, made medical diagnoses and improved the energy efficiency in data centers to name a few discoveries [41, 13, 1]. These improvements are in large due to the technical innovations of neural networks.

While neural networks can achieve better results than humans on some tasks [19], it often comes at the price of time. For instance, teaching neural networks to recognize pictures better than humans might requires weeks of training [20]. This introduces several drawbacks; the time that developers have to wait to get results, the difficulty of testing different approaches, and the need for specialized computers with multiple GPU's - which consume an extensive amount of energy, to name a few. Training neural networks also requires annotated data, something that might be hard and expensive to come by [11]. As neural networks grow more popular, there is a big need for improvements in this field - something this thesis hopes to contribute to.

There are a lot of different problems and domains where neural networks can be utilized to solve different problems. An example of one of those problems is an autonomous vehicle that needs to adapt and make decisions based on its environment. The vehicle needs to continuously learn new information, while simultaneously utilizing and retaining previously known knowledge. Problems that require this type of learning are in the domain of *lifelong learning* or *continual learning* [34]. One of the key difficulties with lifelong learning is to avoid *catastrophic forgetting* - when newly learned knowledge overwrites existing knowledge. Problems within the lifelong learning domain has introduced multiple long-standing challenges [46]. Algorithms that are designed to solve continual learning tasks are therefore of high importance.

One of the key Capabilities of the brain, which still is missing from neural networks, is *neuroplasticity*. Neuroplasticity is the ability to exist in an ever changing environment by altering the nervous system [35]. Neuroplasticity occurs when new neurons are added to the brain [23, 9], new connections are made between neurons [15, 8] or

when neurons are removed - mainly due to apoptosis, cell death [31]. Incorporating the idea behind neuroplasticity into neural networks and AI, could potentially bring us closer to a solution [18].

This thesis proposes to implement, tweak and investigate the performance of a completely new training algorithm *Lifelong Learning starting from Zero (LL0)*. The LL0 model makes use of a dynamic neural network structure, building a network over time. The new algorithm [44] is a different approach to the technique seen in [43], developed by, among others, Claes Strannegård at Chalmers University of Technology. The new algorithm also aims to focus on one-shot learning, an approach that requires less data to train neural networks. Neuroplasticity can easily be translated into a neural network by allowing the network to add new nodes on demand, add connections - weights, between nodes that did not exist before and by removing obsolete nodes. Neuroplactisity of several different forms lies as the core foundation of the algorithm, and is the main inspiration of the method.

# 2

# Theory

This section aims to give enough background information to the different techniques used and discussed in this thesis. The section aims to provide an overview over neural networks and its key properties, tasks within neural networks, and different approaches to lifelong learning, one-shot learning and transfer learning.

## 2.1 Neural Networks

An *artificial neural network* is a rough mathematical model of a natural nervous system. It is represented as a graph structure, consisting of nodes and edges that connect the nodes in different layers. The nodes represent the *neurons* in the brain, and the edges represent the *connections* between the neurons, which in the brain corresponds to *synapses*.

Fully-connected networks, which have the structure of a *Directed Acyclic Graph - DAG*, are most common. A DAG is a finite directed graph with no cycles. In a *fully-connected network*, all nodes on layer $l$ are connected with an edge to all nodes on the next layer, $l+1$. More precisely each node $n_i^{(l)}$ in layer $l$ with identifier $i$, have connections $e = (i, j)$ with weight $w_{i,j}$ to all $j \in Layer(l+1)$. $Layer(x)$ specifies the indices to all nodes on layer $x$.

A node higher up in the hierarchy are often called *parents* to the nodes it is connected to further down the hierarchy. These nodes further down the hierarchy are in contrast called *children* to the ones further up. A fully-connected network can be seen in Figure 2.1.

Neural networks are often used to give predictions given specific inputs. These predictions can either be in regards to *classification*, which is explained in Section 2.2, or in tasks such as *regression* - which is not within the scope for this thesis.

A feed forward neural network is given its structure in order to create a function $NET(x)$ which solves some task, for instance a classification task. By feeding input $x$ to the network, an output $y = NET(x)$ is calculated by the network. Calculating the output is called *forward propagation* or a *forward pass*. During a forward pass a neuron $n_i^{(l)}$ in the network calculates the *weighted sum* of its incoming edges.

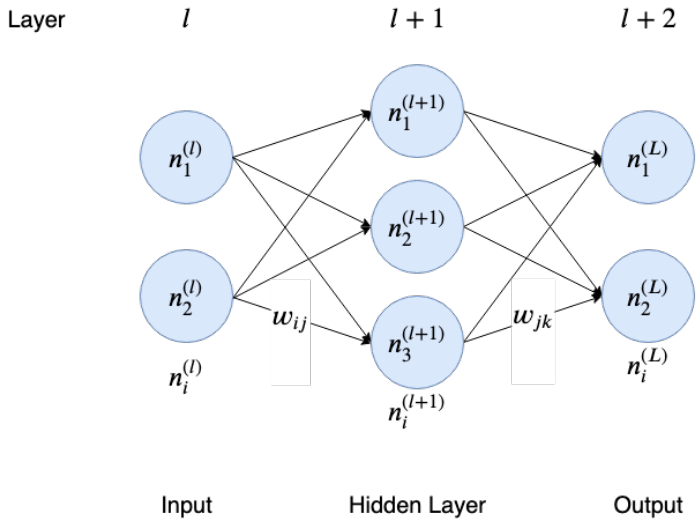


**Figure 2.1: A visualization of a fully-connected neural network.** Each node on layer $l$ is connected to the nodes on layer $l+1$, by weights $w_{ij}$. $w_{ij}$ means that the weight goes from node $i$ to node $j$. The first layer is the input layer, and the last layer is the output layer. The nodes $n_1^{(l)}$ and $n_2^{(l)}$ are parents to nodes $n_1^{(l+1)}$, $n_2^{(l+1)}$ and $n_3^{(l+1)}$. Nodes $n_1^{(L)}$ and $n_2^{(L)}$ are children to nodes $n_1^{(l+1)}$, $n_2^{(l+1)}$ and $n_3^{(l+1)}$ and so forth.

The weighted sum $s_i^{(l)} = \sum_j y_j^{(l-1)} w_{j,i}$, $j \in Layer(l-1)$ is the sum of the neurons incoming values multiplied by the weight on the edge between the neuron and its parents. Each neuron also has a bias $\theta_i^{(l)}$ which is added to the weighted sum. Therefore the first step of each neuron is to calculate $z_i^{(l)} = s_i^{(l)} + \theta_i^{(l)}$. The weighted sum will in this thesis be referred to as $z_i^{(l)}$ which includes the bias term.

After a neuron has calculated its weighted sum $z$, it calculates $f(z)$, where $f(\cdot)$ is the *activation function*. The activation function is typically a function that squashes the weighted sum to keep the values through the network in a reasonable interval. Different activation functions are explained in Section 2.3. The *output* or *activation* of the neuron $y = f(z)$ is sent out on all of the neuron's outgoing edges to its children, who calculate their weighted sum. This process continues throughout the entire network, layer by layer, until the output nodes calculate the result of the network.

For the network to be meaningful it has to be trained to learn a task. The network is trained on a dataset $D = (X, T)$, where $X$ is a set of inputs and $T$ is a set of labels. The network is trained through the process of *backpropagation*. Backpropagation, or a *backward pass*, is the process of going backwards in the network, from the output layer to the input layer. During the backwards pass the effect that each parameter had on the error is calculated, so that these parameters can be tuned to minimize the error, and therefore learn the dataset. For a datapoint $(x, t)$ an error is calculated by some error function $E(NET(x), t)$ in order to evaluate the impact of each parameter. The impact is calculated by the derivative of the error. The

error function is further explained in Section 2.4, together with backpropagation. The parameters that are updated in the network are all the weights, as well as the biases.

The forwardpropagation and the backpropagation are used iteratively during the *training phase*, where the network is trained to learn the dataset. During the training phase, the parameters are tweaked to make better predictions. This typically requires the dataset to be passed through the network multiple times. One forward pass and one backward pass for every datapoint in the dataset is typically referred to as one *epoch*. The training period for a task is typically specified by the amount of epochs a network should run.

Most neural network models require an immense amount of labeled data to train on. One common saying is that your model is only as good as your dataset, which raises an issue with many of the existing models - there just is not enough data. Another big issue is the amount of energy consumed by the neural networks. With many layers and many nodes in each layer, each iteration of the network is computationally heavy. As the network often needs vast amounts of data, and a large number of iterations for each datapoint, the energy consumed quickly accumulates towards very high numbers.

Even though the feed forward neural network is the simplest form of a neural network it can be extremely powerful. If given the correct structure, and the time to tune its parameters, a feed forward neural network can approximate almost any continuous function [22].

## 2.2 Classification

*Classification* is the task of classifying elements to groups. A very simple example is by imagining a fruit basket with a handful of fruits. One datapoint corresponds to one fruit, and we classify each element separately to the correct fruit. A banana is not an apple, and so forth.

In the field of machine learning and neural networks, classification is the task to correctly map inputs to labels. More specifically, given an input $x$, assign to $x$ a *label* $t$ such that $t \in T$, where $T$ denotes all possible labels for the task. Classification tasks within neural networks have a widespread area of applications. They range from predicting diseases in humans [49] to correctly identifying and labeling objects in of autonomous driving [7].

During the training phase of a neural network, a variety of inputs $x \in X$ with corresponding labels $t$ are fed to the network. A prediction of the correct label $y$ is made, and an error estimating how far away the estimated label $y$ is to $t$ is calculated. How the error can be calculated will be discussed in Section 2.4. The network is adjusted by updating the different parameters by the rules of backpropagation, which also will be explained more in detail in Section 2.4.

The labels are represented by a one-hot encoded vector. This means that all labels are converted from their initial integer into a binary vector, where the position $i$ of the vector represents the integer. For instance, if we one-hot encode the labels 1,2 and 3, we would get the corresponding three vectors; [1,0,0], [0,1,0] and [0,0,1]. The main reason why one would want to use one-hot encoding is its ability to represent probabilities. If the machine learning technique used makes sure that the output of a prediction represents probabilities for each class, we get a vector of probabilities which sums to one. This is an output which is easy to understand and gives more information than just which label the network predicted.

To validate how well a network estimates the classification labels, one simply measures the ratio of how many correct predictions the network has made.

$$y_{correct} = \begin{cases} 1 & \text{if } argmax(t_i) = argmax(y_i) \\ 0 & \text{otherwise} \end{cases}$$

$$Accuracy = \frac{1}{n}\sum_{i=1}^{n} y_{correct}$$

Where $argmax(\cdot)$ returns the index of the maximum value in a vector.

One key element when validating the networks performance is to estimate the accuracy on inputs the network has not seen before, i.e. trained on. Therefore it is very common to divide the data set into three separate parts; a *training*, *validation* and *test set*. The network is trained on the training-set, and continuously validated on the validation set. When the accuracy on the validation set is deemed as good enough, or the network has started to *overfit* on the training data, the training is cancelled. The accuracy on the test-set is the accuracy that the model of the current network is appointed. One of the main reasons to use the technique of splitting a data set into three parts is to avoid aforementioned overfitting. Overfitting occurs when a network has been trained on a finite number of inputs for too long. For each input, the network gets tweaked to more precisely remember the input, and thus becomes tailored to the training set to the extent that it does not recognize unseen inputs. Overfitting is usually also called poor generalization. On the contrary, when overfitting is avoided and the network recognizes unseen data, it is usually said that the network generalizes well.

All the possible parameters that are updated during the training phase of a network, are what make up a neural network; a configuration of parameters that are connected to each other in a graph-like structure to solve a task.

## 2.3 Activation Functions

There are several different activation functions used in an artificial neural network, usually in order to squash the output of the neurons. One wants to squash the

output of a neuron so its output stays within a reasonable interval. Their use in forwardpropagation was explained in section 2.1. The activation functions used in this thesis either squashes their input in the interval $[0, 1]$ or not at all. However, there exist other activation functions that squashes in other intervals. Another important requirement of the activation functions is that they have to be differentiable. This is mainly due to the fact that backpropagation needs to calculate the derivatives of the activation functions, which is further explained in section 2.4. Some activation functions can also have different parameters which specify different characteristics of the function.

### 2.3.1 Identity Activation Function

The *identity activation function* is the simplest activation function out of all the activation functions. It does not squash the input at all, but instead only propagates the same value forward. The identity function is defined as:

$$identity(x) = x \tag{2.1}$$

This activation function can be used if some neuron in a network does not want to affect the output from the weighted sum. A visualization of the identity activation function can be observed in Figure 2.2

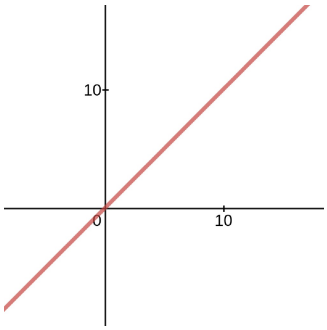


**Figure 2.2: The identity activation function** $identity(x) = x$ which can be used in a neuron.

### 2.3.2 Sigmoid Activation Function

The *sigmoid activation function*, also referred to as *sigmoid* in this thesis, is quite commonly used in neural networks. It squashes the input within the interval $[0, 1]$ in a smooth $s$-like curve, as can be observed in figure 2.3. The two different sigmoid curves in the figure have different multipliers to its inputs which results in the

different characteristics of the curves. The sigmoid function is defined as:

$$simgoid(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

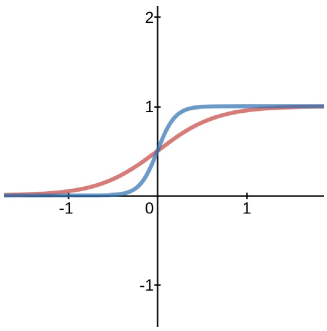


**Figure 2.3: The sigmoid activation function** $simgoid(x) = \frac{1}{1+e^{-x}}$. The two different functions show how the function can be shaped with different parameters. The blue function plots $sigmoid(10x)$ and the red function plots $sigmoid(3x)$.

### 2.3.3 Gaussian Activation Function

The *gaussian activation function*, also refered to as *gaussian* in this thesis, is not as commonly used in neural networks. This is mainly due to the fact that it has one additional tuning parameter than more commonly used activation functions. This extra parameter is important in order to form the function during backpropagation, however, it increases the computational complexity of backpropagation by one additional parameter. The gaussian activation function is heavier to calculate as the function is more complex than most other activation functions. The gaussian function is defined as:

$$gaussian(x) = exp\Big( - \frac{(x - \mu)^2}{2\sigma^2} \Big) \tag{2.3}$$

Where $\mu$ decides the center of the activation function. The center of the gaussian function is the x-value where the function has its maximum value.

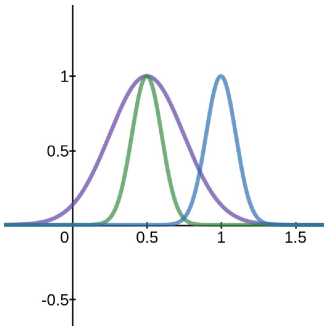


**Figure 2.4: The gaussian activation function** $gaussian(x) = exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big)$. The three different functions show how the function can be shaped based on the parameters $\sigma$ and $\mu$. The purple function has $\sigma = 0.25$ and $\mu = 0.5$, the green function has $\sigma = 0.1$ and $\mu = 0.5$, and the blue function has $\sigma = 0.1$ and $\mu = 1.0$.

One of the big advantages of the gaussian activation function lies in its symmetric characteristic. The function is symmetric around its center, which makes it possible for the function to model the distance from its center with a smooth differentiable curve. This is an important property which is utilized later in this thesis. This can be compared to the identity and sigmoid activation functions which do not have the property of naturally calculating a distance from a certain point.

### 2.3.4 Softmax Activation Function

The *softmax activation function*, also referred to as *softmax* in this thesis, usually only occur in the neurons of the last layer of the network, the output layer. Each softmax neuron typically represents one class, and its output represent the probability of that class being the correct one. This gives the softmax activation function a dependency between all softmax neurons since the probabilities must sum to one. It follows that if there are a lot of softmax neurons, the calculation of softmax can be very computationally heavy. The softmax function is defined as:

$$softmax(z_i^{(L)}) = \frac{e^{z_i^{(L)}}}{\sum_j e^{z_j^{(L)}}} \tag{2.4}$$

Where $z_i^{(L)}$ denotes the weighted sum (including bias) for the $i$'th softmax neuron. $j$ denotes the identifiers for all softmax neurons. An example for the softmax activation function can be illustrated in a network solving a classification task. The task is to predict the correct berry out of three possible, given a couple of inputs. There are three output neurons, and they recieve the following input: $(z_0^{(L)}, z_1^{(L)}, z_2^{(L)}) = (2, 1, 0.1)$. Softmax neuron 0 models the class blueberry, neuron 1 models the class raspberry, and neuron 3 models the class lingonberry. Applying the softmax function will give the output $(0.66, 0.24, 0.1)$, which represents the

probability for each class being the correct class. This output is easier to understand and gives more information than just which label the network predicted. The most common way of choosing a class based on its probabilities is by the argmax function. Using probabilities also enables the use of the cross-entropy error function, which will be explained in the next section.

