Modelling Languages for Deep Learning based Cyber-Physical Systems

Master Thesis

presented by

Timmermanns, Thomas

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr.-Ing. S. Kowalewski

Advisor: Evgeny Kusmenko

The present work was submitted to the Chair of Software Engineering

Aachen, May 31, 2018

Eidesstattliche Versicherung

_____   _____

Name, Vorname   Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

_____

_____

_____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____   _____

Ort, Datum   Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

_____   _____

Ort, Datum   Unterschrift

Abstract

Deep learning becomes increasingly important in the domain of cyber-physical systems with each passing year. C&C models are widely used in design and implementation of cyber-physical systems to increase safety and reliability by dividing a complex system into smaller components that can be individually tested. The objective of this thesis was to develop a modeling language that combines regular computations with deep learning techniques in a C&C design with a focus on cyber-physical systems. Thus, we developed the descriptive modeling language CNNArch for convolutional neural networks and integrated it into the textual modeling family MontiCAR that is based on the C&C paradigm. Furthermore, we created a code generator for the resulting models, which combine neural networks with regular mathematical computations, and tested the generated code successfully with a small C++ application that can recognize certain objects in images.

Contents

Chapter 1

Introduction

Deep learning becomes increasingly important with each passing year. It shows exceptional results in the areas of computer vision, artificial intelligence, natural language processing and many more. Especially self-driving cars and robot navigation systems rely more and more on convolutional neural networks and deep learning for all kinds of visual tasks, e.g. the detection of other cars based on camera images. In these areas, conventional software cannot compete with a deep learning algorithm. However, *cyber-physical systems* (CPS) in the automotive and robotics domain have special requirements regarding safety and reliability since each error can lead to real damage in the physical realm. The ability to ensure the correct behavior of a component of a system strongly depends on the used software architecture. Thus, a model-driven development is much more convenient to be able to design modular and reusable software components, which can be tested individually. Moreover, a developer can focus on the actual behavior of a software component and can be assured that low-level operations are handled correctly by the modeling tool. This saves time and increases productivity of the development team. In fact, *component and connector* (C&C) models are widely used in design and implementation of CPS for these reasons. However, there exists currently no modeling tool for C&C architectures that allows us to combine deep learning techniques with regular computations in a type-safe manner. This results in systems that are more difficult to test and components that are hard to reuse.

1.1 Task

The task of this thesis was to develop a descriptive modeling language for the architectures of convolutional neural networks that can be integrated into the textual modeling family MontiCAR, which is based on the C&C paradigm and designed to increase development efficiency of CPS. This modeling language should be easy to use and should check the validity of a neural network at the time of model creation. It should ensure that the strong type system of MontiCAR is followed. Furthermore, we have to integrate the language into MontiCAR and develop a code generator that can generate code for a modeled system that combines deep learning components, which are based on the developed modeling language, with regular components for mathematical computations.

1.2 Structure

This thesis will start with the preliminaries in chapter 2. In the course of this chapter, we will introduce and explain the basic concepts, which are needed to understand this thesis, and the main software tools, which we used to fulfill our task. In the following chapter 3, we will present and compare many existing deep learning frameworks so that we can design our modeling language based on the gained knowledge. The chapter 4 will first discuss the context in which the modeling language was developed so that we can formulate the requirements of the developed tool. Thereafter, we will explain how we designed the language and the code generator to fulfill these requirements. The chapter 4 is about he implementation of designed language and code generator and in the following chapter 5 we will evaluate the generated product of the developed language. Finally, we will summarize the results of this thesis in chapter 7.

Chapter 2

Preliminaries

In this chapter, we will cover the basics needed to understand this thesis. First, we will state the mathematical notation used in this thesis. Then, we will explain directed graphs which are referenced multiple times throughout the thesis. After that, machine learning will be presented with a focus on deep learning. The emphasis will be specifically on the architectures of convolutional neural networks. Then, we will discuss the data storage format HDF5 in the context of deep learning. Thereafter, we will first present component and connector architectures and then the Freemarker template engine, which we will use for code generation. Finally, we will explain the language workbench MontiCore that we will use to develop a DSL for deep learning.

2.1 Mathematical Notation

We will generally use standard mathematical notation throughout this thesis and we will assume in some instances that the reader has a basic knowledge of calculus and linear algebra. However, we want to clarify some of the used symbols and notations in the following. The symbols $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ denote respectively the sets of the natural, whole and rational numbers. The notation $[a, b]$ denotes the *closed* interval between a and b, that is the set of all numbers which are greater or equal than a and less or equal than b. Furthermore, the notation $(v_1, \ldots, v_n)$ denotes a vector with n elements and is written as the bold letter $\boldsymbol{v}$. In a similar way, matrices are also written as bold letters. In contrast, sets are always upper case letters. The Cartesian product of two sets A and B is $A \times B$ and the n-ary Cartesian power of the set A is written as A^n. Finally, the relative complement (or difference) of set A in respect to set B is written as $B \backslash A$.

2.2 Directed Graphs

A *graph* in mathematics is a pair $G = (V, E)$ consisting of a set of *vertices* (or *nodes*) $V \neq \emptyset$ and a set of *edges* E [JJ99]. A graph can be *directed* or *undirected*. The set of vertices is equal in both cases but we define edges differently. In an undirected graph, an edge e is an unordered pair a, b where a and b are elements of V. The vertices a and b are called the *end vertices* of edge e. In a *directed graph* (or *digraph*), the edges are directed which means that an edge e is an ordered pair (a, b) where $a \neq b$ are vertices of the graph
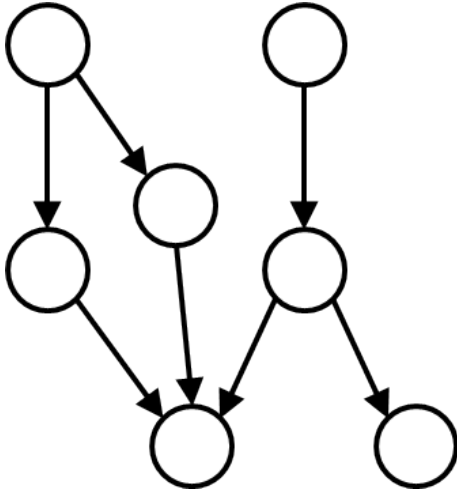
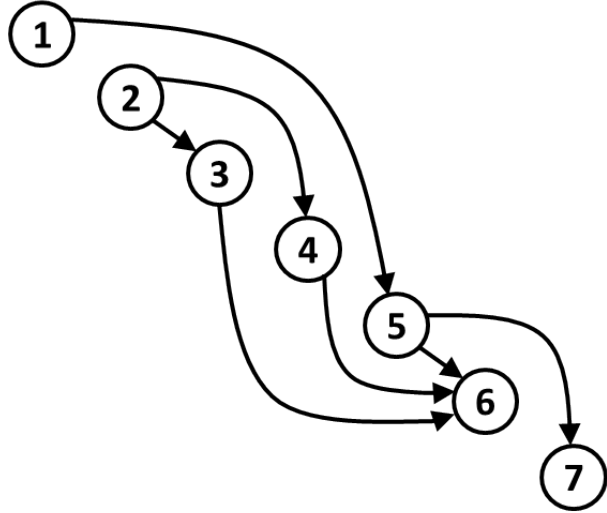Figure 2.1: Example of a directed acyclic graph.

Figure 2.2: Visualization of a topological ordering of the graph in figure 2.1. All connections point in the same direction.

[JJ99]. The vertex a is called the *start vertex* or *tail* and b the *end vertex* or *head* of e. For both types of graphs holds that a and b are called *adjacent* or *neighbors* of each others if an edge exists which contains these vertices [Wil79]. Hereinafter, we will only focus on directed graphs and assume all graphs are directed if not otherwise specified.

Let $(e_1, \ldots e_m)$ a sequence of edges in graph G with $m \in \mathbb{N}$. We call this sequence a *walk* if there are vertices $v_0, \ldots, v_m$ such that $e_i = (v_{i-1}, v_i)$ holds for $i \in \{1, \ldots, m\}$. A walk that has distinct edges e_i is called a *trail*. If additionally the vertices are distinct (except, possibly, $v_0 = v_m$), we call the sequence a path of G. A walk, trail or path is *closed* if v_0 is equal to v_m. A closed path containing at least one edge is called a *cycle* [Wil79]. A graph is *acyclic* if it contains no cycle.

A graph is called *connected* if it cannot be expressed as a union of two graphs. The union of the two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as the graph $G_3 = (V_1 \cup V_2, E_2 \cup E_2)$. A digraph is called *strongly connected* if, for any two vertices v and w, there is a path from v to w [Wil79]. Every strongly connected graph is connected but the opposite is generally not true. For example, the digraph in figure 2.1 is connected but not strongly connected. Furthermore, the *in-degree* of a subset of vertices $W \subseteq V$ is defined as the number of edges whose heads are in W and tails are in $V \backslash W$ [BJG08]. The *out-degree* similarly defined as the number of edges whose tails are in W and heads are in $V \backslash W$. The in-degree and out-degree can be calculated in the same way on a single vertex.

Directed acyclic graphs (DAG) are a well studied family of digraphs and appear often in computer science, e.g. version control systems, dataflow networks, task scheduling and more. An important property of a DAG is that every DAG has a *topological ordering* of its vertices [BJG08]. A topological ordering of graph G is a ordered sequence of vertices $(v_1, \ldots, v_m)$ such that for each edge $e = (v_i, v_j)$ holds that $i < j$. Moreover, each graph with a topological ordering is acyclic. The process of creating an topological ordering from a DAG is called *topological sorting*. Another property is that every DAG has a vertex of in-degree zero as well as a vertex of out-degree zero [BJG08]. The figure 2.2 visualizes a topological ordering of the graph in figure 2.1. Note that the topological ordering is not

unique. For example, it is possible to switch the position of the nodes with index 6 and 7.

2.3 Machine Learning

Machine learning is a subfield of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence. There are two main branches of machine learning, i.e. supervised learning and unsupervised learning. In supervised learning the data consists of instances of the problem and target labels whereas unsupervised learning only looks at the instances and tries to find structures in the data [Bis06]. In this thesis, we focus on supervised learning. A supervised learning algorithm iteratively learns from data consisting of instances and target labels and generalizes the obtained knowledge into a model. The purpose of machine learning models is to predict target labels for novel and unseen instances. Models can be seen as black box functions which get a data point as input and produce the associated target label.

The insight that these models have into the problems is usual impossible to replicate with human written programs. A typical example is the recognition of handwritten digits given in figure 2.3. Each image of a handwritten digit contains 28×28 pixels. These can be represented by a vector $x \in \mathbb{N}^{784}$, where each natural number describes the intensity of a pixel. The target labels $t \in \{0, \ldots, 9\}$ are the digits themselves. It is practically impossible to write clear rules how to recognize a digit because of the wide variability of handwriting. It is an easy task for a human but very challenging for a computer. Instead, a learning algorithm is used that receives a large set of n instances $\{x_1, \ldots, x_n\}$ along with their associated target labels $\{t_1, \ldots, t_n\}$ as input. Both together are called the *training set*. The training set is used by the learning algorithm to tune the model parameters such that the resulting model can recognize the handwritten digits with reasonable accuracy.

Supervised learning differentiates between the tasks of classification and regression. The aim of classification is to assign an input vector to one of a finite number of discrete categories [Bis06]. These categories are called classes and the resulting model is called classification model. The recognition of handwritten digits is an example of classification. In regression, the output contains one or more continuous values. A simple example of regression is curve fitting. The task of estimating the best fitting function for a set of two dimensional data points. A more complicated example would be the estimation of the distance between a camera and an object based on the camera images as input. Classification can be seen as a special case of regression. A model based on regression is called a predictive model. In practice, classification models with a number of classes C are often technically predictive models because their output is a continuous vector $[0, 1]^C$, which can be interpreted as the probability that the input belongs to the corresponding class.

Figure 2.3: Example of handwritten digits from the MNIST dataset [LBD$^+$89].

There exists a variety of learning algorithm, for example decision trees, support vector machines, naive Bayes classifiers, neural networks and deep networks. Also there is a variety of deep network architectures such as *convolutional neural networks* (CNN), *deep belief networks*, deep *boltzmann machines* or *recurrent neural networks* (RNN). Deep networks produce state of the art results on many problems. CNNs perform especially well on image recognition and other computer vision related tasks. The MNIST dataset of handwritten digits is a very simple example of image recognition. Even relatively simple convolutional neural networks can achieve human level performance on it [LBD+89].

Classical machine learning algorithms did not work directly on the input data because the raw data was too big and too complex for the learning algorithm. Instead techniques were used to compute specific features of the data. These features could be anything as long as they describe the data well. For example, features in computer vision could be simple ones like colors, edges and corners, or more complicated ones like Scale-Invariant Feature Transform (SIFT) [Low04] and many others. A good outcome highly depended on the used features and it often required a lot of research to find good features. Moreover, feature engineering was a manual and time-consuming task. The main difference of deep learning, which stands for machine learning with deep neural networks, and classical machine learning (or shallow learning) is that deep networks are able to operate on raw data. They can learn the features themselves. This makes them much more powerful than previous techniques. Thus, deep learning is currently one of the most researched topics in machine learning and related fields.

In this thesis, we are focused on convolutional neural networks. Therefore, we will first explain how neural networks and convolutional neural networks function and present multiple architectures of CNNs used in practice.

2.3.1 Neural Networks

Artificail neural networks or short *neural networks* are computation models which have their origin in an attempt to find a mathematical representation of information processing in biological systems [Bis06]. A neural network is built from a collection of *neural units* (or *neurons*) which are connected to each other. Each connection transmits a signal in form of a number from one neuron to another. The receiving neuron processes the signals and sends the result to the following neurons. The connections between neural units are in most cases *directed* which means that the signals only flows in one direction. Notable exceptions to this are boltzmann machines [AHS85] and deep belief networks [Hin09]. However, these are rarely used in practice and we assume hereinafter that all connections between neural units are directed.

The simplest form of neural network, also called single-layer *perceptron* [Ros58], consists of a collection of inputs which are connected to a layer of output neurons. The output neuron with index k calculates its result in the following way:

$$y_k\left(\boldsymbol{x}\right) = \sigma\left(\sum_{j=1}^{M} w_{kj}x_j + w_{k0}\right) \tag{2.1}$$

where M is the number of input neurons, $\boldsymbol{x}$ is a vector containing the data or in other words the signal from each input neuron and σ is the so called *activation function*. The
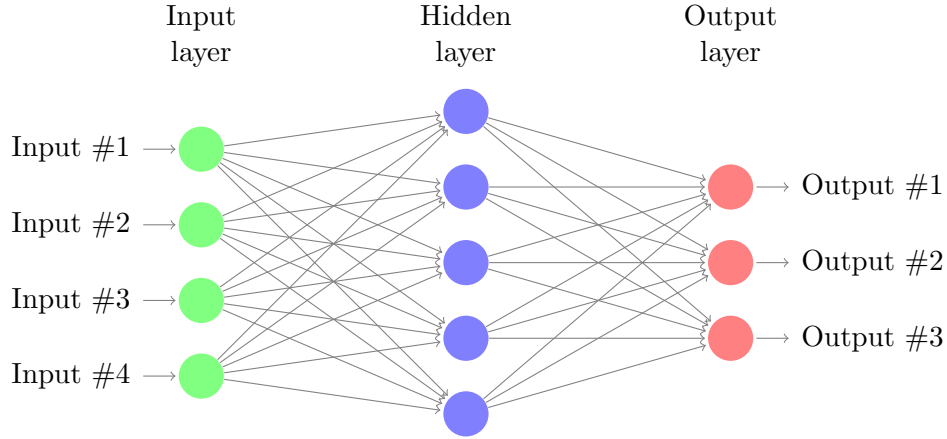
Figure 2.4: A neural network with two fully connected layers. It consists of a 4 input units, a hidden layer with 5 neurons and a output layer with 3 neurons.

matrix w contains the *weights* of the neural network. The weight w_{k0} is called the bias of the output neuron with index k. For all $j > 0$, the weight w_{kj} belongs to the connection from input neuron j to the output neuron k. The weights are learned parameters and determine the output of the perceptron. In the *training* of the network, the weights are optimized to predict the labels of the training set (see section 2.3.2). Furthermore, the result of an activation function is called the *activation* of that neuron.

A single layer perceptron is not enough to compute all functions [MPB17]. For example, it is not even possible to compute the logical XOR function. However, we can expand the network by introducing additional layers, called *hidden layers*, between the input and the output layer. The figure 2.4 shows an example of a neural network with one hidden layer and multiple output units. The activation of the output neuron with index k can be calculated in the following way:

$$y_k\left(\boldsymbol{x}\right) = \sigma^{(2)}\left(\sum_{j=1}^{M} w_{kj}^{(2)} z_j\left(\boldsymbol{x}\right) + w_{k0}^{(2)}\right), \text{ where } z_j\left(\boldsymbol{x}\right) = \sigma^{(1)}\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) \quad (2.2)$$

where D is the number of input neurons, M the number of units in the hidden layer. The superscripts of the weights and the activation functions denote to which *fully connected layer* they belong. The superscript (1) denotes the hidden layer and (2) denotes the output layer. A fully connected layer is a collection of neurons where each neuron of the previous layer has a connection to a neural unit from the fully connected layer. More hidden layers can be added to a network by nesting more activations similar to $z_j\left(\boldsymbol{x}\right)$. Note that the neural network is a predictive model and not a classification model. Classifications with neural networks are implemented by creating one output neuron for each class and by treating the result as a probability or confidence factor for that class. To get proper probabilities for each class, it is necessary to choose the right activation function for the output layer. In practice, the *softmax* activation function is used for that purpose. This function and others will be explained in section 2.3.4.

Activation functions between layers are necessary to bound outliers and to introduce a non-linearity into the network. All networks without a non-linear activation function could be reduced into a single layer perceptron and would still not be able to compute the XOR

function. However, a neural network with one hidden layer and an activation function on that layer can approximate all possible functions. This was proved by K. Hornig in 1991 [Hor91]. Approximation in this context means that the neural network can get arbitrarily close to the computed function by increasing the number of hidden units (neurons in the hidden layer). Increasing the number of hidden layers does not change the computational power in theory but it can reduce the number of hidden units necessary to approximate the desired function well. In practice, neural networks with only one hidden layer are not sufficient to learn complicated functions and deeper networks are required for most tasks. However, deep networks, which are all networks with more than 2 or 3 hidden layers, are more difficult to train which is why it took a long time to make deep networks viable. Attempts to train deep networks before the year 2006 were not successful [GB10].

2.3.2   Network Training

The training of a neural network is an iterative optimization process. At each step the weights of the network are modified to better predict the target labels of the training set. For that an algorithm, called *optimizer*, is used which minimizes the *loss* of the network. The loss strongly relates to the error that the network makes on the training set. The difference between loss and error is that the loss has to be computed in such a way that optimization is possible. This means for example that the *loss function* has to be differentiable. There exist multiple loss functions for different purposes but all compute the loss based on the predicted values of the current network and the real target labels of the input data. Different loss functions are presented in section 2.3.5. For this section, we assume that a differentiable loss function E exists.

The training process of a network starts always by initializing the network parameters with random values. The initialization itself is very important and has a big impact on the training. Bad initialization was one of reasons early attempts on training deep networks failed [GB10]. Parameters of the network have to be initialized according to a certain probability distributions to avoid these problems. A common way to initialize parameters is the *Glorot initializer* (or *Xavier initializer*) named after Xavier Glorot who proposed it in [GB10]. The idea is to draw random values for each neuron from a probability distributions (usually a normal distribution) with zero mean and the following variance:

$$Var(\boldsymbol{w}) = \frac{2}{n_{in} + n_{out}} \tag{2.3}$$

where $\boldsymbol{w}$ denotes the initialized weights of the neuron in question, and n_{in} and n_{out} are respectively the number of inputs and outputs of the neuron.

All optimizers are based on the technique of *gradient descent*. Gradient descent is a general algorithm to find minima or maxima of arbitrary differentiable functions. The process of gradient descent is visualized in its simplest from in figure 2.5. Here, we use it to train our network by minimizing the loss function. To do that, we iteratively update all weights $\boldsymbol{w}$ of the network in the direction of the negative gradient of the loss function E according to the following formula [Bis06]:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E(\boldsymbol{w}^{(\tau)}) \tag{2.4}$$

where the parameter $\eta > 0$ is known as the *learning rate* and τ denotes the iteration step. The symbol $\boldsymbol{w}$ is the vector of size n that contains all weights of the network in arbitrary
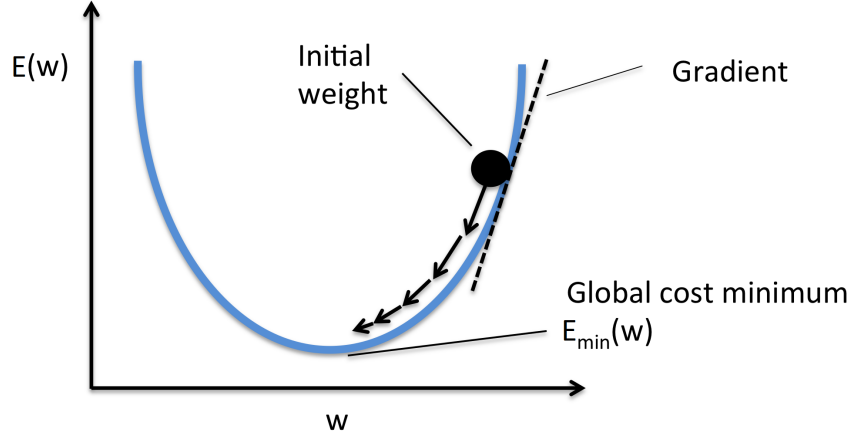
Figure 2.5: Simplified visualization of gradient descent.

order and ∇ is the Nabla-operator which computes the gradient in the following way:

$$\nabla E(\boldsymbol{w}) = \left(\frac{\partial E(\boldsymbol{w})}{\partial w_1}, \dots, \frac{\partial E(\boldsymbol{w})}{\partial w_n} \right) \tag{2.5}$$

The derivations for each weight are calculated with the backpropagation algorithm which is explained in section 2.3.3. Note that the loss function is defined with respect to the training set. Thus, it is necessary to process the entire training set to evaluate the gradient of E. Techniques which use the whole training set at once in an iteration step are called *batch* methods [Bis06]. In practice, it turns out that batch gradient descent is a poor algorithm for multiple reasons. One reason is that it is necessary to store the complete training set in RAM which is difficult for large datasets and deep learning requires a lot of data to train optimally.

However, there is an on-line version of gradient descent, known as *stochastic gradient descent* (SGD) which has proved itself useful on large data sets [L$^+$89]. Stochastic gradient descent in its original form uses the same update rule 2.4 but computes the loss function only on a single instance of the training set in each iteration step. SGD is often implemented by cycling through the data set in sequence. A whole cycle is called an *epoch*. Often, the training set is shuffled between epochs which can lead to better results. Stochastic gradient descent has several advantages over batch gradient descent. First, a single iteration step is computationally fast and does not require a lot of additional memory. Secondly, it handles redundancy in the data much more efficient than batch learning [Bis06]. Thirdly, it is possible to escape from a local minimum of the loss function because of the noisy update process [Bis06]. A deep network will always have a lot of local minima and we can never be sure that we have trained the optimal weights for the network [Ben12]. Therefore, a noisy update is often beneficial to avoid just choosing the local minimum that is nearest from the initialization. However, using SGD with a single instance is often too noisy and the process can never settle on a minimum. Additionally, SGD is as a whole more computationally expensive because it needs much more iteration steps and thus more gradient computations.

As a compromise between these algorithm, *mini-batch gradient descent* was developed. Mini-batch gradient descent is a generalization of classical SGD. Instead of using a single instance, we use the next n instances of the training set as a batch to compute the gradients

of the loss function. With a *batch-size* of $n = 1$, it is equivalent to classical SGD. For that reason, mini-batch gradient descent is in practice often just called SGD. Mini-batch gradient descent combines the advantages of both algorithms if a proper batch-size is chosen.

In practice, mini-batches are used with all neural network optimizers and the batch-size is an important *hyperparameter*. Hyperparameters (or configuration parameters) are paramters of the training process and have to be chosen by the user. Good hyperparamters heavily influence the result of the training process. The learning rate is also a very important if not the most important hyperparameter [Ben12]. The learning rate determines how much the weights change in each iteration step. If the learning rate is to large, the loss can actually be increasing instead of decreasing since the updates overshoot the minimum. And if the learning rate is too low, the network will a long time to train and will not be able to escape bad local minima. For that reason, we often use a learning rate schedule instead of a fixed learning such that the training can start with a relatively high learning rate that will decrease in the training process. A common way to do this, is the *step decay function* that uses exponential decay together with a step size to reduce the learning rate after a certain amount of iteration steps. The current learning rate is calculated based on a decay factor $\gamma < 1$ and the current iteration step in the following way:

$$learning\_rate = base \cdot \gamma^{\lfloor iteration/step\_size \rfloor} \tag{2.6}$$

The number of epochs that the network is trained on can also be viewed as a hyperparameter. However, the number of epochs is often dynamically decided by stopping the training process after a certain amount of time or after a certain accuracy is reached. Furthermore, more advanced optimizers than SGD like *Adam*, *RMSProp* and *AdaDelta* add additional hyperparameters to the training process which we will not discuss in this thesis.

2.3.3   Backpropagation

The error *backpropagation* algorithm is an efficient way to evaluate the gradient of the loss function [Bis06]. Backpropagation works in two steps: a *forward* and a *backward pass*. The forward pass is the prediction or classification of the data where all activations and intermediate values are saved. We use the following notations for the forward pass:

$$a_j^{(k)} = \sum_{i=0}^{m_k} w_{ji}^{(k)} z_i^{(k-1)} \qquad\qquad \forall j, k \geq 1 \tag{2.7}$$

$$z_j^{(k)} = \sigma_k(a_j^{(k)}) \qquad\qquad \forall j, k \geq 1 \tag{2.8}$$

The symbols $a_j^{(k)}$ and $z_j^{(k)}$ denote respectively the weighted sum and the activation of the neuron j in the layer k. The bias of neuron j in layer k is represented by $w_{j0}^{(k)} z_0^{(k)}$ where $z_0^{(k)}$ is always equal to 1. Furthermore, $z_i^{(0)}$ is for all $i \geq 1$ the network input, m_k is the number of inputs of the fully connected layer k and σ_k is the activation function of layer k. For a network with a single layer, $z_j^{(1)}$ is equivalent to y_j in the equation 2.1.

In the backward pass, we use the chain rule of calculus to compute the derivatives of each weight based on the values of the forward pass. To do that efficiently, we start from the

output layer and use the known derivatives of a layer to compute the derivatives of the previous layers. First, we define the symbol δ in the following way:

$$\delta_{jk} = \frac{\partial E(\boldsymbol{z}^{(n)})}{\partial a_j^{(k)}} \tag{2.9}$$

where j denotes the neuron, k denotes the layer and E is the loss function that we compute based on the activations of the output layer n. Note that the loss function depends on the iteration step, i.e the target labels, which we omit for the sake of simplicity. Furthermore, due to equation 2.7, we know that the following holds:

$$\frac{\partial a_j^{(k)}}{\partial w_{ji}^{(k)}} = z_i^{(k-1)} \tag{2.10}$$

and if we know all δ_{jk}, we can compute the elements of the gradient with the following equation:

$$\frac{\partial E(\boldsymbol{z}^{(k)})}{\partial w_{ji}^{(k)}} = \frac{\partial E(\boldsymbol{z}^{(n)})}{\partial a_j^{(k)}} \frac{\partial a_j^{(k)}}{\partial w_{ji}^{(k)}} = \delta_{jk} z_i^{(k-1)} \tag{2.11}$$

For the output layer n, we can calculate δ_{jn} directly. It is the product of the derivatives of loss and activation function:

$$\delta_{jn} = \frac{\partial E(\boldsymbol{z}^{(n)})}{\partial z_j^{(n)}} \frac{\partial z_j^{(n)}}{\partial a_j^{(n)}} = \frac{\partial E(\boldsymbol{z}^{(n)})}{\partial z_j^{(n)}} \sigma_n'(a_j^{(n)}) \tag{2.12}$$

We cannot compute δ_{jk} directly for the previous layers. However, we can use the rules of calculus to compute the derivatives of a layer if we know the derivatives of the following layer [Bis06]:

$$\delta_{j(k-1)} = \frac{\partial E(\boldsymbol{z}^{(n)})}{\partial a_j^{(k-1)}} = \sum_{i=1}^{m_k} \frac{\partial E(\boldsymbol{z}^{(n)})}{\partial a_i^{(k)}} \frac{\partial a_i^{(k)}}{\partial a_j^{(k-1)}} = \sigma_{k-1}'(a_j^{(k-1)}) \sum_{i=1}^{m_k} \delta_{ik} w_{ij}^{(k-1)} \tag{2.13}$$

To summarize, the algorithm is based on the following steps. First, compute all activations of the network based on the network input. Secondly, evaluate δ_{jk} for all neurons of the output layer by calculating the derivatives of the loss function. Thirdly, backpropagate the δ's using the equation 2.13 to obtain δ_{jk} for all neurons j and layers k. At last, evaluate the complete weight gradient with the equation 2.11.

2.3.4 Activation Functions

There exist several activation functions for different task. Some are only used for the output layer, some only in hidden layers and some in both areas. In the following, we will present four commonly used non-linear activation functions.

The classical activation function is the *logistic sigmoid*. It is a smooth and continuously differentiable function and can be calculated in the following way:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.14}$$

The result of the logistic sigmoid is always in the range between 0 and 1. It converts large negative numbers to zero and large positive number to 1. This is one of the main reasons why it was initially used. It reduces the effect of extreme parameters in the network because the activation cannot get arbitrarily large. In the output layer, it can be interpreted as a probability for a binary value, e.g. an activation of 0.8 for the single output neuron stands for a probability of 80% that the label to the given data is equal to 1 (or `true`). The
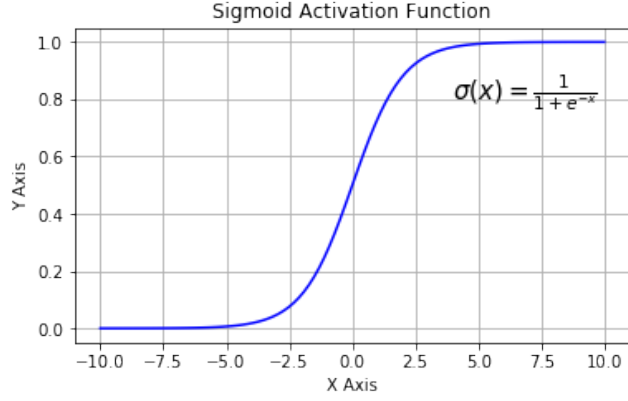

Figure 2.6: The logistic sigmoid.

task of predicting a label which is a binary vector $t \in \{0,1\}^n$ is called *logistic regression*. Nowadays, the logistic sigmoid is in general only used in the output layer for the purpose of general logistic regression or classification with exactly 2 classes since other activation functions perform better in other areas. Classification with 2 classes is special since it can be modeled with a single binary value as label by using the activation of the single output neuron as the probability of the first class. The probability of the second class is calculated by subtracting the probability of the first class from 1.

Historically, logistic regression with the sigmoid was also used to learn classification tasks with multiple classes by comparing the activation for each class and choosing the one with the highest value. This makes sense because labels in classification tasks with n classes can also be viewed as a binary vectors $t \in \{0,1\}^n$ where the value with the index of the correct class is equal to 1 and all other values are 0. However, logistic regression with the logistic sigmoid is not classification. It does not predict a proper probability that the input data belongs to a certain class since the sum of all predicted probabilities, which are computed independently, can be bigger or smaller than 1. For this reason, the *softmax* activation function was developed to properly calculate the probabilities for each class.

The *softmax* function, also called normalized exponential function, is special since its results depend on all neurons in the layer and not only the neuron on which it is applied. It is a generalization of the logistic sigmoid and can be calculated in the following way:

$$softmax(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{m} e^{x_k}} \tag{2.15}$$

where m is the number of neurons in the layer and j denotes the specific neuron in the layer. It is a generalization of the logistic sigmoid because the softmax function with two classes is equivalent to the classification with the logistic sigmoid with two classes. The softmax function is only used as an activation function for the output layer and never for hidden layers. Moreover, it is only used for the purpose of classification. The sum of all activations in a layer with the softmax activation function is equal to 1. Thus, the network can give proper probabilities for each class.

