POLITECNICO DI TORINO

Master Degree Course in Electronic Engineering

Master Degree Thesis

Approximate Computing for Softmax and Squash functions in Capsule Networks

Supervisors
prof. Guido Masera
prof. Maurizio Martina

Candidate
Edoardo Salvati

July 2021

Summary

This work presents a spectrum of approximations of the *softmax* and *squash* functions used in Capsule Neural Networks, building upon previous related research as well as introducing new ideas and solutions. The main focus of the work is to explore *approximate computing* techniques for softmax and squash at the algorithmic level, in order to demonstrate a trade-off between complexity of the hardware implementation of the functions and inference accuracy of the Capsule Network. In particular, area usage and power consumption are considered as hardware cost metrics and the inference pass is performed with different Capsule Network models on different benchmark image datasets. Three approximate softmax and three approximate squash architectures are proposed with the ultimate goal of making comparisons between multiple instances of the same function type, in terms of area/energy costs and accuracy of the overall Capsule Network. The motivational idea behind the application of the approximate computing paradigm to the softmax and squash functions is to reduce hardware costs at the expense of lower quality classification results, in the context of *error-tolerant applications*. As a matter of fact, a wide variety of applications exists for which a circuit that produces some incorrect outputs may be still acceptable, provided that the error rate is kept within application-specific thresholds.

The following work is organised in different chapters and sections:

- **Chapter 1** provides an introduction to Deep Learning and Capsule networks, by reviewing the state-of-the-art models and their main applications. It also presents an overview of the approximate computing design methodology and its application to the non-linear functions used in deep neural networks.
- **Chapter 2** describes the thesis work focus, main purposes, development phase and results. In particular, a detailed step-by-step description of the design process of the approximate softmax and squash functions is reported, with critical analysis of the results.
- Finally, **chapter 3** includes the main conclusions of the work, summary of obtained results and final comments. It also outlines possible future works that can be explored to further develop the proposed contributions. In the closing appendix, auxiliary diagrams are included for ease of reference.

Contents

List of Figures

Chapter 1

Introduction and related work

1.1 Introduction to Artificial Intelligence and Machine Learning

Artificial intelligence is a critical topic nowadays as AI applications are becoming widely spread in the modern society. Artificial intelligence refers to computer programs with the ability to mimic the cognitive function of humans, such as reasoning and learning. In particular, an effective definition of the AI field is the effort to automate intellectual tasks normally performed by humans.

The dominant paradigm in AI is machine learning, a revolutionary programming approach that arises from a simple yet groundbreaking question: *could a computer algorithm automatically learn how to perform a specified task by looking at data without the need of being explicitly programmed?*

In classical programming, programmers develop data-processing rules by hand, so that input data is processed according to these rules to obtain output results. On the other hand, a machine learning algorithm is trained rather than explicitly programmed to perform a specific task, that is the algorithm is presented with data as well as expected results and extracts the rules for automating the task from the data.

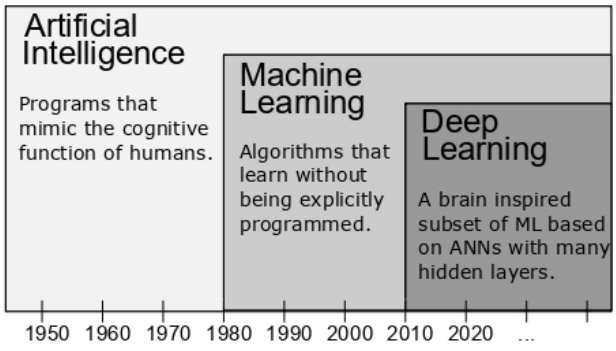


Figure 1.1: Artificial Intelligence field and subfields [1]

Machine learning technology is increasingly present in many aspects of modern society: actually machine learning systems are exploited to identify objects in images, transcribe speech into text as well as select relevant results from web searches, filter contents on social networks and make recommendations on e-commerce websites based on the users' interests.

Machine learning algorithms are classified in two main categories based on their functionality and the approach used to learn from experience: supervised and unsupervised learning algorithms. Supervised machine learning techniques take as input a known set of data and responses to the data and learn to generate reasonable predictions as a response to new data. The supervision to the algorithm is provided in the form of the desired output for each input example that the algorithm learns from. On the other side, unsupervised machine learning techniques are used to draw inferences from datasets consisting of data without labeled responses. The unsupervised nature of the technique is given by the fact that no known output data is given to the algorithm.

Depending on the type of predictions, the supervised learning algorithms are grouped into classification and regression techniques. In particular, the classification models classify input data into discrete categories, while regression techniques predict the value of continuous variables. As regards the unsupervised learning algorithms, the most common techniques perform clustering, that is used for data analysis to find patterns or structures in the input data.

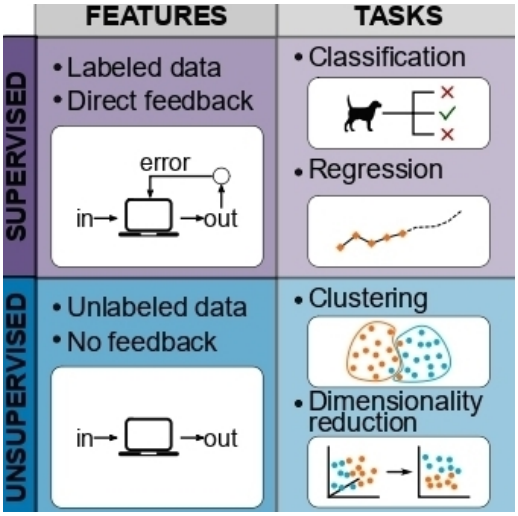


Figure 1.2: Machine Learning overview [2]

Typical applications of supervised learning algorithms include image classification, speech recognition and energy demand forecasting. On the other side, unsupervised clustering techniques are exploited to perform tasks like genomic sequence analysis and market research to segment customers into groups with similar preferences.

The most common classification algorithms are logistic regression, k nearest neighbour, support vector machine, decision tree and neural network, while the most popular regression techniques are linear regression, generalised linear model and regression tree. On the other hand, the clustering algorithms can be distinguished in two groups: hard and soft clustering

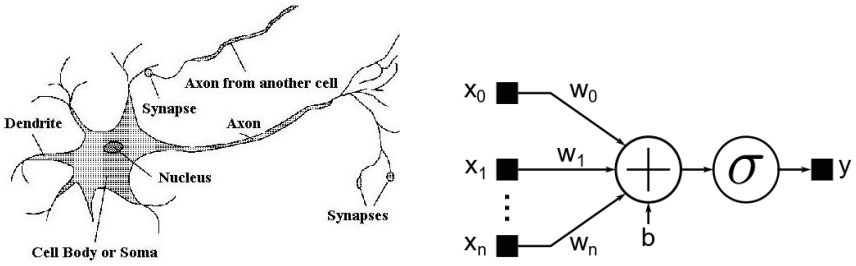
techniques, based on the fact that each data point can belong to only one or more than one cluster. One of the most common hard clustering algorithms is k means, while among the most popular soft clustering techniques is gaussian mixture model.

Selecting the right machine learning algorithm to perform a specific task is a trade-off between the characteristics of the available algorithms, such as accuracy on new data, training speed, memory usage and algorithm interpretability.

As a matter of fact, regarding the classification algorithms, a decision tree is a good option when interpretability, fast training and low memory usage are the key requirements, while high predictive accuracy is less important. If memory usage is a lesser concern, k nearest neighbour is a well-suited candidate as a simple algorithm to classify objects. When an easy to interpret and accurate classifier is needed, a support vector machine can be used for high-dimensional, non-linearly separable data. If the algorithm interpretability is not a key concern, a neural network is a good solution for modelling highly nonlinear systems to classify the input data. Finally, logistic regression is typically used as a baseline for binary classification problems because of the algorithm simplicity.