## 2.4 Backpropagation

In order for the network to make correct predictions, the parameters in the network has to be tuned, a task that is handled in the training phase. For a datapoint $d = (x, t)$, where $x$ is the input and $t$ is the correct label, the parameters are tuned in order to make the output $y = NET(x)$ of the network match the correct label $t$. In order to change the parameters, the gradient of an error function with respect to each parameter needs to be calculated. This is systematically done from the output nodes, layer by layer, until the input nodes are reached. In each layer the gradient of the error with respect to the parameters in that layer are calculated. Calculating the gradient in this fashion is called *backpropagation* [21].

One of the commonly used error functions for classification, together with a softmax layer, is the *cross-entropy* error function [26]. The cross-entropy calculates an error based on the true probability distribution, which is the distribution the network should model. In this setting the true distribution is the target label $t$. The error function compares the true probability distribution with the output of the network, calculating an error to minimize. The cross-entropy error is defined as:

$$E = -\sum_{i}^{D} = t_i log(y_i) \tag{2.5}$$

Where $D$ denotes the output dimension.

Once the error has been calculated, the next step in backpropagation is to calculate the gradients of the error with respect to all the parameters. The backpropagation is defined as a recursion, where the recursive step goes through each layer $l$. In each recursive step the chain rule is used to calculate the gradients $\frac{\partial E}{\partial p_i^{(l)}}$ for all the relevant parameters $p_i^{(l)}$ with identifier $i$ in layer $l$. The relevant parameters are typically the bias for each neuron, all weights between the layers, as well as any parameters for the activation functions.

Once the derivatives have been calculated with respect to all the parameters, the backpropagation step is completed. However, one last step remains in order to tune the parameters. Through an optimizer the parameters are updated in the negative direction of the gradient in order to get closer to a minima of the error function. The optimizer used in this thesis, and one of the most common optimizers, is the *gradient descent algorithm*. The gradient descent algorithm updates the parameters

in the negative direction of the gradient, multiplied by a *learning rate* $\delta$. Hence the update rule is defined as:

$$p_i^{(l)} \leftarrow p_i^{(l)} - \delta \frac{\partial E}{\partial p_i^{(l)}} \tag{2.6}$$

An approach that is commonly used to both speed up the training and get a more stable training path is *batch training*. Batch training is similar to regular *online training* which is described above, except for when it comes to updating the parameters. In batch training, the parameters are updated with accumulated gradients instead of updating each datapoint separately. A *batch size* is specified which decides the number of gradients to accumulate before the gradient descent step is applied on the accumulated gradients.

The backpropagation and gradient descent algorithms are applied multiple times in order to train the network over many epochs, where each step hopefully brings the the network closer to the optimal solution.

## 2.5 Lifelong learning

*Lifelong learning* is one of the most central problems to solve in the area of machine learning. Today, most models are based on the assumption that there is one single training phase with a finite amount of training data. Lifelong learning is a learning mechanism in which the training phase is continuous, and the data is potentially infinite.

A Lifelong learning system is an adaptive algorithm which adapts to a continuous stream of data where the number of tasks is not predefined [33]. For example, the number of classes to classify in a classification task can change over time. This often leads to more complex networks being evolved over time as the network learns more tasks. Lifelong learning is one of the main objectives being investigated in regards to the dynamic algorithm of this thesis.

While dynamic networks might be understandable for the purpose of lifelong learning, the process of starting small and dynamically expanding the network has also shown great promise in networks that do not focus on lifelong learning [42]. Another benefit of using this approach is that a trial and error phase of finding a good topology for the network is removed and done automatically.

Avoiding, or by any means reducing, *catastrophic forgetting* is critical to achieve lifelong learning. Catastrophic forgetting is when new knowledge overwrites old knowledge in the network. *Catastrophic interference* occurs when training new tasks interfere with already learned and stored information. Catastrophic forgetting occurs if the interference is so severe that previous tasks are forgotten. Combining the avoidance of catastrophic forgetting with a lifelong learning process is one of

the major difficulties with lifelong learning algorithms [12]. Catastrophic forgetting can quite simply be avoided by retraining the entire network when a new class is introduced in a classification task. However this is very ineffective and a need for specialized algorithms that focus on solving catastrophic forgetting is required [24]. Minimizing catastrophic forgetting, or at least catastrophic inference, is the focus and a necessity for all lifelong learning algorithms.

## 2.6 Transfer Learning

When catastrophic forgetting is avoided, previously learned tasks are preserved. These tasks can be used to more efficiently learn new tasks. We can transfer knowledge from previously learned tasks to help solve a new task which results in an improved convergence speed. This process is known as *transfer learning* [45].

Transfer learning has its roots in the psychological field, and stems from humans. We can utilize existing knowledge to learn something new, an idea that researchers are trying to incorporate into machine learning. In machine learning, transfer learning is about domains, $D$, and tasks, $T$ [32]. The existing knowledge in the model exists in the source domain - $D_s$, and in the source task - $T_s$. The new information that the model wants to learn lies in the target domain - $D_t$, and the target task - $T_t$. The tasks can be equal, but then the domains must not be - or vice verse. For instance, if we want to learn what a truck is after learning what a car is, the two domains are equal, but the tasks are not. If we instead have trained the model on real cars and wanted to learn what a hand-drawn car is, the domain is different while the tasks are equal. If both domains and tasks are equal, it is simply a normal machine learning problem [32].

Transfer learning can affect three different metrics [47] which are

- A higher starting accuracy
- A steeper learning curve
- A higher final accuracy

A higher starting accuracy implies that the model recognizes more inputs after being trained on a similar task, compared to a model that has not been trained on similar tasks. A steeper learning curve means that it takes fewer iterations to learn the inputs, and fewer iterations to converge. A higher final accuracy simply means that the model performs better with transfer learning than without. If none of these improvements occur, but instead the opposites, negative transfer learning has taken place [47]. The most desired effect is to have a positive transfer learning on tasks that are related to one another, while avoiding negative transfer learning on tasks that are not.

## 2.7 One-shot Learning

Another technique to speed up training is *one-shot learning*. According to [4] humans can recognize between 5,000 and 30,000 different objects. An object is often recognized after only seeing an object a few times. In contrast, a neural network often needs thousands of images to learn a new task, and usually needs to see each image many times. One-shot learning in machine learning attempts to replicate this feature of the human brain, allowing a network to learn new knowledge with only a few images from the training set, instead of thousands.

Using one-shot learning, a neural network can learn tasks with only a small portion of labeled data. Combining this with one-shot learning, a neural network could learn hundreds of task with a minimal training set, and a reduced training time.

# 3

# Related Work

Lifelong learning is one of the harder problems to solve in machine learning and neural networks [18], and many approaches to tackle the issue have been proposed. One of the earliest models is the *cascade-correlation algorithm* [10]. It is a dynamic model, which incrementally adds nodes to the model, one at the time. The algorithm freezes all weights before adding a new node, and then train until convergence when the new node is added. Therefore, when adding a new node, the training will not interfere with the frozen weights, which to some extent mitigate catastrophic inference.

Another approach to create a dynamic model is implemented in *NeuroEvolution of Augmenting Topologies (NEAT)* which has shown great improvements in faster convergence on reinforcement learning tasks [42]. NEAT makes use of genetic algorithms to reach its dynamic structure. They evolve a population of neural networks, where the population is mutated and can create offspring in order to find new successful structures. To find an architecture, NEAT starts minimally, and continuously extends the network until a solution has been found. In [17] an *Evolvable Neural Turing Machine (ENTM)* is proposed. It manages to achieve one-shot learning together with avoiding catastrophic forgetting [30]. This is achieved by creating a *Neural Turing Machine (NTM)*, which uses a neural network as a controller to read and write to an external memory [16]. ENTM further uses the NEAT strategy to find a good architecture for the controlling neural network.

As mentioned earlier, when solving lifelong learning tasks, catastrophic forgetting is an unwanted property. [33] proposes three different approaches to mitigate catastrophic forgetting. The first method is to retrain the whole network while regularizing [25]. The second approach is to expand the network when necessary to represent new tasks [38, 10]. The third method is to use *memory replay.* In [14] memory replay is used by replaying previously sampled data using a short term memory during something they call sleep phases.

Another desired learning property for lifelong learning is transfer learning. In 1997 [37] proposed an agent *CHILD*, which is capable of *Continual, Hierarchical, Incremental Learning and Development*, as one of of the first approaches to transfer learning in the form of incremental learning. At a high abstraction CHILD uses a simple one layered neural network, and adds new nodes whenever useful. This can be whenever the same weight is pulled strongly in different directions during

training. When this occur, a new node is added to compensate by setting the weight correctly in one of the contexts. By adding new units, new knowledge can be learned when needed, while also retaining part of the old knowledge which therefore achieves incremental learning. This way of training also utilizes transfer learning, as the old knowledge is used to learn new tasks.

A more recent approach for transfer learning is the progressive neural network [38]. The progressive network has a growing architecture, where new sub-networks can be added after demand. The addition of the sub-networks is not done automatically, but has to be specified beforehand by dividing the dataset into the seperate tasks. For each task $t$, the progressive networks add a sub-network, such that if there are a total of $k$-many tasks, then the model has $k$-many sub-networks. The old networks are connected to the new network, and uses existing knowledge to learn faster. The progressive network managed to speed up the learning of a robot in a real life environment substantially by transferring knowledge from learning in a simulation [39].

Another successful approach to transfer learning has been to first learn a set of different skills or tasks, which are part of a more complex task, to then combine them into a single network, as proposed in [11]. They model the skills as deep neural networks, called *Deep Skill Networks (DSN)*, and they are trained independently. Once these individual skills are learned, they are incorporated in a *Hierarchical Deep Reinforcement Learning Network (H-DRLN)*. The H-DRLN is trained on the complex task and learns to use the DSNs appropriately to solve it. This approach successfully managed to solve a complex reinforcement learning task in the game Minecraft, something the commonly used DDQN [48] did not manage to do.

Another technique that is important for lifelong learning is one-shot learning. In [27] a siamese network [5] is used to create one-shot learning of an image classification task. This approach differs a lot from most neural networks for classification, as it does not model a datapoint to a class, but instead a *similarity function* of two different datapoints. In this way the siamese network can store a reference datapoint for each class, which new datapoints can be compared with to calculate the similarity. Hence the network can after make useful predictions after seeing just one datapoint. Another approach to one-shot learning is proposed in [40], where an NTM is used. With NTMs ability to use external memory they can quickly encode the relevant information for remembering a new datapoint, without interfering with already stored memory.

The recent publications of *AdaNet* [6] and *Dynamically Expandable Network (DEN)* [50] gives evidence that networks with dynamic architectures are promising in the field of machine learning. AdaNet dynamically extends its structure when appropriate by adding blocks of nodes. These blocks are fully connected to other blocks of nodes. AdaNet has the ability to extend by increasing depth, or making an existing layer wider. This approach has the possibility of generating sparse networks with connections that are not only connected to adjacent layers. DEN takes a slightly different approach. It uses selective retraining for new tasks to try to improve perfor-

mance by tuning important nodes for the given task. If these nodes reach adequate performance, new nodes are added for increased performance. DEN also has the ability to split. It splits if weights have drifted too far away from their initial value. The split makes two copies of the modules that have drifted, giving one of them the initial values. This process will in turn avoid catastrophic forgetting, as learned tasks are never allowed to drift too far away from the original parameters. Another recent approach is proposed in [29], which generates a neural network called *AOGnet* using *AND-OR grammar (AOG)* [51]. The structure is built using three different building blocks; a terminal node, an OR node and an AND node. These building blocks are put on top of trained convolutional layers, something that in the article is referred to as stem. Each node in the network performs a complex transformation function on its input. In the result section of the article, the transformation function is a complex sequence of convolutional layers, batch normalization layers and ReLU layers. The AOGnet is mostly used for computer vision, which is why there is a need for many convolutional layers.

# 4

# The LL0 Model

This section describes the implementation of a lifelong learning algorithm named *Lifelong Learning Starting From Zero (LL0)* in detail. It begins with a broad overview of the algorithm, and continues with a more detailed description of each key concept. The key concepts of the algorithm are extension, generalization, forgetting, and backpropagation. These concepts all have their origin in biology and are critical for the model. When utilized correctly, they create a powerful neural network. How this is implemented is explained in this section.

## 4.1 Overview of the LL0 model

The LL0 algorithm takes an approach which is quite rare within the field of neural networks; the algorithm has the ability to change its structure on demand. The network starts from scratch, consisting of only *Input Nodes* and *Output Nodes*, and whenever the algorithm makes an erroneous prediction, new nodes are added to the network to compensate for the error. This process is called the *extension step*, and will be explained more in detail in section 4.2. If any node in the network becomes obsolete, it will be purged from the network by the *forgetting step*, which will be discussed in more detail in section 4.5. By implementing these two steps, we get a network with a dynamic structure with the ability to grow or shrink and adapt to new situations.

When extension and forgetting occur, they are carried out in a systematic way, and only on *Concept Nodes* and the belonging *Value Nodes*. These are two of the four possible node types. The four node types are:

- *Input Nodes* - An Input Node propagates its value to its children, which are always Value Nodes. The Input Node has an identity activation function
- *Value Nodes* - with the responsibilities to remember one single value. They always have a constant weight with value 1 as incoming edge and one outgoing edge with a trainable weight. The Value Nodes represents how well an incoming value is remembered by an output $y \in [0, 1]$. The Value Node has a gaussian activation function, with its center at the value it should remember, and its standard deviation, $\sigma$, defining how wide the function should be. How

the gaussian activation function responds to different values is visualized in Figure 4.2. A Value Node always propagates its value to a Concept Node, which will receive how well a certain value is recognized.

- *Concept Nodes* - with the responsibilities to remember different concepts, a concept is supposed to model a group of datapoints. When a concept is added, it is added to remember a datapoint with the room for some noise. A concept is represented by the Concept Node by having many incoming edges from Value Nodes, which is first summarized and then passed through the Concept Nodes sigmoid activation function. It therefore has an output $y \in [0, 1]$ which represents how well the concept is remembered. The Concept Node can have many outgoing edges, which connects to Value nodes (which then evaluates how well the concept is remembered). Furthermore, a Concept Node always has one outgoing weighted edge connected to each Output Node.
- *Output Nodes* - with the responsibilities to give a prediction of the input. Each Output Node has an incoming edge from every Concept Node in the network. The Output Node has a softmax activation function.

The four node types can be seen in Figure 4.1.

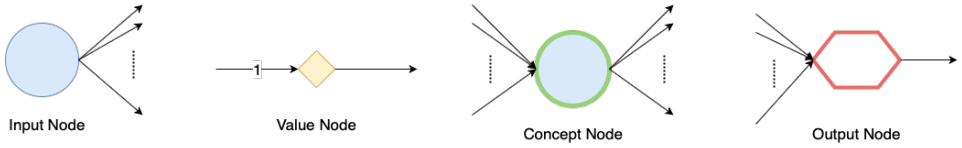


**Figure 4.1: The four node types of the network,** namely the Input Node, Value Node, Concept Node and Output Node. The Input Node has many outgoing edges to Value Nodes. The Value Node has one fixed incoming edge, and one outgoing edge with a trainable weight. The Concept Node has many outgoing and incoming edges from and to Value Nodes. All Concept Nodes are also connected to the Output Nodes with trainable Weights.

Lifelong learning is facilitated through the aforementioned concept of neuroplacsticity, which have been translated to the rules forgetting, extension and generalization. Forgetting and extension are done by adding or removing one Concept Node, together with its parents (Value Nodes). Therefore, whenever an extension is done, a concept is created to remember the label $t$ for the input $x$. Forgetting is the opposite, and instead removes the concept. As mentioned previously, the rule for extension is carried out in a systematic way. By taking certain Concept Nodes with high output, i.e. *activation*, we create a *focus set*. The focus set consists of the nodes which will be parents to the new Value Nodes that are created in the extensions step. The new Concept Node that is extended to the network will be a child to these Value Nodes. When a new Concept Node is added it is very important that the node reacts to the input $x$ and gives output $t$, with a reasonable amount of noise. How parameters are calculated and set in order to fulfill these one-shot properties are explained in section 4.3

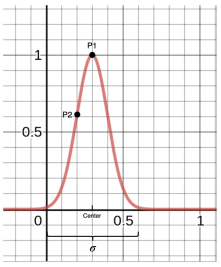


**Figure 4.2: Visualization of a possible gaussian activation function for one of the Value Nodes.** This specific function has its center at 0.3, and a standard deviation $\sigma = 0.1$. Since the center is placed at 0.3 the nodes responsibility is to remember value 0.3. We can see that point $P1$ located at 0.3 gets a value of 1, which means that 0.3 was the value the node remembers. For point $P2$, which has value 0.2, its activation value drops to 0.6. This means that the value is close to what we remembered, but not exact.

A visualization of a possible network is illustrated in Figure 4.3. In the illustration we have three Input Nodes, representing tastes. The Input Nodes are all connected to a set of Value Nodes (diamond shaped). We also have two Concept Nodes, one for blueberry and one for raspberry, which are connected to two Output Nodes (hexagon shaped). The blueberry Concept Node has the highest activation, due to the fact that the datapoint $(0.6, 0.4, 0.2)$ is better recognized by the Value Node connecting to the Concept Node representing the blueberry, compared to the one representing raspberry. We can also observe that this leads to a prediction $(1, 0)$, which most likely represents a blueberry class.

