PART II

Test Case Functions

6

Deep Learning for Advanced Driver Assistance Systems

Florian Giesemann[1], Guillermo Payá-Vayá[1], Holger Blume[1], Matthias Limmer[2] and Werner R. Ritter[2]

[1]Institute of Microelectronic Systems, Leibniz Universität Hannover, Hannover, Germany
[2]Vision Enhancement, Daimler AG, Germany

6.1 Introduction

Today, vehicles contain a wide range of electronic driver assistance systems. These systems, for example *Anti-lock Braking System* (ABS) or *Electronic Stability Control* (ESC), increase car safety and on a more general level even road safety. More complex *Advanced Driver Assistance Systems* (ADAS), like *Lane Departure Warning*, *Overtaking Assistant*, *Collision Warning* or *Emergency Breaking* do not only observe the parameters of the vehicle itself, but also require information regarding the environment. Future applications, which target autonomous driving, need an even more detailed understanding of the vehicle's environment and the current driving situation. Therefore, vehicles are equipped with a number of sensors, which enable the perception of the vehicle's surroundings including other road users. But the sensors generaly used deliver a huge amount of raw and unrefined data, from which the necessary information needs to be extracted. For instance, for camera sensors, an algorithm called *Scene Labeling* can be used to detect relevant objects in camera images. It assigns every pixel of an input image to a semantic class (e.g., road, car, free space etc.) and can therefore be used to extract detailed information from the scene.

The increasing complexity of algorithms and the increasing amount of data that has to be processed requires a high amount of processing power. At the same time, processing hardware is subject to restrictions regarding power

consumption and size. These conditions make the field of embedded hardware platforms for driver assistance systems challenging.

This chapter is organized as follows: Section 6.2 gives an introduction to Scene Labeling techniques and their application in Advanced Driver Assistance Systems. Section 6.3 explains the concepts of Convolutional Neural Networks and Deep Learning. In Section 6.4, an exemplary CNN is presented and evaluated. Section 6.5 describes different hardware platforms for Scene Labeling. Finally, Section 6.6 summarizes the chapter.

6.2 Scene Labeling in Advanced Driver Assistance Systems

Getting a thorough understanding of the vehicle's environment is an important step in the development of advanced driver assistance systems. Different techniques for detection and classification of objects have been developed. Literature offers a wide range of algorithms for detecting traffic signs, traffic lights, driving lanes, and also other vehicles and pedestrians. In order to build up a comprehensive understanding of the environment, not only single objects have to be detected, but also the objects in relation to each other have to be determined. This is commonly referred to as *Scene Labeling*.

Scene Labeling is a technique to classify images on different levels of detail. Image-level Scene Labeling (e.g., [1]) is used to derive one or more labels for the whole image that describe different scene types, e.g., urban, inter-urban, or highway. On another level, labels are deduced for small sub regions of an image, so called *regions of interest*. This allows for a more detailed understanding of the scene in terms of objects, like pedestrians, vehicles, driving lanes, traffic signs and so on. On a third level of detail, each pixel in an input image is classified and provided with a semantic label. The information provided by these labels can be used in different applications, for example in pedestrian/obstacle detection, close range lane course estimation or relative map positioning.

Scene Labeling can also be combined with other detection methods in order to increase reliability and thereby increase the integrity level of safety functions. Moreover, it can replace different detection modules in order to save resources.

The Scene Labeling task is usually performed in two steps. The first step extracts features from the input image; the second step computes a classification of the image, the region, or the pixels from the extracted features.

Several different features are used in order to perform image segmentation and semantic labeling. Some algorithms rely on single, low-level features, like color [2], texture [3, 4], shape [3, 5], geometry [6], and edge features [7]. Object detection algorithms are used to extract high-level features, e.g., pedestrian detection [8], traffic sign detection [9], and lane detection [10]. Some algorithms perform labeling using image segmentation techniques, e.g., Super Pixels [11] or sliding windows using Boosting [12] to detect regions of one certain class, e.g., pedestrians or traffic signs.

Classification of extracted features is performed using different techniques, like Support Vector Machines [13], Genetic Algorithms [7], or Neural Networks [14]. Probabilistic models like Conditional Random Fields (CRF) [15] and graph-based optimization methods (e.g., Graph Cut [16]) are used to combine different features and include smoothness constraints or neighbor relationships.

Recent advances in the field of deep learning and neural networks yielded a new technique for the scene labeling problem, which is described in the next section.

6.3 Convolutional Neural Networks and Deep Learning

Typical systems for detection and recognition of objects or situations use a two-step data processing scheme. In a first step, features are computed from data gathered through different sensors, like cameras, radar, etc. Then, a second step uses the previously computed features in order to classify the candidates into the object classes. The implementation of the classification step might involve the use of machine learning techniques, i.e., the training of a classifier. One difficulty in this scenario is the selection of features to be used. Often, these features are hand-crafted and a lot of work might be involved in tuning the parameters in order to find a set of features that can be used for reliable detection and recognition of objects.

Another way of building recognition systems that evolved recently is the use of learning techniques and especially the technique of *deep learning* with close coupling between the feature extraction and feature classification steps. Deep learning describes methods, in which feature extractors are not hand-crafted but automatically learned from a set of training data. Multiple layers of feature extractors can be used in a hierarchical structure in order to allow deeper layers to extract features of higher order from previous layers. The idea behind this technique is that the learning algorithm is capable of detecting the best features for the following classification step itself. Commonly

used implementations of the deep learning methodology are *artificial neural networks*.

6.3.1 Introduction to Neural Networks

Inspired by processes in the biological neural networks of the central nervous systems and especially the brain, different computational models of artificial neural networks have been developed [17]. Artificial neural networks are built as a collection of relatively simple units, so called neurons, that are connected together to form a network which can process a complicated task. One of the first models of neural networks is called *perceptron* [18]. The simple perceptron neurons perform binary decisions depending on their input values. The input signals x_i are weighted and accumulated. The neuron "fires", i.e., produces an output signal y of 1, if the weighted sum of the input signal exceeds a given threshold value, and outputs 0 otherwise. The first networks had one single layer of neurons and were only capable of computing linear classifications. More complex networks with multiple layers were capable of computing more complex classifications. Nowadays, neural networks use a different model for the artificial neurons [19, 20], as depicted in Figure 6.1. The input values, which are now real numbered values, are weighted and accumulated. Afterwards, a non-linear activation function is applied to the sum. Commonly used activation functions are the *sigmoid function*, which can be interpreted as a smoothed threshold. Recently, *rectifier linear units (ReLU)* have been reported to have several advantages over the sigmoid functions [21]. Some exemplary activation functions are shown in Figure 6.2.