Another activation function is a the *tangens hyperbolicus* (TanH). TanH is a scaled version of the logistic sigmoid and calculated in the following way:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.16}$$

The idea behind TanH is to use a activation function which is similar to the logistic sigmoid but symmetric around the origin. It is only used in hidden layers and not in the output layer. The network learns in general faster with TanH than with the logistic sigmoid [LBBH98]. The reason for this is that a neural network learns better if the mean activations in each layer are close to 0 which is the case for TanH but not for the logistic function.
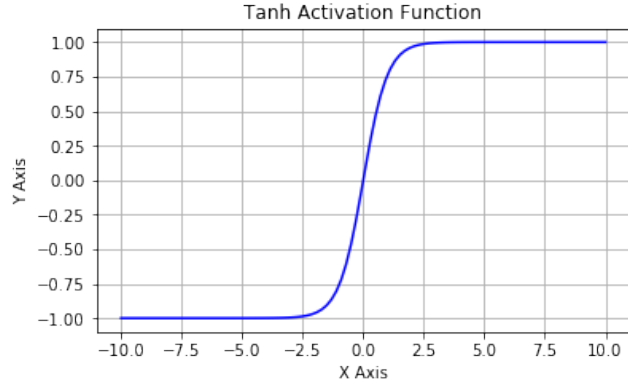
Figure 2.7: The tangens hyperbolicus.

The *rectified linear unit* (ReLU) is probably the most commonly used activation function in deep networks. The ReLU function is only used in the middle of the network and not at the output layer. It is calculated in the following way:

$$relu(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{otherwise} \end{cases} \qquad (2.17)$$

A problem from which both the logistic sigmoid and TanH can suffer is a vanishing gradient. The gradient of the function gets smaller and smaller as the absolute value of the activation increases. This can lead to the neural network getting stuck in training because the parameters barely change in each iteration of gradient descent. Moreover, this effect is amplified by the number of layers in the network which makes the logistic sigmoid and TanH unsuitable for hidden layers of very deep networks.

The derivative of ReLU is always constant but the function is still nonlinear. It is 0 for $x < 0$ and it is 1 for all other values. Thus, the problem of vanishing gradient does not exist for ReLU, which is the reason why it is so popular in deep networks. Additionally, the fact that the activation is 0 for all negative values is often beneficial for deep networks because it induces sparsity in the hidden layers. Certain neurons can be deactivated through high negative parameters which allows deep networks to be trained more efficiently.
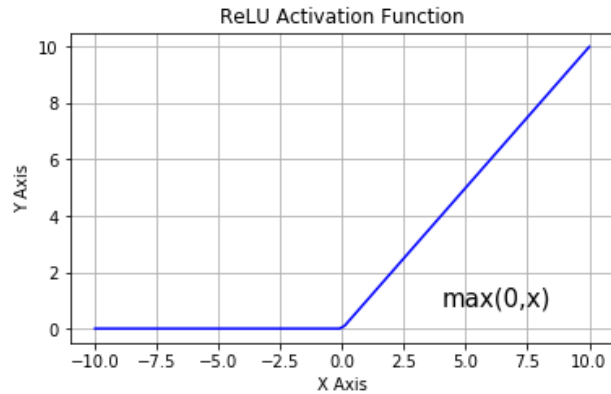
Figure 2.8: The rectified linear unit.

2.3.5 Loss Functions

The ability to correctly predict the target labels of new data that is different from the training set but generated from the same distribution is called *generalization*. The *generalization error* is the average error a model makes on unseen data [DHS12]. Ideally, the goal of training should be to minimize the generalization error. However, this is not

possible. We can only train based on the training set. Moreover, we also cannot use the simple error function which returns a 0 for a correct prediction and 1 otherwise since it is not enough to know that a prediction is wrong. We need to know how much and in which direction we need to change the weights to improve the network. Thus, we use a differentiable loss function with a single minimum for training. There exist multiple possible loss functions. In this section, we will look at the *euclidean loss* (or *square error loss*) and the *cross-entropy loss*.

The euclidean loss is a simple function which can be used for both: classification and prediction. It does not require any specific activation function in the last layer of the network. It is defined by the following equation:

$$E_1(\boldsymbol{x}, \boldsymbol{t}) = \sum_i^n (y_i(\boldsymbol{x}) - t_i)^2 \tag{2.18}$$

where $y_i(\boldsymbol{x})$ computes the output i of the network based on the network input $\boldsymbol{x}$, n is the number of outputs and t_i is the target label for output i. In case of classification, the target label t_i is either 1 if the input belongs to class i or 0. This loss function is very sensitive to outliers [Bis06] which can be a problem as models in practice never make perfect predictions. Training sets for deep networks are big and some instances of these sets can be substantially harder to predict than others. A high sensitivity to outliers can make the training slower. Additionally, the euclidean loss penalizes unnecessarily *too correct* predictions if it is used in classification. Because of this, other loss functions are used for classification in practice.

The cross-entropy loss is in practice heavily used for deep networks. It can be used for logistic regression and classification but not for other types of regression since the loss function requires the network outputs to be between 0 and 1, i.e. softmax or logistic sigmoid as the activation functions of the output layer. The cross-entropy can be calculated in the following way:

$$E_2(\boldsymbol{x}, \boldsymbol{t}) = -\sum_i^n \left(t_i \log(y_i(\boldsymbol{x})) - (1 - t_i) \log(1 - y_i(\boldsymbol{x})) \right) \tag{2.19}$$

This loss function increases almost linearly for misclassified instances and it does not penalize too correct predictions like the euclidean loss. Thus, the cross-entropy loss solves the main problems of the euclidean loss for classification.

The figure 2.9 compares different loss functions. The graph shows the loss for a classification with two classes, which are represented as the labels 1 and -1. The value z is the correctness of the prediction. A value $z > 0$ means that the prediction is correct. Points further away from the origin show more extreme predictions. High negative values on the z-axis are outliers and high positive values are predictions with high confidence. The *optimal loss* is equivalent to the simple error, which only differentiates between correct and incorrect predictions. The *hinge loss* is the loss function commonly used in support vector machines. It is robust to outliers and neither penalizes nor rewards too correct predictions. Unfortunately, the hinge loss is not differentiable at $z = 1$ and can not be optimized directly. However, differentiable approximation of the hinge were developed which can be used in the training of neural networks [SRJ05].
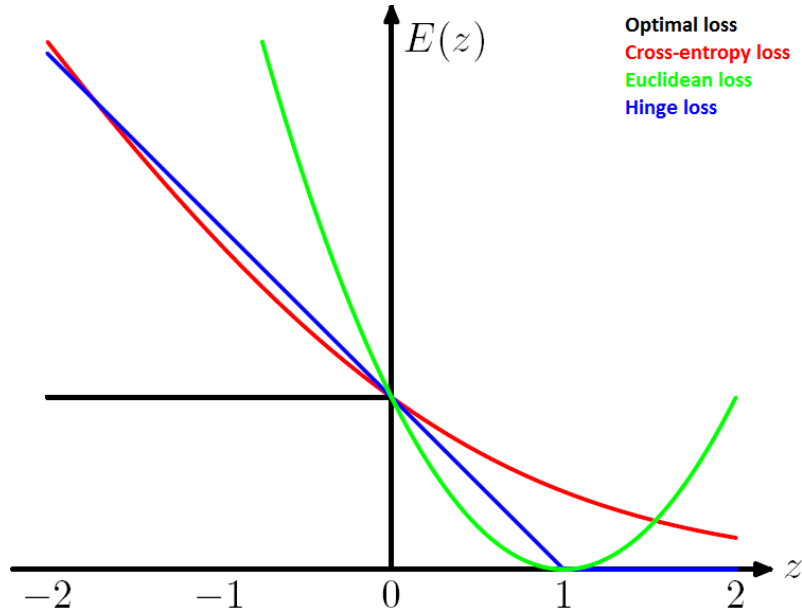
Figure 2.9: Comparison of loss functions for classification with two classes [Bis06]. $E(z)$ is the loss and $z = ty(\boldsymbol{x})$ with $t \in \{1, -1\}$ is the correctness of the prediction.

2.3.6 Overfitting and Regularization

The goal of machine learning models should be to accurately predict the desired target values for all possible data points. One kind of measure for this is given by the *training error*. This is the error the model makes on the known set of data which was used to develop the model. However, this measure is a very bad approximation of the true performance. The reason is that the model can learn certain patterns which only appear in the training set and not in the overall distribution of the problem. In fact, it is possible to have a model which just memorizes the data itself and consequently behaves perfect on the training set but has almost no correct predictions on previously unseen data. This problem of learning specifics about the individual dataset is known as *overfitting*.

Generally, overfitting occurs if the model is too complex for the available data [Bis06]. The figure 2.10 shows how model complexity affects the training and the generalization error. If the amount of data is too low, the trained model will likely overfit to the training set. The same holds true if the known data is not diverse enough or contains a large amount of noise and is therefore a bad representation of the overall distribution. In other words, the model should not be more complex than the information it can gain about the problem from the training set. Naturally, a model should also not be too simple, or the error would increase through a lack of knowledge about the problem. This is called *underfitting*. Consequently, both overfitting and underfitting make the model less accurate and should be avoided. Unfortunately, it is in practice not possible to know beforehand if a model is too complex or too simple.

Therefore, it is always necessary to not only have a training set but also a *test set*. The test set consists of instances not used in the training set and is used to approximate the generalization error of the model. Note that the test set has to be diverse and big enough such that the test error accurately represents the true generalized error. However, knowing the test error is not enough. Overfitting itself has to be avoided to train a good

15
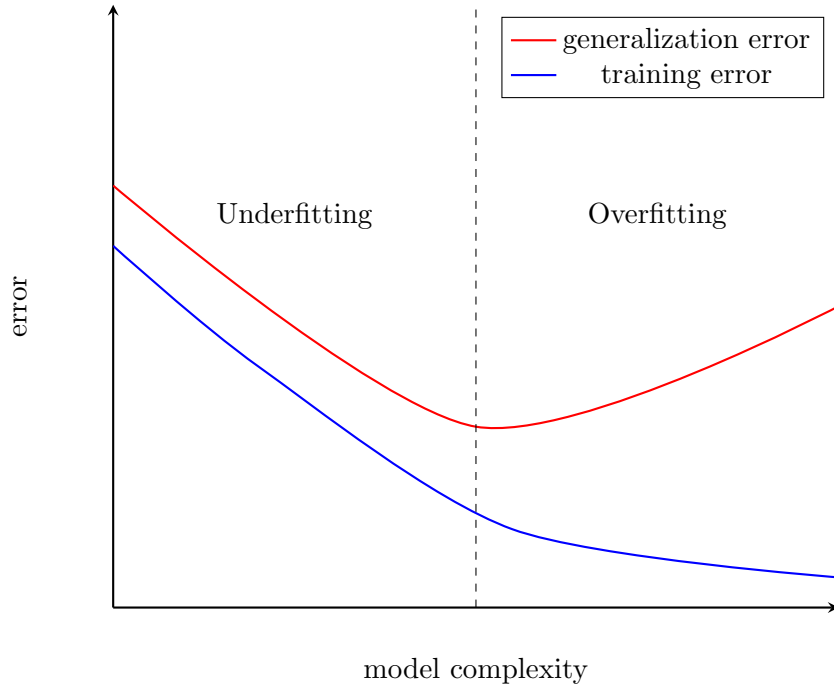
Figure 2.10: Difference between training and generalization error for a fixed dataset.

model. It can be reduced by increasing the amount and diversity of the data. However, getting a large amount of high quality data is often very difficult in practice. Instead of increasing the amount of data, it is also possible to limit the way a model can learn by introducing additional information which discourages a complex explanation of the data. This is known as *regularization*. Especially in the domain of deep networks, which are by design very complex models, are regularization techniques like *early stopping*, *weight decay*, *dropout* or *batch normalization* a very effective way to decrease overfitting.

In short, Early stopping means that the network is continuously tested after each epoch and the training stops after the test error starts to increase to avoid the obvious overfitting. It is an effective technique but it does not actually help the network to generalize, which other techniques do.

By using the technique weight decay, we add a hyperparameter to the model. This hyperparameter defines a penalty for high weight values. The reasoning behind this is that large weights corresponds to a high complexity and are a result of overfitting. It has been shown that weight decay can substantially decrease overfitting and aid in generalization. However, the weight decay hyperparameter can reduce the ability of the network to learn properly and should not be chosen too large.

Dropout is a techniques developed by N. Srivastava et al. [SHK+14]. If Dropout is applied to a layer of a neural network, half of the neurons of the layer will be deactivated in each iteration step. Deactivated means that they are forced to have an activation of zero and their weights will not be modified in the update step. The deactivated neurons are randomly chosen in each iteration step. This helps the network to learn more general features. It has be shown that Dropout applied to fully connected layer can substantially improve the performance of neural networks.

Batch normalization is a technique developed by S. Ioffe and C. Szegedy [IS15]. It is

often used in combination with very deep convolutional neural networks. The idea is to normalize the activations of each layer during training based on the mini-batch of the iteration step. Additionally, batch normalization adds two trainable parameters to the network which can scale and shift all activations of the specific layer. Batch normalizations added to state-of-the-art classification models lead to a substantial speed-up in training [IS15]. Moreover, batch normalizations also have an regularization effect. It has been shown that Dropout can be removed from networks with batch normalization without losing accuracy in the trained model [IS15].

2.3.7 Convolutional Neural Networks

Convolutional neural networks (CNNs) are networks specialized for the image domain. They outperform all other machine learning models in *image recognition* and in all kinds of visual tasks. Image recognition is the task of identifying objects or patterns in images. CNNs are also successfully used in other domains. For example, Google's AlphaGo, which can play the extremely complicated game Go better than any human player, is based on a CNN [Bur16]. However, before we start to explain how CNNs function, we first want to show why deep fully-connected networks do not perform well for image recognition tasks. An image can be represented by a three-dimensional matrix, e.g. in the case of the RGB format, this matrix has the shape $(h, w, 3)$ where h and w denote respectively the height and width of the image. The last dimension in this format denotes the number of color channels, which is always 3 for RGB. If we want to use an image as an input for a fully connected network, we have to transform the matrix into a one-dimensional vector of size $n = h \cdot w \cdot 3$. Assuming the image a height of width of 256, a single neuron in the first fully connected layer requires $n = 196608$ weights. A fully connected with a reasonable size would have too many parameters to train it efficiently. Moreover, even if the training time were not an issue, the large amount of weights would quickly lead to overfitting. Thus, fully connected network can only be applied to images if we reduce the size of the images drastically or if we precompute features like other machine learning models in computer vision.

To classify images directly with neural networks, we need a way to reduce the amount of learnable parameters in the network by generalizing the learned knowledge. *Convolutional layers*, which are the core functionality of CNNs, do exactly that by using the same weights for all neurons in a layer. Additionally, they use the relations between neighboring pixels to
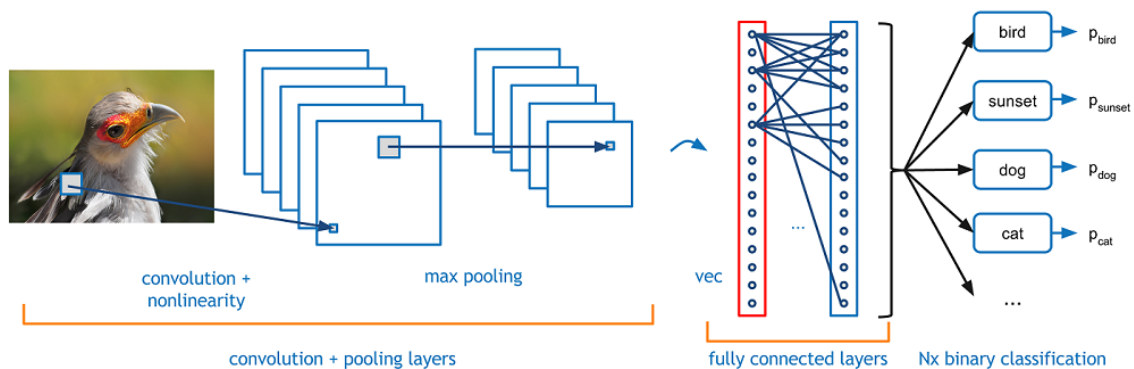

Figure 2.11: A convolutional neural network for image classification [Des16].
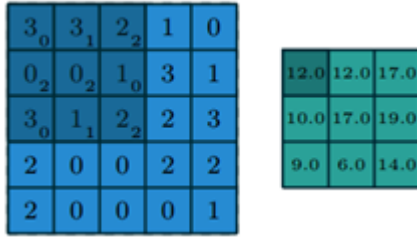
Figure 2.12: Convolutional arithmetic with a 3×3 filter, a stride of 1 and no padding. [DV16]. The blue square is the input tensor and the green square is the output of the convolution.
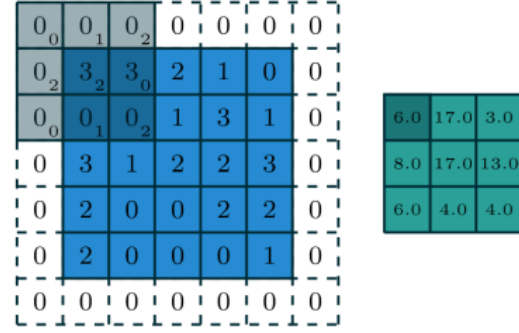
Figure 2.13: Convolutional arithmetic with a 3×3 filter, zero padding of size 1 and a stride of 2 [DV16]. The blue square is the input tensor, the white border is zero-padding and the green square is the output of the convolution.

learn general features of the raw data. Unlike fully connected networks, a CNN organizes the data between layers similar to images in volumes with a shape of $(heigth, width, depth)$. We call such a volume a *tensor*. Each layer in a CNN transforms one tensor into another. In this way, a CNN can process an image directly by interpreting it as a tensor. CNNs are an extension of standard fully connected networks. They still use fully connected layers, activation functions, a loss function and are trained with the backpropagation algorithm. Moreover, classification is implemented in the same way, i.e. with a softmax activation function in the last layer and cross-entropy as the loss function. In addition, they consist of the aforementioned convolutional layers and *pooling layers*, which exist to reduce the *spatial size* (height and width) of a tensor. The figure 2.11 shows the overall structure of a CNN used for classification. A CNN always starts with convolutional layer and ends with a fully connected layer. Pooling layers are only used after convolutions. In the following we will explain the new layer types in detail.

At the core of a convolutional layer is a set of learnable *filters* (or *kernel*). Each filter is a three-dimensional matrix of weights. The *receptive field* of a filter is the height and width of this matrix, denoted as $heigth \times width$. A filter has usually a small receptive field but extends through the full depth of the input tensor. For example, a typical filter in a CNN has a receptive field of 3×3. This filter applied to input tensor of size $(5, 5, 1)$ will have a size of $(3, 3, 1)$. If a filter with the same receptive field is used in a layer with an input tensor of size $(16, 16, 64)$, it will have a size of $(3, 3, 64)$. During the forward pass of the CNN, we slide each filter across the height and width of the input tensor starting from the top left position and ending in the bottom right. At each position, we compute the dot product between the weight matrix of the filter and the input tensor. Thus, we compute the weighted sum of a neuron for each possible position of the filter. Note that these neurons have all the same weights. The only thing that changes for each neuron are the inputs on which these weights are applied. The figure 2.12 shows a simple convolution with a input tensor of size $(5, 5)$ and a single filter with a receptive field of 3×3. The result of a convolution with a single filter is a two-dimensional matrix. This matrix is the height and width dimension of the output tensor of a convolutional layer. The depth of the output tensor is given by the number of filters. The receptive field and the number of filters for each layer are hyperparameters and have to be decided by the user.

Other hyperparameters are the amount of *zero-padding* and the *stride* of the convolution.
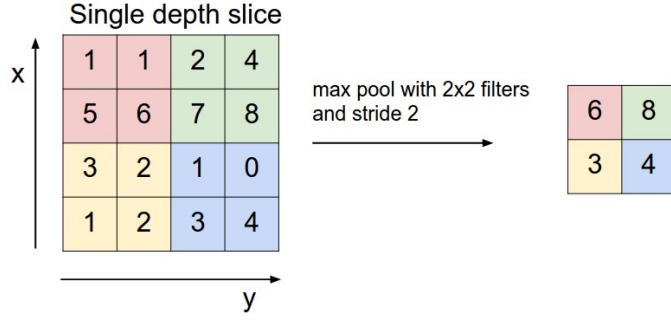
Figure 2.14: Max pooling with a (2×2) filter and a stride of 2 [Kar16].

The stride is a value which describes how far the filter moves when it slides over the input and zero-padding defines the amount of rows of zeros that are added around the border of the input tensor. Padding is often used to keep the height and width of input and output tensor equal if the stride is equal to 1. It is also important to avoid losing data. This can happen if the stride is greater than one. For example, the convolution from figure 2.12 with a stride of 3 would only have a single output value and more than half of the data would not be used in the layer. By adding a padding of one, all values of the input tensor would be used in the convolution. The figure 2.13 shows a convolution with a zero-padding of 1 and a stride of 2.

It is common to insert pooling layers between successive convolutional layers. A pooling layer reduces the spatial size of the input tensor. Similar to convolutional layers, the pooling layer also has a filter with receptive field which slides over the input tensor. However, the filter does not contain learnable parameters but instead applies a constant function to the values inside the receptive field. The pooling layer has the same hyperparameters as the convolutional layer with one exception. It does not have a hyperparameter which defines the number of filters because it operates independently on every depth slice of the input tensor. There are two common types of pooling layers: *max pooling* and *average pooling*. Max pooling is the most commonly used type of pooling. It uses the maximum function as the filter. Only the highest value is chosen in each position. Max pooling is shown in the figure 2.14. Average pooling computes instead the mean of all values inside the receptive field. Average pooling is usually only used at the last convolutional layer of a CNN in combination with *global pooling*. A global pooling layer is a pooling layer with a receptive field equal to the height and width of the input tensor. Global average pooling is used in some CNNs to reduce the height and width dimension to 1 such that a fully connected layer can be used afterwards. Other CNNs simply *flatten* the input tensor, which means that the tensor is just reshaped into a one-dimensional vector.

2.3.8 CNN Architectures

In this section, we present multiple CNN architectures. We will start with the *LeNet* architecture which was developed by Y. Lecun in the year 1989 for the recognition of handwritten digits [LBD+89]. It is one of the oldest architectures and cannot compete with the state-of-the-art. However, it is heavily used in tutorials in combination with the MNIST dataset of handwritten digits as the deep learning equivalent of a "Hello, World!" program. LeNet consist of two convolutional layers, two max pooling layers and two fully connected layers. The activation function for all convolutional and fully connected layer,

with the exception of the last layer, is the ReLU activation function. The last layer is a fully connected layer with 10 neurons and the softmax activation function. The convolutional layer have a filter size of 5×5 without padding and a stride of 1 and the pooling layers have a 2×2 kernel with a stride of 2. The number of filters in the convolutional layers and the number neural units in the first fully connected layer vary in different implementations. The figure 2.15 shows the architecture of LeNet.

The creation of the architecture *AlexNet* in 2012 was a breakthrough for the fields machine learning and computer vision. AlexNet was the first CNN which significantly outperformed all other machine learning models. It was developed by Alex Krizhevsky et al. and competed with other machine learning models in the ImageNet Large Scale Visual Recognition Challenge (ILSVC) [Sta]. AlexNet won the challenge and beat the state-of-the-art with a very big margin compared to the second place. It had a top-5 error rate of 15% and a top-1 error rate of 26% while the second place had a top-5 error rate of 26% and a top-1 error rate of 50%. An important part of the success of AlexNet and deep learning in general was the big training set used in the challenge provided by ImageNet with 1.2 million images labeled with 1000 different classes. AlexNet consists of five convolutional layers, three max pooling layer and three fully connected layer 2.16. The pooling layer follow the first the second and the fifth convolutional layer. The last layer has 1000 neurons and uses the softmax activation function while all other convolutional and fully connected layers use the ReLU activation function. The first two fully connected layer use the regularization technique dropout (see section 2.3.6). The original AlexNet is a special kind of CNN because it is split up in two streams, which only see half of the original input tensor. This was done for performance reasons to support training on multiple GPUs. The complete architecture is shown in figure 2.16.

Another groundbreaking work in deep learning was the development of *residual blocks*, which are used in the construction of deep *residual networks*. Residual networks were developed by Kaiming He et al. in 2016 [HZRS16]. Residual blocks make it possible to train extremely deep networks with up to hundreds of layers. The smallest residual network is ResNet-34, which consists of 33 convolutional layers and one fully connected layer. In addition, all residual networks use one max pooling layer after the first convolution and one global average pooling after the last convolutional layer. Residual networks always use the ReLU activation function for all convolutional layers and the softmax function for the only fully connected layer. The largest and best-performing residual network proposed by K. He was the ResNet-152 which consists of 152 layers.

The core idea of residual blocks is to introduce so called *skip connections* (or *identity shortcut connection*) which skip one or more layer and use element-wise addition to combine the input tensor of the residual block with the output tensor of the last convolutional layer in the block. In general, very deep CNNs without residual blocks cannot be trained effectively because the derivatives get smaller and smaller for the first layers in the backpropagation algorithm. The reason is that the last layers have a significant higher impact on the result of the network than the first layers. This problem disappears by using residual blocks, which allow the gradient to propagate directly to the first layers through the skip connections. A residual block with two convolutional layers is shown in 2.17. This kind of block is used in smaller residual networks like ResNet-34 while bigger residual networks like ResNet-152 use 3 convolutional layer inside the block. In general, a two-layer residual block uses convolutions with 3×3 filters and the three-layer block uses one 3×3 convolution in the center and two 1×1 convolutions at the start and the end. A three-layer residual block can be seen on the left side of figure 2.18.
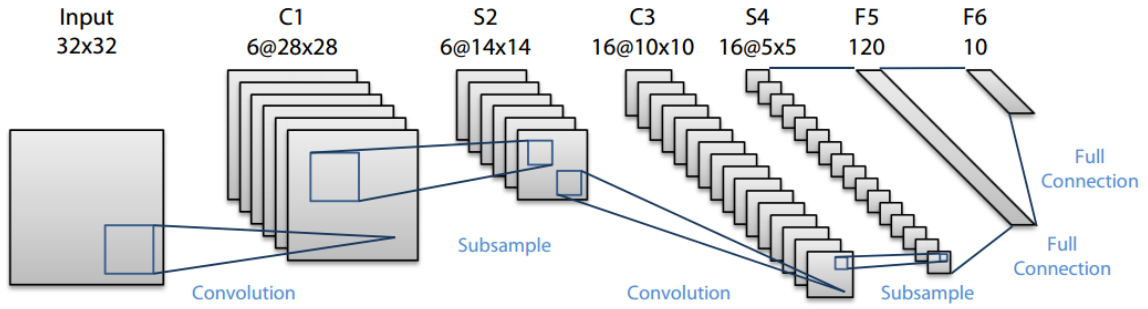
Figure 2.15: The architecture of LeNet, a convolutional neural network for digit recognition. It has two convolutional layers and two fully connected layers [Mit10].
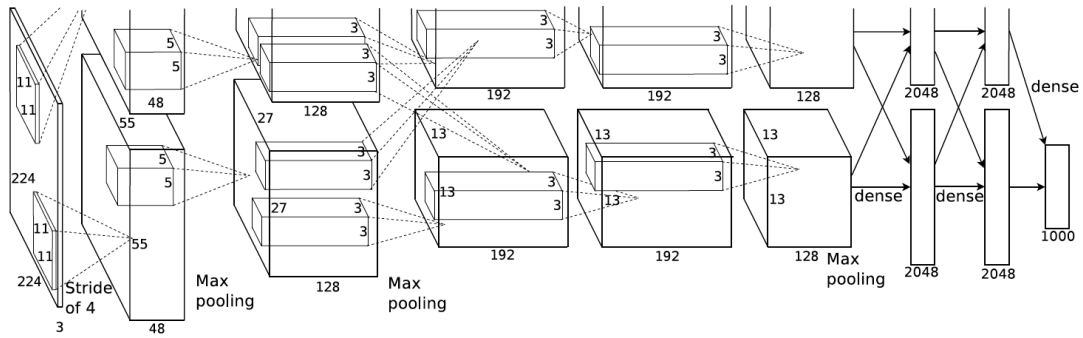
Figure 2.16: The architecture of AlexNet, a convolutional neural network for image recognition [KSH12].
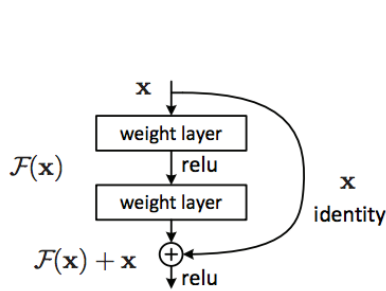
Figure 2.17: A residual block with two layers [HZRS16].
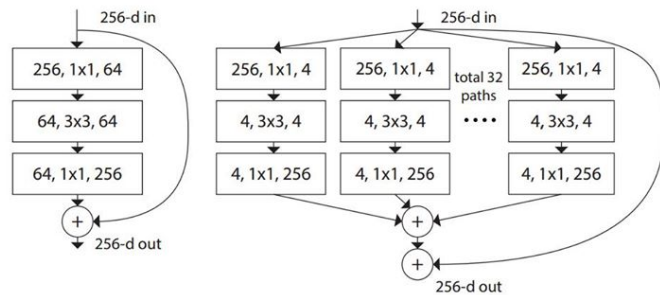
Figure 2.18: **Left**: A block of ResNet with three layers. **Right**: A block of ResNeXt with a cardinality of 32. A layer is shown as (input depth, filter size, output depth) [XGD+17].

The figure 2.19 shows the complete architecture of ResNet-34. The dotted connections in the figure are special skip connections which use a 1×1 convolution to increase the depth of the tensor. They are necessary because the element-wise addition at the end of the skip connection only works if both tensor have the same depth. The power behind such extremely deep CNNs like the residual networks is that stacked convolutions with smaller filter sizes can do the same computations as single convolutions with a large filter size while using less parameters. For example, two stacked 3×3 convolutions work on the same area as one 5×5 convolution but the two small convolutions use together 18 weights and the bigger convolution uses 25 weights.

Another architecture is ResNeXt developed by Saining Xie et al. in 2017 [XGD+17]. ResNeXt is build on the ResNet architecture and extends the definition of a residual block. ResNeXt uses another dimension aside from height, width and depth in the CNN which the authors call *cardinality*. The cardinality defines a number of convolutions which are used in parallel at the same serial position in the network. The right side of the figure 2.18 shows a residual block used in ResNeXt with a cardinality of 32. In the residual block, the output tensors of the parallel convolutions are merged with the addition operator. The authors of ResNeXt showed that it performs better than a normal ResNet architecture with the same amount of layers [XGD+17]. Furthermore, parallel convolutions were also used in the original AlexNet. Thus, most convolutional layers in AlexNet have a cardinality of 2. However, AlexNet splits the input tensor such that each stream only works on half of the data. This is not true for parallel convolutions in general.

2.3.9 Feedforward and Recurrent Neural Networks

A *feedforward neural network* (FNN) is any network where connections between units do not form a cycle. Thus, a FNN can be represented as a directed acyclic graph which is often done in practice. All network architectures presented in the previous sections are FNNs. The opposite of FNNs are *recurrent neural networks* (RNN). RNNs have feedback connections which make an activation depended on previous activations. This means usually that a neuron has a weighted connections to itself. In this way, current and previous network input affect together the current network output. RNNs are used in practice for time-dependent or sequential data, e.g. natural language processing. These applications require some form of memory which is provided by the feedback connections. More sophisticated RNNs use *Long short-term memory* (LSTM) units or *Gated recurrent units* (GRU). These are special neural units that improve the memorization capability of the network. Note that the architecture of a RNN is often still represented as a DAG by using layers as vertices and by encapsulating the feedback
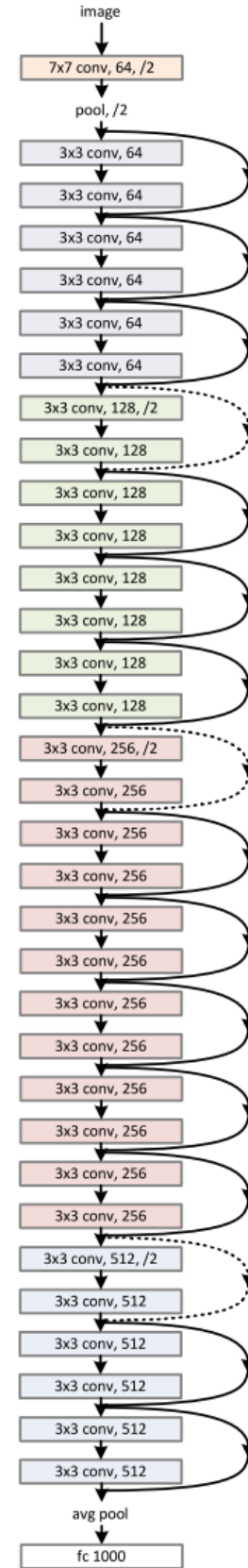
Figure 2.19: ResNet-34 [HZRS16].

22

connections in special RNN-layers.