1.2 Fundamentals of Artificial Neural Networks

A neural network is a computational model that is inspired by the biological network of neurons found in the human brain. The building block of the model is a computational unit that receives n input values and returns a scalar output. First of all, each input is multiplied by a coefficient, called weight and the weighted inputs are summed together with a bias term. Then, the output of the unit is determined by applying a non linear function, called activation function, to the computed sum: $y = \sigma(\sum_{i=0}^{n} w_i\, x_i + b)$. The typical activation function used in the current neural networks is a non linear function called Rectified Linear Unit.



(a) *Biological neuron* **(b)** *Artificial neuron [1]*

Figure 1.3: Neurons

The processing of the basic unit of the neural network model is a brain-inspired computing paradigm that resembles the behaviour of the biological neuron. Typically, the biological neuron accepts the signals entering it via elements called dendrites, perform a computation on those input signals inside its cellular body called soma and generates an output signal on an element leaving it called axon. The neuron responds differently to each input signal depending on a connection present on the dendrite called synapse that can

scale the signal crossing it. Following the scaling performed by the synapses, the weighted signals are combined inside the neuron cell and the neuron activates if the combination of the scaled signals cross an activation threshold.

The neural network model is a direct graph consisting of interconnected nodes, called neurons, that represent the computational units of the model. The neurons are organised in layers and the neural network is composed of multiple interconnected layers of neurons. An input layer of neurons receives some input values and propagate them to the neurons in the first intermediate layer of the network, called hidden layer. The output values from the last hidden layer are finally received by the output layer of neurons, which computes the final outputs of the network. To be consistent with brain-inspired terminology, the outputs of the neurons in the neural network are referred to as activations.

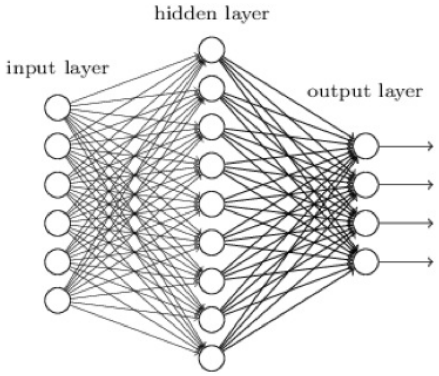


Figure 1.4: Neural network model

If the graph of the neural network is acyclic, the neural network is called feed-forward, while if the graph is cyclic the network is defined recurrent and the outputs of the network do not depend only on the current inputs but also on the previous ones.

1.2.1 Training and Inference

As an instance of a supervised machine learning classification algorithm, the neural network learns to perform its predictions by exploiting pairs of inputs and desired outputs, so that the model becomes capable of producing the desired output given an input it has never seen before, i.e. generalising from known examples.

In the case of neural networks, the process by which the network learns to perform its classification task is called training and it consists of determining the value of the weights and biases used by the neurons in the neural network. Using the model parameters determined during the training phase, the neural network can perform the classification of unlabelled input data in a process called inference.

The outlined training process is inspired to the way the human brain is believed to learn, that is by adjusting the strength of the synaptic connection between interconnected biological neurons in response to a learning stimulus. In particular, different weights associated with the synapses result in different responses of the neuron to the incoming signals.

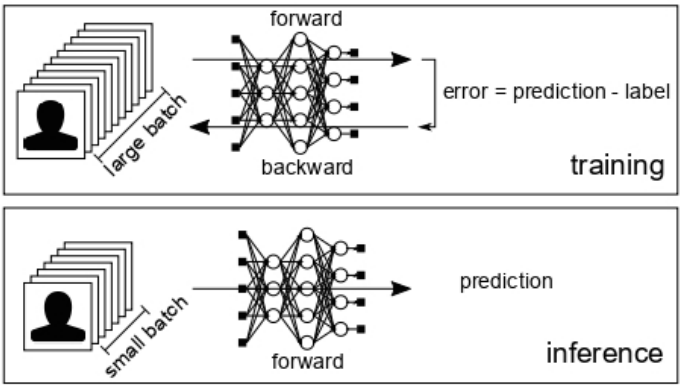


Figure 1.5: Training and Inference [1]

The training process of a neural network consists of three main phases: a forward pass, a backward pass and a parameters update pass. During the forward pass, an input is fed into the network that generates an output. At the beginning of the backward pass, the loss for the current input is computed, by comparing the generated output and the desired output. At this point, an algorithm called backpropagation is used to compute the gradient of the loss with respect to each parameter (weights and biases) of the network, by applying the chain rule of calculus and passing values backwards through the network. As a last step, the network parameters are updated by using an optimisation method called stochastic gradient descent, that exploits the computed gradient of the loss with respect to each weight or bias to adjust the corresponding weight or bias according to the formula $\theta^{t+1} = \theta^t - \eta \frac{\partial L}{\partial \theta^t}$, where the scaling factor η is referred to as learning rate. The training process is repeated iteratively until a local minimum of the loss is found or a convergence criterion is met.

As a matter of fact, the ultimate goal of the training is to determine a set of network parameters (weights and biases) that minimises the average loss over the training dataset. In order to improve the speed and robustness of training, the parameters update pass is not performed for every input data, but for multiple sets of input data, called batches. Thus, an average gradient over the batch of input data is used in the stochastic gradient descent algorithm. The algorithm to minimise the average loss of the network is called stochastic because each batch gives an approximate estimate of the average gradient over all examples, that would be required to minimize the average loss over the whole training set. Typically, multiple cycles through the full training dataset are performed, that means the network is trained for multiple epochs. During one epoch, the training loss and accuracy are available for each batch of training data, while at the end of each epoch the accuracy of the model is evaluated on a validation set consisting of unlabelled input data, in order to asses the generalisation capability of the currently trained neural network.

1.2.2 Training Issues

While training a neural network, one of the most common problems is overfitting, i.e. the model overfits the data in the training set by learning to detect noisy data trends that are not found in the data in the test set. As a consequence, the trained neural network becomes accurate in classifying the training data, but is not able to generalize well on the unseen test data. Typically, overfitting is caused by an overly complex model with many parameters or by a training set that is not well representative of the data from the problem domain. To combat overfitting, a number of techniques can be used. The early stopping technique stops the training process after a given number of cycles when the model begins to overfit, that is the test error starts to increase. By acting on the training set, the data augmentation technique generates new realistic training examples from existing ones, increasing the size of the training set and reducing overfitting. To improve the generalisation capability of a neural network, the complexity of the model can be decreased by reducing the number of parameters through the removal of layers and neurons from the network. L1 and L2 regularization techniques can be exploited to reduce the freedom of the model by adding a penalty on the weights by means of a regularisation term in the loss function. Finally, the most popular technique to reduce overfitting in neural networks is probably dropout, which consists in temporarily removing neurons from the network during the training process in a random way with a given dropout rate, so that the model will be less likely to fit the noise of the training data.