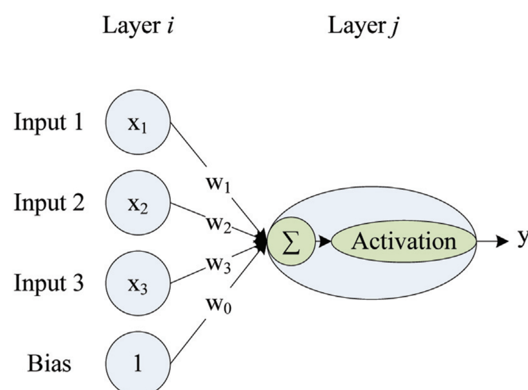

Figure 6.1 Model of an artificial neuron.

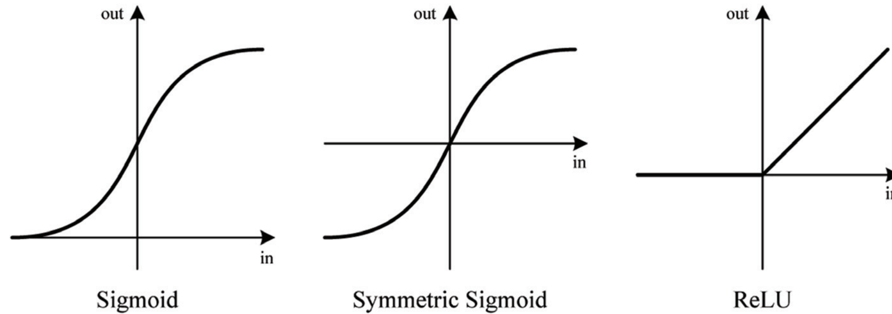

Figure 6.2 Exemplary activation functions used in neural networks.

The bias is another value summed up along with the weighted inputs. This parameter influences the neuron's general activity or the likelihood for an output activation of the neuron. For simplicity, the bias can be interpreted as the weight for a constant input value of 1, so that all parameters of the network can be interpreted as weights. Therefore, a neuron with inputs $x_1, x_2 \ldots, x_n$, weights $w_1, \ldots, w_n$, bias w_0, (with $x_0 = 1$) and activation function f can be described mathematically as

$$y = f\left(\sum_{i=0}^{n} w_i x_i\right).$$

In so called *Multi Layer Perceptrons* (MLP), neurons are arranged in layers. The neurons of one layer are connected to neurons in the following layers. No connections exist between neurons of one layer and the graph formed by the neurons and connections is a directed acyclic graph. Therefore, MLPs are called *feed forward networks*.

The task performed by the neural network depends on the parameters, namely the weights and biases. Therefore, the network parameters have to be adjusted before the network produces the correct outputs. This adjustment is called *training*. Different methods for training multi-layer feed-forward networks have been devised. The most commonly used technique is the backpropagation of error [22].

6.3.2 Supervised Learning

In a neural network, the internal parameters (weights of the neurons) are also called *trainable parameters*, since they can be trained to approximate a desired function. In case of Scene Labeling, this function would map a pixel of an image to a specific label, using the pixel's neighborhood. For classification

tasks with a given set of classes, *supervised learning* schemes are used. A set of *training samples* contains input images together with the desired output. In combination with an error function, the training set can be used to adjust the internal parameters of the network.

The Cost Function and Backpropagation

Supervised learning for neural networks is performed by measuring the neural net's estimated output against the expected output with a so called *cost function*. The goal of a supervised training is to find the internal parameters which minimize this cost function regarding a set of training examples. Since the network in general models a highly non-linear function, *gradient descent* can be used as an optimization procedure. This is done by computing the gradient of the cost function and leveraging the chain rule to propagate the cost and the gradient back through each layer of the network. The weights in each layer are updated according to the current gradient of the backpropagated cost. This algorithm is therefore called *backpropagation*.

A successful training converges against the minimum value of the cost function. It is important to choose the cost function suitable for the task that the neural network needs to perform. For classification tasks, a combination of the softmax function and (multinomial) logistic regression is often performed to train the internal parameters. The softmax function serves as a normalization function, which maps input values x_j of arbitrary range to values in the range $(0, 1)$ that add up to 1. The maximum of the input values maps close to 1 while the other values map close to 0. The function is defined by

$$\mathrm{softmax}(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \quad \text{for } j = 1, \ldots, K.$$

The softmax directly serves as the multinomial version of the logistic function used in logistic regression. The resulting cost function is defined by

$$\mathrm{cost}(x) = -\ln(\mathrm{softmax}(x'_k)),$$

with x'_k as the predicted output of the neural network for the actual class *k*. The cost is therefore the negative log-likelihood of the expected class, which minimizes, when the estimated probability for that class is 1.

Stochastic Gradient Descent

Gradient descent is an algorithm that finds a local minimum by following iteratively the negative gradient of a function $F(x)$ at each point x. It can be defined as

$$x_{i+1} = x_i - \eta_i \nabla F(x_i).$$

Here, η_i is the so called *learning rate* at iteration i. Choosing the right η in every iteration of the algorithm is crucial for the success and the convergence speed of the optimization. If η is too small, it takes many iterations to find a local minimum. Furthermore, the detected local minimum might just be a plateau with better local minima in the neighborhood. If the chosen learning rate is too big, it is possible to jump repeatedly over the local minimum, but never reaching it. In severe cases, it is even possible that the algorithm diverges. There are several schemes for choosing the learning rate adaptively. Resulting in most cases in a computational overhead, which is due to an additional analysis step at the current point of the function. A fixed learning rate is often used, which is scaled down in every iteration. Later iterations are supposed to be close to a minimum and require therefore a finer grained learning rate.

Given the basic gradient descent update rule, the term $\eta_i \nabla F(x_i)$ can be called update ν_i of iteration i. Since these updates only rely on the current gradient, small bumps in the error function might lead to a jittering path in the gradient descent, which increases the number of iterations until a local minimum is found. This might especially occur in stochastic gradient descent, which does not use every training sample in each iteration. To overcome this, many learning schemes extend the update rule by a *momentum term.* The update rule is then defined by

$$x_{i+1} = x_i - (\eta_i \nabla F(x_i) + \mu \nu_{i-1})$$

with a new definition for the update ν_i:

$$\nu_i = \eta_i \nabla F(x_i) + \mu \nu_{i-1} \quad \text{and} \quad \nu_0 = 0.$$

The parameter $\mu \in \mathbb{R}(\mu \geq 0)$ denotes the influence of the update from the previous iteration. If $\mu = 0$, no momentum is used to calculate the current update. Update steps are stabilized and the "velocity" in flat valleys of the error function is increased by using a momentum. However, this property is not always desired in all gradient descent schemes, because the momentum might also cause the update to overshoot. Hence, the momentum term should be used with care.

In a learning environment, a point x of the cost function is the set of internal parameters unified with the expected net output. Since there is not only one training example but many, there are also many expected output points. The cost of more than one data point is therefore the sum of all costs.