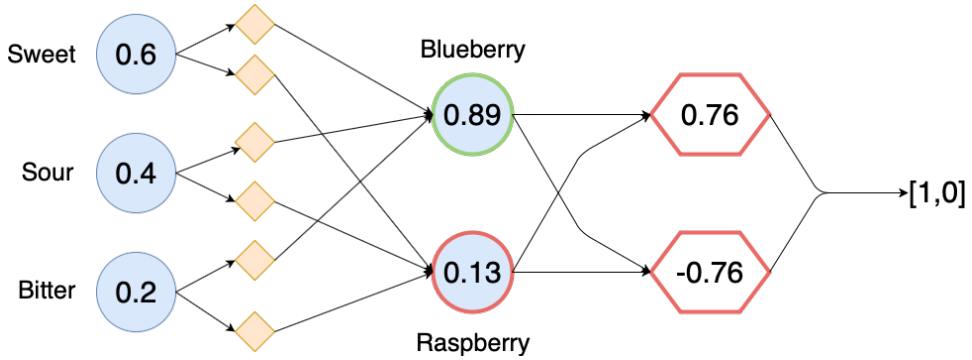


**Figure 4.3: Small illustration of a possible network created by LL0.** Fed into the network was two different berries with different inputs. The blue circles are Input Nodes, the diamonds are Value Nodes, the blue circles with a colored border are Concept Nodes, and the hexagons are Output Nodes. In the illustration the blueberry Concept Node responds with the highest activation for the datapoint $(0.6, 0.4, 0.2)$.

In addition the forgetting and extension, the network structure can be altered by *generalization*. Generalization can be done in the following two ways:

- *Sub concept* - A new Concept Node is created out of an already existing concept, but with fewer Value Nodes as parents.
- *Intersect concept* - A new Concept Node is created as the intersection of at least two existing Concept Nodes which share traits.

Generalization is done by the addition of a new Concept Node that is a small modification of already existing concept(s). The intention is that the new node should be a more general version of the concept(s) it is generalized from and thus lets the network generalize better. Generalization is explained in detail in section 4.4.

Allowing the network to have a dynamic structure has many benefits. However, this property by itself is not enough for the network to reach good performance. Like most other training algorithms, LL0 uses backpropagation in order to tweak the parameters in the network. How backpropagation works in LL0, with different types of activation functions, sparsely connected nodes, and edges that can skip layers, is introduced in section 4.6.

How these different steps work jointly can be observed in algorithm 1 which explains the LL0 algorithm in its simplest form. In the next section the first rule of the LL0 model, the extension rule, will be covered.

```
for epoch do
    for datapoint(x, t) do
        output, error, extend_set = forwardpropagate(x)
        if argmax(output) != argmax(t) then
            if nbr_high_value_nodes > θ_g then
                generalize_sub()
            else
                extend(extend_set, t)
            end
        else
            backpropagate(error, output, t)
        end
    end
    generalize_intersection()
    forget()
end
return model
```

**Algorithm 1: The main loop of the training algorithm**, showing how the four different rules extension, forgetting, generalization and backpropagation are utilized.

## 4.2 Extend

The most crucial ingredient in the LL0 model is to allow the network to grow by adding new layers and nodes. This is where the rule for extension plays the major part. Extension has the sole purpose of expanding the network during the training phase when a pattern that is fed to the network is not recognized. The extension step is triggered when an erroneous prediction is done. The addition of new nodes is treated differently depending on the values of the input, and has several edge cases that must be taken care of. The extension step is probably the most important rule for two reasons. Firstly, it gives the algorithm a dynamic structure that adapts to the problem at hand, adding more nodes if required by the problem. Secondly, the extension phase opens up a possibility for one-shot learning by setting appropriate parameters when nodes are created, which have the potential to speed up the training phase immensely. This is explained in detail in section 4.3.

### 4.2.1 Extension with Real Numbers

When dealing with input values in the interval $v \in [0, 1]$ we extend in regards to three rules. While doing the forward propagation of the input, the focus set is formed. The focus set consists of Concept Nodes with an activation $\geq \theta_a$, where $\theta_a$ is a predefined threshold. If two Concept Nodes in the focus set have a relation such that one node is a parent to the other, we remove the parent from the focus set. If an erroneous prediction was made, it triggers the rule for extension, and we extend on the focus set. If there are no Concept Nodes in the focus set, we extend from all Input Nodes. If there is only one node in the focus set we try to extend from that node together with all Input Nodes that are not predecessors to the node. If there are no such nodes, we extend from only the Input Nodes. If the number of Concept Nodes are above a certain threshold in the extend set, we extend from all nodes in the extend set, otherwise we extend from the Input Nodes. These different extensions can be seen in Figure 4.4.

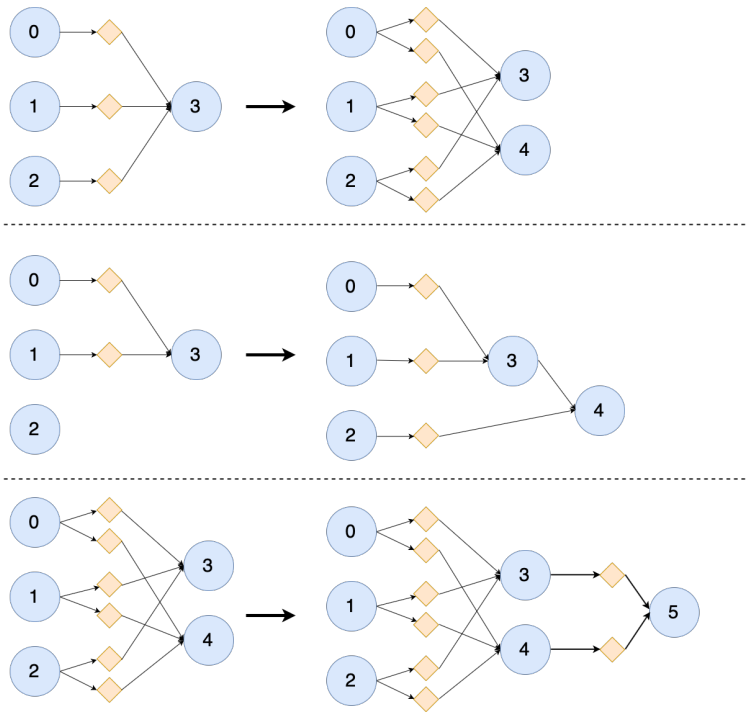


**Figure 4.4: A small network with three Input Nodes.** The input is not recognized, and a new Concept Node (node four and node five) is added by the three different ways of extension. The yellow diamonds indicate Value Nodes. The top most image represents when we extend from nothing but the Input Nodes. The middle image represents when we extend from the extend set and all input nodes that are not predecessors to the extend set. The last image represents the extension from the extend set when we have enough Concept Nodes in the extend set.

A rough overview of the flow of the extension rule with real numbers can be seen in Algorithm 2.

**if** *|extend set| == 0* **then**
  Extend from Input Nodes
**else**
  **if** *|extend set| > extend threshold* **then**
    Extend from extend set
  **end**
  **if** *|extend set| < extend threshold, but there are input nodes that are not predecessors* **then**
    Extend from extend set + Input Nodes that are not predecessors
  **else**
    Extend from Input Nodes
  **end**
**end**

**Algorithm 2:** Extension algorithm for input values $v \in [0, 1]$

### 4.2.2 Extension with discrete values

When dealing with input values $v \in \{0, 1\}$, such as the N-parity problem, we have to remodel how the extension works. Now, each node is instead considered active or inactive, depending on its value. This limits the extension rule to only apply to the nodes that are active, i.e. the nodes with an activation of 1, instead of a lower threshold as in the case with real numbers.

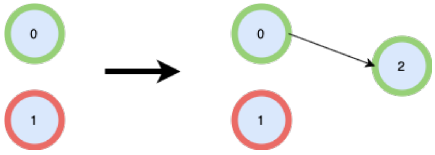


**Figure 4.5: Extension in its simplest form.** Two Concept Nodes (Value Nodes are ignored in this visualization), whereas one is active (green border), are extended. As one of the Concept Nodes is inactive, it is not part of the extension. The extended node is active, as its only parent is active.

This means that we get the structure as seen in Figure 4.5, where only the active nodes are used for extension. This type of extension differs from that proposed in Section 4.2.1. If we use the rules from Section 4.2.1 we could get the following scenario: If we consider having trained on the pattern $[1, 1, 1]$ on a previously untrained network, we would simply extend all three Input Nodes to a new Concept Node. If we then feed the pattern $[1, 1, 0]$, which is a new pattern, and therefore not recognized, we would get the following structure as seen in Figure 4.6.

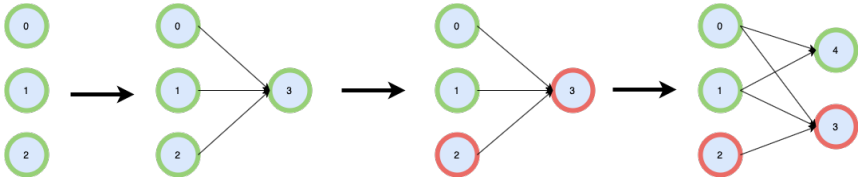


**Figure 4.6: Extension that leads to a faulty architecture.** The pattern [1,1,1] is fed to the network (visualized by the nodes 0, 1 and 2 having a green border). A new concept is formed, node 3. We then feed [1,1,0] to the network, and as thus only nodes 0 and 1 are active. The pattern is not recognized, and a new node is added. The Value Nodes are ignored in this visualization.

If we then feed the first pattern, $[1, 1, 1]$, we would get that both Concept Nodes are active, and we therefore would try to output either two different answers, or do an average of the two - which both would result in an error, and we would have forgotten the first pattern we trained on, as seen in Figure 4.7.

Instead, we propose two different cases of extension. The first case is the one that solves the issue with Figure 4.7. If two nodes are active, but have an inactive child, we simply add another concept between the two active nodes and the child. Furthermore, we remove the connections between the two active parents and the

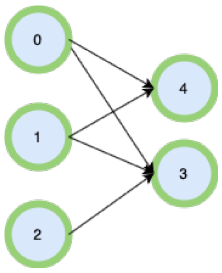


**Figure 4.7: Forwardpropagation on the model from Figure 4.6** with all three Input Nodes active, which results in a faulty activation of both Concept Nodes (nodes 3 and 4). The Value Nodes are ignored in this visualization.

child, and instead add a new connection from the added node to the child. The same applies if only one Input Node is active.

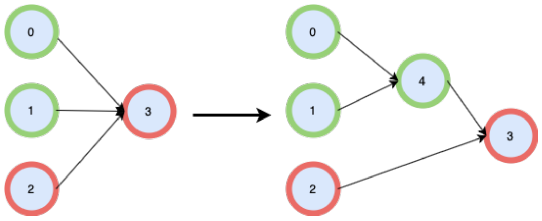


**Figure 4.8: A correct extension made from the initial structure seen in Figure 4.6**. Two active Concept Nodes have an inactive child, and add another child in between. The Value Nodes are ignored in this visualization

The first rule also applies in the cases when there is no immediate child that is shared by the active nodes, but there exists one further down the graph. If one of the active nodes has a direct connection to the first shared child, we also remove the connection between those nodes as seen in Figure 4.9.

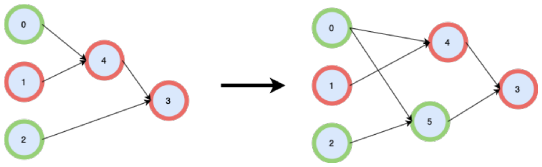


**Figure 4.9: Extension when two active Concept Nodes share a successor but not a child.** The Value Nodes are ignored in this visualization

The second case is when we have two clusters of active nodes, but with no shared parents. We then treat each cluster separately, and do extensions appropriate to that cluster. When the extension has been made, we join the two clusters by a final extension. The flow of the extension rule for discrete values can be seen in

Algorithm 3.

```
if successors(most active nodes) == 0 then
 | Extend from most active nodes
else
 | if active nodes share child then
 |  | Extend by inserting node between active nodes and child
 | else
 |  | if active nodes share successor then
 |  |  | Extend by inserting node between active nodes and successor
 |  | else
 |  |  | Repeat from first step but for each cluster of nodes that share successor
 |  | end
 | end
end
```

**Algorithm 3:** Extension algorithm for input values $v \in \{0, 1\}$

## 4.3 Initial Node Values For Extension

In order to achieve one-shot learning, the initial parameters for the nodes that are added by the extension step are very important. With correct parameters the nodes will get a correct activation when appropriate, without the need for backpropagation. If this is not done properly, wrong Concept Nodes will have a high activation at inappropriate times, and therefore lead to incorrect inference. As an example; if all gaussian parameters are initialized too small they do not allow for noise, which will lead to overfitting. On the other hand, if they are initialized too large, they will recognize all inputs as noise and will therefore not be able to distinguish different inputs from one another. These parameters can also be required to be different for different problems. Distinct problems have different noise for input parameters, which would require the nodes to be more or less sensitive to noise depending on the problem.

The parameters that influence the one-shot learning capabilities the most are those in the activation functions and the weights. More specifically the standard deviation $\sigma$ and the center $\mu$ for the Value Nodes, the bias $\theta$ for the Concept Nodes, and the weights $w$. In order to initialize these parameters in a practical and understandable way, two new hyper-parameters $a_v$ and $b_c$ are added, with the purpose of giving initial values to the other parameters. The new hyper-parameters specify the *drop-off time*. We define the drop-off time as the change needed by the $x$-value to drop from value 1 to 0.5 on the y-axis for the Value Nodes, and from 0.9 to 0.5 on the y-axis for the Concept Nodes. The reason why we start from 0.9 for the sigmoid activation function is due to one of the sigmoid functions properties. For the sigmoid function $\lim_{x\to\infty} sigmoid(x) = 1$. Hence, if we wanted to drop from 1 to 0.5, $x$ would have to be set to $\infty$, which is not plausible. How the drop-off parameters affect the activation functions can be observed in Figure 4.10.

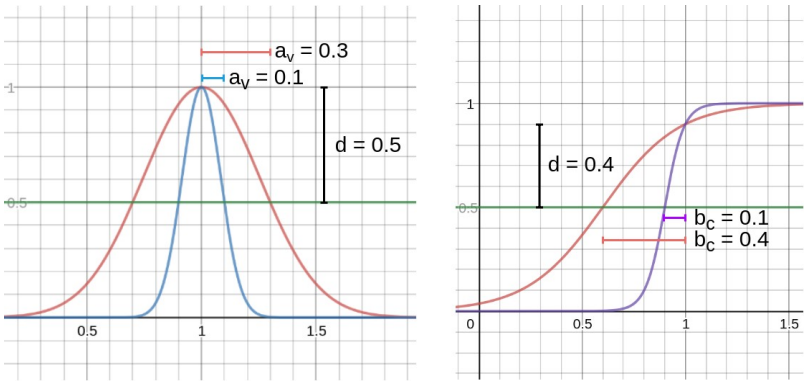


**Figure 4.10: Visualization of possible activation functions and how the drop-off parameters $a_v$ and $b_c$ affect them.** To the left in the figure we see a gaussian activation function with two different values for $a_v$. We see that for $a_v = 0.1$, it requires a step of 0.1 on the x-axis to reach 0.5 on the y-axis, and for $a_v = 0.3$ it requires a step of 0.3. Similarly for the sigmoid function we see in the right picture a value of $b_c = 0.1$ requires one step of 0.1 on the x-axis while $b_c = 0.4$ requires a step of 0.4 to reach the value 0.5 on the y-axis.

The equation used to get the parameter $a_v$ for the gaussian activation function is:

$$gauss(x) = exp\left(\frac{(x-\mu)^2}{2\sigma^2}\right) = 0.5, x = \mu + a_v \tag{4.1}$$

We set $x = \mu + a_v$ as we want a distance $a_v$ away from the center to give us value 0.5. This process can be observed in Figure 4.10. The center is always set as the activation of the Value Nodes parent when the Value Node is created. Thus, if we go a distance $a_v$ away from the parents activation we will get a value of 0.5, i.e. we recognize the input to 50%. Since the incoming weights to the Value Node are frozen and set to 1, they do not affect the behaviour of the Value Node.

Solving equation 4.1 for the standard deviation, $\sigma$, gives us the initial value of $\sigma$ dependent on $a_v$, which is the $\sigma$ that will be the initial value for all Value Nodes based on $a_v$.

$$\sigma = \pm\frac{a_v}{\sqrt{2}\sqrt{ln(2)}} \tag{4.2}$$

For the sigmoid activation function we set up the following equations:

$$sigmoid(nw + \theta) = \frac{1}{1 + e^{-(nw+\theta)}} = 0.9 \tag{4.3}$$

$$sigmoid(a_c nw - b_c nw + \theta) = \frac{1}{1 + e^{-(a_c nw - b_c nw + \theta)}} = 0.5 \tag{4.4}$$

Where $\theta$ is the bias, $w$ are the incoming weights to the Concept Node, $n$ is the number of parents to the Concept Node and $a_c$,$b_c$ are new variables. As we utilize 0.9 as the max value for the sigmoid, the input to the Concept Node will be $\sum_{i=0}^{n} w_i$ when all Value Nodes have the value 1. Since we initiate all weights to the same value, $\sum_{i=0}^{n} w_i = nw$. The parameter $a_c$ is typically 1 which lets $b_c$ specify the ratio of Value Nodes that can have a value of 0 such that the sigmoid output 0.5. Solving the equations for $w$ and $\theta$ we get the initial values for the Concept Node based on the tuning parameters $a_c$ and $b_c$.

$$w \approx \frac{-2.19722}{n(a_c - b_c - 1)}, n(a_c - b_c - 1) \neq 0 \tag{4.5}$$

$$\theta = logit(0.9) - \frac{2.19722}{-a_c + b_c + 1}, (-a_c + b_c + 1) \neq 0 \tag{4.6}$$

where $logit(x)$ is the inverse function of the sigmoid function.