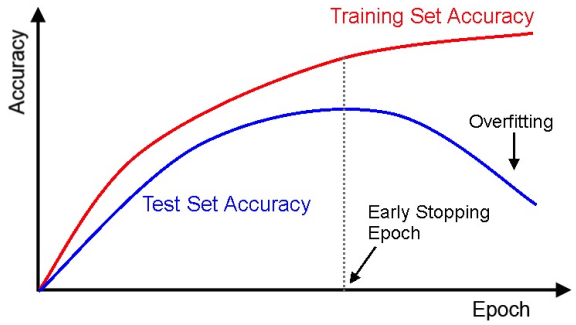
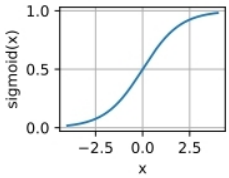


Figure 1.6: Overfitting problem

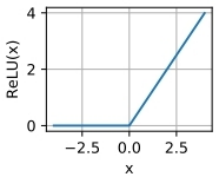
As the number of layers in the neural network increases, the training process can suffer from a difficulty referred to as vanishing gradient problem, that prevents the network from learning and tuning the parameters of the earlier layers using the gradient descent algorithm. In the vanishing gradient problem, the gradients of the network loss with respect to the parameters in the early layers become extremely small because of the peculiar activation function used in the neurons of the network. Actually, many activation functions map their input in a very small output range in a non-linear fashion, so that a large change in the input produces a small change in the output, leading to a small gradient. To avoid the vanishing gradient problem, activation functions that do not have the property of squashing their input space into a small output region can be adopted: a popular choice is the Rectified Linear Unit that maps its input to the highest non-negative value.

$$y = \frac{1}{1 + e^{-x}}$$



(a) *Sigmoid*

$$y = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$



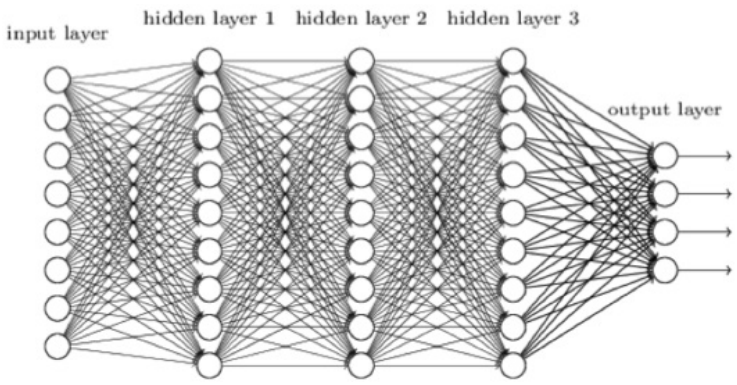
(b) *Rectified Linear Unit*

Figure 1.7: Non linear activation functions [2]

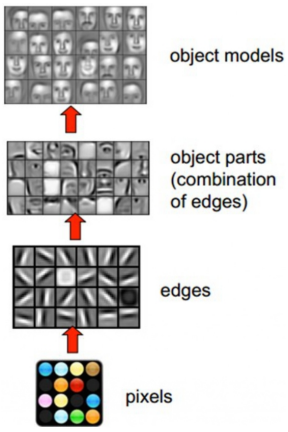
1.3 Deep Learning and Deep Neural Networks

Deep learning is a subfield of Machine learning within the domain of neural networks. The neural networks used in deep learning have more than one hidden layer and they are defined in literature as deep neural networks. In particular, the number of network layers can range from five to more than a thousand.

Traditional machine learning algorithms are limited in their ability to process data in their raw form. As a matter of fact, feature extraction is required to transform the input raw data into a suitable representation that the machine learning algorithm can use to detect or classify patterns in the input. By stacking multiple layers of neurons, a deep neural network is able to directly process raw data and automatically extract representations of data with multiple levels of abstraction. The ability of the deep neural network to extract features from raw data is obtained by learning multiple layers of feature detectors from labelled data using a supervised learning procedure.



(a) *DNN model*



(b) *Hierarchical feature learning*

Figure 1.8: Deep neural networks

Being capable of learning a feature hierarchy, deep neural networks can achieve superior performance in a wide range of tasks with respect to shallower neural networks. In particular, dnns have significantly improved the accuracy of computer vision tasks like

image classification and object detection, as well as speech recognition and natural language processing tasks. The superior accuracy of dnns in multimedia applications, however, comes at the cost of high computational complexity. In particular, dnns require a large amount of weighted sum computations because of the high number of layers of neurons.

In an image classification task, an input image in the form of an array of pixel values is fed into the first layer of a dnn, which extracts low-level features, such as lines and edges, in different locations of the image. The second layer detects higher-level features like arrangements of edges and shapes. The third layer determines the presence of parts of predefined objects and finally the subsequent layers detect objects as combinations of the previously extracted parts. The output layer of the dnn provides a vector of probabilities, one for each object class and the class with the highest probability represents the most likely class of object in the image.

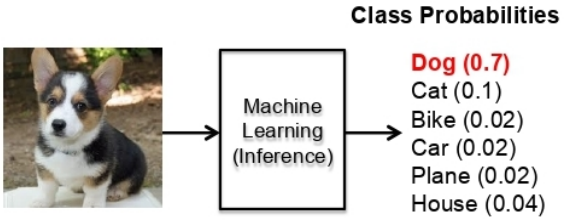
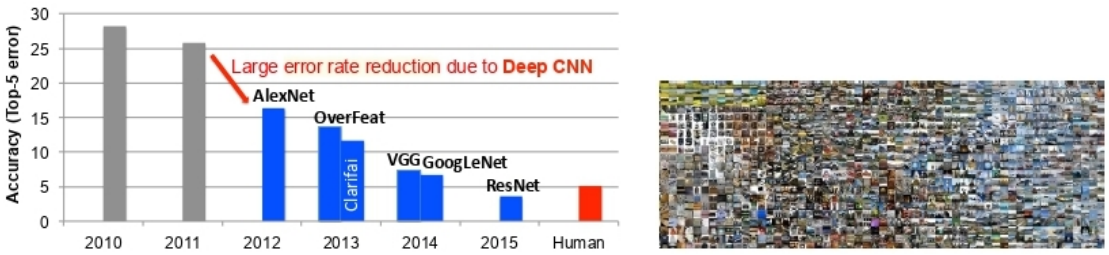


Figure 1.9: Image classification task [3]

The success of the deep neural networks is shown by the performance of the best algorithms in the ImageNet challenge, a competition involving an image classification task with 1000 object categories. As illustrated by the results from the ImageNet challenge, in 2012 a dnn model named AlexNet reported a dramatic improvement of the Top-5 accuracy by approximately 10% with respect to the state-of-the-art. The accomplishment of AlexNet led to an increasing number of deep learning algorithms competing in the challenge and in 2015 the dnn called ResNet overcame human-level accuracy with a Top-5 error rate below 5%. Top-5 accuracy means that a classification algorithm is considered to correctly classify an image if the correct class of the object in the image is one of the 5 classes with the highest scores provided by the algorithm.



(a) *Top-5 error results* **(b)** *ImageNet dataset*

Figure 1.10: ImageNet Challenge [3]

1.3.1 DNN Layers

In deep neural networks, neurons are organised in layers with various characteristics and shapes.

A fully-connected layer is a layer where neurons are organised in a 1-D array shape, such that each neuron in the layer receives the activations of all the neurons in the previous layer. The total number of weights of the layer is given by the product of the number of neurons in the current layer and the number of neurons in the previous layer, that is the number of inputs of the current layer. If some of the inter-layer connections are removed by setting the corresponding weight to zero, the layer is referred to as a sparsely-connected layer.

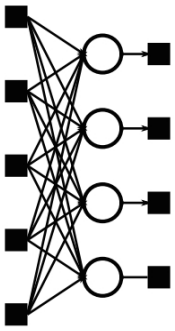


Figure 1.11: Fully-connected layer [1]

A convolutional layer is a windowed and weight-shared layer of neurons, where each neuron receives a fixed-size window of input activations from the previous layer and the same set of weights is shared for every neuron in the current layer. The neurons in a convolutional layer are organised in a 2-D grid called feature map and a convolutional layer may consist of multiple feature maps, whose number is referred to as number of output channels of the conv layer. Each neuron of the same feature map processes a different group of input activations of dimension $H_k \times W_k \times C_i$, where C_i is the number of input channels of the current conv layer, by using the same kernel of $H_k \times W_k \times C_i$ weights. The total number of weights of a conv layer is given by the product of the kernel size of each feature map and the number of feature maps. Mathematically, the filtering operation performed by the neurons of a feature map is a discrete high-dimensional convolution of the weights kernel and the input activations.