This is called *objective function*. It follows, that in an iteration (epoch) of the gradient descent algorithm, all data points need to be processed. This is called *batch gradient descent*. In many cases though, processing all data points in one epoch is not feasible because of the size of the dataset. In this case, *stochastic gradient descent* is used. Instead of predicting all data points per epoch, a random subset for each epoch is generated. If the subsampling is random enough in each epoch, this method optimizes an approximation of the objective function. Though each individual epoch might not sufficiently approximate the objective function, the repeated random sampling does. Stochastic gradient descent is therefore a common approach to train a neural network with big datasets.

6.3.3 Convolutional Neural Networks

A *Convolutional Neural Network* (CNN) is an extension to the common MLP, originally designed for two-dimensional data, like images. As the name suggests, it adds *convolutional layers* to the set of possible layers in an MLP. There is an analogy here with the primary visual cortex of a cat, which also uses convolution-like simple cells to extract information from spatially close overlapping regions of the field of view [23]. In [24], the authors showed that the backpropagation algorithm can be extended for the training of CNNs by introducing an update and backpropagation rule for convolutional layers.

Convolutional Layer

The convolution layer differs in two ways from the common fully connected layer of an MLP:

1. Convolution layers only sum up a fixed window of the input signal. They are therefore only *locally connected*. This connection window is called *receptive field* of the layer.
2. Each possible position of a receptive field uses the same weights to produce an output. This is called *weight sharing*.

The output signal is produced in a sliding window fashion, by applying a weighted summation of the receptive field for each possible receptive field position. The output contains as many values as possible positions. It is exactly a convolution of the input signal, where the layer weights form the convolution filter (kernel). A convolution layer can have several filters, thus forming a *filter bank*, which is analogous to the amount of hidden units in this layer.

Pooling Layer

Another important extension of the MLP is the *pooling layer*. A pooling layer performs a subsampling of the input signal, by "combining" small windows of the input signal into several singular values. A common pooling function is *max-pooling*, which calculates the maximum of its receptive field. Another pooling function is *average-pooling* which computes the average value in its receptive field. A pooling can be seen as a convolution with a special function and a *stride* that equals the filter size of the *pooling kernel*. Regular convolutions have a stride of 1, meaning every pixel position is computed in the convolution. A stride of 2 means that every other pixel position is computed. The purpose of pooling is not only to reduce the spatial size of the input signal, but also to increase the robustness of translational invariance of the activations.

Multiscale CNN

A variation of convolutional neural networks is the *Multiscale CNN*. Instead of processing an input signal as it is, the Multiscale CNN processes several scaled down versions of the signal simultaneously. This approach increases the ability to extract scale invariant features, without the need to increase the size for the extracted pixel neighborhood patch windows. The extracted feature maps of each scale are finally combined to produce a joint feature map. This can be done by a fully connected layer that takes all feature maps as an input to compute its output. For the Scene Labeling application, an image pyramid has to be created prior to the extraction of image patches for each scale, which are then fed to the Multiscale CNN.

Patch Based and Image Based Application

Neural networks for image classification tasks were traditionally designed so that they process a complete image of fixed size and produce classification results of a fixed size as well. Big image sizes automatically implied that the fully connected hidden layers had also a great amount of hidden units. This resulted in the reduction of the input images sizes to keep the neural networks scalable and computable. In order to apply neural networks in a pixel classification scheme, image patches had to be extracted at each pixel position that needs to be classified. In many cases, these extractions are applied sparsely across the image to produce a coarse pixel classification.

A patch based application of CNNs for pixel classification tasks is computationally very inefficient, because image patches for neighboring pixels overlap. Therefore, the same convolutions are computed multiple times.

This redundancy can be omitted by applying CNNs in an image-based fashion. This has an effect on several aforementioned components of the neural network, since they have been designed in regard to a patched based application. The fully connected layer especially is not applicable in an image based application, because *full connectivity* is contrary to the *local connectivity* of the convolution layers for arbitrary image sizes. The adequate translation of a fully connected layer in a patch based approach is actually another convolution layer, with a 1×1 convolution on all locally connected input values.

Another layer type that works differently in an image based application is the pooling layer. A naïve translation would result in a huge loss of output resolution, since pooling layers in patch based mode are designed to subsample the input signal. A patch based application on every possible pixel location though doesn't share this subsampling property. This is why the patched based approach really evaluates every pixel location, while an image based approach implicitly only fully evaluates a subset of all pixel location due to the subsampling. To remove the subsampling property, a pooling must be applied in a convolutional manner (overlapping pooling). Looking at the output maps of such an overlapping pooling, it is clear, that they differ from maps of a non-overlapping pooling. In particular, neighboring pixels from a non-overlapping pooling are not neighbors anymore. If a convolution layer follows, it results in a wrong calculation of the output maps. This can be corrected by reordering the pixels after the pooling layer into n subimages, where n is the size of the pooling kernel or the stride, and apply the following layers on each subimage independently [25]. The reordering is hence defined as *fragmentation*, because the input map is fragmented into smaller output maps. Figure 6.3 shows such a fragmentation after the application of a 2×2 pooling producing 2×2 subimages.

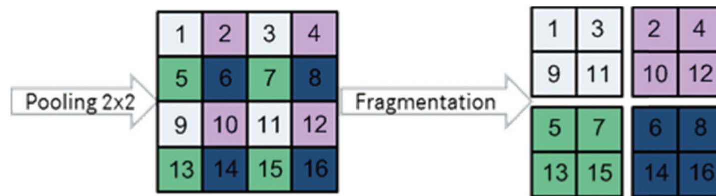

Figure 6.3 Example of a fragmentation after a 2×2 pooling. The naïve approach would only produce the bright pixels, while an overlapping pooling produces all other possible pixels (purple, green, and blue). These pixels must be reordered to be able to correctly continue with the forward propagation of the neural network.

For Multiscale CNNs, an image based application introduces another difficulty, which needs to be solved. In a patch based approach, the image patches for each scale have to be extracted and each patch has the same size. In an image based approach however, the feature maps for different scales are of different size. This becomes a challenge in the fully connected layer, which combines the feature maps of all scales. Since there are no fully connected layers in the image based approach, the feature maps of each scale need to be transformed so that a regular convolution layer can handle them. The simplest solution is to scale the smaller maps up so that they all match in size. If the maps have been fragmented because of a pooling layer, they need to be *defragmented* before they are scaled up. Defragmentation is the reverse function of fragmentation, turning multiple smaller maps into one bigger map.

6.4 CNN for Scene Labeling

There are many ways to perform Scene Labeling on images. CNNs have proven themselves useful on this task, because they achieve state of the art performance without the need to develop complex multi cue frameworks that combine different inputs and sensors. Additionally, many frameworks for modeling, training and execution of CNNs exist, e.g., Caffe [26], Torch7 [27], Theano [28], Pylearn2 which is built on top of Theano, and cuda-convnet [29]. These frameworks exploit the CNN's parallelizability to provide fast and time efficient implementations using *General Purpose GPUs* (GPGPU). Furthermore, the research community is actively training and publishing models, which can often be adapted to a specific task by resuming the training with corresponding data. Most frequently used models are AlexNet [30], GoogleNet [31] or VGG [32]. They differ in complexity and run time efficiency, but reached state of the art performance during their time of publishing for certain challenges on datasets like ImageNet [33]. A high *network capacity* is needed to achieve a high accuracy on such complex tasks. So the trained models are rather big and need a huge amount of computational power. Incorporating this into an embedded system with low power consumption, as is needed for ADAS, is still a great challenge.