When the tuning parameters are set, equations 4.5 and 4.6 are used to get good initial values for the Concept Node. However, since each input feature can come from different distributions, the same initial values can not be expected to perform perfectly. The initial parameter values are not known beforehand, thus the network must handle setting parameters dynamically. If these inaccuracies are small, back-propagation will make the final adjustments. If these inaccuracies are large, it is up to the initial values of the extension rule. The extension rule has to set the values as explained in this section for the weights, biases, centers, and standard deviation for the newly created nodes, to compensate for the large inaccuracy.

One situation where the need of differently set parameters for different Concept Nodes become apparent is during some cases of classification. For example if a Concept Node has a high activation, meaning it is confident that it has the correct prediction - but makes a wrongful prediction. This leads to that the node with high activation, and thus confident that it is correct, has had a big impact on a wrongful decision. This is most likely due to the initial parameters being inaccurate, therefore drastic changes must be made to the network.

If the hyper-parameters are inaccurate and drastic changes must be made, what we perceived as correct was actually wrong. This can be compared to humans; when we realize some assumption is too general, and we need to change that assumption to be more specific. For example if we see a leopard for the first time, it is a reasonable assumption that all larger cat animals with spots are leopards. If we then see a jaguar for the first time - which is also a large cat with spots, we need to change our mental model of the leopard so we do not mistake it for a jaguar in the future. In a similar way when the network predicts the wrong class, it means that some concept is too general. This is handled by letting the weights to the output for the new jaguar concept to be negative. By letting the jaguar Concept Node have negative outgoing weights towards the leopard Output Node, the jaguar concept cancels out the influence of the leopard concept, thereby enabling the network to

correctly identify a jaguar. By setting negative weights in this way we do not need to dynamically change $a_c$ and $b_c$ for the cases when they are too general. Instead, the concepts that are too general will be canceled out and made more specific by other concepts interfering with the concepts that are too general.

More specifically; When a new Concept Node $n_c$ with activation $a_c$ is created it cancels out other impacting Concept Nodes by first creating the vector $v_1 = output * \frac{1}{a_c}$, where *output* is the output of the network that triggered the extension. By setting $w_{oc}$ to $v_1$, where $w_{oc}$ is $n_c$'s weights which are connected to the output, we cancel out all other concept nodes for this input. However, canceling out all other Concept Nodes fully could be harmful. There is a risk that $n_c$ will have some activation when other nodes make inferences, and therefore negatively impact other nodes predictions. Therefore $w_{oc}$ is multiplied by a new tuning parameter $\delta < 1$ to lower the interference with other nodes. $w_{oc}$ is then incremented by the vector $v_2 = \vec{0}$, with $v_{2,argmax(t)} = \frac{1}{a_c}$, where $t$ is the correct label. We increase the value at index $argmax(x)$ since that is the class we want to predict. This gives us the final formula:

$$w_{oc} = v_1 * \delta + v_2 \tag{4.7}$$

Once $a_v$, $b_c$, and $\delta$ have been tuned to suitable values they have a large impact on the quick learning rate of the network. They are all used in order to achieve one shot learning through the extend rule. To better catch unseen datapoints, as well as removing noise from the dataset, we also have a generalization step, which is discussed next.

## 4.4 Generalization

The second rule in the algorithm is generalization, which aims to generalize the network for unseen patterns. The rule for generalization is carried out in two different manners. The first way to generalize is to create a *sub concept*, which is achieved during extension step. When an input is fed into the network and the extension rule is triggered, we first check if some other Concept Node has *high Value Nodes*. A Concept Node has a high Value Node if the Value Node is above above a certain threshold, $\theta_h$. If any Concept Node has at least $\theta_g$ amount of high Value Nodes, a Sub concept is constructed instead of carrying out the extend step. $\theta_h$ and $\theta_g$ are predefined parameters.

The second manner in which generalization takes place is by creating an *intersection concept*. The process of creating the Intersection concept is not bound to any specific input, instead we compare all nodes that contribute positively to the same output for every layer. More precisely, if two nodes activations multiplied with their weights to the output layer are contributing positively to the same output node, they are considered alike.

If two or more nodes have the aforementioned property, while simultaneously sharing

parents according to a specific criteria - which is discussed later in this section, we generalize by intersection.

### 4.4.1 Sub Concept

A sub concept is created in order to make a single concept more general and is achieved by creating a copy of the concept and removing some of the incoming data (connections) to the new concept. This can result in unnecessary noise being removed from a concept. An example of this is illustrated in Figure 4.11, where a sub concept is formed after eating an apple. The amount of energy received for eating an apple should be independent from what temperature it is outside, even if temperature was part of the input. This shows the key property and importance of the sub concepts. Instead of creating one Concept Node for each temperature giving the same energy, we can generalize to a sub concept that ignores the temperature input. This can also be seen in normal fully-connected networks, where weigths can be trained towards zero to efficiently remove a specific input parameter.

The sub concept is created by searching through each Concept Node. If the number of high Value Nodes for some Concept Node $n$ are above the threshold $\theta_g$, we form a sub concept $n_{sub}$ from $n$. $n_{sub}$ is added as a layer between node $n$ and the nodes that have outgoing connections to node $n$. The high Value Nodes that previously were connected to node $n$ is after the generalization instead connected to $n_{sub}$. We then let $n_{sub}$ be connected to $n$. This is illustrated further in Figure 4.11, where a node has been generalized - it had two high Value Nodes, and we have formed a new node as a sub concept.

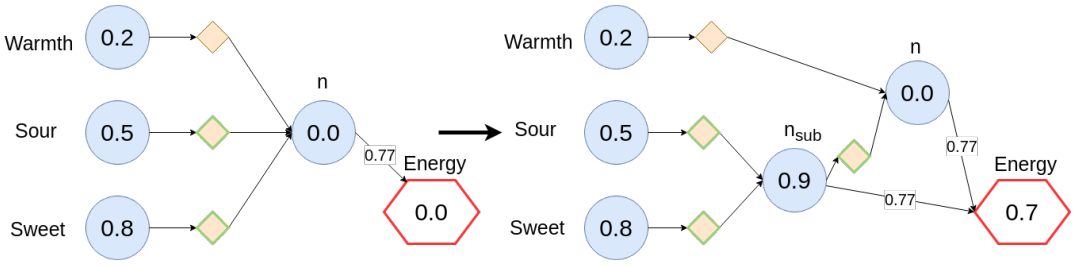


**Figure 4.11: Visualization of the creation of a sub-concept.** On the left hand side of the picture we have first created the concept $n$ when eating an apple with warmth of 0.8. We then feed the data point $d = (0.2, 0.5, 0.8)$, creating the state in the figure. When the network is fed $d$, which represents eating an apple when its cold, we get the result on the right hand side of the figure. A sub concept $n_{sub}$ has been constructed from $n$ by triggering generalization instead of extend. The resulting sub concept ignores the warmth input and maps to the correct energy output.

It is really important when we add $n_{sub}$ that it does not affect $n$. For instance, if we have learned the concept of a red apple, and we generalize from the red apple to an apple concept, our perception of the red apple should not change; the red

apple must remain exactly as it was. On top of these requirement we also want $n_{sub}$ to have as high of an activation as possible when added, and the same drop-off time as all other Concept Nodes. A special architecture is required to fulfill these requirements.

The impact of $n_{sub}$ is controlled by setting its weights and biases correctly, as well as adding two new type of nodes, the *Split Node* - $n_{SN}$, and the *Sum Node* - $n_{sum}$. The Split Node ensures that the exact same value that was propagated to $n$ before the generalization still reaches $n$, but now through the Split Node instead. The Split Node has an outgoing connection to $n$ and an incoming connection from the Sum Node. The Sum Node is simply a node with an identity activation function and a bias set to zero, and therefore only sums the incoming values. The Sum Node has its incoming edges from the high values nodes and outgoing edges to the Split Node and $n_{sub}$. This architecture can be visualized in Figure 4.12. The parameters in the Figure and how they are initially set will be explained in the following paragraphs.

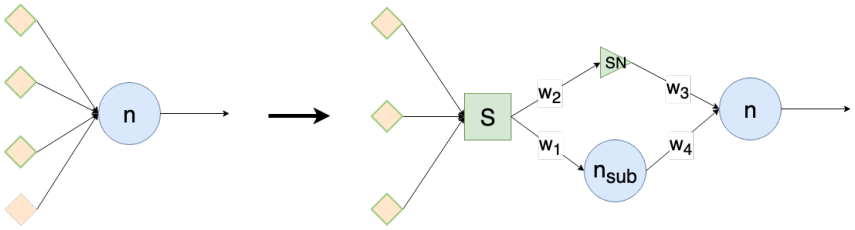


**Figure 4.12: A detailed visualization of the creation of a sub-concept.** The node denoted S is a Sum Node, SN is a Split Node, $n_{sub}$ is the sub-concept, and $n$ is the node being generalized. This Figure shows the actual architecture, while in Figure 4.11 a conceptual figure of the creation of a sub-concept can be observed. In Figure 4.11 S and $n_{sub}$ have been merged in to one node, depicted as $n_{sub}$.

When adding $n_{sub}$, the new parameters $w_2$ and $w_3$ must be set such that they do not influence $n$. $w_2$ is defined as $\frac{1}{w_{sum}}$, where $w_{sum}$ is the sum of the weights going in to $n_{sum}$. $w_1$ is defined such that $n_{SN}$ can not get an activation higher than 1. The weight $w_3$ is set to $w_{sum}$. This ensures that the incoming value to $n$ after generalization equals $f(p_v * \frac{1}{w_{sum}}) * w_{sum} = p_v$, where $f(x) = x$ and $p_v$ is the value incoming to $n$ before the generalization. The Split Node's sole purpose is to function as a Value Node; it sets an activation $a \in [0, 1]$ of how well the input is recognized. By implementing the generalization as mentioned above, we get the exact same activation on $n$ as prior to the generalization.

On top of ensuring that $n$ keep its activation, we also need the correct drop off time for $n_{sub}$, and let its activation peak at the point of creation. This is done by setting the parameters $w_1$, $w_4$ and $\theta_{sub}$ - the bias for $n_{sub}$, in the following manner:

1. $w_1 = \frac{ln(9)}{(-a_c+b_c)w_{sum}+w_{sum}}$
2. $\theta_{sub} = -w_{sum} * w_1 + ln(9)$

3. $w_4 = 0$

We obtain $w_1$ and $\theta_{sub}$ by the two equations:

$$sigmoid(w_{sum} * w_1 + \theta_{sub}) = \frac{1}{1 + e^{-(w_{sum} * w_1 + \theta_{sub})}} = 0.9 \tag{4.8}$$

$$sigmoid((a_c - b_c)w_{sum} * w_1 + \theta_{sub}) = \frac{1}{1 + e^{-((a_c - b_c)w_{sum} * w_1 + \theta_{sub})}} = 0.5 \tag{4.9}$$

We set up these equation by the same motivation as discussed in Section 4.3. Variables $a_c$ and $b_c$ can also be found in section 4.3, with a description of how they influence the drop off time. $w_4$ is set to zero so it does not influence $n$, while fulfilling the wanted hierarchical parent-child relationship.

The last parameters that need to be set is the weights to the Output Nodes. We set the weights almost exactly as in equation 4.7 in Section 4.3. However, since $n_{sub}$ is a bit of a guess and does not directly model an entire data point but only a sub-part of it, we do not want it to have the same impact as a regular Concept Node. Therefore $n_{sub}$'s output weights $w_{o,sub}$ are multiplied by an additional parameter $\delta_{sub} \in [0, 1]$. This results in the final equation:

$$w_{o,sub} = w_{oc} * \delta_{sub} \tag{4.10}$$

Setting all these parameters as discussed in this section will lead to a sub Concept Node that does not impact the node it generalized from. Furthermore, the sub Concept will have its peak activation when the network gets the same data point as when it was added, and have the same drop off time as all other Concept Nodes when created. The Sub Concept will hopefully respond to the datapoint, with some Input Nodes being cut off, leading to a more sparse network and a more general perception of the given data point.

### 4.4.2 Intersect Concept

The intersect concept solves a different problem than the sub concept. Instead of generalizing a single concept, this step generalizes the intersection of two or more different concepts. When two or more Concept Nodes share parents and similar traits, a new node is added to generalize them. The resulting Concept Node has less predecessors, as the parents of the new node is the intersection of the parents of the original nodes. The new Concept Node is also an attempt to bundle properties that are related to one another together. As an example, instead of having multiple Concept Nodes describing different kinds of apples separated from each other, a Concept Node for an apple could be added. This would mitigate the continual growth of the architecture, and allow for new and unseen kinds of apples to be

correctly labeled as apples in a smaller manner. The new Concept Nodes values' are simply the average from the values that it was generalized from, and the incoming Value Nodes' values are averaged from the generalized Value Nodes' values.

The intersection step is not run every time extension is triggered, but instead at a predefined interval - every tenth of an epoch. If we change this interval to occur more often, the training time increases, and we do more generalizations. If we change it to occur less often, the training time decreases, and we simply do less generalizations. The reasoning behind the magic number 10 is that after a tenth of an epoch the majority of the nodes have been created. All nodes that are located on the same layer $l$ are investigated for an intersection generalization. There are two criteria that a subgroup of nodes need to fulfill to be generalized upon:

- The size of the intersected parents set is above a certain threshold.

$$|interesected\_parents| \geq generalization\_threshold$$

- The intersected parents are not equal to the parents to one of the nodes.

$$\forall : nodes, (node\_parents) \neq intersected\_parents$$

In Figure 4.13 a small network with four Input Nodes and two Concept Nodes is shown. Following the rule for intersected generalization, the Input Nodes 1 and 2 should form a new Concept Node, as they share two children which we assume contribute to the output in a similar manner. Node 4 and 5 also have more parents (0 and 3) than only the intersected nodes (1 and 2).

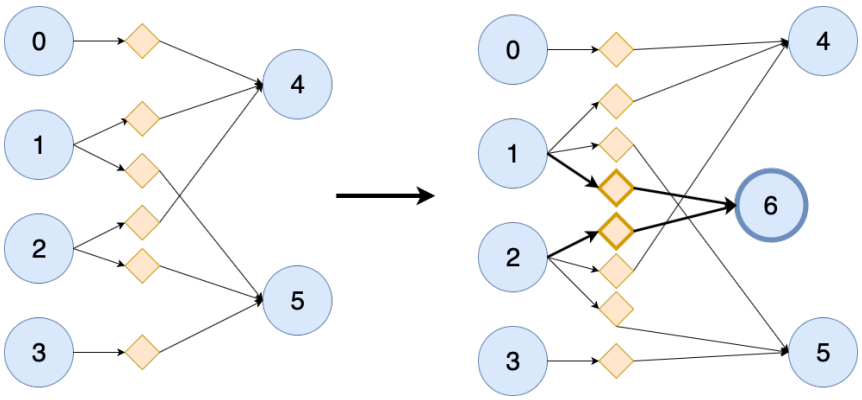


**Figure 4.13: Intersected generalization on two Concept Nodes.** Node four and five share two parents, one and two. They also have parents that are not in the intersection, node zero and three. The added Value Nodes, connections and finally Concept Node (node six) are all highlighted with thicker borders and lines.

## 4.5 Forgetting

As previously mentioned, one key principle to the dynamic network structure is the ability to continually grow and expand when necessary. However, to reduce the possibility for growing indefinitely, it is important that the network also can remove nodes and concepts that are deemed unimportant. There exist different approaches to how forgetting should take place. In LL0 we have taken the approach of removing nodes that on average have very low activation during training. This results in redundant nodes that do not contribute to the system to be removed from the network.

This approach can effectively reduce the size of the network, and as the nodes with low average activation are likely to be special edge-cases that only map to one datapoint, hopefully also increase the generalization effect of unseen values. However, this will allow for catastrophic forgetting in the case of lifelong learning, since we purge the network of stored information that it has previously learned.

The average activity is used as a measure of whether a node should be removed. We therefore define *Average Activity* $AAvg_c(t)$ of Concept Node $c$ at time $t$ as

$$AAvg_c(t) = \frac{\sum_{i=t_0}^{t} a_i}{t - t_0}$$

where $t_0$ is the iteration at which $c$ was added and $a_i$ is the activation of $c$ at $t$. By removing nodes with a low activity, the network is purged of nodes with smallest impact.

Forgetting is carried out in two different manners in the LL0 model, both of which can coexist during training. Forgetting is primarily done on a Concept Node $c$ at time $t$ whenever $AAvg_c(t)$ drops below a certain predefined threshold. Forgetting can also occur by defining a maximum size of the network and making sure that the size is not exceeded. When the maximum size is reached and a new Concept Node is about to be added to the network at time $t_f$, the Concept Node $v$ with the smallest $AAvg_c(t_f)$ out of all nodes is removed from the network. The new node can then be added whilst fulfilling the size requirements. Having a fixed size can limit networks performance, as the amount of allowed nodes can be too small to reach better performance. However, having a fixed size might be appropriate for some problems where the model is in an environment with limited memory.