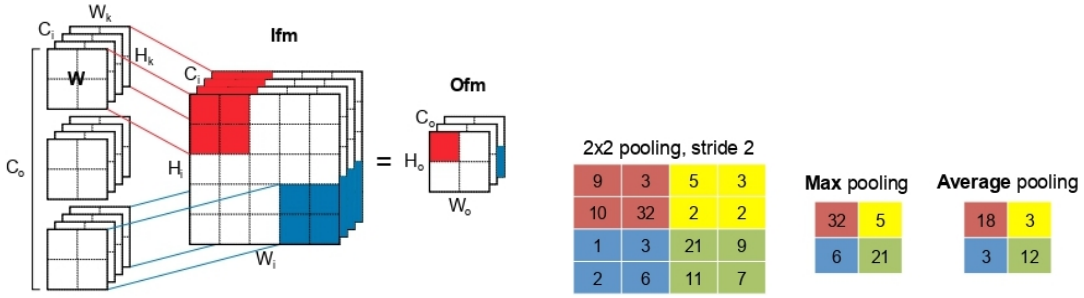
In addition to fully-connected and convolutional layers, a number of auxiliary layers can be found in a dnn, i.e. non-linearity, normalisation and pooling layer.

A non-linear activation function is typically applied after each fully-connected or convolutional layer in order to introduce non-linearity into a deep neural network model. Possible non-linear functions are sigmoid or hyperbolic tangent, as well as rectified linear unit, that is widely adopted in the current dnns for its simplicity and ability to allow fast training.

A batch normalisation layer is applied after the fully-connected or convolutional layer in order to obtain a normalised distribution of the layer activations, with zero mean and

unit standard deviation. By inserting a normalisation layer between the fc or conv layer and the non-linear function, saturation of the values inside the network due to saturating non-linear functions is avoided, so that fast training and improved accuracy are obtained.

A pooling layer is applied after a convolutional layer with two primary purposes. Firstly, the pooling layer allows to reduce the dimensionality of the output feature maps of the conv layer and secondly, it gives the conv layer invariance to small shifts and distortions in the input 2-D data. For each output channel of the conv layer, equal-size adjacent non-overlapping grids are selected and a single value is computed for each selected grid, i.e. the maximum value or the average value depending on the pooling operation. In order to select non-overlapping sub-grids of each feature map, the size of the pooling receptive field is chosen equal to the stride used to scan each feature map of the conv layer.



(a) *Conv layer [2]* **(b)** *Pooling layers [3]*

Figure 1.12: Convolutional and Pooling layers

1.3.2 Convolutional Neural Networks

A Convolutional neural network is a particular type of deep feed-forward neural network that is widely adopted in computer vision tasks. A cnn is designed to process input data in the form of multiple arrays of values, such as a colour image represented by three 2-D arrays including the pixel intensities in the three colour channels RGB. A typical convolutional neural network consists of a series of stages, that are composed of three main types of layers: convolutional, non-linearity and pooling layer.

The layers allow to generate a representation of the input data with multiple levels of abstraction. In particular, the convolutional layer detects local patterns of features and the pooling layer creates an invariance of the network to small changes in position and appearance of the elements in the input data. As a matter of fact, in array data such as images, local group of values forming patterns are correlated and the same pattern can be detected in different parts of the array. On top of the multiple stacked convolutional, non-linearity and pooling layers, a small number of fully-connected layers are typically applied in cnns used for a classification task, in order to obtain the class scores required to classify the input data. As the convolutional layers, the fully-connected layers apply filters on their input activations but without the windowing and weight sharing property of the conv layers. As with regular neural networks, the weights in all the filter kernels used in

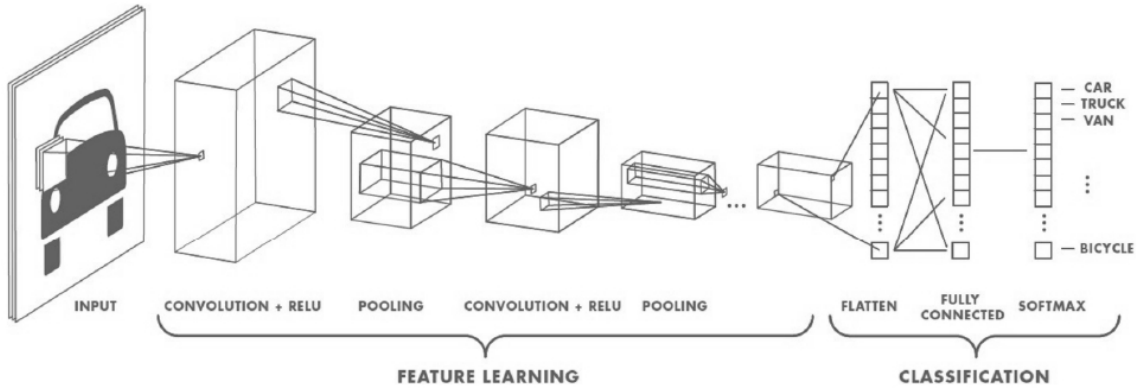


Figure 1.13: Convolutional neural network

cnn can be determined through a training process, by exploiting the stochastic gradient descent method and the back propagation algorithm.

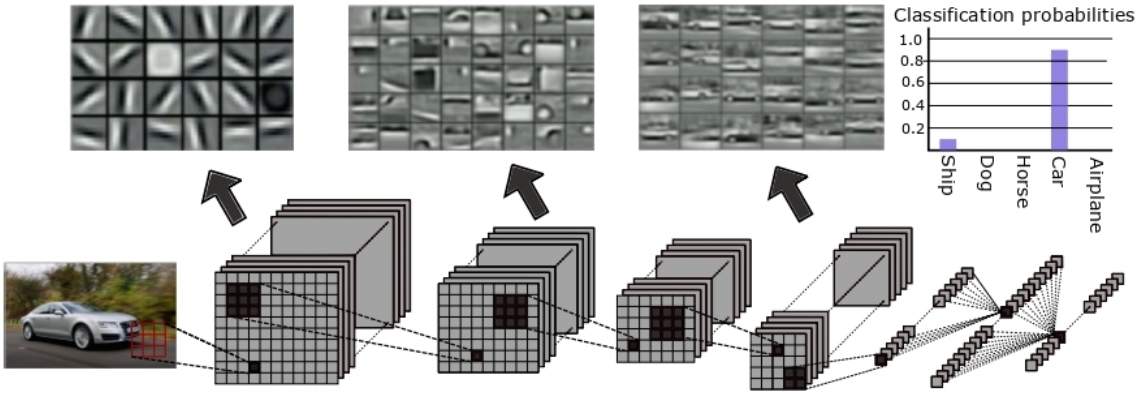


Figure 1.14: CNN hierarchical feature detection

The first popular convolutional neural network has been historically LeNet, that performs the task of handwritten digits classification. The version LeNet-5 consists of 5 main layers, i.e. three convolutional layers and two fully-connected layers. Each convolutional layer applies 3-D filters of size 5×5, with 6 filters for the first conv layer, 16 filters for the second conv layer and 120 filters for the last conv layer. After each conv layer, a sigmoid non-linearity is used and after the first two conv layers, average pooling is performed with field size 2×2 and stride of 2. On top of the previous stages, two fully-connected layers are inserted for classification purposes, with the first fc layer using a sigmoid activation function and the second fc layer providing the class probabilities for the input numerical digit image through a softmax function.