The following section describes one possible model with reduced complexity, selected for implementation in the course of the DESERVE project. Its purpose is to detect the road, vehicles and vulnerable road users, which can then be utilized for lane prediction and pedestrian detection.

6.4.1 Exemplary Network for Scene Labeling

The proposed model is derived from the Multiscale CNN used in [34]. It consists of 2 convolutional layers and 2 pooling layers. The activation function, used after the convolutional layers, is the ReLU function (see Figure 6.2). Each convolution layer contains a bank of $16 \times (7 \times 7)$ filter kernels. These four layers are applied on three scales of the input image and combined by a fully connected layer, producing 6 output channels: *background*, *road*, *vehicle* (including cars, trucks, busses, ...), *vru* (vulnerable road users: pedestrians, cyclists, ...), *sky* and *infrastructure* (buildings, signs, barriers, traffic lights, ...). Those channels are normalized by a softmax layer to produce class probability maps for each class. By applying an argmax on these maps a class membership map is produced returning the most probable class for each pixel. The input images are preprocessed by transforming them into an image pyramid and locally normalizing them afterwards to zero mean unit variance in a 15×15 neighborhood. Figure 6.4 shows the complete toolchain and Figure 6.5 the network topology in more detail.

6.4.2 Evaluation

The topology described in subsection 6.4.1 was trained with 6895 labeled night time images of a near infrared camera used in the NV3 night vision system of a Mercedes Benz S-Class. The images show mainly rural, but also urban, road scenes under different weather conditions and different seasons. To augment the heavily under-represented *vru* class, 15174 images are added to the aforementioned set of images, where only the *pedestrian* and *cyclist* labels are used. This is called the *learn set*. The training scheme is stochastic gradient descent with the logistic regression objective function for 6 classes.

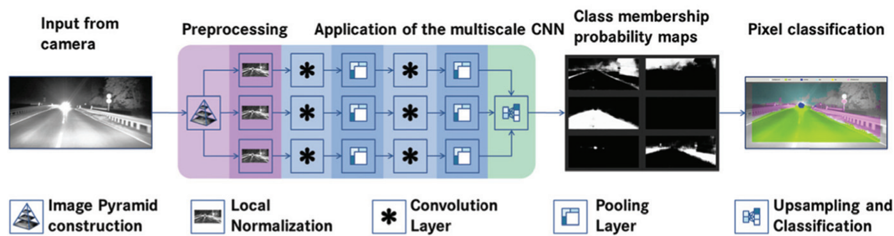

Figure 6.4 The complete processing chain from input image to a scene labeled image is displayed. After building an image pyramid of 3 layers and the local normalization every scale is fed to its own processing chain. This produces 6 class membership probability maps. They can be interpreted and augmented as seen in the output image.

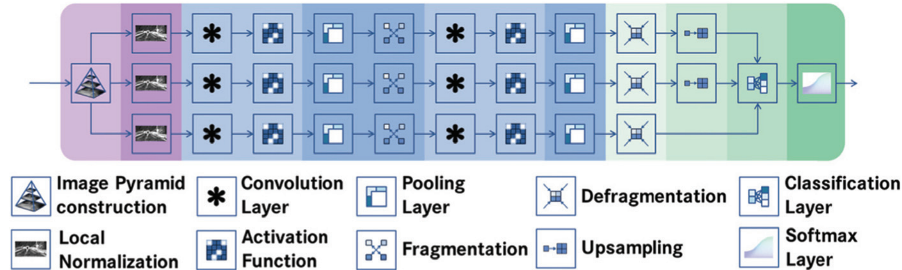

Figure 6.5 The image pyramid construction layer produces 3 scales that are locally normalized in 15×15 windows. Every scale is propagated independently. There are in total 2 convolution layers with $16 \times 7 \times 7$ filter kernels using the ReLU activation function. After activation a 2×2 max-pooling is performed followed by a fragmentation in the first pooling layer. A second fragmentation is not necessary since the second pooling layer is followed by a defragmentation. The small scaled feature maps are sampled up and fed to a classification layer, being a $6 \times 1 \times 1$ convolution layer. Finally, a pixel wise softmax is applied.

It is trained 10.000 epochs with 40960 balanced training examples (patches) per epoch. The learning rate was determined following several short runs of 100 epochs with different learning rates. The best progressing learning rate was then chosen. During training, the learning rate was linearly reduced after 5000 epochs by a factor of 0.995 per epoch. Figure 6.6 shows the training progress (2-2-16 topology) in relation to the objective function on the learn set. Two other topologies were also trained in the same way. One introduced a third convolution layer including the ReLU activation function after the second pooling (3-2-16 topology). The third topology is similar to the 3-2-16 topology, but uses 32 filters per convolution (3-2-32 topology). Figure 6.6 shows that the topology with the least trainable parameters (2-2-16 topology) performed worst during training. The introduction of another convolution layer (3-2-16 topology) resulted in a better learn curve. However, doubling the amount of filters (3-2-32 topology) increased the learn performance yet again.

Since the classifier of topology 3-2-32 appears to have the best performance, it is evaluated on the evaluation set of images containing 200 images that have not been part of the learn set, called the *eval set*. Evaluation in multiclass problems is done by analyzing the *confusion matrix*. The confusion matrix for topology 3-2-32 is displayed in Table 6.1. It shows the class predictions in relation to the actual class. The diagonal entries form the *true positives* (pixels that were classified correctly, TP) for each class, while the remaining entries of a line or column display the individual *false negatives*

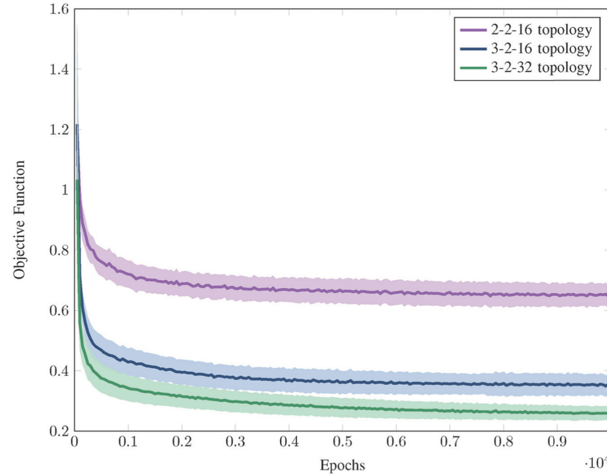

Figure 6.6 Displayed are the learn curves of three different network topologies. Each topology was trained three times and the learn curves were averaged. The averaged learn curves are displayed as solid lines while the standard deviation for 50 epochs is displayed as the area around the lines.