2.4 HDF5

Data storage is an important topic in deep learning because training sets are in general very large and cannot be stored in the main memory. This can make access to the data a limiting factor of the training speed since reading files from a hard drive is very slow compared to an iteration step of the training process. One data format which is used in the deep learning domain is the *Hierarchical Data Format 5* (HDF5). HDF5 is a very versatile data model which is designed for an efficient access to complex data. HDF5 stores data in a single portable file format without upper limit to the file size. The data model is hierarchically structured. The file consist of a number of *groups* which in turn store a number of objects which can be other groups or so called datasets. Hereinafter, we call these dataset-objects *h5-datasets* to differentiate them from the standard meaning of the word dataset. The h5-datasets are arrays which contain a number of *dataspaces* which are the actual data points stored as multi-dimensional arrays. All stored values have a fixed data type declared by the h5-dataset. Groups and datasets can both be accessed by name. HDF5 is described in detail in the HDF5 User's Guide [G$^+$11].

The flexibility of HDF5 can be useful in a deep learning application if the training set consist of multi-dimensional labels or multiple inputs, which is a possible in deep networks. This is not supported in some other data storage formats used in the deep learning domain like *LevelDB*. Many deep learning frameworks like Caffe, TensorFlow and MxNet support the HDF5 format.

2.5 C&C Software Architectures

A *component and connectors* (C&C) architecture is a type of *information-flow architecture*. Like the name suggests, a C&C model consists of a set of components which are connected to each other. A software component is an architectural entity which encapsulates a subset of the system's functionality [Düd14]. It can realize a computation or store data. Each component has an interface which consists of a number of input and output *ports*. A connector of the model transports data from an output port of one component to the input port of another component. The information flow of the connectors is the driving force of the computation and each component is independently executed if data is available on all input ports. The interfaces hide the actual implementation and a component generally only interacts with the rest of the model through its output ports. C&C models are hierarchically structured. Instead of implementing a computation, a component can consist of multiple sub-components which again can be made up of even smaller components.

Thus, a C&C architecture is very modular. Each component has a clearly defined functionality and can be individually tested. This makes design and implementation of large and complex systems considerably easier since they can be divided into smaller independent tasks that are implemented by different developers. For these reasons, C&C models are used in many application domains. These include cyber-physical systems, web services and enterprise applications [KRRvW17]. One tool to model C&C systems, which is heavily used in industry and research, is *Simulink*.
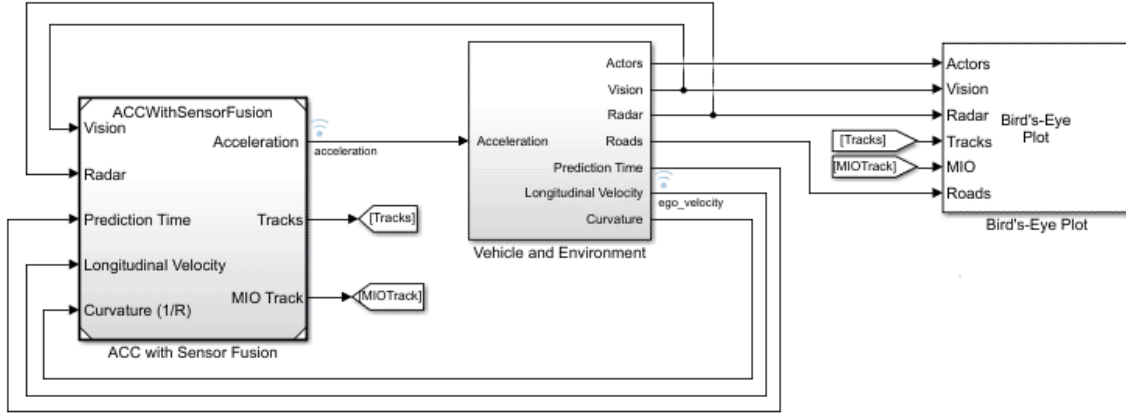
Figure 2.20: A simulink model for adaptive cruise control.

Simulink is a proprietary software developed by MathWorks [Mat16]. Simulink is a graphical modeling tool which is tightly integrated into the numerical computing environment *Matlab*. The Matlab programming language can be used to implement the functionality of the components. Additionally, Simulink offers a large library of pre-defined components (or *blocks* in Simulink), which implement commonly used operations. The figure 2.20 shows an example of a C&C model in Simulink. Note that the complete implementation of the components is hidden and only the interface is visible. All three components in the example are `subsystems` which means that they consist of smaller components. Subsystems can be opened to view the sub-components.

2.6   Freemarker Template Engine

Apache Freemarker is a template-engine for the Java programming language. A template-engine is a software which uses *templates* to generate code or documents by replacing placeholders with content based on a data model. Template-engines are very useful for code generation because templates can be easily modified and they reduce necessary overhead to a minimum. Freemarker is an open-source project under the Apache license 2.0. It uses its own script language called *Freemarker Template Language* (FTL) which is based on the Java syntax. It is well integrated in Java and allows access of arbitrary Java objects and functions from a template. The FTL is explained in detail in the *Apache FreeMarker Manual* [RSD⁺]. In the following, we will present the basic structure of FTL.

The core functionality of Freemarker is given by the *interpolations*. Interpolations are placeholders which will be replaced by calculated values. They always start with the dollar symbol "$" followed by an FTL-expression in brackets. Interpolations can reference Java objects and methods but they are also used to reference variables that were created with FTL. If an interpolation without specific context is executed in the template, Freemarker will call the `toString` method of the referenced object in Jave and replace the interpolation in the generated code by the returned value. There are a few exceptions to this. For example, integers or doubles will be printed in a special format which is different than the string representation in Java if not otherwise specified. The following code is an example with a interpolation which prints the `toString` method of the object with the name `name`, which has to be contained in the data model:

```
${name}
```

To reference an object in an interpolation, it has to be contained in the data model of the Freemarker instance. The data model is created in Java by the user and given to Freemarker so that it can process the templates. It is usually created as a Java Map with keys that are the names of the references in FTL and values that are the referenced objects. Furthermore, it is possible to use Java methods in an interpolation which is shown in the following example:

```
${allEvents.indexOf(event)?c}
```

The above example prints the integer that is the index of the element `event` in the list `allEvents`. Both names `event` and `allEvents` are references in the data model. The reference `allEvents` does not have to be a list but it has to be an instance of a class which implements an `indexOf`-method. The so called *built-in* reference `?c` is a command that forces Freemarker to print numbers in the normal computer-readable format. Otherwise Freemarker will add decimal separators to the numbers. There are many different built-ins for numbers, booleans, strings and sequences which implement commonly used functions for these types.

Furthermore, Freemarker offers multiple other abbreviations for expressions which increase the readable of the code. For example, the notation `[expression]` is short for `.get(expression)` and the notation `.variable` is an abbreviation of `.getVariable()`. The following example would print the value of the variable `name` of the first element in the list `allEvents`:

```
${allEvents[0].name}
```

Another part of Freemarkers are the *directives* (or *FTL tags*) which contain instructions which will not be printed to the output. There exist many directives in Freemarker. We will explain the directives `set`, `if` and `list` in the following. The `assign` directive denotes a variable assignment. It can change existing references or create new ones. The assigned value can be an arbitrary expression in FTL. This includes arithmetic expressions, logical expressions, strings, referenced objects and methods. Multiple examples of this are given in the following template:

```
<#assign hallo = "hello world!">
<#assign bool = true && false>
<#assign number = 42 + 42>
<#assign firstEvent = allEvents[0]>
```

The directive `if` can be used to conditionally skip sections of the template if the condition evaluates to true. The end of the block has to be explicitly denoted with `</if>`. The followoing is an example of this directive:

```
<#if x == 1>
    This line will be generated if ${x} is equal to one.
<#else>
    This line will be generated if ${x} is not equal to one.
</#if>
```

The `list`-directive is used to iterate through arrays and objects which implement the Java interface Iterable. Similar to the `if` directives, the end of the directive has to be explicitly denoted with `</#list>`. To create a loop with n iterations, Freemarkers offers the notation `1..n` which creates a list representing the range from `1` to `n`. Additionally, Freemarkers has the built-in references `?index` and `?has_next` for the loop variable
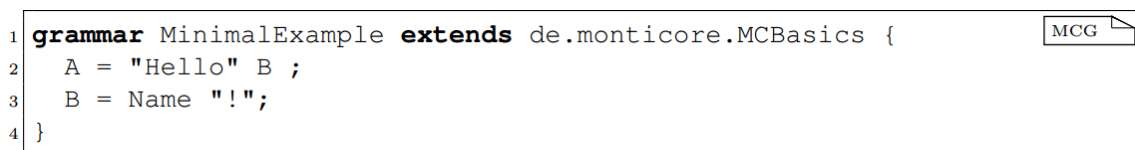
which return respectively the index of the loop and a boolean value that is only false if the current iteration is the last one. Furthermore, comments in Freemarker can be written starting with the notation `<#--` and ending with `-->`. The following template shows an example of a comment and two loops:

```
<#-- This is a comment -->
<#list 0..9 as i>
    ${i} is identical with ${i?index}.
</#list>
<#list tc.architectureInputs as input>
    The element ${input} has the index ${input?index}.
</#list>
```

2.7 MontiCore

MontiCore is a language workbench developed at the Software Engineering Chair of the RWTH Aachen University [RH17]. It is developed in Java and it employs Apache Maven for build management. MontiCore aids in design and implementation of textual *domain specific languages* (DSL). It uses a grammar description language which defines an extended *context-free grammar* (CFG) to generate a parser and an abstract syntax for the DSL. The generated parser can then be used to translate an instance of the DSL to an *abstract syntax tree* (AST) which contains all relevant information of the instance in form of Java objects. These objects are the *nodes* of the AST. The CFG consists of a set of nonterminals with different production rules. A simple example of a grammar is shown in figure 2.21.

The grammar in this example can parse the string "Hello World!" but also the string "Hello Bob!". A and B are the *nonterminals* of the grammar. The production rule of A produces the string "Hello" followed by the result of the production of B which is the token Name followed by an exclamation mark. The token Name is a variable string of common characters. The AST that this example would produce consists of 2 classes. One class for each nonterminal. Both classes have attributes that depend on their production rule. The class of B would have a string attribute with the name name and the class of A would have an attribute of type B with the name b. The attribute names can also be specified explicitly in the grammar, e.g. the notation world:B would define a B attribute with the name world. In general, all nonterminals of the grammar are classes in the AST and all nonterminals and tokens in a production rule are the attributes of the respective classes. Tokens can be defined by the user but MontiCore also offers pre-defined tokens like the token Name, which can be used for all kinds of named entities in the DSL. Furthermore, the grammar description language also offers similar to regular expressions the notations `*`, `+`, `?` and `|`. The first two denote that the previous expression can be repeated an arbitrary number of times but with + it has to occur at least once. The notation `?` means that

```
1  grammar MinimalExample extends de.monticore.MCBasics {    MCG
2    A = "Hello" B ;
3    B = Name "!";
4  }
```

Figure 2.21: A minimal example of a context-free grammar in MontiCore [RH17].
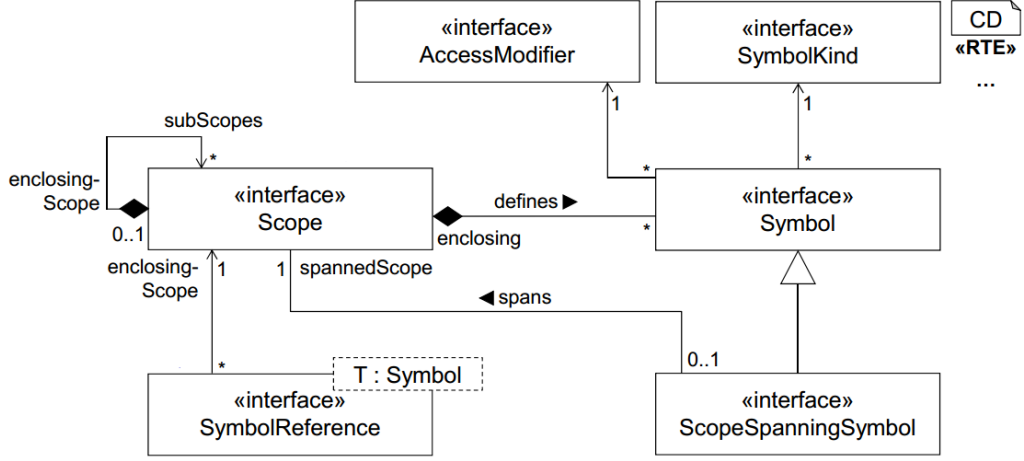
Figure 2.22: The main interfaces of the symbol management infrastructure [MSN17].

the previous expression is optional and | means that either the previous or the following expression is used. The AST uses Java List or Optional objects to store the corresponding nonterminals or tokens.

Additionally, MontiCore provides and generates Java classes which support the implementation of the *symboltable* and the *context conditions* (CoCos), which will be explained in the following. The symboltable has many purposes. It often defines the complete structure of the DSL model, while the AST is mainly used to create the symboltable. Every textual software language uses names to define entities, e.g. variables, methods and classes. These names can then be used to identify and reference the respective entities. A DSL-tool has to have a mechanism to resolve names in order to find the object which stores the information about the entity. This is realized by the *symbol management infrastructure* (SMI) in MontiCore [MSN17]. The SMI can be used to create and resolve the named entities in form of abstract *symbols* in different *scopes*. A scope, sometimes called *block* or *namespace* defines a specific area in which symbols are defined and impacts their visibility [MSN17]. A typical scope in a programming language is the scope of a method, e.g. in Java. A local variable inside this method is not visible from the outside but a variable defined in the scope of the class is visible in all methods. This behavior can be realized in MonitCore with nested scopes. Each symbol is contained in a scope and some symbols span named scopes, e.g. a method. To differentiate between multiple symbols with the same name, each symbol has a SymbolKind, which defines the type of the symbol. The symboltable contains all symbols and their relation to each other. The symbols in a symboltable are implemented by the user as classes that implement the Symbol-interface of MontiCore. They store all necessary information of the model but can also contain additional convenience methods. While the AST is a strict representation of the grammar, the symbols are an abstract representation of the language model. The figure 2.22 shows an overview of the main interfaces of the SMI.

The CoCos of a DSL exist to model context-sensitive constraints. For example, a constraint could be that a variable has to be declared before it can be used. This cannot be modeled in the context-free grammar. Additionally, CoCos are also often used in cases where the constraint would be cumbersome to express in a context-free way. A CoCo works on a single class of the AST and can automatically check all instances of this class. In general, they exist to find and log errors in the model.

27

Furthermore, MontiCore supports language inheritance as well as language embedding and language aggregation [MSN17]. In addition, it has its own logging system which makes error handling in constructed DSLs easier. The behavior of a DSL can be realized through the use of code generators which translate the created model to a chosen target language. MontiCore has a lot more features than presented in this section. They are explained in detail in the MontiCore reference manual [RH17].

Chapter 3

Comparison of Deep Learning Frameworks

The growing interest and successes of deep learning in the past ten years was only possible due to software tools which make execution and development time of deep leaning applications feasible. The training of a deep network is not only very computationally expensive but the corresponding algorithms are also complicated to implement. This means that before the existence of designated deep learning frameworks, research in deep learning was only possible for a handful of researchers who not only understood the theory behind it but could also implement the necessary algorithms. These algorithms have to be implemented efficiently on a low-level programming language and CPUs are often not powerful enough to train deep networks in a reasonable time frame. Instead, deep networks are usually trained on GPUs.

A lot of deep learning frameworks were created in the past ten years to enable research and applications for deep learning. Some of these are more focused on certain aspects of deep learning while others are general purpose frameworks. Most of the frameworks are still in heavy development and get continuously improved. Big tech companies like Google, Amazon, Microsoft and Facebook are supporting different frameworks. So it is not clear at this point which will prevail to be the standard in the future.

A very important aspect of all deep learning frameworks is how fast they train and how well they support parallelization on multiple GPUs. In the year 2014, the graphics card manufacturer Nvidia has introduced *cuDNN* which is a set of optimized low-level primitives to boost the processing speed of deep neural networks on Nvidia GPUs [Bro14]. Since then, cuDNN was integrated in all deep learning frameworks and some rely only on cuDNN for GPU computations, which is why Nvidia GPUs are dominant in the deep learning domain. Furthermore, most deep networks for research and production are nowadays trained on online services like Amazon's *Deep Learning AMIs* which offer computational power and a specialized infrastructure for deep learning applications [AWS].

In the following, we will first present a selection of deep learning frameworks which we deem as the most important and then we will compare their performance and features.

3.1 Deep Learning Frameworks

In this section, we will present the deep learning frameworks *Theano*, *Keras*, *Torch*, *Py-Torch*, *Caffe*, *Caffe2*, *TensorFlow*, *MxNet* and the *Matlab Neural Network Toolbox*. We will give a small code example for each framework. Other frameworks like the *Microsoft Cognitive Toolkit* (CNTK), *Deeplearning4j* and *Nervana Neon* will not be discussed in detail. In short, CNTK is a framework from Microsoft which is very good for recurrent neural networks and speech recognition but overshadowed by other frameworks in other areas. Deeplearning4j is a deep learning framework based on Java and relevant for those who want to use deep learning in a Java application. And Nervana Neon is a deep learning framework by Intel which has a good performance but it is not a general purpose framework and not very popular in the deep learning community.

3.1.1 Theano

Theano was one of the first widely used toolkits for deep learning. It was developed by Yoshua Bengio and his research team at the University of Montreal and released as open source project under the BSD license [ARAA⁺16]. The first release was version 0.1 in the year 2009. In the past years, Theano was overtaken by newer more popular frameworks. Hence, the latest version was released in 2017 and the team ceased the development. However, even though Theano is arguably already outdated, it still inspired newer frameworks like TensorFlow and MxNet which implement a similar approach to deep learning.

Theano is a Python library but all computationally expensive functions are implemented in C++ and Cuda. Despite being mainly developed and used for machine learning and deep learning, Theano is not a deep learning framework in itself but instead a general purpose framework for the manipulation of symbolic mathematical expressions [ARAA⁺16]. Thus, it is not only a fast but also a very powerful framework. Theano represents math expressions as static computation graphs and can compute gradients of mathematical expressions through symbolic differentiation on these graphs. The graph is static which means Theano has to compile the graph first before it can be executed and the graph cannot be changed after it is compiled. A computation graph in Theano is a bipartite directed acyclic graph. The nodes of these graphs are either *variable nodes*, which represent data, or *apply nodes*, which represent the application of mathematical operations [ARAA⁺16]. Variable nodes have a fixed data type (float32, int64, etc.) and occur usually in form of tensors. Variable nodes can be graph inputs, graph outputs and intermediate values. During the execution of a graph, intermediate and output values will be computed from provided input values. The mathematical expressions in Theano can be written in a syntax similar to the Python library *numpy*. The figure 3.1 shows a simple example of a symbolic function which takes a vector as input and returns another vector as output.

```python
import theano
a = theano.tensor.vector()        # declare variable
out = a + a ** 10                 # build symbolic expression
f = theano.function([a], out)     # compile function
print(f([0, 1, 2]))               # output: [0. 2. 1026.]
```

Figure 3.1: Example of a very simple function in Theano.

A consequence of not being designed to be only a deep learning framework is that neural networks have to be mostly constructed from low level operations. Thus, the construction of a network can be cumbersome and hard to learn which is one disadvantage of Theano compared to other deep learning frameworks. Low level operations make Theano also more error prone and debugging can be hard because the computation graphs are static and only executed after construction. Therefore, Python error messages cannot show where the actual error is but only where the graph was executed. However, there exist different high-level libraries like Lasagne and Keras which use Theano as backend while providing a user friendly interface for deep learning.

Another disadvantage of Theano is that it is mainly designed for research and not very suited for production. It does not offer special libraries for deployment and has to be always executed in Python despite being mostly implemented in C++.

3.1.2 Keras

Keras is a user friendly high level library which uses other deep learning frameworks as a backend. It is written in Python and is capable of running on top of Theano, TensorFlow, CNTK and MxNet [C$^+$15]. It was first released in 2015 as a open source project under the MIT license. Keras is one of the most popular deep learning frameworks because it is easy to use and available for many frameworks.

Keras uses an API similar to the deep learning framework Torch. A network can be constructed either as a *sequential* model by listing all layers of the network or in a functional way where each layer is applied to a given input. Keras makes saving and loading of models easy. It saves the architecture of a network either in a JSON or YAML file and the weights of the model in a HDF5 file. Additionally, many pretrained state-of-the-art networks can be easily loaded directly over the python module `keras.applications`. Pretrained models can also easily modified. For example, it is possible to change the size of the input or to retrain certain layers while keeping others fixed. The figure 3.2 shows an example of the LeNet architecture constructed in a functional way.

```python
from keras.layers import Input, Dense
from keras.layers import Convolution2D, Activation, MaxPooling2D
from keras.models import Model
def create_lenet(width=28, height=28, depth=1, classes=10):
    inputs = Input(shape=(depth, height, width))
    x = Convolution2D(20, 5, 5, border_mode="valid")(inputs)
    x = Activation("relu")(x)
    x = MaxPooling2D(pool_size=(2, 2), strides=(2, 2))(x)
    x = Convolution2D(50, 5, 5, border_mode="valid")(x)
    x = Activation("relu")(x)
    x = MaxPooling2D(pool_size=(2, 2), strides=(2, 2))(x)
    x = Flatten()(x)
    x = Dense(500)(x)
    x = Activation("relu")(x)
    x = Dense(classes)(x)
    predictions = Activation("softmax")(x)
    return Model(inputs=inputs, outputs=predictions)
```

Figure 3.2: The architecture of the network LeNet with Keras.

```
 1 require 'nn';
 2 net = nn.Sequential()
 3 net:add(nn.SpatialConvolution(1, 20, 5, 5))
 4 net:add(nn.ReLU())
 5 net:add(nn.SpatialMaxPooling(2,2,2,2))
 6 net:add(nn.SpatialConvolution(20, 50, 5, 5))
 7 net:add(nn.ReLU())
 8 net:add(nn.SpatialMaxPooling(2,2,2,2))
 9 net:add(nn.View(50*4*4))
10 net:add(nn.Linear(50*4*4, 500))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(500, 10))
13 net:add(nn.LogSoftMax())
```

Figure 3.3: The LeNet architecture in Torch. It is constructed as a `sequential` network.

3.1.3 Torch and PyTorch

Torch is a versatile numeric computing framework and machine learning library [CKF11]. It is built using the scripting language Lua and has a underlying C implementation. It runs on the LuaJIT (just-in-time) compiler. Torch was developed as a open source project under the BSD license by Ronan Collobert and Soumith Chintala of Facebook, Clement Farabet who was part of Twitter and is now at Nvidia, and Koray Kavukcuoglu of Google Deep Mind. Torch is not only a deep learning framework but also supports general machine learning techniques like *support vector machines* or *hidden Markov models* [CBM02].

Torch uses dynamic computation graphs which have several advantages over static computation graphs. They are called *dynamic* because Torch builds and rebuilds these graphs dynamically while they are executed. Thus, layers of a network can be executed line by line and there is in practice no difference between the construction and the execution of a network. By using dynamic graphs, it is possible to use standard imperative statements and control structures in the construction and execution of the network. It is also more flexible because the architecture of the network can be easily changed at runtime. Moreover, dynamic graphs are a lot easier to debug than static graphs since error messages are thrown at the specific line where they actually occur. The main disadvantage of dynamic computation graphs is that they cannot be optimized in the same way static graphs can.

Torch has a neural network library (`nn`) which can build neural networks as arbitrary acyclic computation graphs with support for automatic differentiation. A network in Torch has a `forward` function that computes the output for a given network input and a `backward` function which calculates the gradient for each parameter. A network can either be built in a sequential way by adding one layer after another, or in a functional way by setting the input of each layer explicitly. The figure 3.3 shows the construction of the LeNet architecture in Torch with the *sequential* container.

The original Torch was released in the year 2002 and is probably the oldest deep learning framework. It had a large community of developers and was heavily used within large organizations like Facebook, Google and Twitter [BRSS16]. However, it was not that popular in the general public because of Lua as the frontend scripting language of Torch. Lua is not well known and has not a big standard library like Python. Therefore, PyTorch was released by Facebook in early 2017 as an open source project. PyTorch is the Python version of Torch and it becomes increasingly popular in the research community due to it's flexibility. The figure 3.4 shows an example of PyTorch.

```
1   import torch.nn as nn
2   import torch.nn.functional as F
3   class Net(nn.Module):
4       def __init__(self):
5           super(Net, self).__init__()
6           self.conv1 = nn.Conv2d(1, 20, kernel_size=5)
7           self.conv2 = nn.Conv2d(20, 50, kernel_size=5)
8           self.fc1 = nn.Linear(50*4*4, 500)
9           self.fc2 = nn.Linear(500, 10)
10
11      def forward(self, x):
12          x = F.relu(F.max_pool2d(self.conv1(x), 2))
13          x = F.relu(F.max_pool2d(self.conv2(x), 2))
14          x = x.view(-1, 50*4*4)
15          x = F.relu(self.fc1(x))
16          x = self.fc2(x)
17          return F.log_softmax(x, dim=1)
```

Figure 3.4: The LeNet architecture with PyTorch. The code does not use the `sequential` class like the Torch example but instead constructs the network in a `functional` way.

3.1.4 Caffe and Caffe2

Caffe was developed by Yangqing Jia and the Berkeley Vision and Learning Center (BVLC) [JSD+14]. It is written in C++ as an open source project under the BSD license since 2014. Caffe focuses on the efficient implementation of CNNs and an easy off-the-shelf deployment of pretrained models. Caffe was one of the most popular deep learning frameworks especially in the industry due to the fact that it was the first framework which allowed for an easy way to share network architectures and trained state-of-the-art networks.

Additionally, Caffe can be run entirely over command line because the networks in caffe are not constructed in a programming language but are instead defined in the plaintext protocol buffer schema called *prototxt*. It has also interfaces for C++, Python and Matlab but the neural networks are always constructed in the prototxt format. There are four different files for a single model. These are the file which describes the network architecture for deployment, the file that models the same network for training and the configuration file which contains additional hyperparameters for the training process, e.g. the learning rate. The fourth file is the stored dataset mean which is automatically computed by Caffe for the sake of dataset normalization. The data has to be provided by the user in the data formats LevelDB, LMDB or HDF5. Caffe uses two descriptions for the same network because the model for training contains additional information like the name of the dataset, the initialization parameters of each layer and the used loss function. Figure 3.5 shows an excerpt of Caffe's LeNet model for training. This excerpt clearly shows one of the main drawbacks of Caffe. The prototxt data format is extremely verbose. This makes big networks like the residual networks difficult to construct and hard to read.

Caffe has a large community that contributes to the neural network repository which is called the *Model Zoo*. However, Caffe is not a general purpose framework. It does not have good support for networks which are no CNNs. And it does not have low-level math operations for network construction. Thus, it is not possible to create new loss functions or new layers without extending Caffe itself.

In April 2017, the latest version (1.0) of Caffe was released and a successor called Caffe2 was announced [Fac17]. Caffe2 is developed by a team at Facebook headed by the creator of Caffe Yangqing Jia. It is also written in C++ but like other frameworks it has a Python binding from the start. Caffe2 was created to overcome the limitations of Caffe which was

```
1  layer {
2    name: "conv1"
3    type: "Convolution"
4    bottom: "data"
5    top: "conv1"
6    param {
7      lr_mult: 1
8    }
9    param {
10     lr_mult: 2
11   }
12   convolution_param {
13     num_output: 20
14     kernel_size: 5
15     stride: 1
16     weight_filler {
17       type: "xavier"
18     }
19     bias_filler {
20       type: "constant"
21 }}}
```

Figure 3.5: The first convolutional layer of the network LeNet in Caffe's prototxt format. The complete file is 168 lines long.

originally designed for only one use case: conventional CNN applications. Caffe2 calls itself a lightweight, modular, and scalable deep learning framework which is designed with expression, speed, and modularity in mind. It is designed to support both: large-scale training and mobile deployment [Fac17].

In Caffe2, networks are modeled in Python and not in the prototxt format. Another difference is that it does not use layers to construct networks but more general *operators* which are much more flexible. Operators in Caffe2 can be network layers or low-level math operators and are documented in the *operators catalog*. This makes Caffe2 a general purpose deep learning framework similar to Theano or Torch. Thus, Caffe2 has much better support for recurrent neural networks than Caffe and the modeling is more similar to TensorFlow and PyTorch than it is to Caffe. To make the modeling of networks easier, Caffe2 offers the `brew` module which lets us define a layer in a single line. With conventional layer operators, it is necessary to initialize the weights and biases separately and to hand them over to the operator as arguments. The figure 3.6 shows the construction of the network LeNet with the `brew` module.

Similar to other frameworks, the architecture and the trained weights can be saved in separate files. These are protocol buffer files with the ending "pb" and are in contrast to prototxt files not human readable. However, it is possible to convert models from Caffe to Caffe2 by using a Python script provided by Caffe2. A disadvantage of both Caffe

```python
1  from caffe2.python import brew
2  def AddLeNetModel(model, data):
3    conv1 = brew.conv(model, data, 'conv1', dim_in=1, dim_out=20, kernel=5)
4    pool1 = brew.max_pool(model, conv1, 'pool1', kernel=2, stride=2)
5    conv2 = brew.conv(model, pool1, 'conv2', dim_in=20, dim_out=50, kernel=5)
6    pool2 = brew.max_pool(model, conv2, 'pool2', kernel=2, stride=2)
7    fc3 = brew.fc(model, pool2, 'fc3', dim_in=50 * 4 * 4, dim_out=500)
8    relu3 = brew.relu(model, fc3, 'relu3')
9    pred = brew.fc(model, relu3, 'pred', dim_in=500, dim_out=10)
10   softmax = brew.softmax(model, pred, 'softmax')
11   return softmax
```

Figure 3.6: The architecture of the network LeNet with Caffe2 in Python.

and Caffe2 is a lacking documentation compared to other frameworks. Caffe2 noticeable improves in this regard but it is currently still a new deep learning framework and some parts are not properly documented. For example, there seem to be no documentation for the *brew* functions and the description of some operators in the *operators catalog* only says: "No documentation yet".

3.1.5 TensorFlow

TensorFlow is a deep learning framework developed by Google. It became the most popular deep learning framework in recent years. Reasons for the popularity of TensorFlow are the marketing and support from Google, and the extensive documentation and tutorials it offers. TensorFlow takes the same general approach as Theano, i.e. it uses static computation graphs and allows for symbolic differentiation of arbitrary math expressions. TensorFlow grew out of an earlier library of Google called DistBelief, which was a proprietary deep net library developed as part of the Google Brain project [ABC+16]. TensorFlow is an open source project under the Apache 2.0 license and was released in 2015. TensorFlow has the same disadvantages as Theano: bad error messages and a complex API. Therefore, TensorFlow is often used in conjunction with the higher level library Keras, which was officially added to the TensorFlow API in 2017. TensorFlow also offers the high level module `layers` which is shown in figure 3.7,

TensorFlow is very similar to Theano but offers more features and better support. Thus, it replaced Theano in most areas. However, TensorFlow had at least in the past a worse performance than Theano (see section 3.2) which is why Theano is still sometimes preferred as a backend for Keras. In contrast to Theano, TensorFlow is available in multiple languages which are Python, C++, Java and Go. However, the APIs in languages other than Python are not yet covered by TensorFlow's API stability promises [Tena]. Additionally, it offers an easy way to log and visualize the training process with the *summary*-module and *TensorBoard*. TensorBoard can be used to visualize the computation graph, to plot the quantitative metrics about the execution of the graph, and to show additional data like images that pass through it [Tenb]. TensorFlow also offers subgraph execution which means that it is possible to retrieve the results of any node in the graph

```python
import tensorflow as tf
def create_lenet(data):
  conv1 = tf.layers.conv2d(inputs=data,
      filters=20,
      kernel_size=[5, 5],
      activation=tf.nn.relu)
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
  conv2 = tf.layers.conv2d(inputs=pool1,
      filters=50,
      kernel_size=[5, 5],
      activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)
  pool2_flat = tf.layers.flatten(pool2)
  dense = tf.layers.dense(inputs=pool2_flat,
      units=500,
      activation=tf.nn.relu)
  logits = tf.layers.dense(inputs=dense, units=10)
  softmax = tf.nn.softmax(logits, name="softmax")
  return logits, softmax
```

Figure 3.7: The architecture of the network LeNet with the TensorFlow Python API. The network was constructed with the higher-level API of the `layers` module.

The latest version TensorFlow 1.5 was released in January 2018 and added *Eager Execution* and *TensorFlow Lite* to the project. With *Eager Execution* enabled, it is possible to execute TensorFlow operations immediately in an imperative define-by-run style. In other words, TensorFlow added the option to use dynamic computation graphs in TensorFlow. This addition is very important because users switched from TensorFlow to PyTorch due to the limitations of the static computation graphs of TensorFlow [How17]. *TensorFlow Lite* is a library which was added as a efficient solution for deployment on mobile and embedded systems.

3.1.6 MxNet

Apache Incubator's MXNet is an efficient and flexible deep learning framework [CLL+15]. In November 2016, Amazon declared MxNet to be its deep learning framework of choice due to the scalability, development speed, and portability of MxNet [Vog18]. It is written in C++ and was first released under the Apache License 2.0 in 2015. Similar to Theano and TensorFlow, MxNet supports static computation graphs and automatic differentiation on symbolic functions. However, it has additionally the imperative tensor computation module *NDArray* which also supports automatic differentiation for math expressions with the module *autograd*. So, it was called MxNet (pronounced "mix-net") because it mixes symbolic and imperative programming.