When a Concept Node $c$ is removed, it is removed together with all Value Nodes that are parents to $c$, as well as the appropriate weights connected to these nodes. This also means that there is one requirement on $c$; it is only removed if it has no children. This is due to the fact that even if $c$ has a low activity, it might have children that performs well.

By implementing the forgetting rule together with LL0's ability to also expand, we get a dynamic structure of the network which adapts to the problem at hand. It will try to reach a minimal solution to the problem by purging unnecessary nodes

dynamically. LL0 also has the possibility to peg a maximum size of the network and make sure it is not exceeded, and reach a model that is as good as possible based on the allowed size.

## 4.6 Backpropagation

As in most training algorithms, the LL0 model uses backpropagation to achieve good performance. However, its implementation is a bit different as the network is not fully connected, some weights are frozen and some weights are not connected to nodes in adjacent layers. The following parameters are updated in the backpropagation step:

1. $\theta_i^{(l)}$ - The bias for Concept Node $c_i^{(l)}$ with layer $l$ and identifier $i$. The bias is updated for all Concept Nodes in the network.
2. $\mu_i^{(l)}$ - The mean for Value Node $v_i^{(l)}$ with layer $l$ and identifier $i$. The mean is updated for all Value Nodes in the network.
3. $\sigma_i^{(l)}$ - The standard deviation for Value Node $v_i^{(l)}$ with layer $l$ and identifier $i$. The standard deviation is updated for all Value Nodes in the network.
4. $w_{i,j}$ - The weight of the directed edge $e = (v_i^{(l-1)}, c_j^{(l)})$ for Value Node $v_i^{(l-1)}$ on layer $l-1$ with identifier $i$, and for Concept Node $c_j^{(l)}$ on layer $l$ with identifier $j$. All directed edges from a Value Node to a Concept Node are updated in one backpropagation step.
5. $W_{i,j}$ - The weight of the directed edge $e = (c_i^{(l)}, o_j^{(L)})$ for Concept Node $c_i^{(l)}$ on layer $l$ with identifier $i$, and for Output Node $o_j^{(L)}$ on layer $L$ with identifier $j$. All directed edges from a Concept Node to an Output Node are updated in one backpropagation step.

In Figure 4.14 it is illustrated where these parameters are located in the network. We also observe the notation that will be used when defining the derivatives for the different parameters. We also see how the forward propagation is carried out in the bottom of the figure. The recursive step in the backpropagation algorithm is done from one layer of Concept Nodes to the next layer of Concept Nodes, including all weights and Value Nodes in between those layers. This means that if the network has a total of $k$ layers, the recursive step starts at layer $k$ and handles all parameters on layer $k$ and $k-1$. The next Concept Node layer exists at layer $k-2$, where the next recursive steps continue to layer $k-3$. The next Concept Node layer then exists at layer $k-4$ etc.

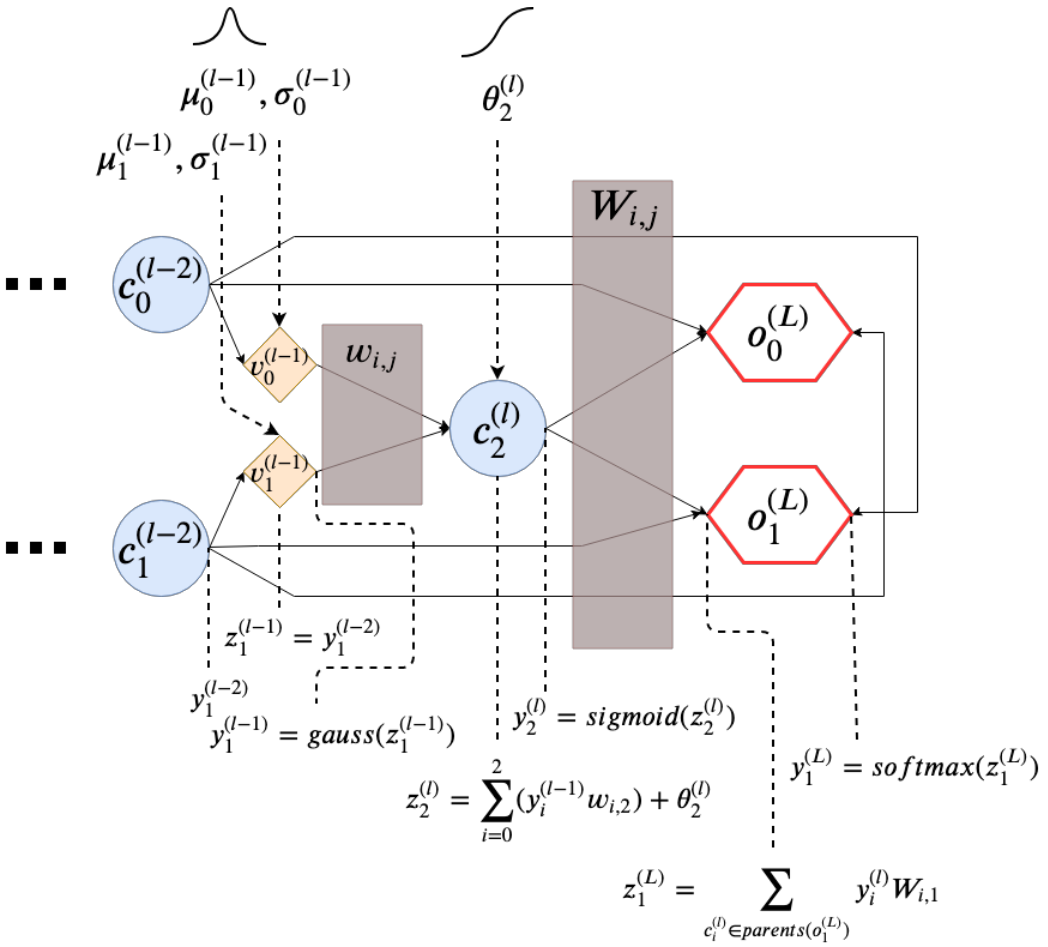


**Figure 4.14: A visualization of the network during the backpropagation step.** At the top of the picture, we observe the parameters for the nodes that the backpropagation step is fine tuning. In the middle of the picture, we observe the weights that are also fine tuned during backpropagation. In the bottom of the picture we see the calculations for the forward propagation. The blue circles are Concept Nodes, the diamonds are Value Nodes and the hexagons are Output Nodes. The notation in the figure will be used throughout this section.

LL0 uses the cross entropy error function:

$$E = -\sum_{d=0}^{D} t_d * ln(y_d^{(L)})$$

Where $D$ is the output dimension of the problem, $t$ is the target label, and $y^{(L)}$ is the output of the network after the softmax activation function. In order to get the update rule for each parameter we need to calculate the derivative of the error with respect to the parameter we want to update. We start by first calculating the derivative for the Output Nodes. We then define the recursive step which calculates

the derivatives for the parameters for two layers at the time; one layer of Concept Nodes, followed by one layer of Value Nodes.

The derivative $\frac{\partial E}{\partial z_i^{(L)}}$ is the first one we calculate. This is the derivative of the error w.r.t the weighted sum of the Output Nodes.

$$\frac{\partial E}{\partial z_i^{(L)}} = y_i^{(L)} - t_i \tag{4.11}$$

The derivative in equation 4.11 is obtained by calculating the derivative of the error, through the softmax, to the weighted sum using the chain rule. This derivative will be used for all Concept Nodes in the recursive step since all Concept Nodes are connected to the Output Nodes.

The recursive step starts with the Concept Node at the deepest layer, and continues until we reach the Input Nodes. For each Concept Node $c_i^{(l)}$ in layer $l$ with identifier $i$ we calculate $\frac{\partial E}{\theta_i^{(l)}}$ and $\frac{\partial E}{\partial W_{i,j}}$ for all Output Nodes $o_j^{(L)}$. These are the derivatives for the biases for the Concept Nodes, as well as the derivatives for the weights between the Concept Nodes and the Output Nodes. They are given the following values:

$$\frac{\partial E}{\partial W_{i,j}} = \frac{\partial E}{\partial z_j^{(L)}} * y_i^{(l)} \tag{4.12}$$

$$\frac{\partial E}{\partial y_i^{(l)}} = \sum_{o_j^{(L)} \in outputNodes} (\frac{\partial E}{\partial z_j^{(L)}} * W_{i,j} * y_i^{(l)}) + \sum_{v_j^{(l+1)} \in children(c_i^{(l)})} (\frac{\partial E}{\partial z_j^{(l+1)}}) \tag{4.13}$$

$$\frac{\partial E}{\partial z_i^{(l)}} = sigmoid(z_i^{(l)})(1 - sigmoid(z_i^{(l)})) * \frac{\partial E}{\partial y_i^{(l)}} \tag{4.14}$$

$$\frac{\partial E}{\partial \theta_i^{(l)}} = -\frac{\partial E}{\partial z_i^{(l)}} \tag{4.15}$$

We reach equation 4.12 by continuing the backpropagation from the Output Nodes to the Concept Nodes at layer $l$ and applying the chain rule. In a similar way we obtain equation 4.13 which defines the incoming derivatives. However, some care must be taken when calculating the incoming derivatives for the Concept Node since we have incoming derivatives from both Output Nodes and Value Nodes. The first sum in equation 4.13 calculates the incoming derivatives from the Output Nodes. The second term calculates the incoming derivatives from Value Nodes. $children(c_i^{(l)})$ denotes at set of $c_i^{(l)}$'s children such that the children are of type Value Node. We also note that for the deepest layer in the network, this term is zero, as the last layer has no outgoing edges except to the Output Nodes.

In the same recursive step we also calculate the derivatives for layer $l-1$. This layer consists of only Value Nodes. We are interested in the weights $w_{i,j}$, the standard

deviations $\sigma_i^{(l-1)}$, and the means $\mu_i^{(l-1)}$ for each Value Node $v_i^{(l-1)}$ in layer $l-1$ with identifier $i$ and Concept Node parent $c_j^{(l)}$. The derivatives used for the Value Nodes are defined in the following way:

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j^{(l)}} y_i^{(l-1)} \tag{4.16}$$

$$\frac{\partial E}{\partial y_i^{(l-1)}} = \frac{\partial E}{\partial z_j^{(l)}} w_{i,j}, \quad j = child\_id(v_i^{(l-1)}) \tag{4.17}$$

$$\frac{\partial E}{\partial \mu_i^{(l-1)}} = \frac{\partial E}{\partial y_i^{(l-1)}} \frac{(z_i^{(l-1)} - \mu_i^{(l-1)}) exp\Big[ - \frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2}{2(\sigma_i^{(l-1)})^2} \Big]}{(\sigma_i^{(l-1)})^2}$$

$$\frac{\partial E}{\partial \sigma_i^{(l-1)}} = \frac{\partial E}{\partial y_i^{(l-1)}} \frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2 exp\Big[ - \frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2}{2(\sigma_i^{(l-1)})^2} \Big]}{(\sigma_i^{(l-1)})^3}$$

$$\frac{\partial E}{\partial z_i^{(l-1)}} = \frac{\partial E}{\partial y_i^{(l-1)}} * - \frac{(z_i^{(l-1)} - \mu_i^{(l-1)}) exp\Big[ - \frac{(z_i^{(l-1)} - \mu_i^{(l-1)})^2}{2(\sigma_i^{(l-1)})^2} \Big]}{(\sigma_i^{(l-1)})^2}$$

These equations are needed for the Value Nodes. The derivatives $\frac{\partial E}{\partial z_i^{(l-1)}}$ are used in the next recursive step when we set $l = l - 2$ which can be observed in equation 4.13. It is important to remember that Value Nodes only have one incoming and one outgoing edge, where the incoming edge is frozen to the value 1 and the outgoing edge is trainable and is obtained in equation 4.16. The last part of the recursive step is the derivatives for the Split Node. Since the Split Node has no trainable parameters, it only back propagates its incoming derivative to its grandparents, which are Value Nodes. Therefore we need to update equation 4.16 and 4.17 to include the Split Nodes.

$$\frac{\partial E}{\partial w_{i,j}} = (\frac{\partial E}{\partial z_j^{(l)}} + SN(v_i^{(l-1)})) y_i^{(l-1)} \tag{4.18}$$

$$\frac{\partial E}{\partial y_i^{(l-1)}} = (\frac{\partial E}{\partial z_j^{(l)}} + SN(v_i^{(l-1)})) w_{i,j}, \quad j = child\_id(v_i^{(l-1)}) \tag{4.19}$$

$$SN(v_i^{(l-1)}) = \sum_{c_j^{(l+2)} \in succSplit(v_i^{(l-1)})} \frac{\partial E}{z_j^{(l+2)}} \tag{4.20}$$

Where we define $succSplit(v_i^{(l-1)})$ in equation 4.20 as a set such that element $c_j^{(l+2)}$ is in the set if and only if it is a successor of $v_i^{(l-1)}$ at exactly layer $l+2$, and has a path from $v_i^{(l-1)}$ to $c_j^{(l+2)}$ through a Split Node at layer $l+1$.

The above equations formulate the entire recursive step. Once it has been run through the two layers, we set $l = l - 2$ and do the recursive step once again. The recursive procedure stops when the Input Nodes are reached, which means that $l = 0$. When the recursive procedure has stopped, we also need to update the parameters in order to fine tune the network towards the minimum of the error function. We update the parameters using gradient descent. We let $p_i \in \{\theta_i^{(l)}, \mu_i^{(l)}, \sigma_i^{(l)}, w_{i,j}, W_{i,j}\}$, and we define the gradient descent step as:

$$p_i \leftarrow p_i - \delta * \frac{\partial E}{\partial p_i} \tag{4.21}$$

where $\delta$ is the learning rate. By executing backpropagation in the fashion explained in this section, we get an algorithm that backpropagates through the network no matter what architecture LL0 develops. It handles connections that skips layers, frozen weights and different activation functions for different nodes. Backpropagation improves the prediction accuracy of the network as it fine tunes all the parameters.

The four different rules explained in this section, joins together to form the LL0 algorithm. The model starts with nothing but input and output nodes, and then creates new nodes by the extension and generalization rules. These nodes are further tweaked by the backpropagation rule, or removed by the forgetting rule.

# 5

# Results

The LL0 model has been benchmarked on three datasets available from `https://scikit-learn.org`, as well as the commonly used spiral dataset. The datasets used are:

**Digits:** 1,797 labeled 8x8 pixel gray scale images of hand-written digits ranging from numbers 0 to 9.

**Sickness:** 569 data points, each of which is a 30-dimensional vector describing features of a radiology image labeled benign or malignant.

**Wines:** 178 data points, each of which is a 13-dimensional vector describing taste features of wine, labeled with one of three regions of origin.

**Spiral:** 2000 data points, the data is represented with 2D points in the form of a Spiral. It is a classification problem. A figure of the dataset is found in Section 5.1.2.

LL0 is compared and benchmarked against four *vanilla* models:

**FC0** A fully connected network with only input and Output Nodes.

**FC10** A fully connected network with one hidden layer of 10 nodes.

**FC10*2** A fully connected network with two hidden layers of 10 nodes each.

**FC10*3** A fully connected network with three hidden layers of 10 nodes each.

LL0 together with the four networks mentioned above are trained using a Stochastic Gradient Descent optimizer, SGD, with a learning rate of 0.01, and a batch size of 10. As the LL0 model currently does not have an implementation of weight decay or momentum for the backpropagation, the vanilla networks have their weight decay and momentum shut off as well. They are not optimized to perform as well as possible on each separate problem, but instead they are optimized to solve the spiral problem, which is the problem that requires the most training time to solve out of the four. The other three datasets are run with the exact same parameters as for the spiral dataset for both the vanilla networks and the LL0 model. The tests are constructed in this way so that comparisons in regards to versatility - the performance on different datasets without any modifications to the networks, can

be shown.

When we in this section refer to the number of Value Nodes, we count both the amount of Value Nodes and the number of Split Nodes, as the Split Nodes function as Value Nodes. The number of Sum Nodes are omitted in the size calculation, as they can be seen as a part of the Concept Node created by generalization.

This section will show results for the aforementioned networks and datasets in regards to prediction accuracy, explainablitiy, one-shot learning, energy usage, and versatility. All experiments shown are the result of 10 runs which have been averaged. In the last chapter of the section there is also a comparison between the LL0 model and a KNN model on a new dataset representing a lifelong learning task. The dataset and task will be explained further in Section 5.6.

## 5.1 Prediction Accuracy

In this section, the models' classification accuracies are compared on a number of datasets. For dataset $(X, Y)$, where $X$ is a set of input vectors, and $Y$ is a set of label vectors. To get a fair evaluation the dataset is split into a test set and a training set. 80% of the dataset is randomly chosen to be in the training set, and the testing set contains the remaining 20% of the datapoints. The same partitions of the datasets are used for all networks.

### 5.1.1 Digits

The results for 3 and 100 epochs for the digits dataset can be observed in Figure 5.1 and 5.2 respectively.

LL0 learns very quickly in the beginning, reaching a level of 95% accuracy in less than three epochs. As with all datasets, the results displayed are the average of 10 runs. The red and blue areas around the testing and training accuracy in Figure 5.1 of the LL0 model is the average $\pm$ the standard deviation. The four vanilla networks do not reach the same accuracy during the four epochs.

The LL0 model converges around 95% accuracy on the test set, while the training accuracy reaches 100%. The LL0 model does not increase in accuracy after $\sim$1.5 epochs, which is why it is only run for three epochs. In Figure 5.2 one can observe that the vanilla models all converge at approximately 95% as well, but they need more than 60 epochs to do so.