Table 6.1 The confusion matrix of topology 3-2-32 and the respective FNR, FPR and IU for each class. The classes are background (Bg), road (Rd), vehicle (Veh), sky, vulnerable road users (VRU) and infrastructure (Inf). Each cell shows the percentage (from all pixels in the dataset) of actual class (row) predicted as class (column)

Act \ Pred	Bg	Rd	Veh	Sky	VRU	Inf
Bg	24.9349[a]	1.9409	1.1226	2.1282	0.3359[b]	5.8754
Rd	1.5685	29.4059[a]	1.0226	0.0034	0.1269	0.3226
Veh	0.1042	0.0829	3.6523[a]	0.0051	0.1156	0.7749
Sky	1.7298	0.0080	0.1744	7.1476[a]	0.0083	0.9632
VRU	0.0058	0.0032	0.0740[b]	0.0001	0.0733[a]	0.0777[b]
Inf	1.6244	0.0459	1.0077	0.3351	0.3538[b]	12.8450[a]
FNR	31.38	**9.38**[c]	22.87[d]	28.75	68.68[b]	20.77
FPR	16.79	**6.61**[c]	48.22[d]	25.70	92.77[b]	38.42
IU	60.27	**85.16**[c]	44.89[d]	57.17	6.24[b]	53.02

(pixels not classified as the desired class, FN) and *false positives* (pixels falsely classified as the desired class, FP). Therefore, the sum over one row of the table gives the percentage of the respective class in the whole training set.

The quality measures of binary classification problems can therefore be applied for each class individually in a “one versus all” fashion. Classic measures contain the *False Negative Rate* (FNR), the *False Positive Rate* (FPR) and the *Intersection over Union* (IU). Those are defined as follows:

$$FNR = \frac{FN}{N}, \quad FPR = \frac{FP}{N}, \quad IU = \frac{TP}{TP \cup FP \cup FN}$$

N denotes the number of all pixels evaluated. FNR and FPR are 0, if the classification is correct and get bigger, if more pixels are classified incorrectly. The IU has a value of 1 in case of a perfect classification and the value gets smaller, if more pixels are classified incorrectly.

Table 6.1 shows the percentage of pixels classified as one of the 6 classes. The last 3 rows display the class-wise FNR, FPR and IU. The confusion matrix shows several interesting features:

a. The diagonal entries show the true positives, the correctly classified pixels. Since the total amount of pixels in the evaluation dataset for each class varies, the maximum possible number for each entry varies as well.
b. For the class *vulnerable road users* (VRU) the classifier performs badly. There are more pixels classified as *vehicles (Veh)* or *infrastructure (Inf)* than VRUs, resulting in a bad FNR. Even worse is the FPR, since the amount of *background (Bg)* or *infrastructure (Inf)* pixels classified as VRU is far greater than the amount of correctly classified pixels. This results in a bad IU.
c. The best performing class is the class *road (Rd)*. It has comparatively few false positives and negatives, which results in a good FNR, FPR and IU.
d. The class *vehicle (Veh)* shows an arbitrary performance. Though the FNR is quite good and better than the class *background (Bg)*, its FPR is second to last. So the IU is greatly affected.

After analyzing each class by itself the question arises of how good this classifier is compared to classifiers, which contain other well and bad performing classes. A common measure to describe the overall performance of a classifier is the *accuracy* (ACC). It is the ratio of correctly classified pixels to all pixels. Let N be the amount of classes and $C_{i,j}$ be the amount of pixels from class i classified as class j. In a multiclass setup, the accuracy can then be defined as:

$$\text{ACC} = \frac{\sum_{k=1}^{N} C_{k,k}}{\sum_{i,j=1}^{N} C_{i,j}}$$

This measure captures in a straight forward way the correctness of a classifier. The value is in the range [0, 1], where a perfect classifier reaches 1. If one or more classes are under-represented in the evaluation dataset, the

expressiveness of this measure suffers, since it does not normalize the amount of samples per class. Other ways to increase the sensitivity to underperforming classes is to average the FNR, FPR or IU over the classes. The *Matthews Correlation Coefficient* (MCC) was designed for binary classifications and computes a correlation between the actual and predicted classifications. It was extended to incorporate more than two classes and is defined by [35] as follows:

$$\mathrm{MCC} = \frac{\sum_{k,l,m=1}^{N} C_{k,k}C_{m,l} - C_{l,k}C_{k,m}}{\sqrt{\sum_{k=1}^{N}\left[\left(\sum_{l=1}^{N} C_{l,k}\right)\left(\sum_{\substack{f,g=1\\ f\neq k}}^{N} C_{g,f}\right)\right]\sum_{k=1}^{N}\left[\left(\sum_{l=1}^{N} C_{k,l}\right)\left(\sum_{\substack{f,g=1\\ f\neq k}}^{N} C_{f,g}\right)\right]}}$$

The Matthews Correlation Coefficient is in the range $[-1, 1]$. An MCC of 1 is a perfect classifier, while -1 is the total contradiction. An MCC of 0 is a random classifier. Table 6.2 shows the ACC, mean IU, MCC and mean FNR for the classifiers trained in Figure 6.6. It can be seen that topology 3-2-32 outperforms the topologies in all defined measures.

6.5 Hardware Platforms for Scene Labeling

Embedded hardware platforms for Advanced Driver Assistance Systems face several challenges. They have to provide a huge amount of processing power to keep up with the rising complexity of applications and the increasing amount of data they have to process. However, the platforms should have low power consumption.

At one end of the spectrum of hardware architectures, *General Purpose Processors* (GPPs) usually do not fulfill all the requirements and restrictions of embedded systems in advanced driver assistance systems. They offer a high degree of flexibility due to the arbitrary programmability, but they cannot usually comply with the high demand on processing power while holding the restrictions in power consumption.

Table 6.2 Displayed are the measures Accuracy (ACC), mean Intersection over Union (mIU), Matthews Correlation Coefficient (MCC) and mean False Negative Rate (mFNR) for 3 topologies

Topology	ACC	mIU	MCC	mFNR
2-2-16	0.60	0.35	0.50	0.44
3-2-16	0.69	0.42	0.60	0.37
3-2-32	**0.78**	**0.51**	**0.71**	**0.30**

At the other end of the spectrum, *Application Specific Integrated Circuits* (ASICs) provide a high degree of processing power and excellent power efficiency. However, they are not flexible as they are fixed after manufacturing and cannot be programmed.

There is a wide range of hardware platforms in between these two extremes, which provide a trade-off between the different characteristics. For example, *Graphical Processing Units* (GPUs) have been used to accelerate the execution of complex algorithms. They provide a certain degree of flexibility, as they are programmable and they achieve high processing power due to a high degree of parallelism. However, the power consumption of GPUs is fairly high and they are therefore not suitable for use in personal cars.