In general, MxNet is more focused on production while Theano and TensorFlow were designed with a stronger focus on research. MxNet is available on many platforms and many programming languages. It runs on all major operating systems and even in a browser with JavaScript. The following programming languages can be used as an interface to MxNet: Python, C++, R, Julia, Matlab, Go, Scala, Perl and JavaScript. However, these do not have all the same features. JavaScript can only used for prediction and not for training. Furthermore, the C++ Api has a lot less features than the Python API.

The figure 3.8 shows a simple example of MxNet code. It is possible to define custom loss functions with symbolic expressions in MxNet but the standard loss functions are predefined and combined with the prediction output in special optimized output layers. Thus, the model looks the same for training and prediction. Thus, the example already contains a loss function which is not true for the examples of the other frameworks. Furthermore, the network can be trained in MxNet over a single function of the *Module* API by providing the data over a MxNet *Data Iterator*. MxNet can store a constructed and

```python
import mxnet as mx
def create_lenet():
    data = mx.symbol.Variable('data')
    conv1 = mx.symbol.Convolution(data=data, kernel=(5,5), num_filter=20)
    relu1 = mx.symbol.Activation(data=conv1, act_type="relu")
    pool1 = mx.symbol.Pooling(data=relu1, pool_type="max", kernel=(2,2),stride=(2,2))
    conv2 = mx.symbol.Convolution(data=pool1, kernel=(5,5), num_filter=50)
    relu2 = mx.symbol.Activation(data=conv2, act_type="relu")
    pool2 = mx.symbol.Pooling(data=relu2, pool_type="max", kernel=(2,2),stride=(2,2))
    flatten = mx.symbol.Flatten(data=pool2)
    fc1 = mx.symbol.FullyConnected(data=flatten, num_hidden=500)
    relu3 = mx.symbol.Activation(data=fc1, act_type="relu")
    fc2 = mx.symbol.FullyConnected(data=relu3, num_hidden=10)
    lenet = mx.symbol.SoftmaxOutput(data=fc2, name='softmax')
    return lenet
```

Figure 3.8: The architecture of the network LeNet with the MxNet Python API.

trained network in two separate file. The first file is a JSON file which describes the architecture of the network and the second file is a binary file with the ending ".params" which contains the trained weights of the model. For deployment, MxNet has a small and fast prediction API written in C++ which contains only those functions which are necessary for prediction. This includes loading an already trained model and applying it to a given input. Additionally, MxNet offers the ability to amalgamate the complete prediction library into a single file without dependencies [MxN]. This is a simple solution to solve dependency problems for deployment in mobile and embedded systems.

In November 2017, Amazon and Microsoft launched the high-level interface *Gluon* for MxNet which is designed to be easy to use while keeping most of the flexibility of a low level API [Mad17]. Gluon combines dynamic computation graphs, similar to PyTorch, with symbolic functions by using so called *HybridBlocks*. A HybridBlock can run fully imperatively with real functions acting on real inputs but they are also capable of running symbolically, acting on placeholders. Microsoft and Amazon claim that Gluon simplifies development of deep learning models without giving up on training speed.

3.1.7 Matlab Neural Network Toolbox

The Matlab Neural Network Toolbox is a deep learning framework developed by Math-Works. It is not open source but part of the proprietary software Matlab. For that reason, it is not a very popular framework in the deep learning community and is usually not mentioned in comparisons between frameworks. The Neural Network Toolbox can be used in the Matlab based software Simulink. Thus, it can be used as part of a C&C architecture.

The Neural Network Toolbox constructs a neural network in form of a static directed acyclic graph of layers [DB09]. A network can be constructed in Matlab by listing layers sequentially in a Matlab vector. The variable `layers` in the example from figure 3.9 demonstrates this by storing the architecture of the network LeNet. This list can then be transformed into a graph with the `layergraph` function shown in line 14 of the example. To create a non-sequential architecture, additional layers can be added to the graph with the `addLayers` function. At first, added layers are not connected to any node of the graph. New edges in the DAG have to be explicitly created by calling the `connectLayers` function. This function gets in addition to the graph itself two arguments, which are the user-defined names of the source and target layer of the edge. A complete network can

```
1  layers = [
2      imageInputLayer([28 28 1],'Name','input')
3      convolution2dLayer(5,20,'Padding','same','Name','conv_1')
4      reluLayer('Name','relu_1')
5      maxPooling2dLayer(2,'Stride',2)
6      convolution2dLayer(5,50,'Padding','same','Stride',2,'Name','conv_2')
7      reluLayer('Name','relu_2')
8      maxPooling2dLayer(2,'Stride',2)
9      fullyConnectedLayer(500,'Name','fc1')
10     reluLayer('Name','relu_3')
11     fullyConnectedLayer(10,'Name','fc2')
12     softmaxLayer('Name','softmax')
13     classificationLayer('Name','classOutput')];
14 lgraph = layerGraph(layers);
```

Figure 3.9: The architecture of the network LeNet in Matlab with the Neural Network Toolbox.

then be trained with the `trainNetwork` function by providing the data and training options as arguments. Matlab also offers the ability to visualize the training progress.

Furthermore, the Neural Network Toolbox supports all feedforward and certain types of recurrent neural networks. Recurrent neural networks are called *dynamic* networks in the Neural Network Toolbox [DB09]. Matlab also offers pretrained state-of-the-art networks which can be easily used. For training, it is possible to use the *GPU-coder* of Matlab to run the computationally expensive process on one or multiple GPUs. However, the Neural Network Toolbox does not offer symbolic differentiation. Thus, to define a custom layer or loss function, it is necessary to not only implement the *forward* function but also a *backward* function, which computes the gradient.

3.2   Performance

It was out of the scope of this thesis to benchmark all of these deep learning frameworks ourselves. So, we had to rely on other sources. However, there are not a lot of good sources on this issue and the ones who exist still have several problems. First, existing benchmarks usually only examine the training performance and not the prediction performance for deployment. Secondly, they do not measure the memory usage of these frameworks. Inefficient memory usage and slow prediction performance can make a deep learning framework unsuitable for production especially in the realm of cyber-physical systems. Thirdly, existing benchmarks only compare a few deep learning frameworks and we found none which compared all of the above frameworks. Lastly, most of these frameworks are still in development which means that the performance could have changed drastically since it was benchmarked due to improvements and bug fixes. For these reasons, we cannot make a definite statement about how well they perform but only relay the available information.

The first and probably best benchmark we will look at is a study by S. Shi et al. from 2017 [SWXC16]. They tested the deep learning frameworks CNTK, Torch, Caffe, MxNet and TensorFlow. They evaluated the training speed of these tools on a single CPU, multiple CPUs, single GPU and multiple GPUs. They used two fully connected networks, two versions of Alexnet and two versions of ResNet. Their results were the following. The frameworks with exception of TensorFlow do not scale well on multiple CPUs. However, TensorFlow performs also by far the worst on a single CPU on smaller batch sizes. Caffe seemed to perform best on a single CPU. Furthermore, they say that "MxNet is outstanding in CNNs" and that Caffe and CNTK perform also well on smaller CNNs. Caffe, CNTK and Torch achieve better results than MxNet and TensorFlow on fully connected networks with a single GPU. Moreover, CNTK performs best on RNNs. With multiple GPUs, Torch and MxNet perform the best on CNNs. In general, MxNet is good on CNNs and GPUs, Caffe is good on a single CPU, Torch is overall very fast and TensorFlow performed in most cases worse than the other tools.

Another study was done by S. Bahrampour et al. in 2016 [BRSS16]. They compared the training performance of Theano, Torch, Caffe and Neon. For neural networks, they used the LeNet and AlexNet architecture. So, they only tested CNNs. In this study, Torch performed best for CPU-based training followed by Theano and then Caffe. Their result for GPU-based training was that Theano is fastest for the smaller network and Torch is fastest for the larger network.

We found another benchmark in the Github repository *deep-learning-benchmark* by Y. Sako [Sak18]. The last update is only two month old. This benchmark compares TensorFlow, PyTorch and Caffe2. The tools were tested on four different GPUs and on two CNNs, which are VGG-16 and ResNet-152. The results show that Caffe2 has the best performance for prediction followed by TensorFlow and then PyTorch. The same is true for training on the smaller CNN. However, TensorFlow is clearly the fastest for the training of the large ResNet-152.

There are no benchmarks for the Matlab Neural Network Toolbox. The developer states that the pretrained models are 7 times faster than TensorFlow and up to 4.5 times faster than Caffe2 [Mat]. This tells us nothing about the training performance but it confirms that Caffe2 is faster for prediction than TensorFlow.

3.3   Features

In this section, we will compare the technical features of the deep learning frameworks. A comparison of the network modeling features of these frameworks and the DSL, which was developed in the scope of this thesis, is done in section 4.4.1. The table 3.1 shows the technical features. Most of these are self-explanatory but we want to clarify the following two.

General purpose framework   We call a framework general purpose if it contains low-level math operations on tensors and is designed to support all kinds of deep learning models. Caffe, Keras and the Matlab Neural Network Toolbox do not support automatic differentiation on math expressions. Instead, they only have high-level layers. They can be extended with custom layers but this requires additional work by the user.

Mobile deployment support   This feature means that the framework has additional support for deployment in mobile or embedded systems. It can still be possible to deploy a model of a deep learning framework for which this feature is not checked on a mobile device. The difference is that TensorFlow and MxNet have independent, small and efficient libraries only for prediction which makes the product faster and reduces dependencies. And Caffe2 was designed with mobile support in mind.

Deep learning framework	Open source	Interfaces	Static/Symbolic computation	Dynamic/Imperative computation	General purpose framework	Mobile deployment support
Theano	√	Python	√	-	√	-
Torch	√	Lua, C	-	√	√	-
PyTorch	√	Python	-	√	√	-
Caffe	√	Command Line, C++, Matlab, Python	√	-	-	-
Caffe2	√	Python, C++	√	-	√	√
TensorFlow	√	Python, C++, Java, Go	√	√	√	√
MxNet	√	Python, C++, R, Julia, Matlab, Go, Scala, Perl, JavaScript	√	√	√	√
Keras	√	Python	√	-	-	-
Matlab NNT	-	Matlab	√	-	-	-

Table 3.1: Comparison of deep learning frameworks, √: yes, P: partially, -: no.

Chapter 4

Context and Design

In this chapter, we want to show the design of the developed DSL. To do that, we will first explain the context of this thesis in detail. After that, we will analyze the task and provide requirements for the overall system. Then, we will present the general concept of our solution. In the following sections, we will explain the design of the modeling language and the code generator.

4.1 Context

MontiCAR, Modelling and Testing of Cyber-Physical Architectures, combines modeling languages with a focus on the domain of cyber-physical systems [KRRvW17]. The figure 4.1 shows the MontiCAR language family. The languages shown are all developed with MontiCore at the Software Engineering Chair of the RWTH Aachen University.

At the core of MontiCar is the language *EmbeddedMontiArc* (EMA) that extends the general purpose architecture description language *MontiArc*. MontiArc is a textual modeling language for C&C architectures. It is used to model web and cloud services in a C&C like manner [KRRvW17]. EMA extends MontiArc with additional features that enhance reusability and safety. This is important in the area of cyber-physical systems which have a higher safety requirements than regular software since an error can cause physical damage.

In contrast to MontiArc, it has port and component arrays and a strong type system that provides unit support and multi-dimensional abstract types that are defined independent from the implementation. While MontiArc uses standard Java types like `int` and `float`, EMA uses general integers `Z`, rational numbers `Q` and complex numbers `C` provided by *NumberUnit* as types. Additionally, these types can be restricted by defining a domain with a minimum, a maximum and a step size. The notation `oo` and `-oo` can be used to denote respectively positive and negative infinity in the domain definition. The values in this domain can have physical units, e.g. `km/h` for speed, `m` for meter or `°` for rotational degree. More complex types can also created with the *Type* language which allows to define enumeration and C-like structures. These types are checked in a MontiCAR language and the physical units can be automatically converted to alternate units of the same type.

We will explain how EMA works based on the example in figure 4.2 which shows an EMA component that can classify images of the MNIST dataset of handwritten digits. This component is part of a package called `mnist` and has two ports: one input port with the
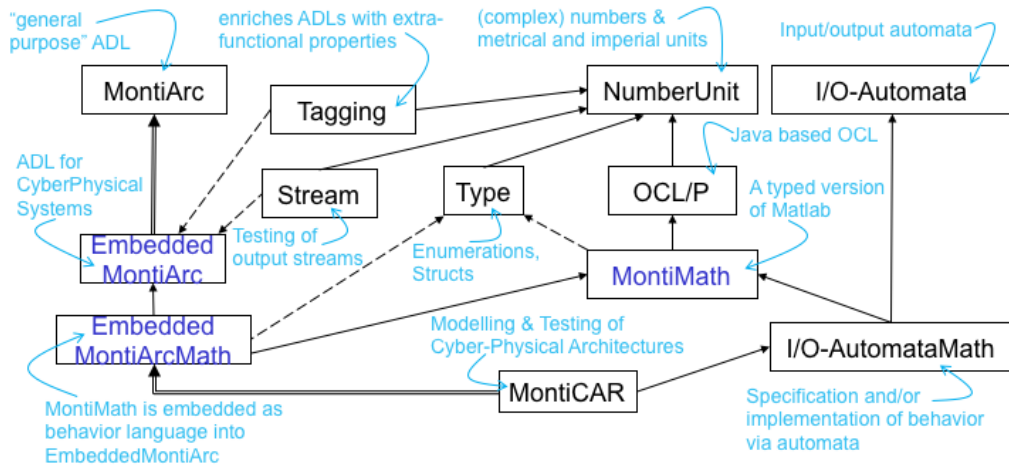
Figure 4.1: The MontiCar language family [KRRvW17].

.

```
1  package mnist;
2  import LeNet;
3  import ArgMax;
4
5  component MnistClassifier{
6      ports in Z(0:255)^{1, 28, 28} image,
7          out Z(0:9) digit;
8
9      instance LeNet<1,28,28,10> net;
10     instance ArgMax<10> calculateClass;
11
12     connect image -> net.data;
13     connect net.predictions -> calculateClass.inputVector;
14     connect calculateClass.maxIndex -> digit;
15 }
```

Figure 4.2: A general EmbeddedMontiArc component.

```
1  component ArgMax<Z(2:oo) n = 2>{
2      ports in Q^{n} inputVector,
3          out Z(0:oo) maxIndex;
4
5      implementation Math{
6          maxIndex = 0;
7          Q maxValue = inputVector(1);
8
9          for i = 2:n
10             if inputVector(i) > maxValue
11                 maxIndex = i - 1;
12                 maxValue = inputVector(i);
13             end
14         end
15     }
16 }
```

Figure 4.3: A EmbeddedMontiArcMath component.

name `image` and one output port with the name `digit`. The input port is a gray-scale image with height and width of 28 defined as a three-dimensional matrix of integers. The integers are restricted to be in the range (`0:255`) which denotes that the values cannot be lower than 0 or higher than 255. The output port is the predicted class of the input image. Thus, it is defined as an integer between 0 and 9. A port or component array can be declared by writing squared brackets with the size of the array after the name. For example, we can write `image[5]` instead of `image` in line 6 of the example to define five identical input ports. A single port of the array can be accessed in a Java-like manner, e.g. `image[0]` is the first port of the array.

Furthermore, The component has two subcomponents that are declared with the `instance` keyword. These are instances of the `LeNet` and the `ArgMax` component with the respective names `net` and `calculateClass`. The first subcomponent `net` should be an implementation of a CNN with the network architecture of LeNet. It computes the predicted probabilities of each class based on an input image which shows a handwritten digit. The second instance has the task of calculating the predicted digit out of the vector of probabilities returned by the first subcomponent. Both instances are created with arguments for the *generic parameters* of the respective components. A generic parameter affects the interface and the behavior of a component instance, e.g. the generic parameter n determines the dimensions of the input port *image* in the component `ArgMax` (see figure 4.3). Another type of parameter is the *configuration parameter*. Configuration parameters are very similar to generic parameters but they should only affect the behavior and not the interface of a component. To implement the connectors in the software architecture, the `connect` keyword is used. In the component `MnistClassifier`, the input port `image` is connected to the input port of the subcomponent `net`, the output port of `net` is connected to the input port of the subcomponent `outCalc`, and the output port of this subcomponent is connected to the output port `digit`.

EmbeddedMontiMath (EMAM) embeds the language *MontiMath* as an implementation language into EMA. MontiMath is a mathematical computation language in the style of Matlab. In contrast to Matlab, it uses the strict type system of MontiCar to minimize runtime errors. It keeps track of algebraic matrix properties and detects math errors at model creation. The figure 4.3 provides an example of a EMAM component. It is the component `ArgMax` used in the `MnistClassifier`. It computes the zero-based *argmax* function on an arbitrary vector of rational numbers by using a `for` loop and an `if` statement, which are defined in the same way as they are in Matlab. This example also shows in line 2 how the generic parameters are declared in an EMA component. A configuration parameter can be declared with round brackets instead of angled brackets. Parameters can have default values that are used if the user does not define an argument for the respective parameter. The component `ArgMax` has the generic parameter n, which can be an arbitrary integer that is higher than 1, with the default value 2. To execute an EMAM component, there exists a code generator called *EMAM2CPP*, which generates efficient C++ code for a modeled system.

4.2   Requirements

The example shown in the above section cannot be modeled completely with the existing MontiCAR languages since there is no way to implement the component `LeNet` with the

Figure 4.4: Requirement Diagram

MontiMath language. Deep learning in general and especially CNNs are getting increasingly important in cyber-physical systems, e.g. modern self-driving cars rely heavily on CNNs for all vision related tasks.

Thus, the task of this thesis is to develop a implementation language similar to MontiMath which makes it possible to model the complete `MnistClassifier`. The modeling language should be able to model common CNN architectures. It should be concise and easy to use. It should enforce correctness of the modeled network architecture by checking the validity of a network at model creation. Furthermore, the language should be properly integrated into EmbeddedMontiArc, which means that ports and parameters have to be correctly used in the created model. Thus, it should also be possible to create instances of modeled architectures with different arguments. This simplifies the process of reusing state-of-the-art network architectures for different tasks with a different amount of classes.

In addition, we had to implement a code generator that extends EMAM2CPP such that we

Figure 4.5: Dependencies of EmbeddedMontiArcDL. Dashed arrows mean that only the code generator of these languages is required for code generation in EmbeddedMontiArcDL. The blocks with blue background denote the languages developed in this thesis.

can generate code for a single EMA system which combines MontiMath and the developed deep learning language. The generated components shall be able to receive and send data to the components generated with EMAM2CPP. Moreover, the generated code should also automate the training process by loading the training and test set that are provided by the user to train the neural network. To do that, it has to be possible to model training parameters in the developed system. The figure 4.4 shows all requirements in form of a SysML requirement diagram.

4.3 Solution Concept

We divided the DSL into three languages which were all created with MontiCore. These are the language *CNNArch* which models CNN architectures, the language *Embedded-MontiArcDL* (EMADL) that integrates CNNArch and MontiMath into EMA, and the configuration language *CNNTrain* which contains parameters for the training process of the modeled network. These languages were developed in this thesis with the exception of CNNTrain, which is developed in the master thesis of Svetlana Pavlitskaya. In addition, we developed the code generator for the EMADL language to be able to execute a modeled system. This code generator uses the generator EMAM2CPP for MontiMath components and additionally generates code for training and prediction of a neural network modeled in a CNNArch component. The language CNNTrain is only used by the code generator to generate code for the training process and is not part of the C&C architecture. The dependencies of the EMADL modeling language are shown in figure 4.5.

The example from section 4.1 is completed with the EMADL component shown in figure 4.6. The parameters of the component define the shape of the input image and the number of classes used in the network. The developed EMADL code generator can be directly applied to the `MnistClassifier` component to generate a whole system. In the following, we will explain the implementation language CNNArch, which was the main focus of this thesis, in detail.

```
1  component LeNet<Z(1:oo) channels=1, Z(1:oo) height=28, Z(1:oo) width=28, Z(2:oo)
   ↪ classes=10>{
2      ports in Z(0:255)^{channels, height, width} data,
3          out Q(0:1)^{classes} predictions;
4
5      implementation CNN {
6          def conv(channels, kernel=1, stride=1){
7              Convolution(kernel=(kernel,kernel),channels=channels) ->
8              Relu() ->
9              Pooling(pool_type="max", kernel=(2,2), stride=(stride,stride))
10         }
11         data ->
12         conv(kernel=5, channels=20, stride=2) ->
13         conv(kernel=5, channels=50, stride=2) ->
14         FullyConnected(units=500) ->
15         Relu() ->
16         Dropout() ->
17         FullyConnected(units=classes) ->
18         Softmax() ->
19         predictions
20     }
21 }
```

Figure 4.6: A EmbeddedMontiArcDL component that models a version of LeNet.

4.4 CNNArch

CNNArch is a descriptive modeling language for CNNs designed as a part of the MontiCAR language family. It is an abstract language that does not describe the model directly but instead describes how it can be created. This means in practice that arithmetic and logical expressions can not only determine single values in the model but can also change the structure of the model considerably. In accordance with MontiCAR, the language supports strongly typed inputs and outputs. All expressions are evaluated at model creation and the validity of the network including the type and shape of input and output ports are checked automatically. The DSL uses a specialized syntax to greatly reduce verbosity and increase clarity in contrast to the network architecture construction of common deep learning frameworks. It is focused on CNNs but allows in principle the creation of any feedforward neural network (see section 4.4.8).

CNNArch can not only be used as part of a EMADL component but also as a stand-alone version. This is important to note because it leads to a slightly different syntax and different file endings for the same network. The stand alone version has the file ending ".cnna" and the EMADL version has the ending ".emadl". In this section, we will use examples of the stand-alone version to demonstrate the language. An example of the embedded was already shown in figure 4.6. The versions have different syntax for port and parameter declarations. However, these are treated in exactly the same way in the network architecture. In the following, we will first describe general features of the modeling language and compare them to other deep learning frameworks. Thereafter, we will explain in detail how the language realizes these features.

4.4.1 Features

In this section, we will describe the abstract features that CNNArch has to be able to fulfill the requirements. We also compare CNNArch with the network modeling language of other deep learning frameworks. In the case of Caffe, we examine the prototxt format.

46

Deep learning framework	C&C integration	Type-safe component interface	Directed acyclic graphs	CNN support	RNN support	Low-level operators	Layer composition	Layer stacking	Layer independence	Network parameterization
CNNArch	√	√	√	√	-	-	√	√	√	√
Theano	-	-	√	√	√	√	√	√	√	√
Torch	-	-	√	√	√	√	√	√	-	√
PyTorch	-	-	√	√	√	√	√	√	-	√
Caffe	P	-	√	√	√	-	-	-	√	-
Caffe2	-	-	√	√	√	√	√	√	-	√
TensorFlow	-	-	√	√	√	√	√	√	√	√
MxNet	P	-	√	√	√	√	√	√	√	√
Keras	-	-	√	√	√	-	√	√	√	√
Matlab NNT	√	-	√	√	√	-	P	P	√	√

Table 4.1: Feature comparison of CNNArch with the modeling language of other deep learning frameworks, √: yes, P: partially, -: no.

In all other cases, we look into the available API for network construction. The table 4.1 shows the comparison of the features. In the following, we will explain the meaning of the features, state which requirement they fulfill and describe how other frameworks realize them.

C&C integration   The feature *C&C integration* means that the modeling language can be directly integrated into a C&C architecture. This is necessary to fulfill the requirement with id 1.2 in the diagram in figure 4.4, which is the integration of CNNArch into EmbeddedMontiArc a component.

The Matlab Neural Network Toolbox, Caffe and MxNet have a Matlab interface which means that they can be used in Simulink which is based on a C&C design. However, Caffe and MxNet do not seem to be feature complete in Matlab and a network in Caffe has to still to be modeled in the prototxt format. Thus, they implement the feature only partially. All other frameworks cannot be easily used in a C&C architecture.

Type-safe component interface   The feature *type-safe component interface* is only possible if a C&C integration is implemented. It means that the language supports strongly typed ports for the components in a C&C model. This feature is also necessary to fulfill the requirement with id 1.2 in the requirement diagram. No other deep learning framework supports this feature.

Directed acyclic graphs   The feature *directed acyclic graphs* means that the modeling language can model neural network in form of arbitrary DAGs of layers. This does not mean that it can model all feedforward networks because the nodes of the respective graph

can be high-level layers like in Caffe, Matlab NNT or CNNArch. However, it does mean that the language can model most FNNs and is not restricted to sequential networks. This feature is desired to fulfill the requirement with id 2.1 in the requirement diagram because Alexnet, ResNet and ResNeXt as a DAG of layers. This feature is implemented by all deep learning frameworks considered.

CNN support The feature *CNN support* means that the language can model all layers necessary for common CNN architectures, e.g. convolutional layers and fully connected layers. In addition, the language must be able to handle tensors between layers to process input images. This feature is also necessary to fulfill the requirement with id 2 in the diagram of figure 4.4. It is supported by all deep learning frameworks considered.

RNN support The feature *RNN support* means that the language can model some kind of recurrent neural network. This is only listed to clarify that CNNArch does not support RNNs.

Low-level operators The feature *low-level operators* means that the modeling language allows the use of low-level math operations on tensors to define parts of the network. This feature in combination with the ability to model arbitrary DAGs allows the language to model all feedforward neural networks. CNNArch does not support this feature and models neural networks only with high-level layers like the frameworks Caffe and Keras.

Layer composition The feature *layer composition* means that existing layers can be easily composed to form new layers. This is very useful to avoid code duplications and to create more readable networks. A common example of a useful layer composition is the residual block of residual networks. We can decrease the size and redundancy of the architecture considerably by reusing the same residual block, which we have constructed once out of smaller layers, multiple times in the network. This feature is desired to fulfill requirement with the id 2 in the diagram properly.

This feature is available in most frameworks since they can use their interface programming language, e.g. Python, to define new functions which act exactly like a layer. This is only partially implemented in the Neural Network Toolbox since a network is connected by name but a composed layer would have different names for input and output which makes it different from a regular layer. It is not possible to compose layers in the prototxt format of Caffe.

Layer stacking The feature *layer stacking* means that it is possible to repeat the same layer an arbitrary number of times with the same or different arguments without having to write each layer explicitly. This reduces redundancy in an architecture and leads to a more concise definition of the network. This feature supports the layer composition in properly fulfilling the requirement 2 from the requirement diagram.

Layer stacking is possible in all deep learning frameworks with the exception of Caffe because they can use loops and functions to construct the network. Caffe cannot do that because the prototxt format is completely static.

Layer independence The feature *layer independence* means that each layer in the architecture can be defined independently from all other layers in the network. Thus, it is not necessary to define the dimensions of the input of the layer like it is in some frameworks which is problematic for CNNs where it is not always clear what the input shape of the first fully connected layer is because the input dimensions depend on all preceding convolutions. Thus, this feature reduces additional work of the user and makes a network easier to modify. Similar to layer composition and layer stacking, this feature is desired to make the language easy to use.

The feature is implemented by all deep learning frameworks with the exception of Torch, PyTorch and Caffe2. In these the input dimensions of a layer has to be specified. This can be seen in line 14 of the PyTorch-example in figure 3.4.

Network parameterization The feature *network parameterization* means that the network architectures can be instantiated with different parameters so that they can be imported and reused. For example, it should be possible to model the network Alexnet once while it can be trained and used multiple times with a different number of classes or different image sizes. This feature is necessary to fulfill the requirement 1.2.3 in the requirement diagram.

The feature is implemented in most deep learning frameworks since they can use the interface programming language, e.g. Python, to construct the network. These functions can then be imported and reused. This is not always done in these frameworks because architectures and trained networks are often shared in form of the static file format that the frameworks use to store it but it is possible. Furthermore, some frameworks like Keras provide function in the official API which construct commonly used architectures based on parameters. This feature is not supported by Caffe because the modeling language prototxt is a static data format and does not support parameters.

4.4.2 Basics

CNNArch uses a syntax similar to python. This decision was made for two reasons. The first reason is that python is the most used language in the area of deep learning which means that the users of the developed DSL are probably familiar with the python syntax. The second reason is that python is an untyped language and it is not necessary to have visible types in CNNArch for anything but the ports because those variables are only relevant at model creation and do not exist at runtime.

The smallest element of a CNNArch network is a layer and not a neural unit. The DSL is not designed as a general purpose deep learning framework but is focused on convolutional neural networks. Thus, lower level functionality is given up in favor of a more concise and easier checkable version of a neural network. Similar to a python method, a layer ends always with a parenthesis around a (possibly empty) set of arguments. Arguments can be optional or required. However, unlike python the arguments of a layer always have to be named.

A layer argument can be an integer, a floating point number, a boolean, a string, a tuple of integers or an arithmetic or logical expression. Arithmetic and logical expressions use the common operators +, −, *, /, &&, ||, ==, !=, <, >, <=, >= and the constants `true` and `false`. Variables in form of layer and component parameters can be used in

```
1  architecture ResNet152(channels=3, height=224, width=224, classes=1000){
2      def input Z(0:255)^{channels, height, width} image
3      def output Q(0:1)^{classes} predictions
4      def conv(channels, kernel=1, stride=1, act=true){
5          Convolution(kernel=(kernel,kernel),channels=channels,stride=(stride,stride)) ->
6          BatchNorm() ->
7          Relu(?=act)
8      }
9      def resLayer(channels, stride=1, addSkipConv=false){
10         (
11             conv(kernel=1, channels=channels, stride=stride) ->
12             conv(kernel=3, channels=channels) ->
13             conv(kernel=1, channels=4*channels, act=false)
14         |
15             conv(channels=4*channels, stride=stride, act=false, ? = addSkipConv)
16         ) ->
17         Add() ->
18         Relu()
19     }
20
21     image ->
22     conv(kernel=7, channels=64, stride=2) ->
23     Pooling(pool_type="max", kernel=(3,3), stride=(2,2)) ->
24     resLayer(channels=64, addSkipConv=true) ->
25     resLayer(channels=64, ->=2) ->
26     resLayer(channels=128, stride=2, addSkipConv=true) ->
27     resLayer(channels=128, ->=7) ->
28     resLayer(channels=256, stride=2, addSkipConv=true) ->
29     resLayer(channels=256, ->=35) ->
30     resLayer(channels=512, stride=2, addSkipConv=true) ->
31     resLayer(channels=512, ->=2) ->
32     GlobalPooling(pool_type="avg") ->
33     FullyConnected(units=classes) ->
34     Softmax() ->
35     predictions
36 }
```

Figure 4.7: The architecture of ResNet-152

these expressions. The arguments of predefined layers are constrained to certain values to ensure the correctness of the network and expressions are evaluated and checked at model creation.

The figure 4.7 shows a complete example of a CNNArch model which is based on the extremely deep CNN ResNet-152 [HZRS16]. It consists of 152 layers and has modeled in the verbose prototxt format of caffe over 6700 lines of code. This is because prototxt has no way of avoiding code duplications and has to model each layer separately. By using structures which implement layer composition and layer stacking, we are able to reduce the file size to 36 lines of readable code.

Furthermore, it is possible to use parameters of the component as variables in the implementation. This can be seen in line 33 of the example in figure 4.7 where the number of neurons in the last fully connected layer depends on the architecture parameter classes, which is defined as a positive integer. Thus, network parameterization is possible in CNNArch.

4.4.3 Data Flow Operators

In CNNArch, a network is constructed through operators which connect layers and determine the data flow through the network. Therefore, The architecture itself can be viewed

as a single expression. This is a major difference between this DSL and common Deep Learning Frameworks. Those either create connections by listing layers in a sequential order (e.g. Keras, Matlab) or create them explicitly by declaring names (e.g. Matlab, Caffe) or variables (e.g. Keras, Tensorflow, MxNet) as inputs. There are two data flow operators, also called layer operators, which construct the architecture of the network. These are the serial connection `->` and the parallelization operator `|`.

The serial connection denotes that the output streams of the left operand are the input streams of the right operand. In other words, all data will be handed forward where the data consists of multiple ordered packages which each could be represented as a 3D matrix. These forwarded data matrices do not need to have the same shape. This means for the simple case where both are convolutional layers that the second layer simply follows the first one. All purely sequential architectures, e.g. LeNet, can be constructed by only using the serial connection operator.

The parallelization operator splits the operands into separate groups which have the exact same input and are not connected to each other. This operator can be applied successively to create multiple parallel groups and a group can also be empty to implement a *skip connection*. The parallelization operator has a lower precedence than the serial connection. Therefore, it is necessary to use parenthesis to be able to combine the outputs of these groups. Let m be a layer, then the expression $(m()|m()->m()|m())->$ creates three parallel groups which are combined in what we call hereinafter a *parallelization block*. The language restricts the number of outputs of each group to be zero or one. The number of output streams of a group can only be zero in the unusual case where the network has multiple output ports and one of those ports is used in the parallelization block. The output of the whole parallelization block is the combined list of the outputs of each group. This list of data streams can then be reduced to a single stream either through the use of the two available merge layers `Concatenate` and `Add` or the selection layer. The selection layer, which can be abbreviated by `[index]`, simply selects one stream out of the list of input streams based on the given zero-based index.