One very successful cnn model is AlexNet, that was the first cnn to win the ImageNet challenge image classification task in 2012 by outperforming the earlier algorithms. AlexNet consists of five convolutional layers and three fully-connected layers. The first conv layer filters the 224×224 input colour image with 96 kernels of size $11 \times 11 \times 3$ with a stride

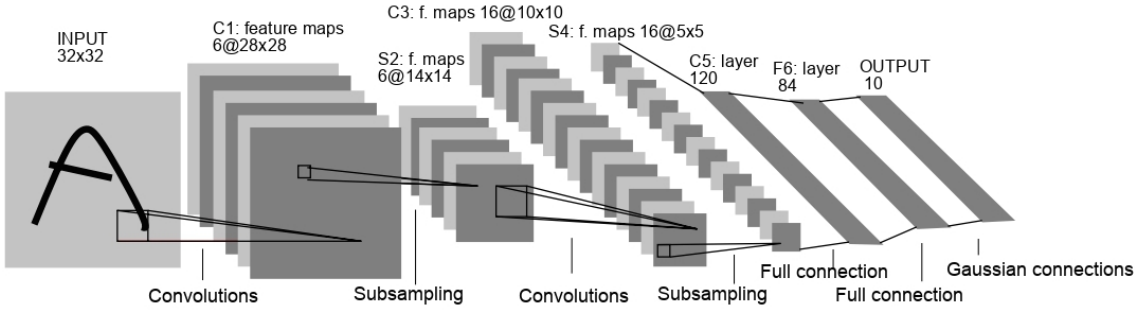


Figure 1.15: LeNet-5 architecture [4]

of 4 pixels to reduce computation complexity. The second conv layer filters the output of the first conv layer with 256 kernels of size $5 \times 5 \times 48$. The third conv layer applies 384 kernels of size $3 \times 3 \times 256$, the fourth conv layer uses 384 kernels of size $3 \times 3 \times 192$ and the last con layer has 256 kernels of size $3 \times 3 \times 192$. A ReLU non-linearity is applied after each convolutional layer. After the first, second and last conv and non linear layers, a max pooling is performed with field size 3×3 and stride of 2. Before the pooling layer in layers 1 and 2, a local response normalisation layer is used to normalize the distribution of the activations. To reduce the amount of weights and computation, the input channels of the conv layers 2, 4 and 5 are split into 2 groups so that each filter in these layers has half the number of weights. Following the convolutional stages, the first two fully-connected layers have 4096 neurons each and ReLU activation function, while the third and last fc layer includes 1000 neurons with a 1000-way softmax function to produce a probability distribution over the 1000 possible class labels of the input image.

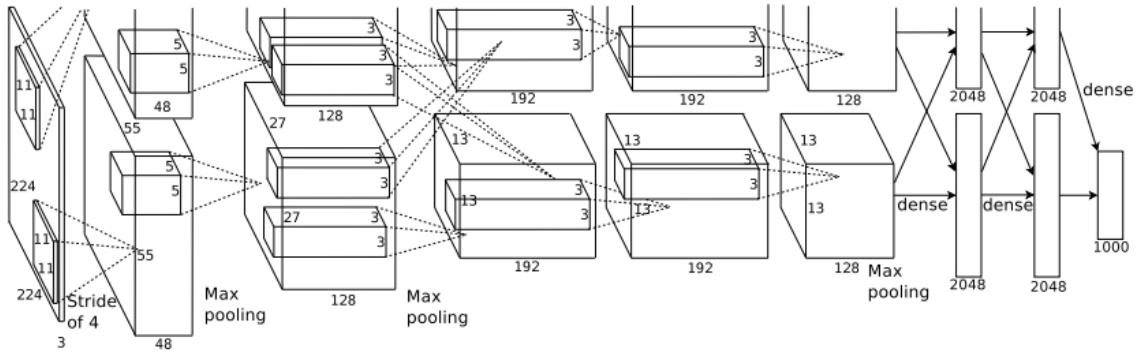


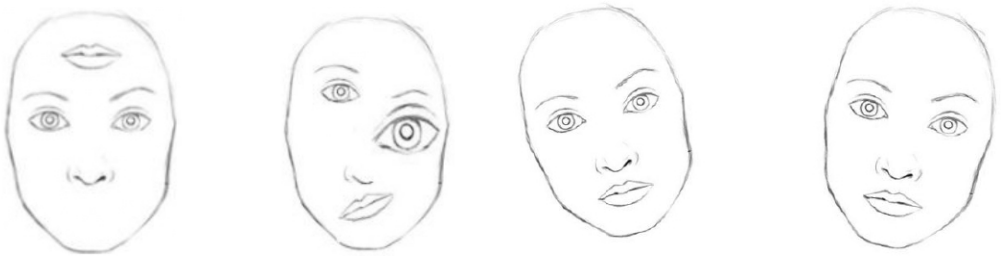
Figure 1.16: AlexNet architecture [5]

ResNet has been the first convolutional neural network to exceed human-level accuracy in the ImageNet challenge in 2015, with a Top-5 error rate below 5%. The superior performance of ResNet is achieved by increasing the depth of the convolutional neural network up to 50 or 152 layers. Increasing the depth of the network, however, leads to the vanish gradient problem during the training process. To combat the vanishing gradient issue, the ResNet model introduces residual blocks or shortcut modules that include an

identity or skip connection, so that the output of a layer can be added to the output of a deeper layer in the network. By using a skip connection, it is possible to provide an alternative path for the gradients of the loss during the backpropagation algorithm and prevent the vanishing of the gradients in the earlier layers of the network.

1.4 An Overview of Capsule Networks

Convolutional neural networks have a limited viewpoint generalisation capability, that is cnns are able to detect the presence of features in the input data, but less effective at extracting spatial relationships among features, such as perspective, relative position, orientation and size. As a consequence, cnns show scarse ability to generalize to new viewpoints in the test data, so that the classification results provided by the cnns are not viewpoint invariant, as it would be desirable in an image classification task. To force the cnn to generalise to novel viewpoints, a possible solution is to train the network over an extensive amount of labelled training data, which is extended by using data augmentation to generate multiple images of the same object from different viewpoints. Using a large training set, however, implies a slow training process.



(a) *Features spatial relationship* **(b)** *Novel viewpoints*

Figure 1.17: CNN limited generalization capabilities

Another key limitation of a cnn used for image classification is the difficulty in detecting overlapping features in the input data. Actually, the pooling layer approach used in cnns allows for the detection of the most dominant feature, while discarding the other overlaid features.

To correctly classify an input image, a classification system should not only detect the likelihood of features, but also determine the spatial properties of features to verify consistence in the orientation and size of the various elements in the image. By taking into account feature spatial information, for instance, the sketch of a distorted human face is not misclassified as a regular human face and the image of a house from a different viewpoint is still classified as a house.

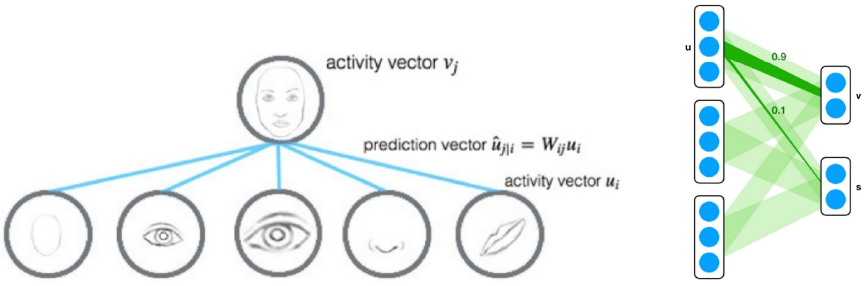
The limited capability of extracting spatial information about features is related to the scalar output of neurons in cnns. The pooling layer in cnns provides translation invariance, but does not enable the model to be viewpoint invariant and to detect overlaid features.