Adapting processor architectures to a given application is a promising approach for designing hardware platforms. *Application-Specific Instruction-Set Processors* (ASIPs) are based on programmable processor architectures. These are adapted to a specific application or a class of similar applications, e.g., by extending the instruction set, by adding dedicated hardware accelerators for frequently used operations, or by changing architectural parameters in order to bypass bottlenecks.

Scene labeling has been implemented on several platforms including CPUs, GPUs, FPGAs, and ASICs. This section gives an overview of recent implementations of convolutional neural networks on different types of computing platforms. At first, the computational complexity of convolutional neural networks is discussed, by deriving a measure of the total number of operations needed in order to compute the forward propagation of one frame through the network. This also serves as a basis for the comparison of different implementations, which is presented later.

6.5.1 Theoretical Performance Requirements

This section describes the computational complexity of convolutional neural networks in terms of operations needed in the forward propagation of a frame. This number of operations clearly depends on the topology of the network.

The most computational intensive task is the convolution, especially, as many convolution layers contain a huge number of filters. For an input image of size $w \times h$ and a convolution kernel of size $n \times n$, the kernel is applied $(w-(n-1))(h-(n-1))$ times. Each time, n^2 multiplications are performed and the results accumulated. Counting the multiply and accumulate operations as two, this leads to a total count of

$$N_{conv}(w,\ h,\ n) = 2(w - (n-1))(h - (n-1))n^2$$

operations for a single convolution.

The activation function is applied to each output pixel of the input layer. Therefore, the total number of operations for an input image of size $w \times h$ is given as

$$N_{act}(w, h, c_{act}) = whc_{act},$$

where c_{act} describes the cost of applying the activation function to one pixel. In case of the ReLU (Rectified Linear Unit), the operation determines the maximum of the input value and 0. Therefore, $c_{ReLU} = 1$.

For the pooling layer, the number of operations depends not only on the size $w \times h$ of the input frame, but also on the kernel size $n \times n$ and the stride s. In some cases, the stride equals the kernel size, but in overlapped pooling, a stride of 1 might be used. In general, the number of operations performed in a pooling layer can be described as

$$N_{pool}(w, h, n, s) = c_{pool} \frac{wh + (s-n)((s-n) + w + h)}{s^2},$$

where c_{pool} is the number of operations per pooling window. For a max-pooling, the number of operations is $c_{\max} = n^2 - 1$, for an average-pooling, the number of operations is $c_{\text{avg}} = n^2 + 1$.

For the exemplary convolutional neural network described in subsection 6.4.1, which is named 2-2-16 in Table 6.2, the following remarks give the numbers of operations for the single layers. The image preprocessing, i.e., the construction of the image pyramid and the normalization, is not counted in this section.

In this exemplary case, the input image has 1024×512 pixels. In the preprocessing step, an image pyramid is generated by an iterative process. In each iteration, the image dimensions are halved by subsampling. Afterwards, the three scaled images from the pyramid are padded by replicating the border pixels in order to maintain the correct output size after the convolutions. The resulting image sizes are listed in Table 6.3.

The first convolution layer performs 16 convolutions with a 7×7 kernel and generates 16 output images. The convolution is only performed for pixels where the convolution kernel fits into the input image, so that the resulting image is reduced by 6 pixels in width and height. The convolution layer is followed by an activation layer, which applies the activation function to each

Table 6.3 Input image sizes for three different scales in the exemplary convolutional neural network

Scale	Pyramid Output	Padded
S	512×256	534×278
M	256×128	278×150
L	128×64	150×86

of the 16 output images of the convolutions. The following max-pooling layer uses a 2×2 patch and a stride of 1 (overlapped pooling). It does not change the total number of pixels but separates one image into four sub images of quarter size. The fragmentation of the images does not contribute to the number of operations since it can be hidden in the other layers. The second convolution layer performs 16 convolutions of size 7×7 on each of the 16 fragmented images and then accumulates them to 16 fragmented output images. The following activation function and pooling layers work the same as after the first convolution layer.

This flow of images through two convolution layers with activation functions and two pooling layers is performed independently for the three scales of the input image. The resulting images are scaled to the same size before they are fed into the classification layer.

The classification layer at the end performs one convolution of size 1×1 per output class, of which there are six in the exemplary convolutional neural network.

With these image and filter sizes, the computational complexity of the convolutional neural network can be estimated using the equations above. Table 6.4 gives the operation counts for the three scales by layer type.

The total number of operations performed for one input image is 4.796.792.784. As expected, the convolution layers contribute the biggest share in the number of operations, with a proportion of 99.2 percent. In order to reach a processing rate of 30 frames per second, 144 billion operations have to be performed per second.

Table 6.4 Number of operations for the exemplary convolutional neural network

Scale	Convolution	Activation	Pooling	Classif.	Operations
S	3.590.995.968	4.444.416	13.220.592	12.582.912	**3.621.243.888**
M	922.435.584	1.175.808	3.470.064	3.145.728	**930.227.184**
L	243.253.248	327.936	954.096	786.432	**245.321.712**
Ops.	**4.756.684.800**	**5.948.160**	**17.644.752**	**16.515.072**	**4.796.792.784**

Table 6.5 lists implementations of convolutional neural networks on different platforms and gives the performance in terms of performed operations per second. When available, two numbers are given for each implementation. The *peak* performance gives the theoretical maximum number of operations per second that the platform can perform. The *real* performance gives the number of operations per second for CNNs of different topologies on the platform. Not all implementations listed in the table are used for scene labeling, but perform other image based detection and classification tasks with convolutional neural networks. Therefore, the networks that are used in the applications may differ in size. This is mentioned, because some implementations do not scale up to bigger networks easily. The subsequent sections give more details to the entries in the table.

Table 6.5 Comparison of different implementations of convolutional neural networks on different platforms

			Perf. [GOPs]	
Author	Year	Device	Peak	Real
		CPU Implementations		
Farabet et al. [39]	2011	Intel Core 2 Duo	10	1.1
Dundar et al. [40]	2013	Intel Core i7 4-core	200	90
Jin et al. [41]	2014	Intel Core i5	45	30
Zhang et al. [42]	2015	Intel Xeon	–	12.87
		GPU Implementations		
Farabet et al. [39]	2011	nVidia GTX 480	1350	294
Dundar et al. [40]	2013	nVidia GTX 780	3977	620
Jin et al. [41]	2014	nVidia GTX 690	5622	530
Cavigelli et al. [43]	2015	nVidia GTX 780	3977	1781
		Mobile GPU Implementations		
Farabet et al. [39]	2011	nVidia GT335m	182	54
Dundar et al. [40]	2013	nVidia GTX650m	182	54
Cavigelli et al. [43]	2015	nVidia Tegra K1	326	76
		FPGA Implementations		
Farabet et al. [39]	2011	Virtex 6 VLX240T	160	147
Dundar et al. [40]	2013	Zync ZC706	–	36
Gokhale et al. [44]	2014	Zync ZC706	–	227
Zhang et al. [42]	2015	Virtex 7 485t	–	61.62
		ASIC Implementations		
Pham et al. [45]	2012	neuFlow in IBM 45 nm	320	294
Chen et al. [46]	2015	Accelerator in 65 nm	–	452
Cavigelli et al. [47]	2015	Accelerator in 65 nm	274	203

6.5.2 CPU-based Platforms

As discussed before, running convolutional neural networks for scene labeling or other image processing tasks incorporates a huge amount of computation. For the use in ADAS, CPUs cannot provide the necessary processing power while also complying to the power budget restrictions. Active work is performed in order to speed up the implementations (e.g., [36]). Also, algorithmic research is conducted in order to speed up the convolutions, e.g., [37, 38].