4.4.4 Layers

CNNArch differentiates between two kinds of layers: predefined layers and constructed layers. Predefined layers are the smallest elements of the model while constructed layers can be defined by using the `def` keyword. The user defines a new layer by constructing it out of existing layers. Parameters can be declared and used in the body of the layer declaration. These can also have default values which makes them optional. Recursion is not allowed in the construction and the language will throw an error at model creation if recursion occurs. The layer `resLayer` in the example 4.7 is a constructed layer which corresponds to the residual block used in ResNet-152. Constructed layers realize the layer composition feature of section 4.4.1.

All predefined layers with the exception of *Concatenate*, *Add*, *Get* and *Split* can only handle a single input and have a single output. Predefined layers always start with a capital letter. This decision was made due to the fact that it is common practice in deep learning frameworks to capitalize layer names. Convolutional, pooling, fully connected and batch normalization layers are respectively implemented as the predefined layer `Convolution`, `Pooling`, `FullyConnected` and `BatchNorm`. For convolutions and pooling we implemented padding in a similar way as Keras. Padding cannot be applied manually. Instead

the user chooses between three modes which calculated the appropriate amount of padding automatically. This reduces work for the user and makes the layers more independent from the input. The three padding modes are called `valid`, `same` and `no_loss`. The first two also exist in Keras. The mode `valid` means that no padding is applied and `same` results in an amount of zero-padding such that the height and width of the output tensor is equivalent to the height and width of the input divided by the stride (rounded up). The third mode `no_loss` was developed in this thesis and means that minimal zero-padding is applied such that no data of the input tensor is discarded. The modes `valid` and `no_loss` are identical if the stride is equal to one.

In addition, we decided to handle activation function in the same way as other layers so that the language is more flexible and can be easier extended with new predefined layers. Thus the activation layers are `Sigmoid`, `Tanh`, `Relu` and `Softmax`. The complete list of predefined layers with their parameters is documented in appendix A.

4.4.5 Inputs and Outputs

The types of input and output ports have to be declared with the `def` keyword in similar way as their are in EMA. A port in CNNArch can either have a one-dimensional shape for flat data or a three-dimensional for images. Other data shapes are not accepted. The data format for images is "NCHW" which stands for the shape with the following four dimensions in the given order: batch size, number of channels, height and width. The batch size is only relevant for training and is therefore not defined in the architecture model but instead in the configuration file of CNNTrain. Thus, the first dimension in a port declaration is the number of channels and should usually be 3 for color and 1 for gray-scale images.

A network architecture in CNNArch can have multiple inputs and outputs. Moreover, the port arrays of EMA are supported. The figure 4.8 shows an example of a model with multiple input ports. This CNN resembles a network by Sun et al. in [SZWZ17] used for flower grading with images of the same flower from three different perspectives. We added the input port `additionalData` and the lines 24 and 25 to demonstrate how an additional input port can be added in the middle of a network. Another way to model the same network would be by including all the layers and ports before line 24 into the parallelization block of line 24. In general, output ports consume existing data streams and input ports create new streams while ignoring all incoming data in the model.

Port arrays can be used in two different ways. Either a single element is accessed or the array is used as a whole. Assuming we are using ports as defined in the above example, we can access the first port of the port array `image` with the expression `image[0]`. The `0` can also be replaced by an arithmetic expression. An error would be thrown at model creation if the result of this arithmetic expression is not a valid index of the port array.

In line 15 of the example 4.8, we use the port array as a whole in the model. This line could be replaced by the line `(image[0]|image[1]|image[2])->` without changing the model. This also means that the expression `image[index]` is equivalent to `image->[index]`. Moreover, assuming `outArray` is an output array of size 2, the line `->outArray` would be identical to:

```
-> ([0]->outArray[0] | [1]->outArray[1])
```

```
1  architecture ThreeInputCNN_M14(classes=3){
2      def input Z(0:255)^{3, 200, 300} data[3]
3      def output Q(0:1)^{classes} predictions
4      def conv(kernel, channels){
5          Convolution(kernel=kernel, channels=channels) ->
6          Relu()
7      }
8
9      def inputGroup(index){
10         [index] ->
11         conv(kernel=(3,3), channels=32, ->=3) ->
12         Pooling(pool_type="max", kernel=(2,2), stride=(2,2))
13     }
14
15     data ->
16     inputGroup(index=[0|..|2]) ->
17     Concatenate() ->
18     conv(kernel=(3,3), channels=64) ->
19     Pooling(pool_type="max", kernel=(2,2), stride=(2,2)) ->
20     FullyConnected(units=32) ->
21     Relu() ->
22     FullyConnected(units=classes) ->
23     Softmax() ->
24     predictions
25 }
```

Figure 4.8: Example of a CNN with multiple inputs

However, multiple output ports are probably never useful. They are only implemented for the sake of completeness.

4.4.6 Argument Sequences

Argument sequences can be used instead of regular arguments to declare that a layer should be repeated with the values of the given sequence. The operator between these so stacked layers is also given by the sequence. Other arguments that only have a single value are neutral to the repetition which means that the single value will be repeated an arbitrary number of times without having influence on the number of repetitions. Argument sequences are a way to implement the layer stacking feature of section 4.4.1.

The following are valid sequences: `[1->2->3->4]`, `[true | false]`, `[1 | 3->2]`, `[ |2->3]` and `[1->..->4]`. All values in these examples could also be replaced by variable names or arithmetic or logical expressions. The last sequence is defined as a range and equal to the first one. A range in CNNArch is closed which means the start and end value are both in the sequence. Moreover, a range has always a step size of one. Thus, the range `[0|..|-4]` would be empty. The data flow operators can be used both in the same argument sequence in which case a single parallelization block is created. A parallel group in this block can be empty, which is why `[ |2->3]` is a valid sequence. If a method contains multiple argument sequences, the language will try to combine them by expanding the smaller one and will throw an error at model creation if this fails. Let `m` be a layer with parameters `a`, `b` and `c`, then the expression `m(a=[3->2],b=1)` is equal to `m(a=3,b=1)->m(a=2,b=1)`. Furthermore, the line `m(a=[5->3],b=[3|4|2],c=2)->` is equal to:

```
(
    m(a=5, b=3, c=2) ->
    m(a=3, b=3, c=2)
|
```

```
    m(a=5, b=4, c=2) ->
    m(a=3, b=4, c=2)
|
    m(a=5, b=2, c=2) ->
    m(a=3, b=2, c=2)
) ->
```

and `m(a=[|5|3->4],b=[||1|2],c=2)` is equal to:

```
(

|
    m(a=5, b=1, c=2)
|
    m(a=3, b=2, c=2) ->
    m(a=4, b=2, c=2)
)
```

However, `m(a=[5->3],b=[2->4->6],c=2)` and `m(a=[5->3],b=[2|4->6],c=2)` would fail because it is not clear how `a` should be expanded such that it is the same size as `b`.

4.4.7 Structural Arguments

Structural arguments are special arguments which can be set for each layer and which do not correspond to a layer parameter. The three structural arguments are `?`, `->` and `|`. The conditional argument `?` is a boolean. It does nothing if it is true and it removes the layer completely if it is false. This argument is only useful for layer composition. The other two structural arguments are non-negative integers which repeat the layer x number of times where x is equal to their value. The layer operator between each repetition looks identical to the respective structural argument. These structural arguments implement the layer stacking feature for identical layers. Structural arguments make it possible to create networks which are much more concise and readable than they are in other deep learning frameworks. Those have to either rely on `for` loops which take more space and make the architecture less readable, or they cannot stack layers at all. The architecture of ResNet-152 in figure 4.7 shows the effectiveness of structural arguments at reducing the size of the model. The only type of structural argument not used in the example is the parallelization argument `|`. This argument is very useful to model a ResNeXt architecture. The architecture of ResNeXt-50 can be found in the appendix B.1.

4.4.8 Modeling of Directed Acyclic Graphs

CNNArch can model all possible *Directed Acyclic Graphs* (DAG) of layers and therefore nearly all feed forward neural networks. Even feedforward neural networks that use unusual connections between neural units could in theory be modeled by using fully connected layers of size 1 to symbolize single neural units. This would be incredible inefficient because the DSL is not designed for low level network construction but it is possible. The only feed forward networks which cannot be modeled are those which use layers that are currently not supported by CNNArch and cannot be modeled with a DAG of neural units, e.g. new activation or normalization functions. However, the DSL can easily be extended by adding these new layers to the list of predefined layers. For more information on how a new layer can be added to the language see section 5.2.8.
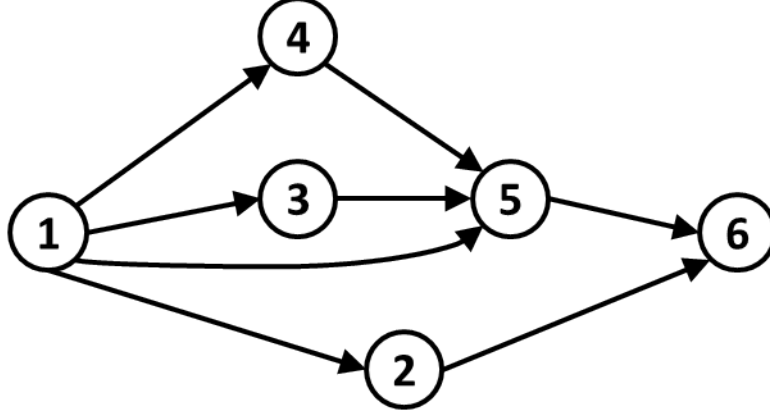
Figure 4.9: Example of a directed acyclic graph

In this section, we will show that the language can in fact model arbitrary DAGs. This will not be a complete formal proof but we will present a way to construct a CNNArch model from a DAG. To do this, we will not only use the two data flow operators but also the selection layer. We include the selection layer in the construction because it makes the proof easier and it allows us in theory not only to model connected DAGs but also disconnected ones. The figure 4.9 shows an example of a DAG which can be constructed by only using the data flow operators in the following way:

$$v_1 \rightarrow (v_2 \mid (\mid v_3 \mid v_4) \rightarrow v_5) \rightarrow v_6 \qquad (4.1)$$

where v_i is the layer or port that represents the node i and the symbol $\rightarrow$ represents the serial connection operator. An input port is equivalent to a node without incoming edges and an output port is equivalent to a node with no outgoing edge. Inputs or outputs with multiple edges can be represented as port arrays in the DSL. A node that has both incoming and outgoing edges is equivalent to a layer in CNNArch.

It is sufficient to show that we can model topologically sorted graphs because a graph is a directed acyclic graph if and only if it has at least one topological ordering (see section 2.2). From the definition of a topological ordering follows that an edge (i, j) of a topological sorted graph can only exist if $i < j$ holds. Furthermore, we can restrict ourselves to those topological orderings where no node with an out-degree greater than 0 appears after a node with an out-degree equals 0.

First, lets redefine the meaning of the sum and product symbol for this construction in the following way:

$$\sum_{i=a}^{n} v_i = v_a \mid \ldots \mid v_n$$
$$\prod_{i=a}^{n} v_i = v_a \rightarrow \cdots \rightarrow v_n \qquad (4.2)$$

Let $G = (V, E)$ be a DAG. It has a topologically sorted set $N = \{1, \ldots, n\}$ of nodes where $n \in \mathbb{N}$ and it always holds that $k < l$ if $l \in N$ has an out-degree equals 0 and $k \in N$ has an out-degree greater than 0. The graph has the set of edges $(i, j) \in E$ with $E \subseteq N \times N$ and $i < j$. Then we can construct the equivalent CNNArch model $C(G)$ with a set of layers

and ports $\{v_1, \ldots, v_n\}$ in the following way:

$$C(G) = \prod_{i=1}^{n} b_i^G \rightarrow () \tag{4.3}$$

$$b_i^G = \begin{cases} v_1 & \text{if } i = 1 \\ ([0] \mid I_2^G \rightarrow v_2) & \text{if } i = 2 \\ \left(\left(\sum_{j=0}^{i-2}[i]\right) \mid I_i^G \rightarrow v_i\right) & \text{if } i > 2 \end{cases} \tag{4.4}$$

$$I_i^G = \sum_{(k,i) \in E} [k-1] \tag{4.5}$$

The idea behind this construction is to forward at every step b_i^G the outputs of all previous layers together with the output of layer v_i. In this way, the only part of the construction which has to change to select different inputs of a layer v_i is the expression I_i^G. This expression uses the parallelization operator to combine the output of all input layers of v_i. Thus, the inputs of a layer v_i in CNNArch are equivalent to the set of nodes $k \in \{1, \ldots, n\}$ for which the edge (k, i) exists. The end part "$\rightarrow$ ()" in $C(G)$ uses an empty parallelization block and is necessary to make the expression valid by closing the remaining open streams. The construction from above applied to the graph G_1 from figure 4.9 results in the following expression:

$$C(G_1) = v_1 \rightarrow ([0] \mid [0] \rightarrow v_2) \rightarrow ([0] \mid [1] \mid [0] \rightarrow v_3) \rightarrow ([0] \mid [1] \mid [2] \mid [0] \rightarrow v_4) \rightarrow$$
$$([0] \mid [1] \mid [2] \mid [3] \mid \quad ([0] \mid [2] \mid [3]) \rightarrow v_5) \rightarrow$$
$$([0] \mid [1] \mid [2] \mid [3] \mid [4] \mid \quad ([1] \mid [4]) \rightarrow v_6) \rightarrow ()$$

which creates an equivalent model to the expression from 4.1.

4.5 Generated Product

We decided to use the template-engine freemarker to implement the code generator. This makes the generator flexible and easily modifiable. The generated code for each layer can be handled in a separate template which makes it easy to add new predefined layers to the code generator. The implementation of the generator is presented in section 5.5. In this section, we want to explain how we designed the generated product.

The code generator has to work with the existing MontiMath generator EMAM2CPP. Hence, the target language is C++. Furthermore, it was completely out of the scope of this thesis to develop a deep learning framework from scratch. Thus we decided to use one of the available frameworks as a backend to implement network training and prediction. The generated code can be separated into two parts: the *trainer* and the *predictor*. The trainer trains the network and is independent from the predictor, which is executed in a component and applies the trained network to the input data to compute the predicted labels. The trainer fulfills the requirements 3.1, 3.4, 4 and 5 of the requirement diagram in figure 4.4.1 while the predictor fulfills the requirement 1.1. To be able to generate the trainer, a configuration file of the language CNNTrain has to exist either in the directory of the main component or in the directory of the EMADL component which defines the architecture. It is possible to use a single configuration file for all networks in a system or

a configurations for each individual instance. The generator will write an error message with the possible names of a configuration file if no configuration could be found for a network.

In the following, we will first discuss which deep learning framework we use as a backend of our system and why we decided to use it. After that we talk about the trainer, the training data and the predictor. A complete code generation example with all generated files can be found in appendix D.1. These listed files are generated based on the model from section 6.2.

4.5.1 Backend

EMA is designed for cyber-physical systems and a common aspect of those is limited memory and computational power. Therefore, we had to choose a deep learning framework that is not only fast in big distributed systems but also usable in smaller systems. The frameworks Theano and Torch are only designed for research. The Matlab Neural Network Toolbox and Keras cannot be used in C++. Furthermore, TensorFlow does not seem to have a good performance in small systems and it is possible that TensorFlow uses the available memory inefficiently. Thus, we mainly considered Caffe, Caffe2 and MxNet as a backend to our system. The main problem of Caffe2 was that it was early in the development when we started to work on this thesis. The documentation was still lacking and it was not clear if it was suitable for our purpose. So we decided against using Caffe2.

The original Caffe has probably a good performance for smaller systems and is suitable but we finally decided to use MxNet. MxNet was chosen because it is more flexible than Caffe and due to its small and fast prediction API. MxNet is a general purpose deep learning framework and even though CNNArch and CNNTrain are currently not meant to do more than Caffe, this could change in the future. For example, it is not possible to define custom loss functions in Caffe but these are a reasonable extension to CNNTrain for future work. Additionally, the amalgamation functionality of MxNet should make deployment with it easier than it is with Caffe.

4.5.2 Trainer

The currently generated trainer is implemented in python while the predictor is written in C++. This is not a problem because the training process is in general very computationally expensive and done before deployment. Note that the MxNet python code uses C++ or cuda functions for the computationally expensive training process. So the actual performance loss is minimal. Moreover, it is a lot easier to execute the trainer as a Python file which does not have to be compiled. After the trainer is used to train the network, it can be discarded and the predictor together with the rest of the system can be compiled for deployment.

The trainer consists of two python files which names depend on the fully qualified name of the component. Let p be a EMA package with the two components `Main` and `Net`, where `Net` has a CNN implementation and "Main" creates an instance of "Net", then these two files are called `CNNTrainer_p_main.py` and `CNNCreator_p_main_net.py`. The CNNCreator class contains the main functionality of a network. It constructs the network, reads the data, trains the network and stores the trained network. The CNNTrainer file

can be called by the user to start the training process for all networks used by the "Main" component. So, there is only one CNNTrainer while there can be multiple CNNCreator if multiple networks are used in the same system. The trained networks are automatically stored in a directory with the name "models" and the training and testing process is logged to a file called "train.log" as well as written to the terminal. The trainer also creates a checkpoint after each epoch and will automatically load the newest checkpoint at the start of training if such a checkpoint exists. The loss function used by the trainer is currently determined by the last layer of the network. It is cross-entropy loss for a softmax or a sigmoid layer and squared-error loss in all other cases.

Furthermore, the trainer can normalize the data by subtracting the mean of the training set element-wise from the network input and by dividing the result of this by the standard deviation. The same normalization is then also automatically applied by the predictor. This can significantly increase the accuracy of many neural networks. However, normalization can also decrease the accuracy for some tasks. For that reason, it can deactivated over the training configuration.

4.5.3   Training Data

The training and testing data have to be provided by the user in the HDF5 data format. The data files are named `train.h5` for the training set and `test.h5` for the testing set and they have to be in a directory which name depends on the fully qualified name of the network component. The path to this directory is `data/p_main_net` for the example from the last section. The data files contain one h5-dataset for each port. The name of this h5-dataset is equal to the name of the port if it is a input port, otherwise it matches the name of the output port plus the postfix "_labels". The CNNCreator will throw an error message if the training data cannot be found or if a h5-dataset is missing. In this message, it will state the correct path for the file or the name of the missing h5-dataset. The data itself has to have the data type `float32`. In case of softmax as the last layer, the labels in the data sets have to be equal to the index of the class. In the other cases, the labels have the same format as the output of the network.

4.5.4   Predictor

The predictor can only be used after the trainer was executed. The predictor only depends on the prediction api of MxNet and the two files that are created by the trainer in the "models" directory. These files which have the file ending ".json" and ".params" store respectively the architecture and weights for MxNet. After these are created, the python files are not needed anymore and can be completely removed from the system.

The predictor consists of 4 files: the component, the CNNPredictor, the "Bufferfile.h" and the "CNNTranslator.h". The Bufferfile class is a simple class used to read the stored network files. The CNNTranslator contains static functions to translate between the different data types that are used in EMAM2CPP and MxNet. The CNNPredictor, which full name also depends on the fully qualified name of the component, is the class that uses the trained MxNet network to predict labels for a given input. The C++ class which represents the component uses an instance of the CNNPredictor and the functions of the CNNTranslator to compute the output of the component based on the input port.

Chapter 5

Implementation

In this chapter, we will explain the implementation of the CNNArch [1] and the EMADL[2] language. The full code is available in the respective Github repository. First, we will present the implementation of the CNNArch language. Thereafter, we will explain how CNNArch is integrated into EMA with EMADL and how the code generator works. Finally, we will state what tests we used to ensure the correct behavior of the languages.

5.1 CNNArch Grammar

In this section, we will explain the MontiCore grammar and the AST of the developed language. CNNArch extends the grammar of MontiMath in order to use arithmetic and logical expressions. The MontiMath grammar in turn extends the basic MontiCAR languages like NumberUnit which allows us to parse numbers and types in the same way as EmbeddedMontiArc. We did not modify the AST classes generated by MontiCore aside from adding simple methods like the `getName` to some interfaces for an easier access to attributes. There is one exception to this which we will mention in this section. In the following, we will present all nonterminals of the grammar and their production rules. We will omit comments and the tokens `NEWLINETOKEN` from the original code when we show the production rules. The token `NEWLINETOKEN` denotes a new line explicitly and is required to integrate the language into EmbeddedMontiArc without errors. Note that some nonterminals will have the prefix `Arch` in their name. The prefix has no semantic meaning and only serves to differentiate them from inherited nonterminals.

Let us first look at the nonterminals which create the simple expressions that can be used as values for variables in the language. These are the nonterminals `ArchSimpleExpression` and `TupleExpressions` that have the following production rules:

```
ArchSimpleExpression = (arithmeticExpression:MathArithmeticExpression
                      | booleanExpression:MathBooleanExpression
                      | tupleExpression:TupleExpression
                      | string:StringLiteral);
TupleExpression = "(" expressions:MathArithmeticExpression ","
                     expressions:(MathArithmeticExpression || ",")* ")";
```

---

[1] CNNArch code is available at: `https://github.com/EmbeddedMontiArc/CNNArchLang`
[2] EMADL code is available at: `https://github.com/EmbeddedMontiArc/EmbeddedMontiArcDL`

The nonterminal `TupleExpression` produces Python-like tuples out of arithmetic expressions separated by ",". The nonterminal `MathArithmeticExpression` for arithmetic expressions is inherited by MontiMath. Furthermore, the nonterminal `Arch-SimpleExpression` produces an arithmetic expression, a logical expression, a tuple or a string.

To produce the argument sequences that we explained in the previous section, we need the following production rules:

```
ArchExpression = (expression:ArchSimpleExpression | sequence:ArchValueSequence);
interface ArchValueSequence;
ArchParallelSequence implements ArchValueSequence =
    "[" parallelValues:(ArchSerialSequence || "|")+ "]";
ArchSerialSequence = serialValues:(ArchSimpleExpression || "->")*;
ArchValueRange implements ArchValueSequence = "[" start:ArchSimpleExpression
                                              (serial:"->" | parallel:"|")
                                              ".."
                                              (serial2:"->" | parallel2:"|")
                                              end:ArchSimpleExpression "]";
```

The nonterminal `ArchExpression` produces expressions for arguments. It can either produce a simple value expression or an argument sequence. The general argument sequence is implemented as the interface `ArchValueSequence`. This interface can either produce a normal argument sequence with nonterminal `ArchParallelSequence` or it can produce a range with the nonterminal `ArchValueRange`. The normal argument sequences are created hierarchically by first producing the parallel part and then the serial part of the sequence.

Furthermore, the following production rules are used for the nonterminals which symbolize variables and arguments:

```
interface Variable;
ArchitectureParameter implements Variable = Name
                                            ("=" default:ArchSimpleExpression)?;
LayerParameter implements Variable = Name
                                     ("=" default:ArchSimpleExpression)?;
interface ArchArgument;
ArchParameterArgument implements ArchArgument = Name "=" rhs:ArchExpression;
ArchSpecialArgument implements ArchArgument = (serial:"->"
                                              | parallel:"|"
                                              | conditional:"?")
                                              "=" rhs:ArchExpression;
```

Variables and arguments are respectively implemented with the interfaces `Variable` and `ArchArgument`. A variable can either be a parameter of the architecture or a parameter of a layer method. These have the exact same production rules and only exist as different nonterminals to easily differentiate between them in the AST. A variable has a name and an optional default value. The nonterminal `ArchitectureParameter` is only used in the stand-alone version of CNNArch. An argument can either be normal argument or a structural argument. The normal arguments given by the nonterminal `ArchParameterArgument` have a name, which references the corresponding parameter, and a value or a sequence of values determined by the production of `ArchExpression`. The nonterminal of the structural argument `ArchSpecialArgument` is similar but it starts with the characters `->`, `|` or `?` which are stored as optionals in the AST. Thus it does not have an explicit name. However, the AST is manually extended in Java such that the class of `ArchSpecialArgument` has a name attribute which is identical to the string representation of the structural argument.

To create a layer in the architecture, we use the nonterminals `IOLayer`, `Layer`, `ParallelLayer`

and `ArrayAccessLayer`. Their production rules are defined in the following way:

```
interface ArchitectureElement;
Layer implements ArchitectureElement = Name "("
                                        arguments:(ArchArgument || ",")* ")";
IOElement implements ArchitectureElement = Name
                                ("[" index:ArchSimpleExpression "]")?;
ArrayAccessLayer implements ArchitectureElement = "[" index:ArchSimpleExpression "]";
ParallelBlock implements ArchitectureElement = "(" groups:ArchBody "|"
                                        groups:(ArchBody || "|")+ ")";
ArchBody = elements:(ArchitectureElement || "->")*;
```

The interface `ArchitectureElement` includes all elements which can be linked with the serial connection operator. These are the layers (`Layer`), the ports (`IOElement`), the selection layers (`ArrayAccessLayer`) and the parallelization blocks (`ParallelBlock`). The nonterminal `ArchBody` produces an list of architecture elements that are connected with the serial connection operator. The parallelization block in turn uses this nonterminal to produce the groups with are connected with the parallelization operator.

Next, we will show the most important nonterminal of CNNArch, which is the nonterminal `Architecture` that has the following production rule:

```
Architecture = methodDeclaration:LayerDeclaration*
                body:ArchBody;
LayerDeclaration = "def"
                    Name "("
                    parameters:(LayerParameter || ",")* ")" "{"
                    body:ArchBody "}";
```

The `Architecture` is the nonterminal which can be embedded into another language like EmbeddedMontiArc. It contains a list of user-defined layers and the network construction itself in form of the nonterminal `ArchBody`. The nonterminal `LayerDeclaration` declares new constructed layers. It has a name, a list of parameters and the layer construction that is also produced by `ArchBody`.

At last, let us look at the remaining nonterminals which are only used in the stand-alone version of CNNArch. These are the nonterminals `CNNArchCompilationUnit`, `IODeclaration`, `ArchType` and `Shape`. They are constructed with the following productions:

```
CNNArchCompilationUnit = "architecture"
                        name:Name
                        ( "(" (ArchitectureParameter || ",")* ")" )? "{"
                        ioDeclarations:IODeclaration*
                        Architecture "}";
IODeclaration = "def"
                (in:"input" | out:"output")
                type:ArchType
                Name
                (ArrayDeclaration)?;
ArchType = ElementType "^" Shape;
Shape = "{" dimensions:(ArchSimpleExpression || ",")* "}";
```

The `CNNArchCompilationUnit` is the main node of the AST and contains the whole file. It defines the name of the network architecture, an optional list of network parameters, a list of ports and the nonterminal `Architecture`. The nonterminal `IODeclaration` defines an input or output port in a similar way as EMA and `ArchType` defines the type of the port. `ArchType` reuses the nonterminal `ElementType` from MontiCAR but changes the production rule for the dimensions of the type to simplify the resolution of architecture parameters that are used in the port definition.

5.2 CNNArch Symboltable

In this section, we want to explain the symbols of CNNArch and how the model creation works. A model in the language is completely based on the symbols and the AST is only used to create the symboltable. Even the CoCos in CNNArch work for the most part on the symbols and not on the AST (see section 5.3). The model creation in CNNArch has three phases. First, the AST is created by the parser that was generated by MontiCore. Second, the Java class `CNNArchSymbolTableCreator`, which extends the `CommonSymbolTableCreator` of MontiCore, creates the symboltable based on the AST. Third, we complete the architecture resolution process of CNNArch in order to be able to check the validity of the neural network. This process is started by calling the `resolve()` method of the `ArchitectureSymbol`, which is automatically called if the static method `checkAll` of the class `CNNArchCocos` is used to check the CoCos of CNNArch. The architecture resolution should not be confused with the symbol resolution of MontiCore. The architecture resolution process has the following two steps:

1. It evaluates all arithmetic and logical expressions in the modeled architecture.

2. It unrolls argument sequences, structural arguments and constructed layers such that the network architecture is constructed out of atomic elements. Atomic elements are explained in section 5.2.5.

The figure 5.1 shows a class diagram of the main symbol classes used in CNNArch. The following classes are not shown in the diagram: the classes of MontiMath like the `MathExpressionSymbol`, the classes of MontiCore like `CommonSymbol` or `Scope-SpanningSymbol`, the `TupleExpressionSymbol` which is a subclass of the `Math-ExpressionSymbol` and the classes of the predefined layers. Additionally, the diagram does not show if a class overrides a method in the super class and it does not show attributes and methods that are inherited from a MontiCore superclass, e.g. the name of the symbol. In the following, we will explain the symbol classes and how they interact in detail.

5.2.1 Expressions

The abstract symbol `ArchExpressionSymbol` corresponds to the nonterminal `Arch-Expression`, which means that it is used by the layer arguments in CNNArch. It has two subclasses which are the `ArchSimpleExpressionSymbol` and the abstract class `ArchAbstractSequenceExpression`, which represents an argument sequences and is in turn extended by `ArchRangeExpressionSymbol` and `ArchSequenceExpression-Symbol`. These expression symbols have similar to the expression symbols in MontiMath no name and are only directly linked by other symbols as can be seen in the class diagram. The `ArchExpressionSymbol` provides many convenience methods that are implemented by the subclasses.

The `ArchRangeExpressionSymbol` and the `ArchSequenceExpressionSymbol` both override the `getElements` method which returns the elements of the argument sequence in a list of list of simple expressions. The first list contains the groups which are separated by the parallelization operator and the second list contains the elements which are

Figure 5.1: Class diagram of the main symbols in CNNArch.
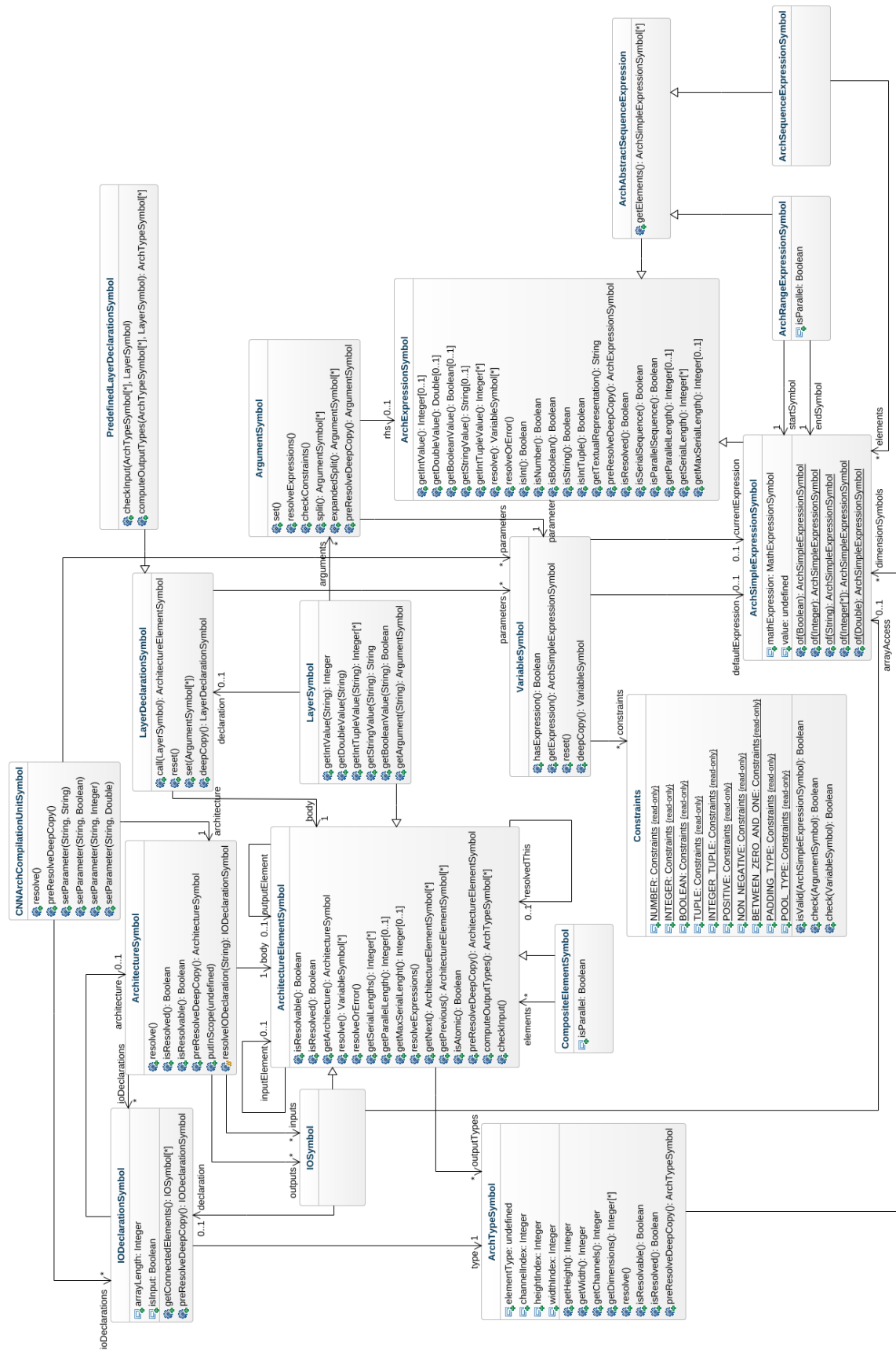
separated by a serial connections in the argument sequence. In case of `ArchSequence-ExpressionSymbol`, the list is stored as an attribute at the time of symboltable creation. However, the `ArchRangeExpressionSymbol` has to compute the list based on the start and end expression in the range. This can only be done after the first step of the architecture resolution process.