To solve the outlined cnn limitations, a novel type of deep neural network called CapsuleNet is introduced. Two main revolutionary ideas are proposed by a Capsule

network: multi-dimensional neurons, called capsules, in place of scalar output neurons used in traditional cnns and the Routing-By-Agreement algorithm that replaces the pooling operation traditionally adopted in the cnns.

A capsule is a group of neurons that is able to encode not only the likelihood but also the spatial properties of a specific feature in the input data. In particular, the output of a capsule is an activity vector, whose magnitude is the probability of detecting the feature and its components represent specific spatial properties of the extracted feature, such as position, orientation and size.

The Routing-by-agreement algorithm is an iterative algorithm that is used to connect capsules belonging to two subsequent capsule layers. The algorithm takes into account the agreement between lower-level capsules, so that the higher-level capsule that gets activated is the one with which the child capsules agree the most. As a measure of agreement between lower-level and higher-level capsules, the algorithm uses the dot product between the parent capsule output vector and each child capsule prediction vector for the parent capsule. By using an agreement approach to connect capsules, the child capsules representing lower-level features (parts) activates the parent capsule extracting a higher-level feature (whole) that is consistent with the spatial properties of the lower-level features.



(a) *Activity and prediction vectors* **(b)** *Routing-by-agreement*

Figure 1.18: Capsules

By combining the concept of capsules and the routing-by-agreement algorithm, Capsule networks show significantly better viewpoint generalisation than traditional cnns, as well as better accuracy in classifying images of overlaid objects. In particular, a Capsule network achieves viewpoint invariance by allowing for viewpoint equivariance of the capsules components, that encode the spatial properties of specific features in the input image, thus changing as the viewpoint of the objects in the image changes. As we read in [6], *"When the capsule is working properly, the probability of the visual entity being present is locally invariant — it does not change as the entity moves over the manifold of possible appearances within the limited domain covered by the capsule. The instantiation parameters, however, are "equivariant"— as the viewing conditions change and the entity moves over the appearance manifold, the instantiation parameters change by a corresponding amount because they are representing the intrinsic coordinates of the entity on the appearance manifold."* By slightly varying one dimension of a capsule and holding the others constant,

it is possible to determine what spatial property of the extracted feature each dimension of the capsule is capturing.

Capsule networks are more suitable than cnns in detecting overlapping objects because higher-level features are extracted from a weighted combination of lower-level features by means of the routing-by-agreement algorithm. On the other hand, the pooling layer in a cnn computes a general metric of a group of activations, such as the maximum value or the average value, thus not allowing to detect overlaid features based on their relevancy.

For a CapsuleNet to be used in an image classification task, a training process is required as in regular dnns to learn the model weight parameters. In particular, in a capsule layer, the trainable parameters are represented by transformation matrixes that allow to compute the predictions of lower-level capsules for higher-level capsules and to form meaningful parts-whole relationships.

1.4.1 Dynamic Routing Algorithm

In the following, an overview of the dynamic routing-by-agreement algorithm is reported. First of all, each lower-level capsule computes its prediction vectors for the higher-level capsules by means of multiplications between its activity vector and transformation matrixes. Secondly, the lower-level capsules send they prediction vectors to the parent capsules without any routing preference at this point, i.e. the prediction vectors of each child capsule are multiplied by equal routing coefficients. Then, the weighted prediction vectors for each parent capsule are summed and the length of the summed vector is large if the orientations of the prediction vectors agree. A non-linearity, referred to as squash activation function, is applied to the weighted sum vector to compute the activity vector of the parent capsule. The squash function ensures that the magnitude of the activity vector is below 1 to represent the existence probability of the extracted feature and preserves the orientation given by the weighted sum of prediction vectors, that is an agreed orientation from the lower-level capsules. At this point, the routing coefficients between child capsules and parent capsules are re-calculated, so that the child capsules send their predictions preferably to the parent capsule with which they agree the most. As a measure of agreement between capsules, the dot product is used between the parent capsule activity vector and the child capsule prediction vector for that parent. The weighted sum of prediction vectors for each parent capsule and the non-linear activation are performed again. The routing-by-agreement algorithm is repeated in practice for three iterations, that are typically enough to assign all the child capsules to their preferred parents.

1.4.2 Pros and Cons of Capsule Networks

Capsule networks achieve state-of-the-art accuracy in an image classification task on the MNIST dataset, that consists of grayscale images of handwritten numerical digits. Moreover, CapsuleNets demonstrate considerably better results than convolutional neural networks at recognising highly overlapping digits from the MultiMNIST dataset, thanks to the use of the routing-by-agreement algorithm. By using routed equivariant capsules, Capsule networks are suitable at performing accurate image segmentation and object detection, that require to detect the precise location and pose of the objects. On the other

side, pooling layers in regular convolutional neuronal nets tend to loose spatial information about the objects throughout the network.

Method	Routing	Reconstruction	MNIST (%)	MultiMNIST (%)
Baseline	-	-	0.39	8.1
CapsNet	1	no	$0.34_{\pm 0.032}$	-
CapsNet	1	yes	$0.29_{\pm 0.011}$	7.5
CapsNet	3	no	$0.35_{\pm 0.036}$	-
CapsNet	3	yes	$\mathbf{0.25}_{\pm 0.005}$	**5.2**

Figure 1.19: CapsNet classification test error on MNIST and MultiMNIST [6]

By encoding the instantiation parameters of the objects in the capsule dimensions, CapsuleNets require less training data than cnns to reach the same accuracy in an image classification task and their classification results are invariant to affine transformations, such as rotations and translations.

Capsule networks are significantly more robust to adversarial attacks than a baseline convolutional neural network. The basic idea behind an adversarial attack is to feed the network with adversarial data, i.e. data that has been slightly modified to trick the classifier into making the wrong classification.

However, Capsule networks have a few drawbacks. First of all, CapsuleNet classification accuracy on more complex image datasets, like CIFAR-10, is not yet state-of-the-art.

Capsule networks are also quite slow to train, mainly because of the inner loop contained in the routing-by-agreement algorithm, that is an iterative procedure.

Finally, CapsuleNets suffer from a problem called “crowding”, that means they are not able to distinguish between two objects of the same type placed close to one another in the input image, because there is only one capsule of any given type in a given location.

The superior representation capabilities of Capsule networks with respect to cnns come at the cost of higher computational complexity. As a matter of fact, the ratio between the number of multiply-and-accumulate operations to perform an inference pass and the memory footprint is a good indicator of the higher compute-intensive nature of CapsuleNets compared to traditional cnns, as shown in figure 1.20.

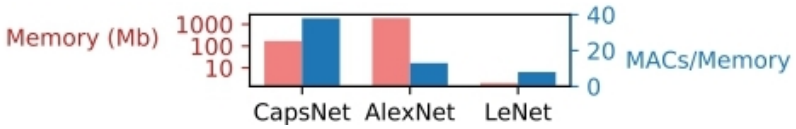


Figure 1.20: CapsNet computational complexity [7]

The high computational cost is due to the larger dimension of the constituent elements of the CapsNets and the high computational effort required to dynamically route the capsules between successive layers. Actually, capsules output an activity vector such that each neuron in a capsule encodes a spatial property of the entity associated with the capsule and the length of the vector estimates the probability that an instance of the entity

is present in the input data. On the contrary, neurons in cnns can only detect features without extracting valuable information about their spatial properties and relationships. In addition to the distributed representation of capsules, the dynamic routing algorithm is an iterative process, that requires multiple iterations to assign parts to their appropriate wholes. The routing-by-agreement in CapsNets is a superior method than the primitive routing mechanism of pooling used in cnns, since it allows to selectively assign lower-level capsule predictions to higher-level capsules based on their agreement and to build a part-whole hierarchy off the ground in a parse tree-like structure. As a consequence, the routing algorithm applied in CapsNets is an effective way of achieving viewpoint generalisation and detective overlapping features, while pooling provides only local translation invariance and tends to loose information about the precise location and pose of the detected features.