The LL0 model constructs an average of $\sim$250 Concept Nodes and $\sim 10,000$ Value Nodes, and reaches an average depth of $\sim$6.6, i.e between 6-8 layers. The LL0 model is sparsely connected.

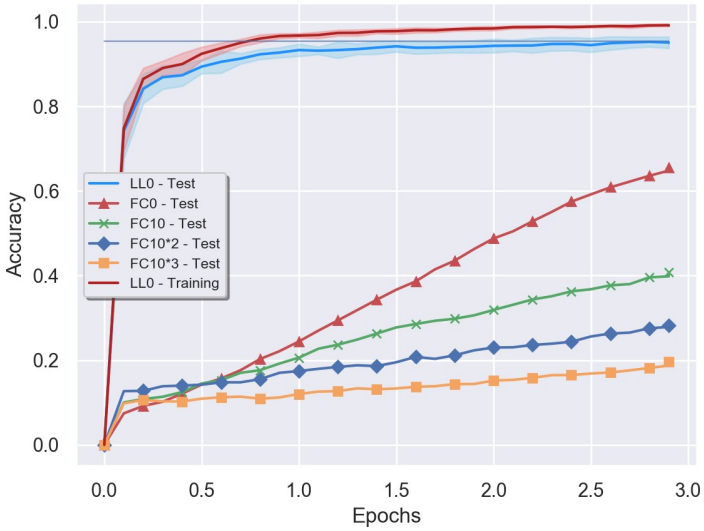


**Figure 5.1: The prediction accuracy during training for the digits dataset during three epochs.** All values are averaged on 10 runs. The red and blue areas around the testing and training accuracy for the LL0 model is the average ± the standard deviation. Only the testing accuracy for the vanilla models is displayed.

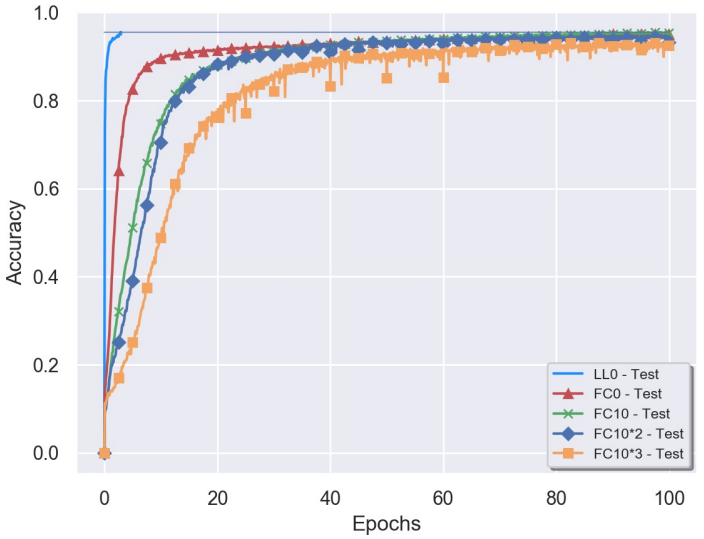


**Figure 5.2: The prediction accuracy during training for the digits dataset during 100 epochs.** All values are averaged on 10 runs. Only the testing accuracies are displayed to avoid cluttering.

### 5.1.2 Spiral

LL0 was run together with the vanilla networks on the spiral dataset. In contrast to the digits dataset, which is quite an easy problem to solve, the spiral dataset is commonly known as a tougher problems for neural networks classification [3]. The dataset can be seen in Figure 5.3.

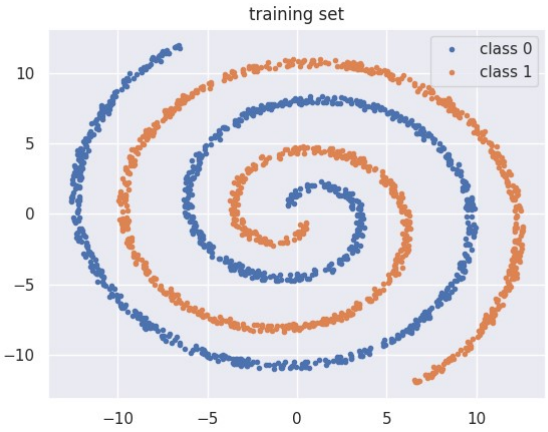


**Figure 5.3: The spiral dataset.** The goal is to correctly categorize the blue datapoints to class zero, and the orange datapoints to class one.

A total of 10 runs was made for each model, and the results displayed are averaged on those runs. In Figure 5.4 the average of 10 runs during three epochs is displayed, and in Figure 5.5 the average of 10 runs for 500 epochs is displayed.

As in Section 5.1.1, the LL0 model learns very promptly, and needs less than one epoch to reach an accuracy of 100% on the test set. The red and blue areas around the testing and training accuracy in Figure 5.4 of the LL0 model is the average $\pm$ the standard deviation. The vanilla networks do not reach a higher accuracy than $\sim$50% during the same time span, which is equivalent to guessing 0 or 1 as the dataset only contains two labels.

The LL0 model constructs $\sim$40 Concept Nodes and $\sim 1,000$ Value Nodes, with a depth of $\sim$5 and is sparsely connected. In contrast to the three epochs needed for the LL0 model to solve the spiral, Figure 5.5 shows that the vanilla networks needs more than 500 epochs to converge, and some of the networks might not even solve the problem at all. FC10 for instance, shows no indication of improving, as it after 500 epochs still is stuck on 60% accuracy. FC10*3 however improves slowly towards 100%, indicating that it can solve the problem.

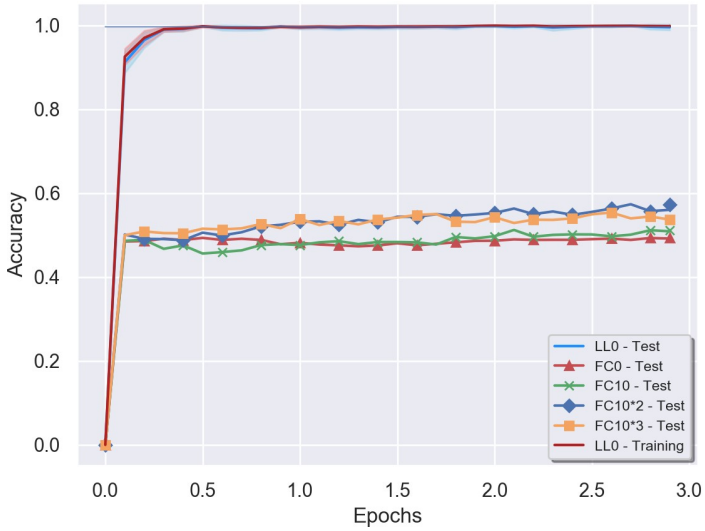


**Figure 5.4: The prediction accuracy during training for the spiral dataset during three epochs.** All values are averaged on 10 runs. The red and blue areas around the testing and training accuracy for the LL0 model is the average ± the standard deviation. Only the testing accuracy for the vanilla models is displayed.

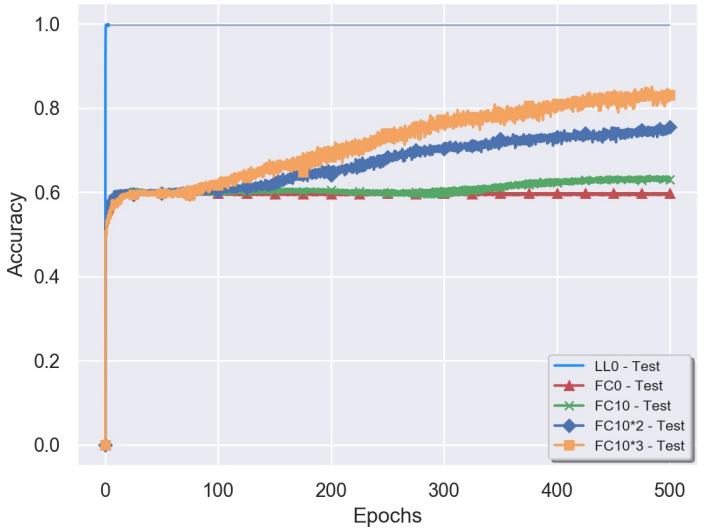


**Figure 5.5: The prediction accuracy during training for the spiral dataset during 500 epochs.** All values are averaged on 10 runs. Only the testing accuracy for the vanilla models is displayed.

### 5.1.3 Sickness

The models were also tested on the *breast-cancer* dataset, which is referred to as sickness for short. The same results as for both the digits and spiral datasets can be observed; the LL0 model needs almost no training to reach a high accuracy percentage, outperforming the other models over the same time span. This can be observed in Figure 5.6. Once again, the red and blue areas is the average accuracy ± the standard deviation for the LL0 model. The standard deviation is higher than for the Digits and spiral datasets, and all the networks are a bit jumpy when they converge.

The vanilla networks reach the same accuracy after about 25 epochs, which can be observed in Figure 5.7. Furthermore, a longer training, i.e. more epochs, does not seem to benefit the models much, as all models fluctuate around 95%.

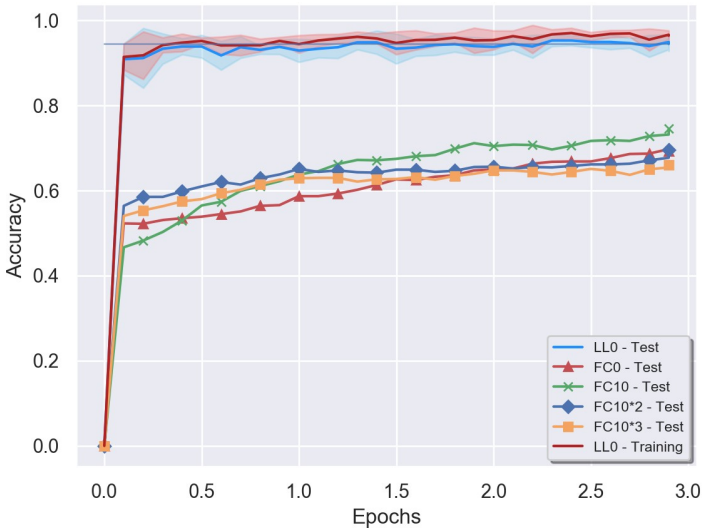


**Figure 5.6: The prediction accuracy during training for the sickness dataset during three epochs.** All values are averaged on 10 runs. The red and blue areas around the testing and training accuracy for the LL0 model are the average ± the standard deviation. Only the testing accuracy for the vanilla models is displayed.

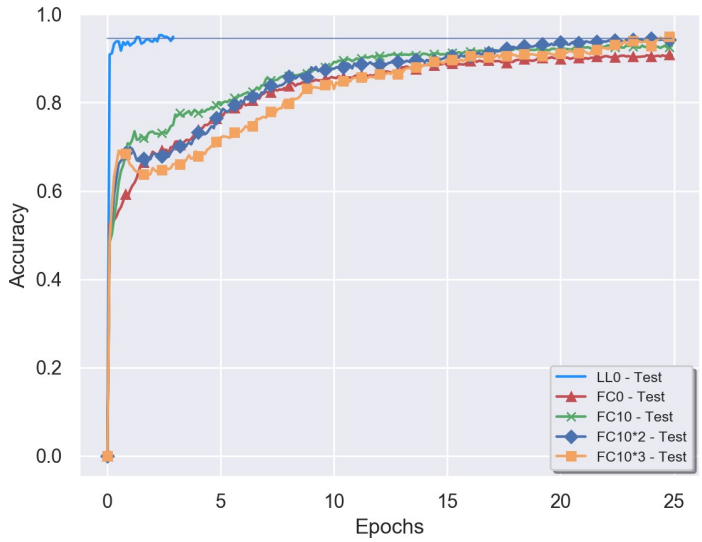


**Figure 5.7: The prediction accuracy during training for the sickness dataset during 25 epochs.** All values are averaged on 10 runs. To avoid too much cluttering in the graph, only the testing accuracy is displayed.

### 5.1.4 Wines

The final dataset the models have been tested on is the wines dataset. As for the previous datasets, the LL0 model needs very little training to reach high accuracy. During the same time span the vanilla networks show little to no improvement. This can be observed in Figure 5.8. As before, the red and blue areas are the average accuracy $\pm$ the standard deviation for the LL0 model.

The vanilla networks need approximately 100 epochs to reach the same, if not even a bit better, accuracy as the LL0 model. This is displayed in Figure 5.9.

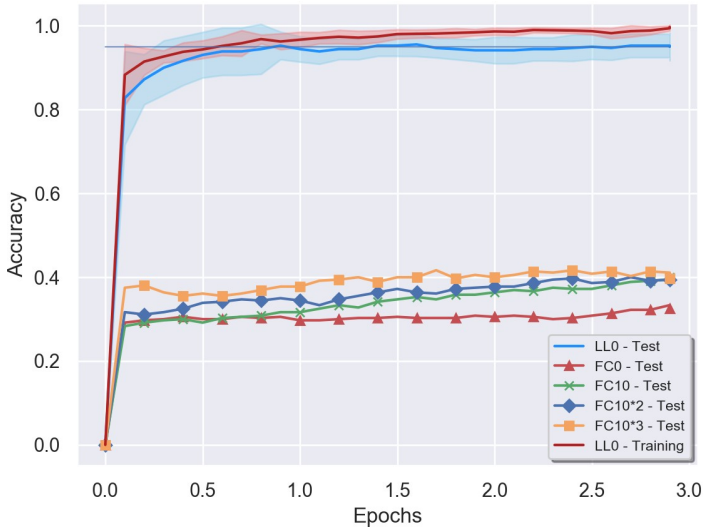


**Figure 5.8: The prediction accuracy during training for the wines dataset during three epochs.** All values are averaged on 10 runs. The red and blue areas around the testing and training accuracy for the LL0 model are the average ± the standard deviation. Only the testing accuracy for the vanilla models is displayed.

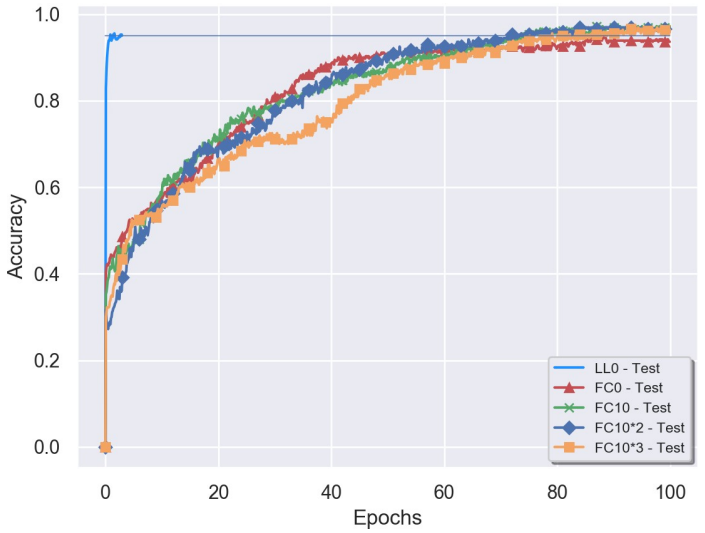


**Figure 5.9: The prediction accuracy during training for the wines dataset during 100 epochs.** All values are averaged on 10 runs. To avoid to much cluttering in the graph, only the testing accuracy is displayed.

## 5.2 Explainability

In contrast to a fully connected neural network, the LL0 model allows for good explainability in terms of structure and decision making. Instead of every weight being a magic number modified by the backpropagation, the LL0 model allows an easier understanding for each node and its impact on the result. This in turn also gives the network another level of traceability. For instance; if one wants to know which nodes in the network impact the result of a specific datapoint $(x, y)$, one can feed the input $x$ to the network and then investigate the state of the network to observe which Concept Nodes have a high activation. The nodes with high activation are the ones that respond to the datapoint and are the ones that affect the decision. There is typically only a few Concept Nodes that respond to a single datapoint at the same time.

The traceablility property gives an opportunity to easily influence separate parts of the network. For example if one wants to remove certain datapoints on a trained model, this could easily be done by just removing their corresponding Concept Nodes, or alternatively let their weights towards the output be set to zero. This could be relevant for ethical reasons, or for erroneous data which is wished to be removed on an already trained model. Removing the recollection of one or multiple datapoints in this fashion is not possible for a regular neural network without negatively impacting the inference of other datapoints.

Another large advantage of the LL0 model can be observed In Figure 5.10, where the LL0 model's decision boundaries in the spiral dataset is depicted. The Concept Nodes (triangles with stronger blue and orange colors) and at which input point they have been added can also be seen. To be able to visualize where in the input space a Concept Node has been added gives some intuition to its responsibilities, and where large errors have occurred. There is not a coincidence that they are placed at similar distances from each other. One should also note that for each layer in the spiral, starting from the center, one can draw a line out towards the edge of the picture where the Concept Nodes occur. Then they often occur on the same line, similar to a star pattern - $*$. This is not a coincidence as those are the inputs where a datapoint is most likely missclassified.

in Figure 5.10 one can also observe the impact that backpropagation has on the model. The figure on the left hand side shows a model that is trained without backpropagation. On the right hand side a figure where backpropagation is activated is depicted. The two images both reach an accuracy of 100%, but the decision boundary is more detailed in the image where backpropagation is activated, and it keeps the decision boundaries at even distance between the two classes. The Concept Nodes are depicted at their starting values, and they might have changed during backpropagation - something that is not accounted for in the image.