The class `ArchSimpleExpressionSymbol` has two attributes. The attribute `mathExpression` is an optional `MathExpressionSymbol` from MontiMath and the attribute `value` is a general Java object. The value of the expression can be a string, a boolean, a double, an integer, or a list of integers, which represents a tuple. Note that the CNNArch is currently restricted to integer tuples since only those are used by existing predefined layers. The class provides the convenience methods `getIntValue`, `getDoubleValue`, `getStringValue`, `getBooleanValue` or `getIntTupleValue` that return the value as an Optional of the corresponding type. The value can either be set at object creation or be computed from the `mathExpression`.

The `mathExpression` is the arithmetic expression, the logical expression or the tuple of arithmetic expressions that is evaluated in the first step of the architecture resolution process. The `TupleExpressionSymbol`, which implements tuples, is a subclass of the `MathExpressionSymbol`. The math expressions are evaluated by calling the `resolve` method. The method first uses the symbol resolution process of MontiCore to resolve the names of variables in the expression and then uses the singleton class `Calculator` that computes the value from a representation of the expression where the names are replaced by their actual values. To compute the values for each name in the expression, the `resolve` method is called recursively on all referenced variables if the value of those variables did not already exist. Tuples are handled in a special way by computing the values of each element in the tuple separately and storing the computed values in a list.

5.2.2 Constraints and Variables

Variables in CNNArch are in general untyped but constraints can be applied to them internally that act similar to types. These constraints are used for parameters of predefined layers to be able to check the validity of the neural network. A constraint can be something general, e.g. the value has to be an integer, or it can be more specific, e.g. the value has to be a number between 0 and 1, or the value has to be a string which is equivalent to either "max" or "avg". A variable can have more than one constraint. If a single constraint is not satisfied, CNNArch will throw an error during the second step of the architecture resolution process. This is the only type of error in CNNArch aside from parsing errors which cannot be checked by a CoCo (see section 5.3). The different constraints are implemented in the Enum `Constraints`. Each constraint overrides the `isValid` method which checks whether the value is valid.

All types of variables in CNNArch are realized with the class `VariableSymbol`. It has a name, an optional default expression, an optional current expression and a set of constraints as explained above. The value of the variable is either given by the default or the current expression. The method `getExpression` returns the current expression, if it is present, or otherwise, the default expression. Note that the value of a variable cannot be an argument sequence. Argument sequences are transformed into simple arguments in the architecture resolution process. Furthermore, the variable has a `reset` method that removes the current expression in order to reset the variable to the default state. Also, the

booleans `true` and `false` are handled in the same way as variables. At the symboltable creation phase, they are added to the model as constant variables.

5.2.3 IODeclarationSymbol and ArchTypeSymbol

The `IODeclarationSymbol` corresponds to the nonterminal `IODeclaration` in the stand-alone version or to the port symbol of EmbeddedMontiArc in the embedded version. It contains all information about an port of the network architecture. It has a boolean attribute that tells us whether it is an input or an output. It has an integer attribute which denotes the length of the port array and is equal to one if the port was not defined as an array. Furthermore, it has a single `ArchTypeSymbol` attribute which denotes the type of the port. It is also linked to the single `ArchitectureSymbol` of the model. However, this link does not exist directly after the symboltable creation phase. Instead, the architecture symbol will create the connection after the port declaration symbol was resolved for the first time with MonitCore. The reason behind this is that the declaration symbol is not known in the symboltable creation phase if the language is embedded into EMA (see section 5.4). Furthermore, the architecture symbol will also store all ports declaration symbols in a list after they are resolved to allow for an easy access to all inputs and outputs of the neural network.

The `ArchTypeSymbol` defines the domain and the dimensions of a tensor. The domain is implemented with the class `ASTElementType` of EmbeddedMontiArc and the dimensions are defined with a list of simple expression symbols. To make the type flexible for different tensor formats, it has integer attributes that define the index of each dimension (height, width and depth) in the list. Access to length of these dimensions is respectively provided by the methods `getHeight`, `getWidth` and `getChannels`.

5.2.4 LayerDeclarationSymbol

The `LayerDeclarationSymbol` is a scope spanning symbol that stores all information about a predefined or constructed layer. In case of constructed layers it corresponds to the nonterminal `LayerDeclaration`. The symbol contains the list of parameters of the layer and the layer construction in form of the attribute `body`, which is a single architecture element symbol. It has a `call` method that is used in step two of the architecture resolution process. The method will be explained in section 5.2.6.

For predefined layers exist the abstract subclass `PredefinedLayerDeclaration-Symbol` which in turn is extended by a class for each individual predefined layer, e.g. the class `Convolution`. All predefined layers are added as symbols to the model at the symboltable creation phase. They are added directly in the scope of the architecture symbol. Furthermore, Each individual class of a predefined layer implements methods that describe the behavior of the predefined layer such that the validity of the network can be checked with a CoCo. How these methods work and how new predefined layers can be added to CNNArch is explained in section 5.2.8.

5.2.5 Architecture Elements

The architecture is built out of instances of the abstract class `ArchitectureElement-Symbol`, which is a scope spanning symbol. It corresponds to the nonterminals

`ArchitectureElement` and `ArchBody`. It uses the composite design pattern to build the entire architecture. It has three subclasses. The composite `CompositeElement-Symbol` and the two leafs `IOSymbol`, which represents the ports, and `LayerSymbol`, which represents the layers of the architecture. The composite contains in addition to the list of architecture elements a boolean attribute called `isParallel` that denotes which data flow operator connects the listed elements. If it is true, the elements are connected with the parallelization operator and the composite is a parallelization block. In the other case, the composite is equivalent to the nonterminal `ArchBody` and the contained elements are connected by the serial connection operator.

The `IOSymbol` corresponds to the nonterminal `IOElement` and represents a port of the architecture. It has name and an optional expression, which denotes the array index of the port if such an index exists. All other information about the port is stored in the `IODeclarationSymbol` with the same name as the port. The attribute `definiton` which links the declaration symbol is initially empty. By calling the method `get-Definition`, the port uses the symbol resolution of MontiCore to find the declaration symbol. However, it does not do this directly but instead finds the single `Architecture-Symbol` of the model and uses the method `resolveIODeclaration` of the architecture symbol, which returns the declaration symbol. This is done for the reasons explained in the previous section.

The `LayerSymbol` corresponds the nonterminals `Layer` and `ArrayAccessLayer`. It represents predefined and constructed layers in the architecture. It has a name, an attribute that stores a list of arguments and an optional attribute that stores the declaration symbol of the layer. The layer uses the symbol resolution mechanism of MontiCore the first time the getter of the attribute is called to find the declaration symbol. In addition, the layer symbol has convenience methods to find an argument or the value of an argument by name.

An architecture element is called *atomic* if it is either a predefined layer, a single port (not a port array), or an *empty composite*. A composite symbol is called empty if the list of architecture elements in the composite is empty. This can happen as a result of the architecture resolution process, e.g. in the case where the conditional structural argument of a layer is false. To check if an architecture element is atomic, it provides the method `isAtmoic`.

Each architecture element also has the optional attribute `resolvedThis`, which stores the resolved version of an element. This attribute is only empty before the architecture resolution phase of model creation. If an element is atomic or a composite, the attribute is equal to the element itself. If the element was a port array without a selected array access index, the resolved version will be composite consisting of the atomic ports of the array. In the case of a constructed layer, it will be a composite consisting of resolved architecture elements where all variables are replaced by the respective argument values and all expressions are evaluated. Additionally, architecture elements have the attribute `outputTypes` that stores the types of all output tensors of the element in a list. These types are calculated in the last step of the architecture resolution process.

Furthermore, all architecture elements store the previous and the following element if one exists. For the purpose of an easy navigation through the model, the architecture element symbol also provides the methods `getPrevious` and `getNext` which can only be used after the architecture resolution process. The `getNext` returns a list with all atomic elements (excluding empty composites) which have the respective element symbol as an

```
1  public Set<VariableSymbol> resolve() throws ArchResolveException {
2      if (!isResolved()) {
3          if (isResolvable()) {
4              resolveExpressions();
5              int parallelLength = getParallelLength().get();
6              int maxSerialLength = getMaxSerialLength().get();
7
8              if (!isActive() || maxSerialLength == 0) {
9                  //set resolvedThis to empty composite to remove the layer.
10                 setResolvedThis(new CompositeElementSymbol.Builder().build());
11             }
12             else if (parallelLength == 1 && maxSerialLength == 1) {
13                 //resolve the layer call
14                 ArchitectureElementSymbol resolvedMethod = getDeclaration().call(this);
15                 setResolvedThis(resolvedMethod);
16             }
17             else {
18                 //split the layer if it contains an argument sequence
19                 ArchitectureElementSymbol splitComposite = resolveSequences(
                        ↪ parallelLength, getSerialLengths().get());
20                 setResolvedThis(splitComposite);
21                 splitComposite.resolveOrError();
22             }
23         }
24     }
25     return getUnresolvableVariables();
26 }
```

Figure 5.2: The `resolve` method of the `LayerSymbol`

input. Thus, it is possible to call the method `getInputs` in the architecture symbol to get all atomic input ports and to use `getNext` recursively to navigate through all predefined layers and ports that construct the network. The method `getPrevious` does the same in the opposite direction.

5.2.6 Architecture Resolution

The architecture resolution process starts by calling the `resolve` method of the architecture symbol. In the stand-alone, it also possible to call the method with the same name in the `CNNArchCompilationUnitSymbol`, which will in addition check if all network parameters have values. After that, the `resolve` method of the architecture element that is stored in the architecture symbol is called. This architecture element is most likely a composite element, which will just relegate the method call to all elements stored in the list.

If the `resolve` method of an `IOSymbol` is called, it will evaluate the array index expression if it exists and check if it is a non-negative integer. After that, it will also evaluate the expressions which defined the dimensions of the `ArchTypeSymbol` in a similar way. Then in the case that the port declaration defines an array, it will create a composite consisting of the corresponding atomic ports and store it with the `resolvedThis` attribute. In doing so, it will also add these atomic ports to the architecture symbol. If the port is an output, it will in addition add selection layers before each atomic port in the composite to realize the behavior as explain in section 4.4.5.

If the `resolve` method of an `LayerSymbol`, which is shown in figure 5.2, is called, it will start by evaluating the expressions of all layer arguments. In doing so, the arguments will check whether the constraints of the corresponding parameters are satisfied. Then, the

67

```
1  public ArchitectureElementSymbol call(MethodLayerSymbol layer) throws
       ↪ ArchResolveException{
2      checkForSequence(layer.getArguments());
3
4      if (isPredefined()){
5          return layer;
6      }
7      else {
8          set(layer.getArguments());
9
10         CompositeElementSymbol copy = getBody().preResolveDeepCopy();
11         copy.putInScope(getSpannedScope());
12         copy.resolveOrError();
13         getSpannedScope().remove(copy);
14         getSpannedScope().removeSubScope(copy.getSpannedScope());
15
16         reset();
17         return copy;
18     }
19 }
```

Figure 5.3: The `call` method of the `LayerDeclarationSymbol`

method resolves the argument sequences and structural arguments. In case of an conditional structural argument which is set to false, the attribute `resolvedThis` will be set to an empty composite. The other two types of structural arguments are internally transformed into an argument sequence and handled the same way as other arguments. If an argument sequence exist in the layer, the `resolve` method creates a new composite that consists of multiple versions of the respective layer with different single value arguments according to the argument sequence. To do this, it uses the `expandedSplit` method of the `ArgumentSymbol` to compute the arguments for each layer in the composite. Thereafter, the `resolve` method of the composite is called recursively. If the layer does not contain an argument sequence, the `call` method of the layer declaration symbol is called with the respective layer as an argument. The figure 5.3 shows the implementation of this method. It returns the layer itself if it is an predefined layer or it creates and returns a resolved copy of the architecture element, which defines the layer construction (the attribute `body`). This copy was resolved by setting the parameter values according to the arguments of the layer symbol.

5.2.7 Instance Creation

It is possible to create instances with different network parameters in CNNArch. This is mainly done through the method `preResolveDeepCopy` in `ArchitectureSymbol` or `CNNArchCompilationUnitSymbol`. This method creates an unresolved copy of the model. This means that the copy is in exactly the same state as the original model was directly after the symboltable creation phase. This even works on architecture symbols which were already resolved. To create instances in the stand-alone version, we can create a copy of the compilation unit symbol and use the `setParameter` method to set values for the network parameters. After that the `resolve` method can be called either manually or by checking the CoCos with the class `CNNArchCocos`. To embed an instance of the `ArchitectureSymbol` into the scope of another symbol, it is useful to call the `putInScope` method that adds the copy properly to the given scope, such that the `resolve` method can be called. We will explain in detail how the language is embedded into EMA in section 5.4.

5.2.8 Adding Predefined Layers

To add a new predefined layer to language it is only necessary to modify the list returned by the static `createList` method in the class `AllPredefinedLayers`, which is shown in figure 5.4. However, the new layer has to be a subclass of the `PredefinedLayer-DeclarationSymbol` and has to implement the methods `checkInput` and `compute-OutputTypes`. The first method has the task to check if the list of input tensors is a valid input to the respective layer. For example, most layers should log an error if the number of inputs is not equal to one. This and other common checks are already implemented in the superclass and can be reused. Another example would be given by the `Concatenate` layer which has to check that all input tensors have the same height and width.

The method `computeOutputTypes` has the task of calculating the list of output tensor types. Note that not the values of the tensor have to be calculated but the dimensions and the number of outputs. Moreover, errors do not have to be handled in this method. It can be assumed that `checkInput` was called previously. Furthermore, to access certain arguments for computation it is recommended to use the constants in the class `AllPredefinedLayers` which define the names of all predefined layer parameters. In this way, the names of parameters can be easily changed without affecting the whole project. Parameters can be added to the new predefined layer by the using the builder of the `VariableSymbol` in the static method that creates the new layer. An example of the complete implementation of a predefined layer in CNNArch is shown in figure 5.5. Furthermore, the code generator also has to be extended to add a new predefined layer to the language. This can be done by adding a single freemarker template with the same as the layer to the code generator. This is explained in more detail in section 5.5.1.

5.3   CNNArch CoCos

There exist three types of CoCos in CNNArch and checking all CoCos is a four step process, which involves the network architecture resolution. This is a consequence of the abstract and user-friendly way CNNArch models neural network architectures. The three types are the CoCos, which can work directly on the AST, the CoCos, which work on symbols before the architecture resolution process, and the CoCos that check symbols after the resolution. The third type of CoCo is necessary for complicated context condition, e.g. checking that each architecture element has a valid input. In addition, having the CoCos based on the symbols allows us to check errors of created instances of a network in the same way. Moreover, the CoCos can be reused completely by any language which embeds CNNArch. All CoCos can be checked by calling the `checkAll` method of the class `CNNArchCocos` that handles the architecture resolution process and checks the CoCos in the correct order. There are in total 20 different types of errors with unique error codes that are defined as constants in the class `ErrorCodes`.

There exist six CoCos of the first type. They use the standard MontiCore mechanism for checking CoCos. Thus, each CoCo implements a single `check` method which gets an instance of a specific AST-class as argument and checks whether the context condition is satisfied for this node. These CoCos can be combined in a so called *CoCoChecker* of MontiCore that traverses the complete AST model and calls the said method for each node of the respective class. The implemented CoCos check, for example, that the following holds: no recursions occurs in a constructed layer, parameters and layers are not defined

```
1  public static List<PredefinedLayerDeclarationSymbol> createList(){
2      return Arrays.asList(
3              FullyConnected.create(),
4              Convolution.create(),
5              Softmax.create(),
6              Sigmoid.create(),
7              Tanh.create(),
8              Relu.create(),
9              Dropout.create(),
10             Flatten.create(),
11             Pooling.create(),
12             GlobalPooling.create(),
13             Lrn.create(),
14             BatchNorm.create(),
15             Split.create(),
16             Get.create(),
17             Add.create(),
18             Concatenate.create());
19 }
```

Figure 5.4: The `createList` method in the class `AllPredefinedLayers`.

```
1  public class FullyConnected extends PredefinedLayerDeclaration {
2      private FullyConnected() {
3          super(AllPredefinedLayers.FULLY_CONNECTED_NAME);
4      }
5
6      @Override
7      public List<ArchTypeSymbol> computeOutputTypes(List<ArchTypeSymbol> inputTypes,
           ↪ LayerSymbol layer) {
8          return Collections.singletonList(new ArchTypeSymbol.Builder()
9                  .height(1)
10                 .width(1)
11                 .channels(layer.getIntValue(AllPredefinedLayers.UNITS_NAME).get())
12                 .elementType("-oo", "oo")
13                 .build());
14     }
15
16     @Override
17     public void checkInput(List<ArchTypeSymbol> inputTypes, LayerSymbol layer) {
18         errorIfInputSizeIsNotOne(inputTypes, layer);
19     }
20
21     public static FullyConnected create(){
22         FullyConnected declaration = new FullyConnected();
23         List<VariableSymbol> parameters = new ArrayList<>(Arrays.asList(
24                 new VariableSymbol.Builder()
25                         .name(AllPredefinedLayers.UNITS_NAME)
26                         .constraints(Constraints.INTEGER, Constraints.POSITIVE)
27                         .build(),
28                 new VariableSymbol.Builder()
29                         .name(AllPredefinedLayers.NOBIAS_NAME)
30                         .constraints(Constraints.BOOLEAN)
31                         .defaultValue(false)
32                         .build()));
33         declaration.setParameters(parameters);
34         return declaration;
35     }
36 }
```

Figure 5.5: Example of a predefined layer class in CNNArch.

multiple times, arguments refer to actual parameters, used layers actually exist, required parameters have arguments, and only valid names are used to define constructed layers or parameters (e.g. `true` and `false` are not valid names).

Furthermore, there are two CoCos of the second type. The first one checks that the ports used in the architecture exist and are valid. And the second one checks that variables used in expressions are actually defined in the respective scope. These CoCos can only work on symbols and not the AST since ports and certain variables are declared outside of the AST of CNNArch if the embedded version is used. Moreover, multiple symbols will have a different behavior but refer to the same node in the AST in different instances of a network architecture. These CoCos are subclasses of the developed `CNNArchSymbolCoCo` which has a `check` method for each symbol type of CNNArch. The symbol CoCos are combined with the `CNNArchSymbolCoCoChecker` that traverses through the model and checks each CoCo for the respective symbols.

There are 4 CoCos of the third type. The most important one is the CoCo `CheckElement-Inputs` that recursively calls the `checkInput` method of each architecture element in the model. This assures that the inputs of each predefined layer and each port are valid. The other CoCos check that all tensor types are valid (e.g. dimensions should only be positive integers), that port arrays are handled in the correct way, and that the network architecture is complete. We call an network complete if there are no open streams at the end of the defined architecture and if there exist at least one input and one output port.

5.4   EmbeddedMontiArcDL

In this section, we will explain how CNNArch is integrated into EmbeddedMontiArc with EMADL. The EMADL language has a very simple grammar that extends EmbeddedMontiArcBehavior, CNNArch and MontiMath.

The nonterminal `BehaviorEmbedding` either produces the nonterminal `Architecture` of CNNArch or the nonterminal `MathStatements` of MontiMath. The grammar is shown in the following:

```
grammar EMADL extends de.monticore.lang.embeddedmontiarc.EmbeddedMontiArcBehavior,
                      de.monticore.lang.monticar.CNNArch{
    EMADLCompilationUnit = EMACompilationUnit;
    BehaviorEmbedding = Architecture | MathStatements;
    BehaviorName = name:"CNN" | name:"Math";
}
```

EMADL uses the language embedding feature of MontiCore to compose a symboltable of the host language EMA and the embedded languages CNNArch and MontiMath. The main symbols created by a EMADL component are the `ComponentSymbol` and the `ExpandedComponentInstanceSymbol` which are imported from EMA. The scope of the component symbol automatically contains the main symbols of the respective implementation language, which are `ArchitectureSymbol` for CNNArch and `Math-StatementsSymbol` for MontiMath.

However, this is not enough to create instances of a network with different parameters. For that, we had to extend the `ExpandedInstanceSymbolCreator` and the `ExpandedComponentInstanceBuilder` of EMA to add a copy of the architecture symbol into the scope of each created component instance. Port symbols and parameters are converted into symbols of CNNArch and the `resolve` method of the architecture

symbol can be called to start the architecture resolution process. This is automatically done by using the static `checkAll` method of the class `EMADLCocos` on the component instance. The method automatically resolves the architecture and checks CoCos of CNNArch, MontiMath and EMA. There exists only one CoCo that is unique to EMADL and it checks that the name specified with the nonterminal `BehaviorName` corresponds to the actual implementation language.

5.5 Generator

In this section, we will explain how the code generator of EMADL is implemented. To do that, we have to first explain how the generated product of the EmbeddedMontiArcMath generator is structured. Each EMAM component is translated into a C++ class. This class has an attribute for each port and for each parameter of the component. Furthermore, it has the two methods `init`, which initializes the attributes of the component, and `execute`, which executes the computation. The classes use the linear algebra library *armadillo* as a backend to be able to efficiently calculate the math expressions of MontiMath. As a consequence, the attributes use the data types of armadillo. Vectors, matrices and three-dimensional matrices are stored respectively with the armadillo types `colvec`, `mat` and `cube`. We keep the same structure with our code generator and reuse the EMAM generator for all components which do not use CNNArch as implementation language.

We decided to split the code generator into two smaller generators to create a more modular software. The first one is the CNNArch generator which works with an CNNArch model and generates all the code that depends on the architecture of the neural network. This includes the *creator*, which is a Python script that can be called with training parameters to construct, train and store the neural network, the *predictor*, which is a C++ class that loads the trained network and predicts labels for deployment, and a C++ code snippet that is the execute method of the component. Note that the CNNArch generator can be used for the stand-alone version of CNNArch. However, the generated code is designed to be used by the second generator. This is the EMADL generator, which combines EMA, MonitMath, CNNArch and CNNTrain to generate code for a complete EMADL system. In the following, we will explain both generators.

5.5.1 CNNArch Generator

The CNNArch code generator is heavily based on the freemarker template engine. There are two ways to start the generation process. For the stand-alone version of CNNArch, it is possible to use the class `CNNArchGeneratorCli` which implements a command line interface. It can be called by providing arguments which denote the model directory, the model name and the target directory of the generation. The other way is by creating an instance of the class `CNNArchGenerator` and calling the method `generateStrings` of this instance. This method accepts an `ArchitectureSymbol` as argument and returns a Java Map where the keys are the names of the generated files and the values are the content of the respective files as strings. The Map has four entries. The first one is the C++ file `CNNBufferFile.h`, which is a simple class used by the predictor to read files. The other three are based on the templates `CNNCreator.ftl`, `CNNPredictor.ftl` and `execute.ftl`. The template `execute.ftl` is special since it does not represent a file in the generated product but instead the execute method of the component. The other

two templates generate respectively the creator and the predictor. These three templates are listed in appendix C.

To process a template, we use the class `CNNArchTemplateController` which handles all interactions with freemarker and the templates. The data model of the mentioned templates only use a reference to the template controller under the name `tc`. We list most of the methods which are used by the templates in the following.

- The reference `tc.architecture` returns the architecture symbol.

- The method `tc.getName` has a single argument which can be a predefined layer or a port of the network. It returns a unique name for the layer or port. This name is used to declare variables in the generated code. The template controller uses the class `LayerNameCreator` to create these unique names for each predefined layer and each port in the network architecture.

- The reference `tc.architectureInputs` returns a list with the names of all inputs of the neural network.

- The reference `tc.architectureOutputs` returns a list with the names of all outputs of the neural network.

- The convenience method `tc.join` has two required arguments which are a list of objects and a string. It calls the `toString` method on each element of the given list and joins the resulting string with the given separator from the second argument. Additionally, the method has two optional string arguments which denote respectively a prefix and a postfix for each element in the list. The method is for example used in the predictor template together with `tc.architectureInputs` and `tc.architectureOutputs` to generate the parameters for the `predict` method.

- The reference `tc.architectureName` returns the name of the network. This is equivalent to the name of the respective component in EMADL.

- The reference `tc.fullArchitectureName` returns the fully qualified name of the network. This is equivalent to the fully qualified name of the respective component in EMADL. Fully qualified means that it also depends on the package and the super-component if one exists. The different are separated by a dot to construct the fully qualified name.

- The reference `tc.fileNameWithoutEnding` returns the respective file name without file extension. This is used to denote the generated Python or C++ class in the creator and the predictor.

However, there are additional templates used in the generation process to make the code generator more modular and extensible. There exist one template for each predefined layer of CNNArch and additionally two templates which generate code for an input or an output port. These templates are included by the `CNNCreator.ftl` and are used to construct the neural network. However, we do not use the `include` directive of freemarker but instead our own `include` method provided by the template controller. This method gets an arbitrary architecture element symbol as argument and not a path to a template. In this way, the template controller keeps track which architecture element is processed at each moment and it can provide information about the current state to the respective

```
1  <#assign input = element.inputs[0]>
2  <#if element.padding??>
3  <#assign input = element.name>
4          ${element.name} = mx.symbol.pad(data=${element.inputs[0]},
5              mode='constant',
6              pad_width=(${tc.join(element.padding, ",")}),
7              constant_value=0)
8  </#if>
9          ${element.name} = mx.symbol.Convolution(data=${input},
10             kernel=(${tc.join(element.kernel, ",")}),
11             stride=(${tc.join(element.stride, ",")}),
12             num_filter=${element.channels?c},
13             no_bias=${element.noBias?string("True","False")},
14             name="${element.name}")
```

Figure 5.6: The template `Convolution.ftl`.

template. This is done via the class `ArchitectureElementData` which is constructed based on the current predefined layer or port. An instance of this class is referenced in the data model of the respective template under the name `element`. Furthermore, the `include` method can also be used on architecture elements which are not atomic. In this case, the method will recursively call itself on all elements contained in the respective architecture element. Serial and parallel composites are treated in exactly the same way. This is possible because the output of each predefined layer is stored as a variable in the generated code. Thus, each predefined layer can be independently generated based on the variables that correspond to the inputs of that layer. The inputs of a predefined layer or output port are provided by the freemarker reference `element.inputs`, which is the list of all input names of the respective architecture element. This list is computed based on the `getPrevious` method of the respective architecture element (see section 5.2.5). Moreover, the variable name of the output of the current element is given by the reference `element.name`.

We will not show all templates used in the construction of the generated network in this thesis since there are 18 different templates and all are based on the same principles. However, the figure 5.6 shows one example, which is the template that generates the convolutional layer. The first part of the template determines whether padding should be used and will apply the zero-based padding to the input tensor if this is the case. After that the actual convolutional layer is defined with the given arguments. There exist convenience methods for each possible argument of a layer in the class `ArchitectureElementData`. For example the references `element.kernel` and `element.stride` are used in the template to get the kernel and the stride argument that are defined by the user in the model. Furthermore, the reference `element.padding` calculates the exact amount of zero-based padding on each side of the tensor based on the padding mode used in the corresponding predefined layer.

A new predefined layer can be added to the code generator by creating a template with the same name as the respective layer. The `include` method will automatically find the template based on the name if it is in the template directory.

5.5.2 EMADL Generator

The class `EMADLGeneratorCli` provides a command line interface for the EMADL generator. It has arguments which denote respectively the path to the model directory, the qualified name of the main EMADL component, and the path of the output directory.

The generator generates the complete deep learning system by recursively calling the `generateComponent` method of the class `EMADLGenerator` for all subcomponents of the system. If a component has a MontiMath implementation or contains subcomponents, it will call the EMAM generator on that specific component and collect the generated files. The interface of a component with a CNNArch implementation is also generated in this way. However, we modify the generated strings by adding the execute method and an attribute, which stores the predictor, to the C++ class that represents the component. The execute method and the additional files for the predictor and the creator are generated by the CNNArch generator. After all components of the system are processed, we generate additionally the static C++ file `CNNTranslator.h` and the *trainer*, which is a single Python script that can train all neural networks of the system according to the given CNNTrain configuration file. The trainer is generated based on the simple template `CNNTrainer.ftl`, which is listed in appendix C. All generated files are collected in a list and created in the target directory at the end of the generation process.

5.6   JUnit Tests

We implemented many JUnit tests for the languages CNNArch and EMADL that get automatically executed when the languages are built with maven. We developed 20 positive tests and 27 negative tests which ensure the correct behavior of the CoCos of CNNArch. The positive tests consists of common CNN architectures like AlexNet and ResNet and simple neural networks, which test that specific features of the language do not lead to errors. The negative tests are invalid neural network architectures which check that a specific error message is thrown by a CoCo. In this way, it is ensured that each type of error in the language occurs correctly. Furthermore, we implemented two negative parser tests which ensure that a specific syntax cannot be parsed by the parser and two tests which ensure that instances of a network architecture are correctly created. In addition, we have six tests for the code generator. These tests check that no freemarker exceptions occurs in the generation process and that the desired code is generated. Moreover, we implemented several tests for each arithmetic and logical operator used in the language so that we can ensure the correct behavior of the calculator class, which computes the results of all mathematical expressions used in the network architecture. We developed less JUnit tests for EMADL since it is only a composition of other languages. However, we still have tests to ensure that instances are correctly created, that the code is correctly generated and that the CNNArch cocos can be properly checked for a component instance with the CNNArch implementation language.

Chapter 6

Evaluation

In this chapter, we want to evaluate the generated product. In order to do this, we decided to create a simple application that uses an EMADL model to classify images. In the following, we will first state which dataset we used to train and test the application and then present the modeled EMADL system. Thereafter, we will discuss the results of the training process of the neural network. Finally, we will describe the complete application and state how we used the generated code.

6.1 Dataset

We decided against using the MNIST dataset of handwritten digits, which we presented earlier, for the main evaluation since it consists of normalized gray-scale images. In practice, images are usually in color and not normalized. Instead, we chose the Cifar10 dataset, which is available at [KNH14], to test our language. This dataset consists of small color images with a resolution of 32×32. The dataset is divided into 10 classes which are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The figure 6.1 shows examples of the dataset. The training set of Cifar10 contains 50000 images and the test set contains 10000 images.
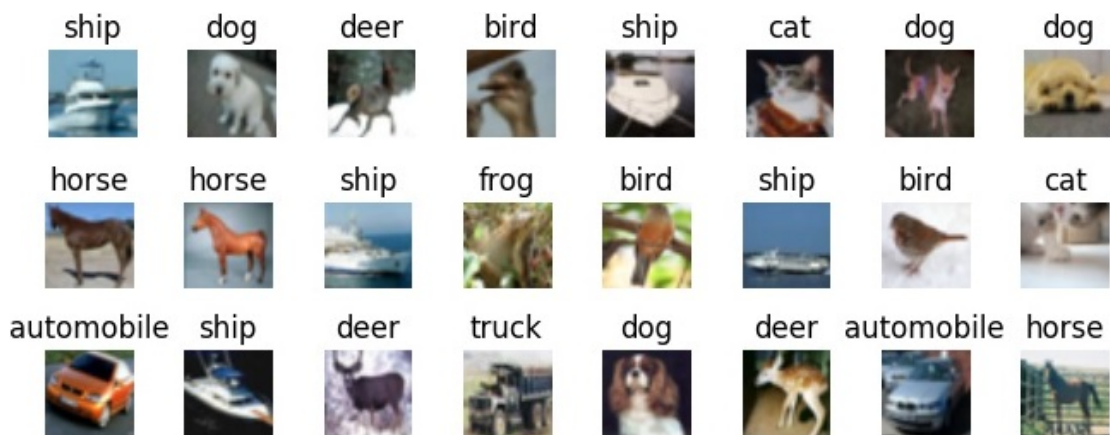

Figure 6.1: Example images of the Cifar10 dataset.

```
1  package cifar10;
2
3  component CifarNetwork<Z(2:oo) classes = 10>{
4      ports in Z(0:255)^{3, 32, 32} data,
5          out Q(0:1)^{classes} softmax;
6
7      implementation CNN {
8          def conv(kernel, channels, stride=1, act=true){
9              Convolution(kernel=(kernel,kernel), channels=channels, stride=(stride,
                  ↪ stride)) ->
10             BatchNorm() ->
11             Relu(?=act)
12         }
13         def resLayer(channels, stride=1, addSkipConv=false){
14             (
15                 conv(kernel=3, channels=channels, stride=stride) ->
16                 conv(kernel=3, channels=channels, act=false)
17             |
18                 conv(kernel=1, channels=channels, stride=stride, act=false, ? =
                      ↪ addSkipConv)
19             ) ->
20             Add() ->
21             Relu()
22         }
23
24         data ->
25         resLayer(channels=8, addSkipConv=true) ->
26         resLayer(channels=16, stride=2, addSkipConv=true) ->
27         resLayer(channels=16, ->=2) ->
28         resLayer(channels=32, stride=2, addSkipConv=true) ->
29         resLayer(channels=32, ->=2) ->
30         resLayer(channels=64, stride=2, addSkipConv=true) ->
31         resLayer(channels=64, ->=2) ->
32         GlobalPooling(pool_type="avg") ->
33         FullyConnected(units=128) ->
34         Dropout()->
35         FullyConnected(units=classes) ->
36         Softmax() ->
37         softmax
38
39     }
40 }
```

Figure 6.2: The CNN used to classify Cifar10 images in the experiment.