1.4.3 CapsNet model for the MNIST dataset

The Capsule network model proposed in the 2017 paper [6] by Geoffrey Hinton et al. introduced the novel idea of capsule-based neural networks in the field of Deep learning. The presented Capsule network consists of three main layers, i.e. a convolutional layer, a convolutional capsule layer and a fully-connected capsule layer. The model is designed to perform an image classification task on the MNIST dataset, including 28×28 grayscale images of handwritten digits.

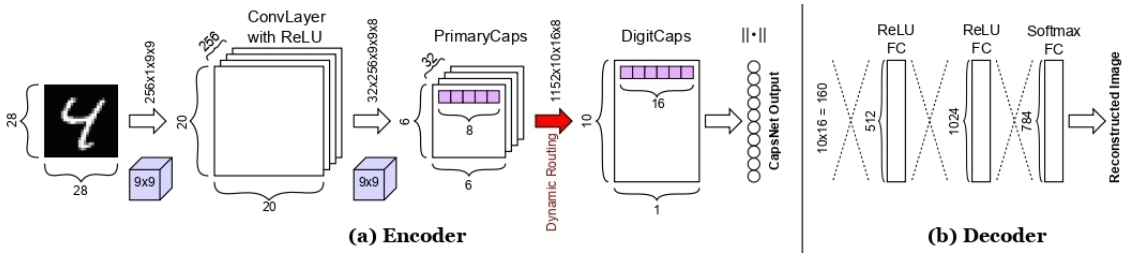


Figure 1.21: CapsNet architecture proposed by Hinton et al. [8]

The first convolutional layer processes the input image with 256 $9 \times 9 \times 1$ kernels to extract features and a ReLU activation function is applied to the output feature maps. The convolutional capsule layer, also referred to as primary capsule layer, applies a second convolutional layer with 256 $9 \times 9 \times 256$ kernels using a stride of 2 and ReLU activation. The array of 256 output feature maps is then reshaped in 32 channels of 8 feature maps each, to obtain 32 8-dimensional capsules for every location. The activity vector of each primary capsule is determined by applying a squashing function, that preserves the vector orientation but squashes it to ensure that its length is between 0 and 1, as the vector length is meant to represent the probability that the entity associated with the capsule is present in the input image.

The primary capsules are organised in a 6×6 grid with 32 8-dimensional capsules in each cell of the grid. In particular, the spatial capsules in the same channel share the same 8 kernels since they look for the same patterns in different locations, while the capsules across the channel direction are computed by using different kernels as they represent

various visual entities in the input image. A key property of the squash activation function is that it is a capsule-wise function, not an element-wise function as ReLU.

The final fully-connected capsule layer, also called digit capsule layer, consists of 10 output capsules, one for each digit class of the MNIST dataset. The digit capsules output 16-dimensional activity vectors, whose length represents the class probability of the digit in the input image. The digit capsule layer performs the dynamic routing-by-agreement algorithm by connecting the primary capsules to the digit capsules.

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\boldsymbol{u}}_{j|i}$, r, l)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax` computes
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash` computes
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Figure 1.22: Dynamic routing algorithm [6]

The activity vectors of the primary capsules are transformed into predictions for the digit capsules output vectors, by means of vector-matrix multiplications, $\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\,\mathbf{u}_i$. Specifically, each primary capsule 8-dimensional vector is multiplied by 10 8×16 weight matrixes, to obtain 10 16-dimensional prediction vectors for the digit capsules. The weight matrixes represent part-whole relationships between each primary capsule and each digit capsule and allow to compute the pose of a higher-level feature from the pose of a lower-level feature. Each primary capsule is connected to the digit capsules and its connection strength is represented by 10 coupling coefficients, that are computed by feeding 10 raw routing coefficients into a non-linear softmax function, as shown in equation (1.1). At the first routing algorithm iteration, the raw coefficients are set to 0, so that the coupling coefficients are set to 0.1. For each digit capsule, the prediction vectors of the primary capsules are summed by weighting each prediction vector with the corresponding coupling coefficients, $\mathbf{s}_j = \sum_i c_{ij}\,\hat{\mathbf{u}}_{j|i}$. The squash activation function is then applied to the summed vector to determine the activity vector of the digit capsule, as shown in equation (1.2).

$$c_{ij} = \frac{e^{b_{ij}}}{\sum_{k=1}^{n} e^{b_{ik}}} \tag{1.1}$$

$$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{1.2}$$

At this point, the coupling coefficients of each primary capsule are updated based on the agreement between the primary capsule predictions and the digit capsules outputs. For each primary capsule, the dot product between each prediction vector and the corresponding digit capsule output vector is computed, then the raw routing coefficients are updated by adding the computed dot product to their previous value and finally the updated coupling

coefficients are determined by applying the softmax function to the updated raw coefficients. After multiple iterations, the digit capsule with the largest activity vector length in the final capsule layer corresponds to the digit class with the highest probability.

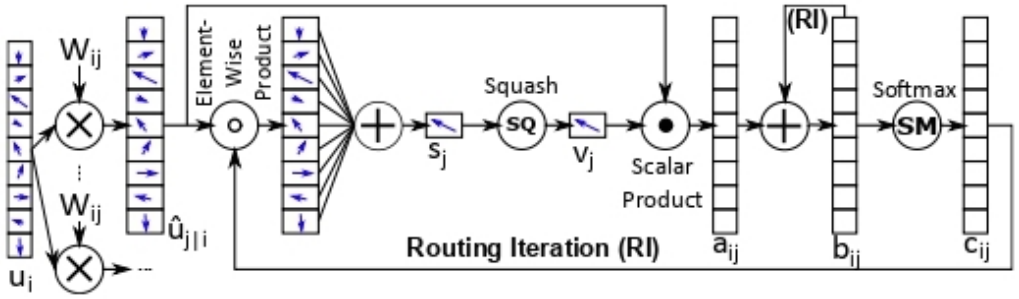


Figure 1.23: Graphical representation of the dynamic routing algorithm [7]

As in regular classification neural networks, the Capsule network model can be trained by adopting a supervised learning procedure and minimising the loss of the network over a training set. During the training process, the CapsNet learns the kernel weights used in the convolutional layer and the convolutional capsule layer, as well as the weights included in the transformation matrixes that are involved in the routing algorithm used in the final fully-connected capsule layer.

The peculiar loss for the CapsNet is the sum of multiple loss contributions, with one separate loss term for each digit capsule, called margin loss. The particular choice of loss function makes it possible to detect multiple digit classes in the same input image. The margin loss is computed by using a different expression for the correct digit class and the absent digit classes. In particular, for a given training example, the margin loss for the correct class is null if the network predicts the correct class with a probability higher than 0.9, while the margin loss for the wrong classes gets assigned a null value if the network estimates a probability lower than 0.1 for those classes.

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

$$\text{Total Margin Loss} = \sum_{k=1}^{n} L_k$$

On top of the digit capsule layer, a decoder network is inserted during training as a regularisation technique, to reduce the risk of overfitting and improve generalisation to new examples. The decoder is composed of three fully-connected layers: the first two layers have a ReLU activation function, while the last layer is sigmoid activated.