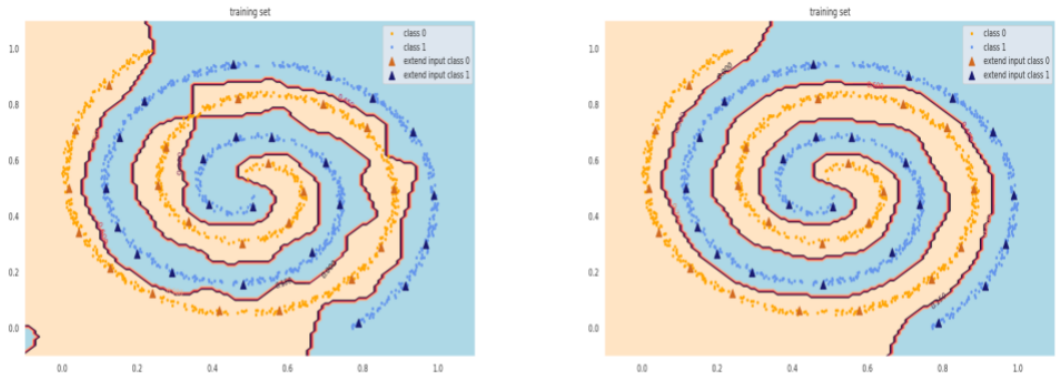


**Figure 5.10: Two contour plots of the spiral dataset created by LL0.** The figure on the left hand side is a contour plot when backpropagation is shut off, and the figure on the right hand side is a contour plot with backpropagation turned on. Both solves the problem, however, one can observe that when backpropagation is turned on the decision boundaries are smoother. They are evenly placed between the points in a smooth round manner. When backpropagation is shut off there is a more unsystematic wave pattern which would make more incorrect predictions if just a little bit of extra noise was added to the test set.

In Figure 5.11 we can see the architecture of a model that is trained to solve the Spiral dataset.

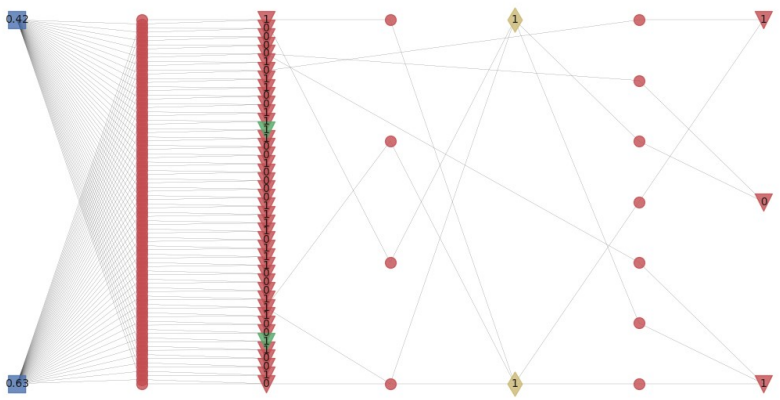

**Figure 5.11: A visualization of a network trained by the LL0 algorithm for the spirals dataset.** In the picture a prediction is done on the datapoint shown in the blue Input Nodes. The triangles are Concept Nodes, the circles are Value Nodes (including Split Nodes), and the diamonds represent the Concept Nodes that have been generalized. The red and yellow color indicates an activation below 0.5 and the green color indicates an activation above 0.5. The numbers in the Concept Nodes indicate which class the nodes belong to.

## 5.3 One-shot Learning

One of the most interesting and important aspects of LL0 is its capability of one-shot learning. After training on less than one epoch, the model achieves an accuracy of ∼90-95% on all benchmarked datasets. This can be observed in the previous sections on accuracy in 5.1. In contrast, the fully connected networks need much more than one epoch to reach the same accuracy, and have barely learned anything after merely one epoch.

Another aspect to note from the graphs in Section 5.1 is that after seeing just 10% of the training datasets LL0 achieves about 90% accuracy on the training set for the datasets Wines, Sickness, and Spirals. On the digits dataset LL0 reaches about 80% after seeing 10%, and 90% after seeing 20%. This shows that LL0 not only directly learns the datapoints it has seen, but also has a general understanding of that datapoint, being able to make correct predictions for datapoints not seen.

## 5.4 Energy Usage

This section compares the models on the basis of computational efficiency. The energy consumption for the baseline models is calculated as the number of parameters times the number of forward and backward passes. The energy consumption of LL0 is calculated similarly for forward and backward passes, where each forward and backward pass is costs the amount of nodes in the network multiplied by three to compensate for the extra parameters. We choose three since each Value Node has three trainable parameters, the bias, center and one weight. Three trainable parameters is maximum amount of trainable parameters for all kinds of nodes, and is therefore an overestimation. The energy consumption of the other rules which are also applied during training are calculated in a similar way. These rules typically requires different searches through the network, which are all combined into a value $v$ of 10x the amount of parameters in the network, which is an overestimation. $v$ is accounted for every time an extension is triggered. This is due to the fact that generalization and extension are all triggered simultaneously, and forgetting requires negligible computational power.

More specifically, the energy consumption per iteration for the vanilla networks is calculated as

$$energy = layer_{input} * layer_i + layer_i * layer_{i+1} + .. + layer_n * layer_{output}$$

where $layer_{input}$ equals the number of nodes in the input layer and $layer_i$ equals the amount of nodes in layer $i$. If there is only an input and output layer, as in FC0, the energy is simply calculated as

$$energy = layer_{input} * layer_{output}$$

For the LL0 model the energy is calculated a bit differently.

$$energy = f_{forward\_propagation} + \begin{cases} f_{extend} & \text{if extension} \\ f_{backpropagation} & \text{if backpropagation} \end{cases}$$

where

$$nbr\_nodes = |Concept\_Nodes| + |Value\_Nodes|$$
$$output\_connections = |Concept\_Nodes| * |Output\_Nodes|$$
$$f_{forward\_propagation} = nbr\_nodes * 3 + output\_connections$$
$$f_{extend} = (nbr\_nodes + output\_connections) * 10$$
$$f_{backpropagation} = nbr\_nodes * 3 + output\_connections$$

where the multiplier of 3 for the forward propagation and backpropagation represents the three parameters that are calculated, and the multiplier of 10 in extend is the overestimated penalty.

Both the LL0 model and the fully connected networks use a batch size of 10, where backpropagation is done every iteration. However the parameters are only updated every tenth iteration, which is ignored in the calculations.

### 5.4.1 Digits

The LL0 model does not outperform the three vanilla networks in energy usage on the digits dataset, as seen in Figure 5.12. As the LL0 model creates quite a large structure to solve the problem, it also consumes quite a lot of energy in terms of parameters calculated. While the vanilla networks have a constant consumption, the LL0 model consumers more the larger it gets. The dynamic structure makes it possible for the network to grow after demand, allowing it to use few nodes in the early stages of training, which results in low energy consumption in the beginning. The quick convergence reduces the need for many iterations, which further reduces the amount of energy consumed. The vanilla networks all grow in similar patterns, as their energy consumption is constant relative to their sizes.

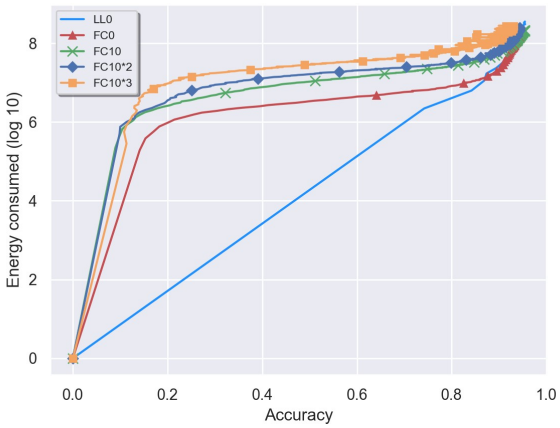


**Figure 5.12: Accuracy of the test set and energy consumption for the digits dataset.** The energy consumption varies widely between the networks, so it is plotted on a logarithmic scale with base 10.

### 5.4.2 Spirals

The LL0 model outperforms the three vanilla networks in the spiral dataset. As the vanilla networks need more than 1500 epochs to converge at the same accuracy percentage, this is not surprising. As seen in Figure 5.13, the LL0 model outperforms the vanilla networks by a factor of 100, and still achieves a higher accuracy.

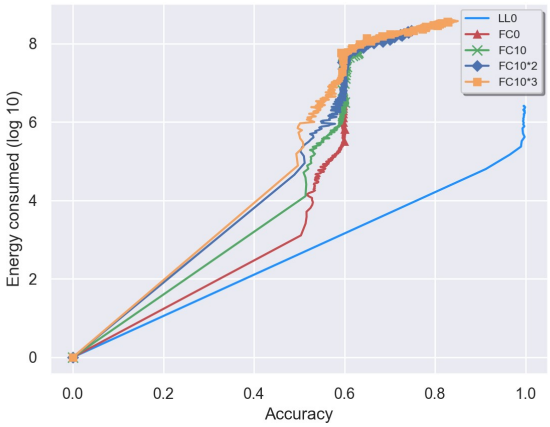


**Figure 5.13: Accuracy of the test set and energy consumption for the spiral dataset.** The energy consumption varies widely between the networks, so it is plotted on a logarithmic scale with base 10.

### 5.4.3 Sickness

The LL0 model outperforms all three vanilla networks in regards to energy on the sickness dataset as well. The difference is not as large as the case with the Spiral dataset, but the vanilla networks consume approximately 10 times the energy of the LL0 model.

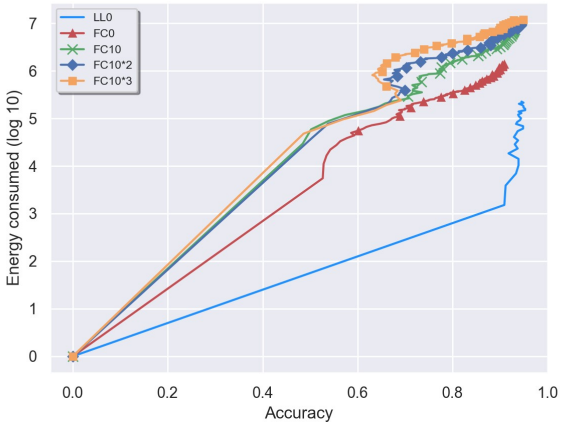


**Figure 5.14: Accuracy of the test set and energy consumption for the sickness dataset.** The energy consumption varies widely between the networks, so it is plotted on a logarithmic scale with base 10.

### 5.4.4 Wines

The LL0 model also outperforms the vanilla networks with a factor $\geq$10 on the wines dataset in regards to energy consumption.

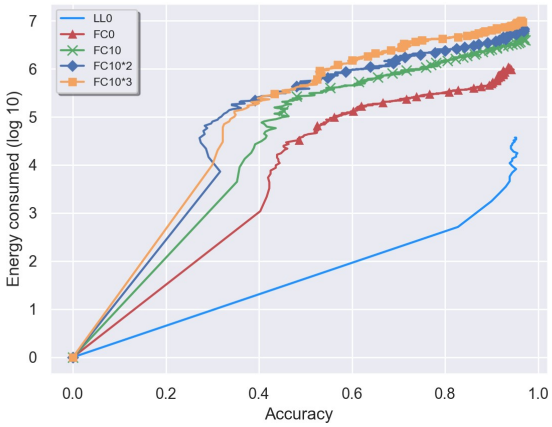


**Figure 5.15: Accuracy of the test set and energy consumption for the wines dataset.** The energy consumption varies widely between the networks, so it is plotted on a logarithmic scale with base 10.

The LL0 model has some clear advantages when it comes to energy consumption. These advantages are mainly encountered early in the training phase, where it performs a lot better than the other networks. In the latter stages of training a single iteration is much more costly. This is mainly due to the fact that the higher accuracy the LL0 model achieves, the more well-built the structure is. A larger structure automatically implies that the model has more nodes, which increases the energy consumption in each forward and backpropagation. The vanilla networks on the other hand, retain the same energy consumption for each point in each iteration.

## 5.5 Versatility

When training the LL0 model on four separate datasets, with the same hyperparameters, the results are very much alike. The model achieves a high accuracy after only seeing a few images (relative to the size of the dataset), while consuming a modest amount of energy to reach said accuracy. This can be observed in the figures for the aformentioned datasets in Section 5.1 The same trend can be observed in the Section 5.4, where each of the four datasets show similar results for the energy consumed in relation to accuracy. However, energy consumption does vary, as the complexity of the problem as well as the size of the input dimensions impacts how many nodes that are being created. While a fully connected network with a fixed input size

and fixed layers always use the same energy per epoch, regardless complexity of the problem, the LL0 model is more unpredictable.

Furthermore, the LL0 model manages to solve all four datasets with a high accuracy, even though the problems are very different. This can be compared to the vanilla networks that only need 25 epochs for the Sickness dataset, but more than 1500 epochs for the Spirals dataset for the best vanilla network. The other vanilla networks are not even able to solve the task.

## 5.6 Generalization

The LL0 model was also compared against a *K-Nearest-Neighbour* algorithm on a simple dataset of berries. This test showcases the importance of the generalization rule. It is through the creation of sub-concepts these results stem from. The berries dataset consist of 100 berries $(X_b, T_b)$, with one input $x_b$ containing 100 input dimensions, where three are always fixed, and the rest are noise, and label $t_b = 1$. More specifically $x_{b,49} = 0.2$, $x_{b,50} = 0.4$, $x_{b,51} = 0.6$, while all other indices $x_{b,i}$ are given a random value $\in [0, 1]$ with equal probability. The dataset also contains 100 non-berries $(x_{nb}, t_{nb})$ with $x_{nb}$ having all its elements $x_{nb,i}$ as random values $\in [0, 1]$ with equal probability, and $t_{nb} = 0$. All $x_{nb}$ further has the requirement $||(x_{nb,49}, x_{nb,50}, x_{nb,51}) - (0.2, 0.4, 0.6)|| > 0.2$, to distinguish the non-berries from the berries.

This is supposed to simulate a real-life task with two scenarios. In the first scenario the agent eats a berry in different situations. All the berries taste the same, and yield the same reward, but they are all eaten in different conditions. Some could for example be eaten when it is hot or cold outside, or some could be eaten at different altitudes, different locations etc. These scenarios are represented by the berries datapoints. The second scenario is all other cases where a berry is not eaten, represented by the non-berry datapoints. The goal is to learn to recognize a berry, independent of what other stimuli input is received.

This can be considered as a real life problem with sensor inputs. It is important in these cases that a model can distinguish which inputs that are relevant for a given task, and which that are considered noise, something called *feature selection* - even during a short training time. These 200 datapoints are used for training, and another 100 berries datapoints are created and used in the test set. Therefore the test set only evaluates how well the berries datapoints are remembered.

In Figure 5.16 the accuracy for the LL0 model and a KNN model is observed. The backpropagation for the LL0 is turned off, and the structure is built entirely on the generalization_sub rule. The vanilla networks perform stochastically with an accuracy $a \in \{0..1\}$, which is why a KNN is used instead. The LL0 also has different parameters than those used in the other simulations. The KNN investigates the $K$-th closest neighbours, where $K$ is set to 3 in this simulation. One can observe in the

figure that LL0 reaches 100% accuracy after 0.2 epochs. This corresponds to the model seeing 40 datapoints. That is approximately 20 berries. Therefore with all these noisy stimuli disturbing the perception of eating a berry, the model recognizes eating a berry after eating 20 of them, and almost recognizes the berry after eating approximately 10 of them.

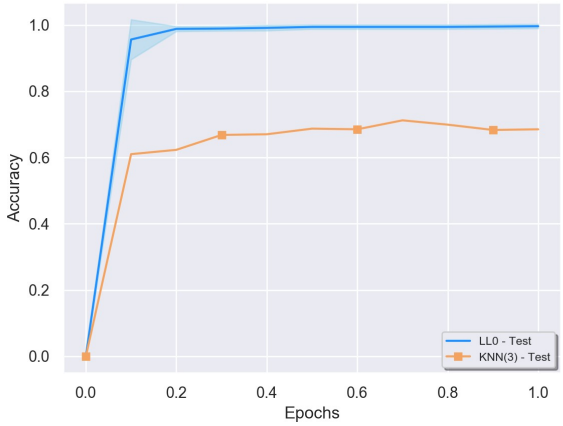


**Figure 5.16: The prediction accuracy for the berries dataset.** The dataset is run for one epoch, and only the training accuracy is visualized. The KNN uses a $K$ of 3.