6.2 Modeled System

We created an EMADL system which is very similar to the component `MnistClassifier` presented in section 4.1. We have a main component which is called `Cifar10Classifier` which has two subcomponents. The first subcomponent models the neural network with CNNArch and the second subcomponent is identical to the `ArgMax` component presented in figure 4.3. The `Cifar10Classifier` takes an image as input and returns with the single output port the predicted class of the image. The convolutional neural network which we used for this experiment is shown in figure 6.2. This network is more complicated than it needs to be for the sole purpose of testing more features of CNNArch. The network is constructed similarly to a residual network but it has two fully connected layers instead of one and a dropout layer applied to the first fully connected layer. The CNNTrain configuration that we used is presented in figure 6.3. We used a learning rate of 0.01 with exponential step decay and the Adam optimizer to train the network. Moreover, we enabled dataset normalization and added a small weight decay factor to the network for the purpose of regularization.

```
1  configuration CifarNetwork{
2      num_epoch:10
3      batch_size:64
4      normalize:true
5      context:gpu
6      load_checkpoint:false
7      optimizer:adam{
8          learning_rate:0.01
9          learning_rate_decay:0.8
10         step_size:1000
11         weight_decay:0.0001
12     }
13 }
```

Figure 6.3: The CNNTrain configuration file `CifarNetwork.cnnt`.

6.3 Training Process

We created the necessary HDF5 files from the Cifar10 dataset with a simple Python script and generated the Python and C++ files from the modeled system with the EMADL generator. All generated files of the model are listed in appendix D.1. After that, we executed the trainer with Python and successfully trained the network. The result was an accuracy ot 75.3% on the test set and an accuracy of 85% on the training set.

In multiple experiments, we found that dataset normalization has an big impact on the prediction accuracy of the neural network. The accuracy of the network presented in the previous section increases only a little with normalization enabled because it uses batch normalization layers which compute an approximation of the dataset normalization. However, by training a simple LeNet architecture instead, we only get an accuracy of 10% on the Cifar10 dataset without normalization. This is equivalent to a random guess. By activating the dataset normalization procedure that we implemented in the trainer, we get an accuracy of over 65% with the same network.

6.4 Application

In order to create an complete application, we wrote a simple C++ program. This program reads an image file based on a command line argument, which denotes the path to the respective file, and transforms the image into a three-dimensional matrix in form of the armadillo `cube` format. After that, it initializes the C++ class of the component `Cifar10Classifier` and assigns the image to the input port attribute. Then, the `execute` method of the component is called and the value of the output port is printed to the terminal. This value denotes the index of the class. We were able to compile the program and we tested the resulting executable on multiple images of the Cifar10 test set. The program successfully classified most of these images. The executable only depends on the trained network which was automatically stored by the trainer in a directory with the name "models". All other files are not required anymore and can be deleted.

The complete program and the associated project file, which was used to compile the code with the IDE Qt Creator, is listed in appendix D.2. Similar application can be easily created for more complex datasets and bigger CNNs. Furthermore, we showed that the code generator successfully combines CNNArch with MontiMath in a single modeled system and that the trainer of the network works as intended.

Chapter 7

Summary and Outlook

The objective of this thesis was to design and to implement a descriptive modeling language for CNNs that can be used as part of a C&C model in order to enable a model driven development of deep learning based cyber-physical systems. In this chapter, we will first summarize the work of this thesis and then present an outlook on future work.

7.1 Summary

We successfully designed and implemented a DSL for deep learning based cyber-physical systems with the language CNNArch. The language constructs a neural network by modeling the data flow between each layer of the network with the serial connection and the parallelization operator. In this way, the language is able to model arbitrary directed acyclic graphs of layers. It can already model most feedforward neural networks including state-of-the-art convolutional neural networks like ResNet and ResNeXt. Additionally, the language can be easily extended with new layers, which can perform arbitrary computations, to model all the feedforward neural networks that are currently not supported. CNNArch can be used stand-alone or integrated in another language. By realizing features such as layer composition, layer stacking and layer independence, CNNArch can model neural networks in a concise, readable and easily modifiable way. In order to keep the modeled network relatively flexible in regards to input image size, we designed padding in an abstract way with different modes such that the exact amount of padding is automatically calculated based on the respective input tensor.

In addition, we integrated both CNNArch and the mathematical computation language MontiMath into the C&C modeling language EmbeddedMontiArc with the composed language EmbeddedMontiArcDL. In this way, we can model complete systems that combine deep learning with regular computations. CNNArch supports the strict type system of EmbeddedMontiArc. Moreover, the language checks the integrity of a modeled network at model creation and ensures that only valid networks are used. If an invalid neural network is modeled, the language will throw a helpful error messages. Additionally, CNNArch allows for instantiation of network architectures with different parameters and thus it is possible to create reusable components that can be modified for different task, e.g. using the same network architecture on a dataset with a different amount of classes.

Furthermore, we developed the code generator of EmbeddedMontiArcDL which generates code for a C++ application. The generated code contains a Python script that can be

81

executed to train all neural network of the modeled system. The training and test set for each network have to be provided by the user in form of a HDF5 file. The training parameters can be modeled for each network in the configuration language CNNTrain. The training process supports weight decay, step decay for the learning rate, saving and loading of checkpoints, different kinds of optimizers like SGD, Adam or RMSProp, and an optional automatic dataset normalization procedure, which does not require additional work by the user. The only requirement to compile the resulting C++ code is the small predictor library of MxNet in addition to the libraries required by MontiMath. We were able to successfully create and test an application which is based on an EMADL system that contains both CNNArch and MontiMath as implementation languages. This application successfully classified selected image files with the trained network.

7.2 Outlook

The modeling language for neural networks can still be improved in some ways. A mechanism could be added that allows the user to declare the data format of the input image so that images can be automatically converted by the generated code into the format (channels×height×width) which is required by the deep learning backend of the system. This would make the language more convenient to use for a real world application. Furthermore, it can be extended by adding support for recursive neural networks. This can be easily done by implementing new predefined layers that model LSTM and GRU layers. In addition, the code generator could be extended by adding support for the unusual case of multiple inputs or multiple outputs in a neural network, which can be modeled in CNNArch. Moreover, the generated product could be extended so that multiple data storage formats for the training and test set are supported.

The overall modeling tool can be improved by developing additional implementation languages for components. For example, a language is needed that implements IO operations in order to be able to model a complete application. However, we think that the first priority of future work should be the development of a generator framework for Embedde-MontiArc which provides a generalized interface for different implementation languages. At the moment, each implementation language has its own generator, which has to generate the complete system. A generator framework could handle all interactions between components so that the code generator for an implementation language, which would be automatically called by the framework, can focus on the behavior of the component.

In conclusion, this thesis bridges the gap between the techniques of deep learning and software engineering, creating a modeling tool that combines neural networks with regular computations in a safe and modular way. This tool still has to be extended and extensively tested to be able to be used for the development of real safety critical systems. However, we showed that it is possible to develop deep learning based cyber-physical systems with a model-based approach. This thesis is one important step on the way of making deep learning based software components safe, reusable and easy to use.

Bibliography

[ABC⁺16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[AHS85] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[ARAA⁺16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 472:473, 2016.

[AWS] AWS. Deep Learning-AMIs. `https://www.allthingsdistributed.com/2016/11/mxnet-default-framework-deep-learning-aws.html`. Accessed: 2018-3-30.

[Ben12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[Bis06] Christopher M Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

[BJG08] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[Bro14] Larry Brown. Accelerate Machine Learning with the cuDNN Deep Neural Network Library. `https://devblogs.nvidia.com/accelerate-machine-learning-cudnn-deep-neural-network-library/`, September 2014.

[BRSS16] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of caffe, neon, theano, and torch for deep learning. 2016.

[Bur16] Christopher Burger. Google deepmind's alphago: How it works. *TasteHit. Available online at https://www. tastehit. com/blog/google-deepmind-alphago-how-it-works/, updated on*, 3(16):2016, 2016.

[C⁺15] François Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7:8, 2015.

[CBM02]   Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[CKF11]   Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376, 2011.

[CLL+15]   Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[DB09]   Howard Demuth and Mark Beale. Matlab neural network toolbox user's guide version 6. the mathworks inc. 2009.

[Des16]   Adit Deshpande. A beginner's guide to understanding convolutional neural networks. *A Beginner's Guide to Understanding Convolutional Neural Networks–Adit Deshpande–CS Undergrad at UCLA ('19). Np*, 20, 2016.

[DHS12]   Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[Düd14]   Boris Düdder. Automatic synthesis of component & connector-software architectures with bounded combinatory logic. 2014.

[DV16]   Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[Fac17]   Facebook. Caffe2 Open Source Brings Cross Platform Machine Learning Tools to Developers. `https://research.fb.com/downloads/caffe2/`, April 2017.

[G+11]   HDF Group et al. Hdf5 user's guide, 2011.

[GB10]   Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[Hin09]   Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[Hor91]   Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[How17]   Jeremy Howard. Introducing pytorch for fast.ai. `http://www.fast.ai/2017/09/08/introducing-pytorch-for-fastai/`, September 2017.

[HZRS16]   Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[IS15]   Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JJ99]   Dieter Jungnickel and D Jungnickel. *Graphs, networks and algorithms*, volume 5. Springer, 1999.

[JSD+14]   Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[Kar16]   Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. *Neural networks*, 1, 2016.

[KNH14]   Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[KRRvW17]   Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *ECMFA*, 2017.

[KSH12]   Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[L+89]   Yann LeCun et al. Generalization and network design strategies. *Connectionism in perspective*, pages 143–155, 1989.

[LBBH98]   Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[LBD+89]   Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4), 1989.

[Low04]   David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 2004.

[Mad17]   Vikram Madan. Gluon announcement. `https://aws.amazon.com/de/blogs/machine-learning/introducing-gluon-an-easy-to-use-programming-interface-for-flexible-deep-learning/`, October 2017.

[Mat]   MathWorks. Matlab for Deep Learning. `https://de.mathworks.com/help/nnet/ug/pretrained-convolutional-neural-networks.html`. Accessed: 2018-3-30.

[Mat16]   Mathworks Inc. Simulink User's Guide. Technical Report R2016b, MATLAB & SIMULINK, 2016.

[Mit10]   Christopher Mitchell. *Applications of convolutional neural networks to facial detection and recognition for augmented reality and wearable computing*. PhD thesis, Cooper Union for the Advancement of Science and Art, Albert Nerken School of Engineering, Graduate Division, 2010.

[MPB17]   Marvin Minsky, Seymour A Papert, and Léon Bottou. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[MSN17]   Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials.* Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MxN]   MxNet. Deep Learning in a Single File for Smart Devices. `https://mxnet.incubator.apache.org/faq/smart_device.html`. Accessed: 2018-3-30.

[RH17]   Bernhard Rumpe and Katrin Hölldobler. Monticore 5 language workbench edition 2017. 2017.

[Ros58]   Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[RSD⁺]   Jonathan Revusky, Attila Szegedi, Dániel Dékány, et al. Apache freemarker manual. [Online; accessed 21-April-2018].

[Sak18]   Yusaku Sako. deep-learning-benchmark. `https://github.com/u39kun/deep-learning-benchmark/blob/master/README.md`, February 2018.

[SHK⁺14]   Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 2014.

[SRJ05]   Nathan Srebro, Jason Rennie, and Tommi S Jaakkola. Maximum-margin matrix factorization. In *Advances in neural information processing systems*, 2005.

[Sta]   Stanford Vision Lab, Stanford University, Princeton University. ImageNet Large Scale Visual Recognition Challenge 2012.

[SWXC16]   Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *Cloud Computing and Big Data (CCBD), 2016 7th International Conference on*, pages 99–104. IEEE, 2016.

[SZWZ17]   Yu Sun, Lin Zhu, Guan Wang, and Fang Zhao. Multi-input convolutional neural network for flower grading. *Journal of Electrical and Computer Engineering*, 2017, 2017.

[Tena]   TensorFlow. Api Documentation. `https://www.tensorflow.org/api_docs/`. Accessed: 2018-3-30.

[Tenb]   TensorFlow. TensorBoard: Visualizing Learning. `https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard`. Accessed: 2018-3-30.

[Vog18]   Werner Vogels. MXNet - Deep Learning Framework of Choice at AWS. `https://www.allthingsdistributed.com/2016/11/mxnet-default-framework-deep-learning-aws.html`, November 2018.

[Wil79]   Robin J Wilson. *Introduction to graph theory.* Pearson Education India, 1979.

[XGD+17] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE, 2017.

Appendix A

All Predefined Layers

- **FullyConnected**

  Creates a fully connected layer and applies flatten to the input if necessary. In the following, we will list all parameters of this layer:

  - **units** (integer > 0, required): number of neural units in the output.
  - **no_bias** (boolean, optional, default=false): Whether to disable the bias parameter.

- **Convolution**

  Creates a convolutional layer. Currently, only 2D convolutions are allowed. In the following, we will list all parameters of this layer:

  - **kernel** (integer tuple > 0, required): The kernel size. The first element of the tuple is the height and the second one is the width.
  - **channels** (integer > 0, required): number of output channels and also the number of convolutional filters.
  - **stride** (integer tuple > 0, optional, default=(1,1)): convolution stride: (height, width).
  - **padding** (String, optional, default="same"): One of "valid", "same" or "no_loss". "valid" means no padding. "same" results in padding the input such that the output has the same length as the original input divided by the stride - rounded up. "no_loss" results in minimal padding such that each input is used by at least one filter which is identical to "valid" if the stride is equals to (1,1).
  - **no_bias** (boolean, optional, default=false): Whether to disable the bias parameter.

- **Softmax**

  Applies the softmax activation function to the input. This layer has no parameters.

- **Tanh**

  Applies the tanh activation function to the input. This layer has no parameters.

- **Sigmoid**

  Applies the sigmoid activation function to the input. This layer has no parameters.

- **Relu**

  Applies relu activation function to the input. This layer has no parameters.

- **Flatten**

  Reshapes the input such that height and width are 1. Usually not necessary because the `FullyConnected` layer applies `Flatten` automatically. This layer has no parameters.

- **Dropout**

  Applies dropout operation to input during training and does nothing outside of the training phase. This layer has the following single parameter:

  - **p** ($1 \geq$ float ≥ 0, optional, default=0.5): Fraction of the input that gets dropped out during training time.

- **Pooling**

  Performs pooling on the input. In the following, we will list all parameters of this layer:

  - **pool_type** (String, required): One of "avg" or "max". It determines the pooling type: average or maximum pooling.
  - **kernel** (integer tuple > 0, required): The kernel size. The first element of the tuple denotes the height and the second one the width.
  - **stride** (integer tuple > 0, optional, default=(1,1)): convolution stride: (height, width).
  - **padding** (String, optional, default="same"): One of "valid", "same" or "no_loss". "valid" means no padding. "same" results in padding the input such that the output has the same length as the original input divided by the stride - rounded up. "no_loss" results in minimal padding such that each input is used by at least one filter which is identical to "valid" if the stride is equals to (1,1).

- **GlobalPooling**

  Performs global pooling on the input. This layer has the following single parameter:

  - **pool_type** (String, required): One of "avg" or "max". It determines the pooling type: average or maximum pooling.

- **Lrn**

  Applies local response normalization to the input. In the following, we will list all parameters of this layer:

  - **nsize** (integer > 0, required): normalization window width in elements.
  - **knorm** (float, optional, default=2): The parameter k in the LRN expression.
  - **alpha** (float, optional, default=0.0001): The variance scaling parameter *alpha* in the LRN expression.
  - **beta** (float, optional, default=0.75): The power parameter *beta* in the LRN expression.

89

- **BatchNorm**

  Applies batch normalization to the input. This layer has the following single parameter:

  - **fix_gamma** (boolean, optional, default=true): Fix gamma while training.

- **Concatenate**

  Merges multiple input streams into one output stream by concatenating the channels. The height and width of all inputs must be identical. The number of channels in the output shape is the sum of the number of channels in the shape of the input streams. This layer has no parameters.

- **Add**

  Merges multiple input streams into one output stream by adding them element-wise together. The height, width and the number of channels of all inputs must be identical. The output shape is identical to each input shape. This layer has no parameters.

- **Get**

  `Get(index=i)` is the *selection layer* and can be abbreviated by `[i]`. Selects one out of multiple input streams. The single output is identical to the selected input. The layer has the following single parameter:

  - **index** (integer $>= 0$, required): The zero-based index of the selected input.

- **Split**

  Opposite of `Concatenate`. Handles a single input and splits it into n outputs. The output streams have the same height and width as the input stream and a number of channels which is in general equal to the number of channels of the input divided by n. The last output will have a higher number of channels than the others if the number of input channels is not divisible by n. This layer was added to be able to model the two data streams of the original AlexNet by A. Krizhevsky [KSH12]. This layer has the following single parameter:

  - **n** (integer > 0, required): Determines the number of output streams. It cannot be higher than the number of input channels.

Appendix B

CNN Architectures in EMADL

B.1 ResNeXt-50

```
1  component ResNeXt50<Z(1:oo) channels=3, Z(1:oo) height=224, Z(1:oo) width=224, Z(2:oo) classes=1000>{
2      ports in Z(0:255)^{channels, height, width} image,
3          out Q(0:1)^{classes} predictions;
4
5      implementation CNN {
6          def conv(kernel, channels, stride=1, act=true){
7              Convolution(kernel=(kernel,kernel), channels=channels, stride=(stride,stride)) ->
8              BatchNorm() ->
9              Relu(?=act)
10         }
11         def resGroup(innerChannels, outChannels, stride=1){
12             conv(kernel=1, channels=innerChannels) ->
13             conv(kernel=3, channels=innerChannels, stride=stride) ->
14             conv(kernel=1, channels=outChannels, act=false)
15         }
16         def resLayer(innerChannels, outChannels, stride=1, changedChannels=false){
17             (
18                 resGroup(innerChannels=innerChannels,
19                         outChannels=outChannels,
20                         stride=stride,
21                         | = 32) ->
22                 Add()
23             |
24                 conv(kernel=1, channels=outChannels, stride=stride, act=false, ? = stride!=1 || changedChannels)
25             ) ->
26             Add() ->
27             Relu()
28         }
29
30         image ->
31         conv(kernel=7, channels=64, stride=2) ->
32         Pooling(pool_type="max", kernel=(3,3), stride=(2,2)) ->
33         resLayer(innerChannels=4, outChannels=256, changedChannels=true, -> = 3) ->
34         resLayer(innerChannels=8, outChannels=512, stride=2) ->
35         resLayer(innerChannels=8, outChannels=512, -> = 3) ->
36         resLayer(innerChannels=16, outChannels=1024, stride=2) ->
37         resLayer(innerChannels=16, outChannels=1024, -> = 5) ->
38         resLayer(innerChannels=32, outChannels=2048, stride=2) ->
39         resLayer(innerChannels=32, outChannels=2048, -> = 2) ->
40         GlobalPooling(pool_type="avg") ->
41         FullyConnected(units=classes) ->
42         Softmax() ->
43         predictions
44     }
45 }
```

B.2 AlexNet

```
1  component AlexNet<Z(2:oo) classes=1000>{
2      ports in Z(0:255)^{3, 224, 224} image,
3          out Q(0:1)^{classes} predictions;
4
5      implementation CNN {
6
7          def split1(i){
8              [i] ->
9              Convolution(kernel=(5,5), channels=128) ->
10             Lrn(nsize=5, alpha=0.0001, beta=0.75) ->
11             Pooling(pool_type="max", kernel=(3,3), stride=(2,2), padding="no_loss") ->
```

```
12          Relu()
13        }
14      def split2(i){
15          [i] ->
16          Convolution(kernel=(3,3), channels=192) ->
17          Relu() ->
18          Convolution(kernel=(3,3), channels=128) ->
19          Pooling(pool_type="max", kernel=(3,3), stride=(2,2), padding="no_loss") ->
20          Relu()
21        }
22      def fc(){
23          FullyConnected(units=4096) ->
24          Relu() ->
25          Dropout()
26        }
27
28      image ->
29      Convolution(kernel=(11,11), channels=96, stride=(4,4), padding="no_loss") ->
30      Lrn(nsize=5, alpha=0.0001, beta=0.75) ->
31      Pooling(pool_type="max", kernel=(3,3), stride=(2,2), padding="no_loss") ->
32      Relu() ->
33      Split(n=2) ->
34      split1(i=[0|1]) ->
35      Concatenate() ->
36      Convolution(kernel=(3,3), channels=384) ->
37      Relu() ->
38      Split(n=2) ->
39      split2(i=[0|1]) ->
40      Concatenate() ->
41      fc(->=2) ->
42      FullyConnected(units=classes) ->
43      Softmax() ->
44      predictions
45    }
46 }
```

Appendix C

Templates

CNNCreator.ftl

```
1   import mxnet as mx
2   import logging
3   import os
4   import errno
5   import shutil
6   import h5py
7   import sys
8   import numpy as np
9
10  @mx.init.register
11  class MyConstant(mx.init.Initializer):
12      def __init__(self, value):
13          super(MyConstant, self).__init__(value=value)
14          self.value = value
15      def _init_weight(self, _, arr):
16          arr[:] = mx.nd.array(self.value)
17
18  class ${tc.fileNameWithoutEnding}:
19
20      module = None
21      _data_dir_ = "data/${tc.fullArchitectureName}/"
22      _model_dir_ = "model/${tc.fullArchitectureName}/"
23      _model_prefix_ = "${tc.architectureName}"
24      _input_names_ = [${tc.join(tc.architectureInputs, ",", "'", "'")}]
25      _input_shapes_ = [<#list tc.architecture.inputs as input>${tc.join(input.definition.type.dimensions, ",")}])</
            ↪ #list>]
26      _output_names_ = [${tc.join(tc.architectureOutputs, ",", "'", "_label'")}]
27
28
29      def load(self, context):
30          lastEpoch = 0
31          param_file = None
32
33          try:
34              os.remove(self._model_dir_ + self._model_prefix_ + "_newest-0000.params")
35          except OSError:
36              pass
37          try:
38              os.remove(self._model_dir_ + self._model_prefix_ + "_newest-symbol.json")
39          except OSError:
40              pass
41
42          if os.path.isdir(self._model_dir_):
43              for file in os.listdir(self._model_dir_):
44                  if ".params" in file and self._model_prefix_ in file:
45                      epochStr = file.replace(".params","").replace(self._model_prefix_ + "-","")
46                      epoch = int(epochStr)
47                      if epoch > lastEpoch:
48                          lastEpoch = epoch
49                          param_file = file
50          if param_file is None:
51              return 0
52          else:
53              logging.info("Loading checkpoint: " + param_file)
54              self.module.load(prefix=self._model_dir_ + self._model_prefix_,
55                               epoch=lastEpoch,
56                               data_names=self._input_names_,
57                               label_names=self._output_names_,
58                               context=context)
59              return lastEpoch
60
61
62      def load_data(self, batch_size):
63          train_h5, test_h5 = self.load_h5_files()
```

```
64
65          data_mean = train_h5[self._input_names_[0]][:].mean(axis=0)
66          data_std = train_h5[self._input_names_[0]][:].std(axis=0) + 1e-5
67
68          train_iter = mx.io.NDArrayIter(train_h5[self._input_names_[0]],
69                                         train_h5[self._output_names_[0]],
70                                         batch_size=batch_size,
71                                         data_name=self._input_names_[0],
72                                         label_name=self._output_names_[0])
73          test_iter = None
74          if test_h5 != None:
75              test_iter = mx.io.NDArrayIter(test_h5[self._input_names_[0]],
76                                            test_h5[self._output_names_[0]],
77                                            batch_size=batch_size,
78                                            data_name=self._input_names_[0],
79                                            label_name=self._output_names_[0])
80          return train_iter, test_iter, data_mean, data_std
81
82      def load_h5_files(self):
83          train_h5 = None
84          test_h5 = None
85          train_path = self._data_dir_ + "train.h5"
86          test_path = self._data_dir_ + "test.h5"
87          if os.path.isfile(train_path):
88              train_h5 = h5py.File(train_path, 'r')
89              if not (self._input_names_[0] in train_h5 and self._output_names_[0] in train_h5):
90                  logging.error("The HDF5 file '" + os.path.abspath(train_path) + "' has to contain the datasets: "
91                                + "'" + self._input_names_[0] + "', '" + self._output_names_[0] + "'")
92                  sys.exit(1)
93              test_iter = None
94              if os.path.isfile(test_path):
95                  test_h5 = h5py.File(test_path, 'r')
96                  if not (self._input_names_[0] in test_h5 and self._output_names_[0] in test_h5):
97                      logging.error("The HDF5 file '" + os.path.abspath(test_path) + "' has to contain the datasets:
     ↪          "
98                                    + "'" + self._input_names_[0] + "', '" + self._output_names_[0] + "'")
99                      sys.exit(1)
100             else:
101                 logging.warning("Couldn't load test set. File '" + os.path.abspath(test_path) + "' does not exist
     ↪          .")
102             return train_h5, test_h5
103         else:
104             logging.error("Data loading failure. File '" + os.path.abspath(train_path) + "' does not exist.")
105             sys.exit(1)
106
107
108     def train(self, batch_size,
109               num_epoch=10,
110               optimizer='adam',
111               optimizer_params=(('learning_rate', 0.001),),
112               load_checkpoint=True,
113               context='gpu',
114               checkpoint_period=5,
115               normalize=True):
116         if context == 'gpu':
117             mx_context = mx.gpu()
118         elif context == 'cpu':
119             mx_context = mx.cpu()
120         else:
121             logging.error("Context argument is '" + context + "'. Only 'cpu' and 'gpu are valid arguments'.")
122
123         if 'weight_decay' in optimizer_params:
124             optimizer_params['wd'] = optimizer_params['weight_decay']
125             del optimizer_params['weight_decay']
126         if 'learning_rate_decay' in optimizer_params:
127             min_learning_rate = 1e-08
128             if 'learning_rate_minimum' in optimizer_params:
129                 min_learning_rate = optimizer_params['learning_rate_minimum']
130                 del optimizer_params['learning_rate_minimum']
131             optimizer_params['lr_scheduler'] = mx.lr_scheduler.FactorScheduler(
132                                                 optimizer_params['step_size'],
133                                                 factor=optimizer_params['learning_rate_decay'],
134                                                 stop_factor_lr=min_learning_rate)
135             del optimizer_params['step_size']
136             del optimizer_params['learning_rate_decay']
137
138
139         train_iter, test_iter, data_mean, data_std = self.load_data(batch_size)
140         if self.module == None:
141             if normalize:
142                 self.construct(mx_context, data_mean, data_std)
143             else:
144                 self.construct(mx_context)
145
146         begin_epoch = 0
147         if load_checkpoint:
148             begin_epoch = self.load(mx_context)
149         else:
150             if os.path.isdir(self._model_dir_):
151                 shutil.rmtree(self._model_dir_)
152
153         try:
154             os.makedirs(self._model_dir_)
155         except OSError:
```

```
156            if not os.path.isdir(self._model_dir_):
157                raise
158
159        self.module.fit(
160            train_data=train_iter,
161            eval_data=test_iter,
162            optimizer=optimizer,
163            optimizer_params=optimizer_params,
164            batch_end_callback=mx.callback.Speedometer(batch_size),
165            epoch_end_callback=mx.callback.do_checkpoint(prefix=self._model_dir_ + self._model_prefix_, period=
                ↪ checkpoint_period),
166            begin_epoch=begin_epoch,
167            num_epoch=num_epoch + begin_epoch)
168        self.module.save_checkpoint(self._model_dir_ + self._model_prefix_, num_epoch + begin_epoch)
169        self.module.save_checkpoint(self._model_dir_ + self._model_prefix_ + '_newest', 0)
170
171
172    def construct(self, context, data_mean=None, data_std=None):
173 ${tc.include(tc.architecture.body)}
174        self.module = mx.mod.Module(symbol=mx.symbol.Group([${tc.join(tc.architectureOutputs, ",")}]),
175                                    data_names=self._input_names_,
176                                    label_names=self._output_names_,
177                                    context=context)
```

CNNPredictor.ftl

```
1  #ifndef ${tc.fileNameWithoutEnding?upper_case}
2  #define ${tc.fileNameWithoutEnding?upper_case}
3
4  #include <mxnet/c_predict_api.h>
5
6  #include <cassert>
7  #include <string>
8  #include <vector>
9
10 #include <CNNBufferFile.h>
11
12 class ${tc.fileNameWithoutEnding}{
13 public:
14     const std::string json_file = "model/${tc.fullArchitectureName}/${tc.architectureName}_newest-symbol.json";
15     const std::string param_file = "model/${tc.fullArchitectureName}/${tc.architectureName}_newest-0000.params";
16     const std::vector<std::string> input_keys = {"data"};
17     //const std::vector<std::string> input_keys = {${tc.join(tc.architectureInputs, ",", "\"", "\"")}};
18     const std::vector<std::vector<mx_uint>> input_shapes = {<#list tc.architecture.inputs as input>{1,${tc.join(
            ↪ input.definition.type.dimensions, ",")}}<#if input?has_next>,</#if></#list>};
19     const bool use_gpu = false;
20
21     PredictorHandle handle;
22
23     explicit ${tc.fileNameWithoutEnding}(){
24         init(json_file, param_file, input_keys, input_shapes, use_gpu);
25     }
26
27     ~${tc.fileNameWithoutEnding}(){
28         if(handle) MXPredFree(handle);
29     }
30
31     void predict(${tc.join(tc.architectureInputs, ", ", "const vector<float> &", "")},
32                  ${tc.join(tc.architectureOutputs, ", ", "vector<float> &", "")}){
33 <#list tc.architectureInputs as inputName>
34         MXPredSetInput(handle, "data", ${inputName}.data(), ${inputName}.size());
35         //MXPredSetInput(handle, "${inputName}", ${inputName}.data(), ${inputName}.size());
36 </#list>
37
38         MXPredForward(handle);
39
40         mx_uint output_index;
41         mx_uint *shape = 0;
42         mx_uint shape_len;
43         size_t size;
44
45 <#list tc.architectureOutputs as outputName>
46         output_index = ${outputName?index?c};
47         MXPredGetOutputShape(handle, output_index, &shape, &shape_len);
48         size = 1;
49         for (mx_uint i = 0; i < shape_len; ++i) size *= shape[i];
50         assert(size == ${outputName}.size());
51         MXPredGetOutput(handle, ${outputName?index?c}, &(${outputName}[0]), ${outputName}.size());
52
53 </#list>
54     }
55
56     void init(const std::string &json_file,
57              const std::string &param_file,
58              const std::vector<std::string> &input_keys,
59              const std::vector<std::vector<mx_uint>> &input_shapes,
60              const bool &use_gpu){
61
62         BufferFile json_data(json_file);
63         BufferFile param_data(param_file);
64
65         int dev_type = use_gpu ? 2 : 1;
```

```
66          int dev_id = 0;
67
68          handle = 0;
69
70          if (json_data.GetLength() == 0 ||
71              param_data.GetLength() == 0) {
72              std::exit(-1);
73          }
74
75          const mx_uint num_input_nodes = input_keys.size();
76
77          const char* input_keys_ptr[num_input_nodes];
78          for(mx_uint i = 0; i < num_input_nodes; i++){
79              input_keys_ptr[i] = input_keys[i].c_str();
80          }
81
82          mx_uint shape_data_size = 0;
83          mx_uint input_shape_indptr[input_shapes.size() + 1];
84          input_shape_indptr[0] = 0;
85          for(mx_uint i = 0; i < input_shapes.size(); i++){
86              input_shape_indptr[i+1] = input_shapes[i].size();
87              shape_data_size += input_shapes[i].size();
88          }
89
90          mx_uint input_shape_data[shape_data_size];
91          mx_uint index = 0;
92          for(mx_uint i = 0; i < input_shapes.size(); i++){
93              for(mx_uint j = 0; j < input_shapes[i].size(); j++){
94                  input_shape_data[index] = input_shapes[i][j];
95                  index++;
96              }
97          }
98
99          MXPredCreate((const char*)json_data.GetBuffer(),
100                      (const char*)param_data.GetBuffer(),
101                      static_cast<size_t>(param_data.GetLength()),
102                      dev_type,
103                      dev_id,
104                      num_input_nodes,
105                      input_keys_ptr,
106                      input_shape_indptr,
107                      input_shape_data,
108                      &handle);
109          assert(handle);
110      }
111 };
112
113 #endif // ${tc.fileNameWithoutEnding?upper_case}
```

execute.ftl

```
1  <#list tc.architecture.outputs as output>
2      <#assign shape = output.definition.type.dimensions>
3      vector<float> CNN_${tc.getName(output)}(<#list shape as dim>${dim?c}<#if  dim?has_next>*</#if></#list>);
4  </#list>
5
6      _cnn_.predict(<#list tc.architecture.inputs as input>CNNTranslator::translate(${input.name}<#if input.
            ↪ arrayAccess.isPresent()>[${input.arrayAccess.get().intValue.get()?c}]</#if>),
7              </#list><#list tc.architecture.outputs as output>CNN_${tc.getName(output)}<#if output?has_next>,
8              </#if></#list>);
9
10 <#list tc.architecture.outputs as output>
11 <#assign shape = output.definition.type.dimensions>
12 <#if shape?size == 1>
13     ${output.name}<#if output.arrayAccess.isPresent()>[${output.arrayAccess.get().intValue.get()?c}]</#if> =
            ↪ CNNTranslator::translateToCol(CNN_${tc.getName(output)}, std::vector<size_t> {${shape[0]?c}});
14 </#if>
15 <#if shape?size == 2>
16     ${output.name}<#if output.arrayAccess.isPresent()>[${output.arrayAccess.get().intValue.get()?c}]</#if> =
            ↪ CNNTranslator::translateToMat(CNN_${tc.getName(output)}, std::vector<size_t> {${shape[0]?c}, ${shape
            ↪ [1]?c}});
17 </#if>
18 <#if shape?size == 3>
19     ${output.name}<#if output.arrayAccess.isPresent()>[${output.arrayAccess.get().intValue.get()?c}]</#if> =
            ↪ CNNTranslator::translateToCube(CNN_${tc.getName(output)}, std::vector<size_t> {${shape[0]?c}, ${shape
            ↪ [1]?c}, ${shape[2]?c}});
20 </#if>
21 </#list>
```