A reference implementation of the exemplary CNN from subsection 6.4.1 was written using C++. It is worth mentioning that the focus in this implementation was not speed or efficiency. Instead, it was intended as a reference for the assembler implementation described later. The implementations of the image processing operations and the different layers of the convolutional neural network make use of templates. This provides the flexibility to use different data types for the pixel values and coefficients. The templates enabled the use of fixed-point data types in order to analyze the compromise of data width and accuracy.

On an Intel Core i5-2400 with 3.1 GHz, the computations for one input image of size 1024×512 with double precision values and coefficients require about 11 seconds, which corresponds to about 436 MOPS. This implementation does not use multiple cores for computation.

6.5.3 GPU-based Platforms

Modern GPUs provide a huge amount of computing power that can be used for general purpose computing (GPGPU). The use of GPUs is most beneficial, if the application provides a high degree of parallelism and regularity. CNNs fall into this category. Therefore, most deep learning frameworks mentioned in the previous section accelerate evaluation and training of networks with GPUs using CUDA, and there are also frameworks specifically developed for GPUs, e.g., cuda-convnet2 [29] and Marvin [48].

A downside of using the powerful GPUs is the amount of power they consume, which makes the use of GPUs in mobile devices infeasible. Nevertheless, GPUs can be used for training the networks, as the training is performed offline. Recently, mobile or embedded GPUs have emerged, aiming to provide low-power high-performance computing platforms.

6.5.4 FPGA-based Platforms

A FPGA, a configurable hardware platform, provides a compromise between the flexibility of a GPU and the efficiency of an ASIC. The high degree of

parallelism that is possible in a FPGA, allows for high performance signal processing. As double precision arithmetic is costly for a hardware-based implementation, the C++ implementation of the algorithm was used to analyze the quality of the classification depending on the data width of pixel values and coefficients. For 32-bit data with 22 fractional bits, the computations are exact and no errors appear. If 16-bit data with 11 fractional bits are used, about 1.4 percent of the pixels are classified incorrectly, which was acceptable in this scenario.

The use of a soft core processor that is mapped to the FPGA also provides software programmability of the design. In order to raise the computational performance, the soft core processor can be extended with dedicated hardware modules (application-specific instruction-set processor, ASIP). For example, the instruction-set can be extended by new functional units for complex operations which are placed in the processor's pipeline and perform as quick as the default operations. Additionally, more complex operations taking more execution cycles can be added as external accelerators tightly coupled with the processor's data path.

In the course of the DESERVE project, an ASIP implementation for convolutional neural networks has been developed. It is based on the TUKU-TURI processor [49, 50], which was developed for image processing and video coding implementations. It is a Very Long Instruction Word (VLIW) processor with two issue slots and 64 bit wide registers that can be split up into subwords of 8, 16, 32, or 64 bits. These subwords are processed in parallel (microSIMD) by all default functional units. Additional features include conditional execution in order to reduce control overhead, and a DMA controller for memory transfer between external and internal memory.

As derived from the CPU-based reference implementation (see subsection 6.5.2), 16 bit wide data is used for the pixel values and the network's coefficients. Therefore, the SIMD-feature can be used to process four values in parallel, which gives a significant speed-up.

As seen in subsection 6.5.1, the convolution is the most computing intensive task in the whole process. Therefore, the TUKUTURI processor was extended with a co-processor that performs 16 convolutions of four pixels at once.

The internal memory of the TUKUTURI is not capable of holding a whole input image. Therefore, the images are processed in blocks. The DMA module supports block transfers, so that a rectangular subsection of the image can be transferred between internal and external memory. The module holds a queue of memory transfers, which are processed independently from the TUKUTURI

processor. This allows the TUKUTURI to program several transfers and process data blocks transferred previously, while the DMA transfers the next blocks in the background.

The first implementation of the exemplary convolutional neural network on the TUKUTURI processor processed one input frame in about 1.2×10^9 cycles. With a clock frequency of 100 MHz, this corresponds to about 0.08 fps. Using the convolution co-processor, the cycle count could be reduced to about 243×10^6 cycles, corresponding to a frame rate of about 0.411 fps. This is a speed-up of factor 5.1. Using the capabilities for background transfers, the total cycle count was reduced to about 101×10^6 cycles per frame, which is an additional speed-up of factor 2.4, leading to about 0.99 fps. According to Table 6.4, we need about 4.8×10^9 operations per frame. Therefore, this implementation reaches about 4.8 GOPs.

6.6 Summary

Convolutional neural networks and methods of deep learning have been used in image processing, segmentation and classification tasks successfully. The huge amount of processing power needed for CNNs for Scene Labeling tasks in advanced driver assistance systems combined with the resource restrictions in embedded systems pose a challenge for hardware architects. FPGAs have been shown as a suitable platform for the implementation of CNNs for Scene Labeling.

References

[1] G. Carneiro and N. Vasconcelos, “Formulating semantic image annotation as a supervised learning problem,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, 2005, pp. 163–168.

[2] E. Saber, A. Tekalp, R. Eschbach and K. Knox, “Automatic Image Annotation Using Adaptive Color Classification,” *Graphical Models and Image Processing,* 1996.

[3] J. Shotton, J. Winn, C. Rother and A. Criminisi, “TextonBoost for Image Understanding: Multi-Class Object Recognition and Segmentation by Jointly Modeling Texture, Layout, and Context,” *International Journal of Computer Vision,* 2009.

[4] M. Pietikäinen, T. Nurmela, T. Mäenpää and M. Turtinen, "View-based recognition of real-world textures," *Journal of Pattern Recognition,* 2004.

[5] X. Ren, L. Bo and D. Fox, "RGB-(D) scene labeling: Features and algorithms," *Computer Vision and Pattern Recognition (CVPR)*, 2012.

[6] P. F. Felzenszwalb and O. Veksler, "Tiered scene labeling with dynamic programming," *Computer Vision and Pattern Recognition (CVPR),* 2010.

[7] S. M. Bhandarkar and H. Zhang, "Image segmentation using evolutionary computation," *IEEE Transactions on Evolutionary Computation,* 1999.