In Table 5.1 the results for $K = \{1..10\}$ is displayed. All values are averaged over 10 runs. The runs are different from that in Figure 5.16, and thus the graph does not represent when $K = 3$ from the table. When $K = 9$ the best result is reached. In comparison, the LL0 model reaches an accuracy of 100% after less than 0.2 epochs.

| **Epoch** | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|
| ***k*** | | | | | | | | | | |
| 1 | 0.59 | 0.57 | 0.58 | 0.61 | 0.62 | 0.61 | 0.62 | 0.63 | 0.64 | 0.65 |
| 2 | 0.27 | 0.38 | 0.38 | 0.38 | 0.41 | 0.42 | 0.41 | 0.39 | 0.39 | 0.39 |
| 3 | 0.57 | 0.62 | **0.65** | 0.65 | 0.66 | 0.65 | 0.66 | 0.67 | 0.69 | 0.68 |
| 4 | 0.40 | 0.42 | 0.44 | 0.45 | 0.48 | 0.49 | 0.49 | 0.48 | 0.51 | 0.53 |
| 5 | 0.53 | 0.61 | 0.63 | 0.63 | 0.63 | 0.65 | 0.66 | 0.68 | 0.69 | 0.71 |
| 6 | 0.47 | 0.50 | 0.51 | 0.55 | 0.57 | 0.57 | 0.56 | 0.55 | 0.57 | 0.58 |
| 7 | 0.51 | 0.55 | 0.63 | **0.69** | 0.68 | 0.68 | 0.70 | 0.70 | 0.72 | 0.72 |
| 8 | 0.52 | 0.57 | 0.52 | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 | 0.61 | 0.62 |
| 9 | 0.51 | 0.61 | **0.65** | 0.66 | **0.70** | **0.69** | **0.71** | **0.72** | **0.74** | **0.75** |
| 10 | **0.60** | **0.69** | **0.65** | 0.63 | 0.64 | 0.68 | 0.64 | 0.63 | 0.62 | 0.65 |

**Table 5.1: Table with the accuracy for a KNN model with 10 different $K$s during one epoch on the berries dataset.** Only the testing accuracy is displayed.

# 6

# Discussion

In this section we discuss the different properties of the LL0 model, and the results that LL0 obtains on the executed experiments. We also discuss ethics and what impact models similar to LL0 might have on society. We also discuss similarities with other machine learning algorithms and future work for the LL0 model.

## 6.1 LL0 model

The model presented in this thesis, the LL0 model, is a new approach to how neural networks can be built and designed. Animals and humans have an ever-changing and learning brain through neuroplasticity, without a fixed structure. Thus it does not really make any sense to model a brain with limited architecture, as most models do.

The concept of transfer learning has been evident in the entire thesis, but has not been displayed in the result section. There is no evidence that actual transfer learning takes place in LL0. However, as the model has shown great capabilities of one-shot learning, it is hard to estimate if transfer learning takes place as the model usually achieves a maximum accuracy within the the beginning of the first epoch. What we do know however, is that the structure is preserved and used. This means that if we train the model on car models, we will have a generalized concept of a car. To add a new car model would then only add one new node from the generalized node. If we were to instead train the model from scratch, on only the new car model, we would build up a new structure from scratch. We have not shown that we have transfer learning in the form of improved accuracy, but transfer learning occur in the form of reusing and extending already built up structure. This can also be put in contrast to fully-connected networks that must often be retrained as soon as a new class is added, whereas LL0 does not. Furthermore, a fully-connected network often overwrites existing knowledge when trained on new tasks, where the LL0 model simply adds another feature.

The LL0 model has gone from idea to working implementation, and its performance is astounding on the datasets that we have tried it on. One drawback with a network that can grow and shrink is that it could possibly be hard to put it to actual use inside

a product. It is hard to know in advance how much memory and computational power that must be allocated given a certain problem for the network to reach its full potential. As the LL0 model has the possibility to be limited to a maximum size, this problem is mitigated.

Currently the LL0 model uses a batch size of 10. If we increase this size to much, we may encounter problems when we want to do an extension. If we have a batch size of 50, and want to do an extension on iteration 48, we need to either deal with the extension, and then updated the parameters or vice versa. If we update the parameters in beforehand, we might not even need the extension at all. Thus it makes more sense to do fewer iterations before updating, as this will likely decrease the risk of a faulty node being added. This is however something that needs to be investigated further, as larger batch sizes can decrease the risk of getting stuck at a local minimum and might therefore be wishful.

The four main ideas of the LL0 model; extension, generalization, forgetting, and backpropagation are quite easy to understand, but it is harder to formulate precise rules for when they should be applied. Forgetting, for instance, currently simply removes the nodes with the lowest activation. In contrast, the rules for extension and backpropagation are quite thoroughly worked through. This opens up for more thorough research on how, when, and to what extent the different rules should be applied, and if there are better strategies than the ones currently implemented.

It is also difficult to determine how the model will perform on large scale datasets, such as real-time images from autonomous driving. The datasets used in this thesis are merely a playing ground for smaller networks, and there is no real evidence that the LL0 model will be able to perform on actual real-life problems, even if the theoretical scaling seems promising.

What this thesis gives evidence to is the feasibility of expanding networks. To reach accuracies of more than 90% after less than one epoch must surely be something that all current implementations of neural networks should be inspired by. This is in large part due to the initialization of the different values in the model, which are pinpointed and carefully calculated, instead of randomized.

## 6.2 Ethics

As there are numerous different ways to apply machine learning, it is important to take ethics into account when implementing and developing new frameworks. Neural networks that can detect which people are more susceptible to harmful kinds of advertisement, must not be allowed to roam free without regulations.

There are many examples of machine learning cases that have gone wrong already, and many theoretical problems that need to be solved before large scale machine learning can become a reality. All machine learning techniques make decisions based on the data it is trained on. This leads to the requirement that the data needs to

be completely unbiased and ethical in order for the machine learning model to be ethical. However, most data in today's society is not completely unbiased. Most data gathered is usually based on human interference in some way, and humans are biased. A machine learning algorithm can make predictions on a task based on religion, gender, skin-color or country of origin. Making predictions based on these features are usually presumed as unethical. One example is Microsoft AI's chatbot Tay, which became corrupted by twitter and started to write antisemitic comments. Another example is an advertising software used by LinkedIn that preferred males to females. But one of the most severe examples is a software used by American courts during sentencing, predicting the likelihood that a person would relapse into crime - a software that turned out to be negatively biased against african-americans [2].

One of the most popular theoretical problems is the *trolley problem*, which is stated in the following way:

*A trolley is on its way to run over five tied up people, you personally can pull a lever to change the course of the trolley to run over a single person instead. The ethical problem is that you either pull the lever, changing the outcome and kill one person, or you do nothing, do not impact the outcome, and five people die.*

This problem can be put into many AI examples but the main point is clear. AI might have to make these difficult decisions, valuing different peoples lives based on some criteria. For example for an autonomous car in a trolley scenario, should it kill five people or just the one? What if there is a child and an old woman instead? What if it is two equally aged people but of different gender or ethnic groups? These are ethical challenges which either has to be pre-determined, or one has to let the data do the choice. However since data is usually unbiased, so will most likely the decision be.

As the LL0 model displays a more detailed level of explainability, one layer of the *black box*-magic that is machine learning has been removed. For instance, if a facial recognition network would, in some way, mistakenly classify humans as animals - which has happened, the structure of the LL0 model allows for an easier understanding to why this would occur, and thus how to prevent it. We can therefore argue for why the model makes the predictions that it does, something that previously has been lacking in fully connected neural networks. This can also be seen under Section 5.2.

The explainability, while being positive, also poses a threat. Companies and people can use machine learning with the intention to harm or trick people. Currently, the explainability possibly imposes a new level to tweak the model to do this more effectively - in the same way that it can be tweaked to reduce it.

## 6.3 Related Work

To the best of our knowledge there is no algorithm working in the same way as the LL0 algorithm. While there are some similarities to other algorithms, we have not found any other algorithm that successfully combines one-shot learning, transfer learning, lifelong learning, and the ability to have a completely dynamic architecture that starts from nothing and can both increase and decrease in size after demand. Most algorithms aim to solve one or two of these problems, but not all at the same time.

DEN, for instance, avoids catastrophic forgetting by its split functionality, and also utilizes transfer learning for new tasks based on previously learned tasks. DEN also has a dynamic architecture, it can however not remove parts of the network if they become obsolete, which might be necessary in a realistic lifelong learning scenario with finite memory. Furthermore, DEN focuses on incremental learning, where the start and stop of each task is pre-defined. Pre-defined tasks might not exist for all lifelong learning scenarios. One might not know from just sensor inputs when a task starts and ends. LL0 on the other hand, do not require pre-defined tasks. DEN is probably the algorithm that is most similar to LL0 even if it differs quite a lot. The LL0 model can also learn from DEN. For instance the split functionality to avoid catastrophic inference could be applied in the LL0 model. Similarly the progressive networks and the H-DRLN algorithms require pre-defined tasks.

The Neural Turing Machine approaches are quite different from LL0 since they use external memory outside of the network to achieve one-shot learning and to avoid catastrophic forgetting. The NEAT algorithm focuses on developing a minimal structure, and the cascade correlation algorithm does not achieve one-shot learning as it trains the network to convergence before it adds a new node.

AdaNet builds a similar structure as LL0; a sparse network which does not only have connections between adjacent layers. However AdaNet is not constructed to specifically solve lifelong learning tasks, but instead to find a good structure for the task it tries to solve. Therefore ignoring some of the key elements in lifelong learning. This nonetheless motivates the validity of a sparse network with connections that skip layers used in LL0.

While LL0 has some similarities with other algorithms, to the best of our knowledge no other algorithm work in a similar way. On top of that, no other algorithm seems to have even focused on solving the same problems that LL0 aims to solve. The combination of a truly dynamic network which can expand and shrink, utilize transfer learning in terms of structure, one-shot learning, try to minimize catastrophic inference and to not use any pre-defined tasks, highlights the novelty of the LL0 algorithm. Altough this thesis does not present any evidence that transfer learning in terms of accuracy takes place, we are quite confident that it does.

## 6.4 Future Work

The tasks that lie ahead for the LL0 model are many. The model must be vectorized to allow for testing on larger datasets, as well as reducing the time required by the training and inference phases. Currently there exist some obstacles to fully vectorizing the model, and there might be a need for the functionality to swap between the current graph-structure and the vectorized structure for extension and forgetting. As the implementation is currently written in Python, the performance could be boosted immensely by switching to a programming language of greater speed, such as C or C++.

The rule for forgetting needs to be examined in further detail. Currently there is not much details for how or when the forgetting is supposed to be carried out. Extension is far more detailed, but there is still a lot of room for improvement. The rule for generalization has been split up into two sub-rules, where one has been merged with the rule for extension. There might be other ways to do this, and it is still possible to only generalize between extensions for better efficiency. Although not implemented, it should be possible to remove nodes with low activity, even though they have children, by re-arranging the network structure.

Another approach is to stochastically remove nodes when the network converges to potentially eliminate the risk of being stuck in a local minimum, while also reducing the risk of overfitting. Introducing a stochastic element might also increase the performance of the algorithm.

The training phase makes use of batch-mode, meaning that we do not backpropagate on a single error, but use batches instead. Batches could also be implemented to work on extension, so we do not extend as soon as we get an error that tells us to extend. Instead, we could extend in batches, and further reduce the amount of *swaps* that must be made between graph-structure and vectors if the code is vectorized.

The parameters currently used work exceedingly well for all the problems we have tried, but there is still a need for a larger investigation if there is even better parameters to use.

Lastly, we would want to test the LL0 model with more proper pre-processing. If the LL0 model could utilize a CNN as pre-processor, it could be used for larger problems that are not limited to the difficulty level of those we have investigated. Or perhaps integrate CNN into each Concept Node similar to how AOGNets work.

# 7

# Conclusion

In this thesis we wanted to investigate the new algorithm for training neural networks proposed by Claes Strannegård. Our results show that the algorithm performs well, and certainly match the chosen benchmarks. The LL0 model outperforms all chosen benchmarks on all chosen datasets, even when not tweaking the algorithm for each new problem. The one-shot learning capabilites allow for a quick learning, reaching accuracies of $\sim$90% in the beginning of the very first epoch. The size of the different models further show that the algorithm does not simply mimic a truth table for the input.

However, as the datasets used are merely smaller datasets, and not of industrial size, there is not sufficient evidence to say that the algorithm is competitive against more rigorous networks. This thesis instead shows a very promising proof of concept, and can be the underlying base for more detailed testing and development.

# Bibliography


[1] Filippo Amato, Alberto López, Eladia María Peña-Méndez, Petr Vaňhara, Aleš Hampl, and Josef Havel. Artificial neural networks in medical diagnosis, 2013.

[2] Julia Angwin, Jeff Larson, Lauren Kirchner, and Surya Mattu. Machine bias, Mar 2019.

[3] GM Behery, AA El-Harby, and Mostafa Y El-Bakry. Reorganizing neural network system for two spirals and linear low-density polyethylene copolymer problems. *Applied Computational Intelligence and Soft Computing*, 2009:6, 2009.

[4] Irving Biederman. Recognition-by-components: a theory of human image understanding. *Psychological review*, 94(2):115, 1987.

[5] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*, pages 737–744, 1994.

[6] Gonzalvo X. Kuznetsov V. Mohri M. Cortes, C. and S. Yang. Adanet: Adaptive structural learning of artificial neural networks. *In Proceedings of the 34th International Conference on Machine Learning*, 70:874–883, 2017.

[7] Michael Darms, Paul Rybski, and Chris Urmson. Classification and tracking of dynamic objects with multiple sensors for autonomous driving in urban environments. In *2008 IEEE Intelligent Vehicles Symposium*, pages 1197–1202. IEEE, 2008.

[8] Bogdam Draganski and Arne May. Training-induced structural changes in the adult human brain. *Behavioural brain research*, 192(1):137–142, 2008.

[9] Peter S Eriksson, Ekaterina Perfilieva, Thomas Björk-Eriksson, Ann-Marie Alborn, Claes Nordborg, Daniel A Peterson, and Fred H Gage. Neurogenesis in the adult human hippocampus. *Nature medicine*, 4(11):1313, 1998.

[10] Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In *Advances in neural information processing systems*, pages 524–532, 1990.

[11] Li Fe-Fei et al. A bayesian approach to unsupervised one-shot learning of object

categories. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1134–1141. IEEE, 2003.

[12] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[13] Jim Gao. Machine learning applications for data center optimization. 2014.

[14] Alexander Gepperth and Cem Karaoguz. A bio-inspired incremental learning architecture for applied perceptual problems. *Cognitive Computation*, 8(5):924–934, 2016.

[15] Kimberly Gerrow and Antoine Triller. Synaptic stability and plasticity in a floating world. *Current opinion in neurobiology*, 20(5):631–639, 2010.

[16] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[17] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. Evolving neural turing machines. In *Neural Information Processing Systems: Reasoning, Attention, Memory Workshop*, 2015.

[18] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258, 2017.

[19] William Grant Hatcher and Wei Yu. A survey of deep learning: platforms, applications and emerging research trends. *IEEE Access*, 6:24411–24432, 2018.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[21] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[22] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[23] Eric R Kandel, James H Schwartz, Thomas M Jessell, Department of Biochemistry, Molecular Biophysics Thomas Jessell, Steven Siegelbaum, and AJ Hudspeth. *Principles of neural science*, volume 4. McGraw-hill New York, 2000.

[24] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[25] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neu-

ral networks. *Proceedings of the national academy of sciences*, page 201611835, 2017.

[26] Douglas M Kline and Victor L Berardi. Revisiting squared-error and cross-entropy functions for training neural network classifiers. *Neural Computing & Applications*, 14(4):310–318, 2005.

[27] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2, 2015.

[28] Philipp J Kraemer and Jonathan M Golding. Adaptive forgetting in animals. *Psychonomic Bulletin & Review*, 4(4):480–491, 1997.

[29] Xilai Li, Tianfu Wu, Xi Song, and Hamid Krim. Aognets: Deep and-or grammar networks for visual recognition. *arXiv preprint arXiv:1711.05847*, 2017.

[30] Benno Lüders, Mikkel Schläger, and Sebastian Risi. Continual learning through evolvable neural turing machines. In *NIPS 2016 Workshop on Continual Learning and Deep Networks (CLDL 2016)*, 2016.

[31] Ronald W Oppenheim. Cell death during development of the nervous system. *Annual review of neuroscience*, 14(1):453–501, 1991.

[32] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[33] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *arXiv preprint arXiv:1802.07569*, 2018.

[34] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 2019.

[35] Jonathan D Power and Bradley L Schlaggar. Neural plasticity across the lifespan. *Wiley Interdisciplinary Reviews: Developmental Biology*, 6(1):e216, 2017.

[36] Blake A Richards and Paul W Frankland. The persistence and transience of memory. *Neuron*, 94(6):1071–1084, 2017.

[37] Mark B Ring. Child: A first step towards continual learning. *Machine Learning*, 28(1):77–104, 1997.

[38] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[39] Andrei A Rusu, Matej Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. *arXiv preprint arXiv:1610.04286*, 2016.

[40] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*, 2016.

[41] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[42] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[43] Claes Strannegård, Wen Xu, Niklas Engsner, and John A. Endler. Evolution and learning in artificial ecosystems. In *Proceedings of IJCAI-18 Workshop on Architectures for Generality & Autonomy*, Stockholm, Sweden, 2018.

[44] Claes Strannegård. Dynamic networks. *Manuscript*, 2 February, 2019.

[45] Matthew E Taylor and Peter Stone. An introduction to intertask transfer for reinforcement learning. *Ai Magazine*, 32(1):15, 2011.

[46] Sebastian Thrun and Tom M Mitchell. Lifelong robot learning. In *The biology and technology of intelligent autonomous agents*, pages 165–196. Springer, 1995.

[47] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI Global, 2010.

[48] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[49] Cheng-Hsiung Weng, Tony Cheng-Kui Huang, and Ruo-Ping Han. Disease prediction with different types of neural network classifiers. *Telematics and Informatics*, 33(2):277–292, 2016.

[50] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

[51] Song-Chun Zhu, David Mumford, et al. A stochastic grammar of images. *Foundations and Trends® in Computer Graphics and Vision*, 2(4):259–362, 2007.