CNNTrainer.ftl

```
1 import logging
2 import mxnet as mx
3 <#list instances as instance>
4 import CNNCreator_${instance.fullName?replace(".", "_")}
5 </#list>
6
7 if __name__ == "__main__":
8     logging.basicConfig(level=logging.DEBUG)
```

```
 9      logger = logging.getLogger()
10      handler = logging.FileHandler("train.log","w", encoding=None, delay="true")
11      logger.addHandler(handler)
12
13 <#list instances as instance>
14      ${instance.fullName?replace(".", "_")} = CNNCreator_${instance.fullName?replace(".", "_")}.CNNCreator_${
            ↪ instance.fullName?replace(".", "_")}()
15      ${instance.fullName?replace(".", "_")}.train(
16      <#if (trainParams[instance?index])??>
17          ${trainParams[instance?index]}
18      </#if>
19      )
20
21 </#list>
```

Appendix D

Evaluation

D.1   Generated Code

CNNTrainer_cifar10_Cifar10Classifier.py

```
1  import logging
2  import mxnet as mx
3  import CNNCreator_cifar10_cifar10Classifier_net
4
5  if __name__ == "__main__":
6      logging.basicConfig(level=logging.DEBUG)
7      logger = logging.getLogger()
8      handler = logging.FileHandler("train.log","w", encoding=None, delay="true")
9      logger.addHandler(handler)
10
11     cifar10_cifar10Classifier_net = CNNCreator_cifar10_cifar10Classifier_net.
           ↪ CNNCreator_cifar10_cifar10Classifier_net()
12     cifar10_cifar10Classifier_net.train(
13         batch_size = 64,
14 num_epoch = 10,
15 load_checkpoint = False,
16 context = 'gpu',
17 normalize = True,
18 optimizer = 'adam',
19 optimizer_params = {
20     'weight_decay': 1.0E-4,
21     'learning_rate': 0.01,
22     'learning_rate_decay': 0.8,
23     'step_size': 1000}
24
25
26
27     )
```

CNNCreator_cifar10_cifar10Classifier_net.py

```
1  import mxnet as mx
2  import logging
3  import os
4  import errno
5  import shutil
6  import h5py
7  import sys
8  import numpy as np
9
10 @mx.init.register
11 class MyConstant(mx.init.Initializer):
12     def __init__(self, value):
13         super(MyConstant, self).__init__(value=value)
14         self.value = value
15     def _init_weight(self, _, arr):
16         arr[:] = mx.nd.array(self.value)
17
18 class CNNCreator_cifar10_cifar10Classifier_net:
19
20     module = None
21     _data_dir_ = "data/cifar10_cifar10Classifier_net/"
22     _model_dir_ = "model/cifar10_cifar10Classifier_net/"
23     _model_prefix_ = "net"
24     _input_names_ = ['data']
25     _input_shapes_ = [(3,32,32)]
```

```python
26        _output_names_ = ['softmax_label']
27
28
29    def load(self, context):
30        lastEpoch = 0
31        param_file = None
32
33        try:
34            os.remove(self._model_dir_ + self._model_prefix_ + "_newest-0000.params")
35        except OSError:
36            pass
37        try:
38            os.remove(self._model_dir_ + self._model_prefix_ + "_newest-symbol.json")
39        except OSError:
40            pass
41
42        if os.path.isdir(self._model_dir_):
43            for file in os.listdir(self._model_dir_):
44                if ".params" in file and self._model_prefix_ in file:
45                    epochStr = file.replace(".params","").replace(self._model_prefix_ + "-","")
46                    epoch = int(epochStr)
47                    if epoch > lastEpoch:
48                        lastEpoch = epoch
49                        param_file = file
50            if param_file is None:
51                return 0
52            else:
53                logging.info("Loading checkpoint: " + param_file)
54                self.module.load(prefix=self._model_dir_ + self._model_prefix_,
55                                 epoch=lastEpoch,
56                                 data_names=self._input_names_,
57                                 label_names=self._output_names_,
58                                 context=context)
59                return lastEpoch
60
61
62    def load_data(self, batch_size):
63        train_h5, test_h5 = self.load_h5_files()
64
65        data_mean = train_h5[self._input_names_[0]][:].mean(axis=0)
66        data_std = train_h5[self._input_names_[0]][:].std(axis=0) + 1e-5
67
68        train_iter = mx.io.NDArrayIter(train_h5[self._input_names_[0]],
69                                       train_h5[self._output_names_[0]],
70                                       batch_size=batch_size,
71                                       data_name=self._input_names_[0],
72                                       label_name=self._output_names_[0])
73        test_iter = None
74        if test_h5 != None:
75            test_iter = mx.io.NDArrayIter(test_h5[self._input_names_[0]],
76                                          test_h5[self._output_names_[0]],
77                                          batch_size=batch_size,
78                                          data_name=self._input_names_[0],
79                                          label_name=self._output_names_[0])
80        return train_iter, test_iter, data_mean, data_std
81
82    def load_h5_files(self):
83        train_h5 = None
84        test_h5 = None
85        train_path = self._data_dir_ + "train.h5"
86        test_path = self._data_dir_ + "test.h5"
87        if os.path.isfile(train_path):
88            train_h5 = h5py.File(train_path, 'r')
89            if not (self._input_names_[0] in train_h5 and self._output_names_[0] in train_h5):
90                logging.error("The HDF5 file '" + os.path.abspath(train_path) + "' has to contain the datasets: "
91                              + "'" + self._input_names_[0] + "', '" + self._output_names_[0] + "'")
92                sys.exit(1)
93            test_iter = None
94            if os.path.isfile(test_path):
95                test_h5 = h5py.File(test_path, 'r')
96                if not (self._input_names_[0] in test_h5 and self._output_names_[0] in test_h5):
97                    logging.error("The HDF5 file '" + os.path.abspath(test_path) + "' has to contain the datasets:
      ↪ "
98                                  + "'" + self._input_names_[0] + "', '" + self._output_names_[0] + "'")
99                    sys.exit(1)
100           else:
101               logging.warning("Couldn't load test set. File '" + os.path.abspath(test_path) + "' does not exist.
      ↪ ")
102           return train_h5, test_h5
103       else:
104           logging.error("Data loading failure. File '" + os.path.abspath(train_path) + "' does not exist.")
105           sys.exit(1)
106
107
108   def train(self, batch_size,
109             num_epoch=10,
110             optimizer='adam',
111             optimizer_params=(('learning_rate', 0.001),),
112             load_checkpoint=True,
113             context=mx.gpu(),
114             checkpoint_period=5,
115             normalize=True):
116
117       if 'weight_decay' in optimizer_params:
```

```
118              optimizer_params['wd'] = optimizer_params['weight_decay']
119              del optimizer_params['weight_decay']
120          if 'learning_rate_decay' in optimizer_params:
121              min_learning_rate = 1e-08
122              if 'learning_rate_minimum' in optimizer_params:
123                  min_learning_rate = optimizer_params['learning_rate_minimum']
124                  del optimizer_params['learning_rate_minimum']
125              optimizer_params['lr_scheduler'] = mx.lr_scheduler.FactorScheduler(
126                                                  optimizer_params['step_size'],
127                                                  factor=optimizer_params['learning_rate_decay'],
128                                                  stop_factor_lr=min_learning_rate)
129              del optimizer_params['step_size']
130              del optimizer_params['learning_rate_decay']
131
132
133          train_iter, test_iter, data_mean, data_std = self.load_data(batch_size)
134          if self.module == None:
135              if normalize:
136                  self.construct(context, data_mean, data_std)
137              else:
138                  self.construct(context)
139
140          begin_epoch = 0
141          if load_checkpoint:
142              begin_epoch = self.load(context)
143          else:
144              if os.path.isdir(self._model_dir_):
145                  shutil.rmtree(self._model_dir_)
146
147          try:
148              os.makedirs(self._model_dir_)
149          except OSError:
150              if not os.path.isdir(self._model_dir_):
151                  raise
152
153          self.module.fit(
154              train_data=train_iter,
155              eval_data=test_iter,
156              optimizer=optimizer,
157              optimizer_params=optimizer_params,
158              batch_end_callback=mx.callback.Speedometer(batch_size),
159              epoch_end_callback=mx.callback.do_checkpoint(prefix=self._model_dir_ + self._model_prefix_, period=
                  ↪ checkpoint_period),
160              begin_epoch=begin_epoch,
161              num_epoch=num_epoch + begin_epoch)
162          self.module.save_checkpoint(self._model_dir_ + self._model_prefix_, num_epoch + begin_epoch)
163          self.module.save_checkpoint(self._model_dir_ + self._model_prefix_ + '_newest', 0)
164
165
166      def construct(self, context, data_mean=None, data_std=None):
167          data = mx.sym.var("data",
168              shape=(0,3,32,32))
169          # data, output shape: {[3,32,32]}
170
171          if not data_mean is None:
172              assert(not data_std is None)
173              _data_mean_ = mx.sym.Variable("_data_mean_", shape=(3,32,32), init=MyConstant(value=data_mean.tolist()
                  ↪ ))
174              _data_mean_ = mx.sym.BlockGrad(_data_mean_)
175              _data_std_ = mx.sym.Variable("_data_std_", shape=(3,32,32), init=MyConstant(value=data_mean.tolist()))
176              _data_std_ = mx.sym.BlockGrad(_data_std_)
177              data = mx.symbol.broadcast_sub(data, _data_mean_)
178              data = mx.symbol.broadcast_div(data, _data_std_)
179          conv2_1_ = mx.symbol.pad(data=data,
180              mode='constant',
181              pad_width=(0,0,0,0,1,1,1,1),
182              constant_value=0)
183          conv2_1_ = mx.symbol.Convolution(data=conv2_1_,
184              kernel=(3,3),
185              stride=(1,1),
186              num_filter=8,
187              no_bias=False,
188              name="conv2_1_")
189          # conv2_1_, output shape: {[8,32,32]}
190
191          batchnorm2_1_ = mx.symbol.BatchNorm(data=conv2_1_,
192              fix_gamma=True,
193              name="batchnorm2_1_")
194          relu2_1_ = mx.symbol.Activation(data=batchnorm2_1_,
195              act_type='relu',
196              name="relu2_1_")
197          conv3_1_ = mx.symbol.pad(data=relu2_1_,
198              mode='constant',
199              pad_width=(0,0,0,0,1,1,1,1),
200              constant_value=0)
201          conv3_1_ = mx.symbol.Convolution(data=conv3_1_,
202              kernel=(3,3),
203              stride=(1,1),
204              num_filter=8,
205              no_bias=False,
206              name="conv3_1_")
207          # conv3_1_, output shape: {[8,32,32]}
208
209          batchnorm3_1_ = mx.symbol.BatchNorm(data=conv3_1_,
```

```
210            fix_gamma=True,
211            name="batchnorm3_1_")
212    conv2_2_ = mx.symbol.Convolution(data=data,
213        kernel=(1,1),
214        stride=(1,1),
215        num_filter=8,
216        no_bias=False,
217        name="conv2_2_")
218    # conv2_2_, output shape: {[8,32,32]}
219
220    batchnorm2_2_ = mx.symbol.BatchNorm(data=conv2_2_,
221        fix_gamma=True,
222        name="batchnorm2_2_")
223    add4_ = batchnorm3_1_ + batchnorm2_2_
224    # add4_, output shape: {[8,32,32]}
225
226    relu4_ = mx.symbol.Activation(data=add4_,
227        act_type='relu',
228        name="relu4_")
229    conv5_1_ = mx.symbol.pad(data=relu4_,
230        mode='constant',
231        pad_width=(0,0,0,0,1,0,1,0),
232        constant_value=0)
233    conv5_1_ = mx.symbol.Convolution(data=conv5_1_,
234        kernel=(3,3),
235        stride=(2,2),
236        num_filter=16,
237        no_bias=False,
238        name="conv5_1_")
239    # conv5_1_, output shape: {[16,16,16]}
240
241    batchnorm5_1_ = mx.symbol.BatchNorm(data=conv5_1_,
242        fix_gamma=True,
243        name="batchnorm5_1_")
244    relu5_1_ = mx.symbol.Activation(data=batchnorm5_1_,
245        act_type='relu',
246        name="relu5_1_")
247    conv6_1_ = mx.symbol.pad(data=relu5_1_,
248        mode='constant',
249        pad_width=(0,0,0,0,1,1,1,1),
250        constant_value=0)
251    conv6_1_ = mx.symbol.Convolution(data=conv6_1_,
252        kernel=(3,3),
253        stride=(1,1),
254        num_filter=16,
255        no_bias=False,
256        name="conv6_1_")
257    # conv6_1_, output shape: {[16,16,16]}
258
259    batchnorm6_1_ = mx.symbol.BatchNorm(data=conv6_1_,
260        fix_gamma=True,
261        name="batchnorm6_1_")
262    conv5_2_ = mx.symbol.Convolution(data=relu4_,
263        kernel=(1,1),
264        stride=(2,2),
265        num_filter=16,
266        no_bias=False,
267        name="conv5_2_")
268    # conv5_2_, output shape: {[16,16,16]}
269
270    batchnorm5_2_ = mx.symbol.BatchNorm(data=conv5_2_,
271        fix_gamma=True,
272        name="batchnorm5_2_")
273    add7_ = batchnorm6_1_ + batchnorm5_2_
274    # add7_, output shape: {[16,16,16]}
275
276    relu7_ = mx.symbol.Activation(data=add7_,
277        act_type='relu',
278        name="relu7_")
279    #This method is 5 pages long.
280    #We shortened it to not waste paper.
281
282    conv29_1_ = mx.symbol.pad(data=relu28_,
283        mode='constant',
284        pad_width=(0,0,0,0,1,1,1,1),
285        constant_value=0)
286    conv29_1_ = mx.symbol.Convolution(data=conv29_1_,
287        kernel=(3,3),
288        stride=(1,1),
289        num_filter=64,
290        no_bias=False,
291        name="conv29_1_")
292    # conv29_1_, output shape: {[64,4,4]}
293
294    batchnorm29_1_ = mx.symbol.BatchNorm(data=conv29_1_,
295        fix_gamma=True,
296        name="batchnorm29_1_")
297    relu29_1_ = mx.symbol.Activation(data=batchnorm29_1_,
298        act_type='relu',
299        name="relu29_1_")
300    conv30_1_ = mx.symbol.pad(data=relu29_1_,
301        mode='constant',
302        pad_width=(0,0,0,0,1,1,1,1),
303        constant_value=0)
```

```
304        conv30_1_ = mx.symbol.Convolution(data=conv30_1_,
305            kernel=(3,3),
306            stride=(1,1),
307            num_filter=64,
308            no_bias=False,
309            name="conv30_1_")
310        # conv30_1_, output shape: {[64,4,4]}
311
312        batchnorm30_1_ = mx.symbol.BatchNorm(data=conv30_1_,
313            fix_gamma=True,
314            name="batchnorm30_1_")
315        add31_ = batchnorm30_1_ + relu28_
316        # add31_, output shape: {[64,4,4]}
317
318        relu31_ = mx.symbol.Activation(data=add31_,
319            act_type='relu',
320            name="relu31_")
321        globalpooling31_ = mx.symbol.Pooling(data=relu31_,
322            global_pool=True,
323            kernel=(1,1),
324            pool_type="avg",
325            name="globalpooling31_")
326        # globalpooling31_, output shape: {[64,1,1]}
327
328        fc31_ = mx.symbol.FullyConnected(data=globalpooling31_,
329            num_hidden=128,
330            no_bias=False,
331            name="fc31_")
332        dropout31_ = mx.symbol.Dropout(data=fc31_,
333            p=0.5,
334            name="dropout31_")
335        fc32_ = mx.symbol.FullyConnected(data=dropout31_,
336            num_hidden=10,
337            no_bias=False,
338            name="fc32_")
339        softmax = mx.symbol.SoftmaxOutput(data=fc32_,
340            name="softmax")
341
342        self.module = mx.mod.Module(symbol=mx.symbol.Group([softmax]),
343                                    data_names=self._input_names_,
344                                    label_names=self._output_names_,
345                                    context=context)
```

cifar10_cifar10Classifier.h

```
1  #ifndef CIFAR10_CIFAR10CLASSIFIER
2  #define CIFAR10_CIFAR10CLASSIFIER
3  #ifndef M_PI
4  #define M_PI 3.14159265358979323846
5  #endif
6  #include "armadillo"
7  #include "cifar10_cifar10Classifier_net.h"
8  #include "cifar10_cifar10Classifier_calculateClass.h"
9  using namespace arma;
10 class cifar10_cifar10Classifier{
11 public:
12 cube image;
13 double classIndex;
14 cifar10_cifar10Classifier_net net;
15 cifar10_cifar10Classifier_calculateClass calculateClass;
16 void init()
17 {
18 image = cube(3, 32, 32);
19 net.init();
20 calculateClass.init();
21 }
22 void execute()
23 {
24 net.data = image;
25 net.execute();
26 calculateClass.inputVector = net.softmax;
27 calculateClass.execute();
28 classIndex = calculateClass.maxIndex;
29 }
30
31 };
32 #endif
```

cifar10_cifar10Classifier_calculateClass.h

```
1  #ifndef CIFAR10_CIFAR10CLASSIFIER_CALCULATECLASS
2  #define CIFAR10_CIFAR10CLASSIFIER_CALCULATECLASS
3  #ifndef M_PI
4  #define M_PI 3.14159265358979323846
5  #endif
6  #include "armadillo"
7  using namespace arma;
8  class cifar10_cifar10Classifier_calculateClass{
9  const int n = 10;
```

```
10   public:
11   colvec inputVector;
12   double maxIndex;
13   void init()
14   {
15   inputVector=colvec(n);
16   }
17   void execute()
18   {
19   maxIndex = 0;
20   double maxValue = inputVector(0);
21   for( auto i=1;i<=(n-1);++i){
22   if((inputVector(i) > maxValue)){
23   maxIndex = i;
24   maxValue = inputVector(i);
25   }
26   }
27   }
28
29   };
30   #endif
```

cifar10_cifar10Classifier_net.h

```
1    #ifndef CIFAR10_CIFAR10CLASSIFIER_NET
2    #define CIFAR10_CIFAR10CLASSIFIER_NET
3    #ifndef M_PI
4    #define M_PI 3.14159265358979323846
5    #endif
6    #include "armadillo"
7    #include "CNNPredictor_cifar10_cifar10Classifier_net.h"
8    #include "CNNTranslator.h"
9    using namespace arma;
10   class cifar10_cifar10Classifier_net{
11   const int classes = 10;
12   public:
13   CNNPredictor_cifar10_cifar10Classifier_net _cnn_;
14   cube data;
15   colvec softmax;
16   void init()
17   {
18   data = cube(3, 32, 32);
19   softmax=colvec(classes);
20   }
21   void execute(){
22       vector<float> CNN_softmax(10);
23
24       _cnn_.predict(CNNTranslator::translate(data),
25                   CNN_softmax);
26
27       softmax = CNNTranslator::translateToCol(CNN_softmax, std::vector<size_t> {10});
28
29   }
30
31   };
32   #endif
```

CNNPredictor_cifar10_cifar10Classifier_net.h

```
1    #ifndef CNNPREDICTOR_CIFAR10_CIFAR10CLASSIFIER_NET
2    #define CNNPREDICTOR_CIFAR10_CIFAR10CLASSIFIER_NET
3
4    #include <mxnet/c_predict_api.h>
5
6    #include <cassert>
7    #include <string>
8    #include <vector>
9
10   #include <CNNBufferFile.h>
11
12   class CNNPredictor_cifar10_cifar10Classifier_net{
13   public:
14       const std::string json_file = "model/cifar10_cifar10Classifier_net/net_newest-symbol.json";
15       const std::string param_file = "model/cifar10_cifar10Classifier_net/net_newest-0000.params";
16       const std::vector<std::string> input_keys = {"data"};
17       //const std::vector<std::string> input_keys = {"data"};
18       const std::vector<std::vector<mx_uint>> input_shapes = {{1,3,32,32}};
19       const bool use_gpu = false;
20
21       PredictorHandle handle;
22
23       explicit CNNPredictor_cifar10_cifar10Classifier_net(){
24           init(json_file, param_file, input_keys, input_shapes, use_gpu);
25       }
26
27       ~CNNPredictor_cifar10_cifar10Classifier_net(){
28           if(handle) MXPredFree(handle);
29       }
30
```

```cpp
31      void predict(const vector<float> &data,
32                   vector<float> &softmax){
33          MXPredSetInput(handle, "data", data.data(), data.size());
34          //MXPredSetInput(handle, "data", data.data(), data.size());
35
36          MXPredForward(handle);
37
38          mx_uint output_index;
39          mx_uint *shape = 0;
40          mx_uint shape_len;
41          size_t size;
42
43          output_index = 0;
44          MXPredGetOutputShape(handle, output_index, &shape, &shape_len);
45          size = 1;
46          for (mx_uint i = 0; i < shape_len; ++i) size *= shape[i];
47          assert(size == softmax.size());
48          MXPredGetOutput(handle, 0, &(softmax[0]), softmax.size());
49
50      }
51
52      void init(const std::string &json_file,
53               const std::string &param_file,
54               const std::vector<std::string> &input_keys,
55               const std::vector<std::vector<mx_uint>> &input_shapes,
56               const bool &use_gpu){
57
58          BufferFile json_data(json_file);
59          BufferFile param_data(param_file);
60
61          int dev_type = use_gpu ? 2 : 1;
62          int dev_id = 0;
63
64          handle = 0;
65
66          if (json_data.GetLength() == 0 ||
67              param_data.GetLength() == 0) {
68              std::exit(-1);
69          }
70
71          const mx_uint num_input_nodes = input_keys.size();
72
73          const char* input_keys_ptr[num_input_nodes];
74          for(mx_uint i = 0; i < num_input_nodes; i++){
75              input_keys_ptr[i] = input_keys[i].c_str();
76          }
77
78          mx_uint shape_data_size = 0;
79          mx_uint input_shape_indptr[input_shapes.size() + 1];
80          input_shape_indptr[0] = 0;
81          for(mx_uint i = 0; i < input_shapes.size(); i++){
82              input_shape_indptr[i+1] = input_shapes[i].size();
83              shape_data_size += input_shapes[i].size();
84          }
85
86          mx_uint input_shape_data[shape_data_size];
87          mx_uint index = 0;
88          for(mx_uint i = 0; i < input_shapes.size(); i++){
89              for(mx_uint j = 0; j < input_shapes[i].size(); j++){
90                  input_shape_data[index] = input_shapes[i][j];
91                  index++;
92              }
93          }
94
95          MXPredCreate((const char*)json_data.GetBuffer(),
96                      (const char*)param_data.GetBuffer(),
97                      static_cast<size_t>(param_data.GetLength()),
98                      dev_type,
99                      dev_id,
100                     num_input_nodes,
101                     input_keys_ptr,
102                     input_shape_indptr,
103                     input_shape_data,
104                     &handle);
105         assert(handle);
106     }
107 };
108
109 #endif // CNNPREDICTOR_CIFAR10_CIFAR10CLASSIFIER_NET
```

CNNTranslator.h

```cpp
1  #ifndef CNNTRANSLATOR_H
2  #define CNNTRANSLATOR_H
3  #include <armadillo>
4  #include <cassert>
5
6  using namespace std;
7  using namespace arma;
8
9  class CNNTranslator{
10 public:
```

```
11      template<typename T> static void addColToSTDVector(const Col<T> &source, vector<float> &data){
12          for(size_t i = 0; i < source.n_elem; i++){
13              data.push_back((float) source(i));
14          }
15      }
16
17      template<typename T> static void addRowToSTDVector(const subview_row<T> &source, vector<float> &data){
18          for(size_t i = 0; i < source.n_elem; i++){
19              data.push_back((float) source(i));
20          }
21      }
22
23      template<typename T> static void addRowToSTDVector(const Row<T> &source, vector<float> &data){
24          for(size_t i = 0; i < source.n_elem; i++){
25              data.push_back((float) source(i));
26          }
27      }
28
29      template<typename T> static void addMatToSTDVector(const Mat<T> &source, vector<float> &data){
30          for(size_t i = 0; i < source.n_rows; i++){
31              addRowToSTDVector(source.row(i), data);
32          }
33      }
34
35
36      template<typename T> static vector<float> translate(const Col<T> &source){
37          size_t size = source.n_elem;
38          vector<float> data;
39          data.reserve(size);
40          addColToSTDVector(source, data);
41          return data;
42      }
43
44      template<typename T> static vector<float> translate(const Row<T> &source){
45          size_t size = source.n_elem;
46          vector<float> data;
47          data.reserve(size);
48          addRowToSTDVector(source, data);
49          return data;
50      }
51
52      template<typename T> static vector<float> translate(const Mat<T> &source){
53          size_t size = source.n_elem;
54          vector<float> data;
55          data.reserve(size);
56          addMatToSTDVector(source, data);
57          return data;
58      }
59
60      template<typename T> static vector<float> translate(const Cube<T> &source){
61          size_t size = source.n_elem;
62          vector<float> data;
63          data.reserve(size);
64          for(size_t i = 0; i < source.n_slices; i++){
65              addMatToSTDVector(source.slice(i), data);
66          }
67          return data;
68      }
69
70      static vec translateToCol(const vector<float> &source, const vector<size_t> &shape){
71          assert(shape.size() == 1);
72          vec column(shape[0]);
73          for(size_t i = 0; i < source.size(); i++){
74              column(i) = (double) source[i];
75          }
76          return column;
77      }
78
79      static mat translateToMat(const vector<float> &source, const vector<size_t> &shape){
80          assert(shape.size() == 2);
81          mat matrix(shape[1], shape[0]); //create transposed version of the matrix
82          int startPos = 0;
83          int endPos = matrix.n_rows;
84          const vector<size_t> columnShape = {matrix.n_rows};
85          for(size_t i = 0; i < matrix.n_cols; i++){
86              vector<float> colSource(&source[startPos], &source[endPos]);
87              matrix.col(i) = translateToCol(colSource, columnShape);
88              startPos = endPos;
89              endPos += matrix.n_rows;
90          }
91          return matrix.t();
92      }
93
94      static cube translateToCube(const vector<float> &source, const vector<size_t> &shape){
95          assert(shape.size() == 3);
96          cube cubeMatrix(shape[1], shape[2], shape[0]);
97          const int matrixSize = shape[1] * shape[2];
98          const vector<size_t> matrixShape = {shape[1], shape[2]};
99          int startPos = 0;
100         int endPos = matrixSize;
101         for(size_t i = 0; i < cubeMatrix.n_slices; i++){
102             vector<float> matrixSource(&source[startPos], &source[endPos]);
103             cubeMatrix.slice(i) = translateToMat(matrixSource, matrixShape);
104             startPos = endPos;
```

```
105         endPos += matrixSize;
106     }
107     return cubeMatrix;
108 }
109
110 template<typename T> static vector<size_t> getShape(const Col<T> &source){
111     return {source.n_elem};
112 }
113
114 template<typename T> static vector<size_t> getShape(const Row<T> &source){
115     return {source.n_elem};
116 }
117
118 template<typename T> static vector<size_t> getShape(const Mat<T> &source){
119     return {source.n_rows, source.n_cols};
120 }
121
122 template<typename T> static vector<size_t> getShape(const Cube<T> &source){
123     return {source.n_slices, source.n_rows, source.n_cols};
124 }
125 };
126
127 #endif
```

BufferFile.h

```
1 #ifndef CNNBUFFERFILE_H
2 #define CNNBUFFERFILE_H
3
4 #include <stdio.h>
5 #include <iostream>
6 #include <fstream>
7
8 // Read file to buffer
9 class BufferFile {
10 public :
11     std::string file_path_;
12     int length_;
13     char* buffer_;
14
15     explicit BufferFile(std::string file_path)
16     :file_path_(file_path) {
17
18         std::ifstream ifs(file_path.c_str(), std::ios::in | std::ios::binary);
19         if (!ifs) {
20             std::cerr << "Can't open the file. Please check " << file_path << ". \n";
21             length_ = 0;
22             buffer_ = NULL;
23             return;
24         }
25
26         ifs.seekg(0, std::ios::end);
27         length_ = ifs.tellg();
28         ifs.seekg(0, std::ios::beg);
29         std::cout << file_path.c_str() << " ... "<< length_ << " bytes\n";
30
31         buffer_ = new char[sizeof(char) * length_];
32         ifs.read(buffer_, length_);
33         ifs.close();
34     }
35
36     int GetLength() {
37         return length_;
38     }
39     char* GetBuffer() {
40         return buffer_;
41     }
42
43     ~BufferFile() {
44         if (buffer_) {
45           delete[] buffer_;
46           buffer_ = NULL;
47         }
48     }
49 };
50
51 #endif // CNNBUFFERFILE_H
```

D.2   Additional Files

Main Program

```
1 #include <armadillo>
2 #include <iomanip>
3 #include <iostream>
4 #include "CNNTranslator.h"
```

```cpp
#include "cifar10_cifar10Classifier.h"
#include <opencv2/highgui/highgui.hpp>
int main(int argc, char *argv[]){
    if(argc < 2){
        std::cout << "Missing Argument: An argument has to denote the path to the tested image " << std::endl;
        exit(1);
    }
    std::string filePath = argv[1];

    ifstream file(filePath);
    if (file.is_open()){
        file.close();
    }
    else{
        std::cout << "File '" << filePath << "' does not exist or couldn't be opened" << std::endl;
        exit(1);
    }
    cv::Mat img = cv::imread(filePath);

    size_t channels = 3;
    size_t height = img.rows;
    size_t width = img.cols;
    vector<float> data(channels*height*width);
    for(size_t j=0; j<height; j++){
        for(size_t k=0; k<width; k++){
            cv::Vec3b intensity = img.at<cv::Vec3b>(j, k);
            for(size_t i=0; i<channels; i++){
                data[i*height*width + j*height + k] = (float) intensity[i];
            }
        }
    }

    cifar10_cifar10Classifier classifier;
    classifier.init();
    classifier.image = CNNTranslator::translateToCube(data, vector<size_t> {channels,height,width});
    classifier.execute();
    std::cout << std::endl;
    std::cout << "predicted class index: " << std::to_string((int)classifier.classIndex) << std::endl;
    return 0;
}
```

Qt Creator Project File (Cifar10Test.pro)

```
CONFIG += c++11

TARGET = Cifar10Test

SOURCES += main.cpp

INCLUDEPATH += $$PWD/../../incubator-mxnet/include

LIBS += -L$$PWD/../../incubator-mxnet/lib/ -lmxnet
LIBS += -llapack -lblas -larmadillo

HEADERS += \
    cifar10_cifar10Classifier.h \
    cifar10_cifar10Classifier_calculateClass.h \
    cifar10_cifar10Classifier_net.h \
    CNNBufferFile.h \
    CNNPredictor_cifar10_cifar10Classifier_net.h \
    CNNTranslator.h

unix: CONFIG += link_pkgconfig
unix: PKGCONFIG += opencv
```