[8] A. Ess, B. Leibe, K. Schindler and L. V. Gool, "A mobile vision system for robust multi-person tracking," *Computer Vision and Pattern Recognition,* 2008.

[9] A. Broggi, P. Cerri, P. Medici, P. P. Porta and G. Ghisio, "Real Time Road Signs Recognition," *2007 IEEE Intelligent Vehicles Symposium,* 2007.

[10] J. C. McCall and M. M. Trivedi, "Video-based lane estimation and tracking for driver assistance: survey, system, and evaluation," *Intelligent Transportation Systems,* 2006.

[11] B. Fulkerson, A. Vedaldi and S. Soatto, "Class segmentation and object localization with superpixel neighborhoods," in *Computer Vision, 2009 IEEE 12th International Conference on*, 2009.

[12] A. Torralba, K. P. Murphy and W. T. Freeman, "Sharing Visual Features for Multiclass and Multiview Object Detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* Vol. 29, No. 5, pp. 854–869, May 2007.

[13] M. Turtinen and M. Pietikäinen, "Contextual Analysis of Textured Scene Images," *British Machine Vision Conference,* 2006.

[14] B. Hariharan, P. Arbelaez, R. Girshick and J. Malik, "Simultaneous Detection and Segmentation," *Computer Vision – ECCV,* 2014.

[15] X. He, R. S. Zemel and M. A. Carreira-Perpinan, "Multiscale conditional random fields for image labeling," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, 2004.

[16] X. Liu, O. Veksler and J. Samarabandu, "Order-Preserving Moves for Graph-Cut-Based Optimization," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* Vol. 32, No. 7, pp. 1182–1196, July 2010.

[17] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics,* Vol. 5, No. 4, pp. 115–133, 1943.
[18] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review,* Vol. 65, No. 6, 1958.
[19] C. von der Malsburg, "Self-organization of orientation sensitive cells in the striate cortex," *Kybernetik,* Vol. 14, No. 2, pp. 85–100, 1973.
[20] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences,* Vol. 79, No. 8, pp. 2554–2558, 1982.
[21] X. Glorot, A. Bordes and Y. Bengio, "Deep sparse rectifier neural networks," *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics,* 2011.
[22] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Lerning representations by back-propagating errors," in *Nature*, Vol. 323, Nature Publishing Group, 1986, pp. 533–536.
[23] K. Fukushima and S. Miyake, "Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position," *Pattern Recognition,* Vol. 15, No. 6, pp. 455–469, 1982.
[24] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* Vol. 86, No. 11, pp. 2278–2324, Nov 1998.
[25] A. Giusti, D. Ciresan, J. Masci, L. Gambardella and J. Schmidhuber, "Fast image scanning with deep max-pooling convolutional neural networks," in *Image Processing (ICIP), 2013 20th IEEE International Conference on*, 2013.
[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, New York, NY, USA, 2014.
[27] R. Collobert, K. Kavukcuoglu and C. Farabet, "Torch7: A Matlab-like Environment for Machine Learning," in *BigLearn, NIPS Workshop*, 2011.
[28] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio, "Theano: A CPU and GPU Math Compiler in Python," in *9th Pytthon in Science Conference (SCIPY 2010), Proceedings of the*, 2010.

[29] A. Krizhevsky, "cuda-convnet2," 2014. [Online]. Available: https://code.google.com/archive/p/cuda-convnet2/. [Accessed März 2016].

[30] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou and K. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105.

[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going Deeper with Convolutions," in *CVPR 2015*, 2015.

[32] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR,* vol. abs/1409.1556, 2014.

[33] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision,* Vol. 115, No. 3, pp. 211–252, 2015.

[34] C. Farabet, C. Couprie, L. Najman and Y. LeCun, "Learning Hierarchical Features for Scene Labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 35, No. 8, pp. 1915–1929, 2013.

[35] G. Jurman, S. Riccadonna and C. Furlanello, "A Comparison of MCC and CEN Error Measures in Multi-Class Prediction," *PLoS ONE,* Vol. 7, No. 8, p. e41882, 08 2012.

[36] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio, "Theano: new features and speed improvements," *CoRR,* vol. abs/1211.5590, 2012.

[37] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets and V. S. Lempitsky, "Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition," *CoRR,* vol. abs/1412.6553, 2014.

[38] J. Cong and B. Xiao, "Minimizing Computation in Convolutional Neural Networks," in *Artificial Neural Networks and Machine Learning – ICANN 2014: 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings*, S. Wermter, C. Weber, W. Duch, T. Honkela, P. Koprinkova-Hristova, S. Magg, G. Palm and A. E. P. Villa, Eds., Springer International Publishing, 2014, pp. 281–290.

[39] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, 2011.

[40] A. Dundar, J. Jin, V. Gokhale, B. Krishnamurthy, A. Canziani, B. Martini and E. Culurciello, "Accelerating deep neural networks on mobile processor with embedded programmable logic," in *Neural information processing systems conference (NIPS)*, 2013.

[41] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini and E. Culurciello, "An efficient implementation of deep convolutional neural networks on a mobile coprocessor," in *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*, 2014.

[42] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, 2015.

[43] L. Cavigelli, M. Magno and L. Benini, "Accelerating Real-time Embedded Scene Labeling with Convolutional Networks," in *Proceedings of the 52nd Annual Design Automation Conference*, New York, NY, USA, 2015.

[44] V. Gokhale, J. Jin, A. Dundar, B. Martini and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, 2014.

[45] P.-H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun and E. Culurciello, "NeuFlow: Dataflow vision processing system-on-a-chip," in *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, 2012.

[46] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen and O. Temam, "A High-Throughput Neural Network Accelerator," *Micro, IEEE,* Vol. 35, No. 3, pp. 24–32, May 2015.

[47] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim and L. Benini, "Origami: A Convolutional Network Accelerator," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, New York, NY, USA, 2015.

[48] "Marvin: A minimalist GPU-only N-dimensional ConvNet framework," [Online]. Available: http://marvin.is. [Accessed 2015].

[49] G. Payá-Vayá, R. Burg and H. Blume, "Dynamic Data-Path Self-Reconfiguration of a VLIW-SIMD Soft-Processor Architecture," *Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS) in conjunction with the 2012 International Conference on Field Programmable Logic and Applications (FPL 2012),* 2012.

[50] S. Nolting, G. Payá-Vayá and H. Blume, "Optimizing VLIW-SIMD Processor Architectures for FPGA Implementation," *Proceedings of the ICT.OPEN 2011 Conference* (Veldhoven, Netherlands), 2011.
[51] A. Ess, T. Mueller, H. Grabner and L. v. Gool, "Segmentation-based urban traffic scene understanding," in *Proceedings of the British Machine Vision Conference*, 2009.
[52] Y. LeCun, K. Kavukcuoglu and C. Farabet, "Convolutional networks and applications in vision," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010.