The last fully-connected sigmoid layer consists of 784 neurons, whose scalar outputs represent the pixel intensities of the reconstructed input image. The squared difference between this reconstructed image and the input image gives the reconstruction loss. By including the reconstruction loss in the loss of the Capsule network during training, the decoder learns to reconstruct the input image from the digit capsule corresponding to the correct digit class and the Capsule network is forced to preserve all the information

required to reconstruct the image up to the top digit capsule layer. When the inference is performed for classification purposes, the reconstruction network can be removed.

$$\text{Reconstruction Loss} = \text{MSE}(\text{reconstructed image, input image})$$
$$\text{Total Loss} = \text{Total Margin Loss} + 0.0005 \cdot \text{Reconstruction Loss}$$

One useful property of the Capsule network is that the capsule activation vectors are often interpretable, i.e. it is possible to determine what property of the detected entity each dimension of the capsule is encoding. To find out what physical parameters the individual capsule dimensions represent, one of the 16 dimensions of the digit capsule activity vector can be gradually modified and the decoder network can be used to obtain the reconstructed image from the digit capsule. In particular, the dimensions of the output capsules represent variations in the way the digit of the corresponding class is instantiated, such as scale, stroke thickness, width, localised skew and digit-specific variations like the length of the tail of a 2, the length of the ascender of a 6 or the size of the higher circle in a 8. By demonstrating that the dimension perturbation of digit capsules leads to a global transformation of the reconstructed image, Capsule networks show transformation equivariance.

1.4.4 DeepCaps: a Deep CapsNet model

The Capsule network model proposed by Hinton et al. in [6] shows limited classification ability on datasets more complex than MNIST, such as CIFAR-10. In particular, the CapsNet model in [6] achieves 10.6% error on CIFAR-10, which is higher than state-of-the-art but it is a similar result to what was obtained by the first cnns. Therefore, the performance of CapsNet on complex benchmark datasets, such as CIFAR-10, is not at the same level as traditional cnns.

Actually, the CIFAR-10 dataset shows images with more complex data and backgrounds with respect to the MNIST dataset as it contains 32×32 colour images in 10 different classes, that represent animals and types of vehicles.

(a) *MNIST* **(b)** *CIFAR-10*

Figure 1.24: Benchmarking image datasets

To improve the classification accuracy of images with complicated data and background, two possible techniques for capsule networks are ensemble averaging and increasing the number of convolutional layers. In ensemble averaging, multiple networks are trained together and their prediction accuracy results are averaged at test time. On the other hand, increasing the number of convolutional layers before the final capsule layer allows to create a more complex image representation to deal with the higher dimensionality of CIFAR-10-like data.

In order to increase the performance of capsule networks on more challenging image datasets, a novel deep capsule network model, called DeepCaps, is introduced in [9]. The two key characteristics of the new capsule network architecture are skip connections and a novel dynamic routing algorithm based on 3-D convolution. In particular, skip connections are required to allow for good gradient flow during backpropagation, while 3-D convolution based routing is needed to stack multiple convolutional capsule layers in a computationally efficient manner. Inspired by the architecture of ResNet, the layers in DeepCaps are organised in a structure with skip connections, in order to combat the vanishing gradient problem arising from the increased depth and the layers non-linear activation functions. The idea behind the routing based on 3-D convolution is to route a localised group of capsules to a higher-level capsule, in order to achieve parameter reduction by sharing. The stride along the depth is equal to the capsule dimension.

In particular, each capsule tensor in a layer predicts all the capsule tensors in the layer above. The localised routing allows to route groups of neighbouring capsules and helps to handle richer datasets than MNIST.

On the contrary, the dynamic routing implemented in fully-connected capsule layers represents a computational bottleneck when it is necessary to stack multiple fully-connected capsule layers to develop deeper networks.

The DeepCaps model consists of an initial convolutional layer, four main middle stages, referred to as CapsCells and a final fully-connected capsule layer.

In an image classification task on the CIFAR-10 dataset, the 32×32 colour images are first rescaled to 64×64 images and then fed into the initial convolutional layer, that uses 128 $3 \times 3 \times 3$ kernels to extract features from the input image. The ReLU activated output feature maps of the convolutional layer are then passed to the CapsCells.

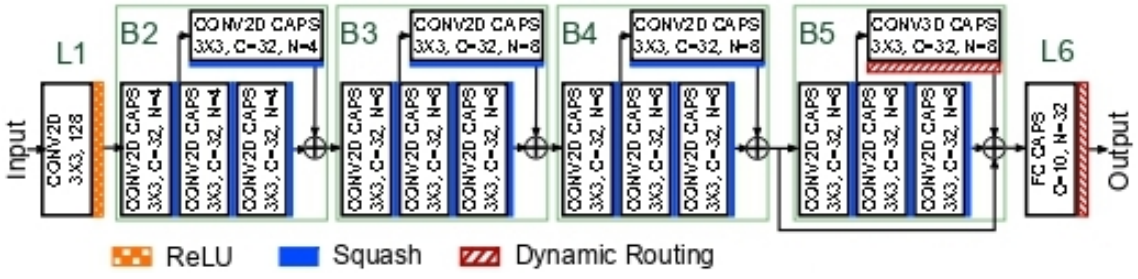


Figure 1.25: DeepCaps model architecture [7]

The first three CapsCells are composed of 4 convolutional capsule layers, with one layer

operating in parallel to the other three layers, so that the first layer output is convolved and skip connected to the last layer output in the cell. The convolutional capsule layers, also called ConvCaps, are squash activated. The fourth CapsCell is organised as the previous three cells, with three sequential and a parallel convolutional capsule layers, but here the parallel layer, called ConvCaps3D, performs the novel 3-D convolution based dynamic routing algorithm, where the number of routing iterations is set to 3. The capsules dimensionality used in the first CapsCell is 4, while the other three CapsCells involve layers of 8-dimensional capsules. The output of the fourth cell is flattened and concatenated to the output of the third cell. As a last step, the concatenated capsules are routed to the 10 class capsules by means of a fully-connected capsule layer where regular dynamic routing is performed. The class capsules output a 16-dimensional activity vector, whose length is the probability that the corresponding dataset class is present in the input image.

Compared to Hinton's CapsNet model, DeepCaps provides a combination of 2- and 3-dimensional convolutional capsule layers. By using 2-D convolutions, all the capsules along depth are filtered together by a 3-D weight kernel. On the other hand, in the 3-D convolutional capsule layer, the capsules along the depth dimension are voted separately by using a stride of the 3-D kernel along the depth equal to the capsule dimension.

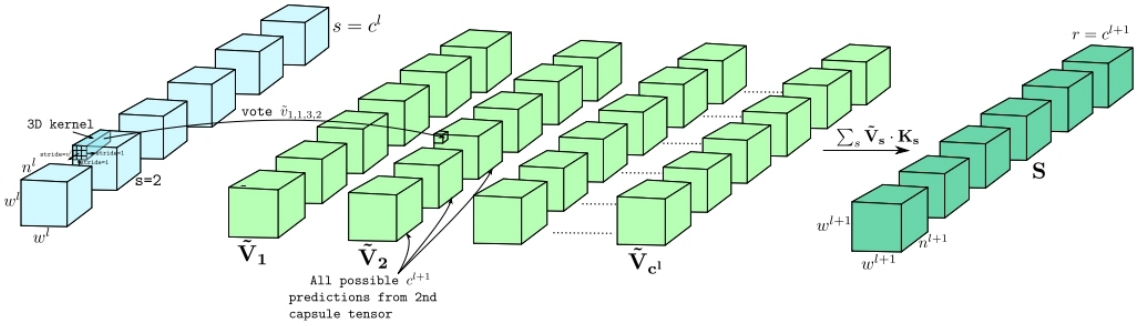


Figure 1.26: 3D convolution-based dynamic routing [9]

As with the CapsNet proposed in [6], the margin loss is used as the loss function for DeepCaps and a decoder network is inserted during training as a regularisation method to reduce model overfitting over the training set. Because of the increased model depth and complexity, a class-specific deconvolutional decoder is used which acts as a better regulariser for the deep capsule network.

1.5 Approximate Computing Methodology in Deep Learning

Deep learning algorithms stand out for their high workload and for being computation-hungry. In particular, DNNs are characterised by large memory footprint and high energy cost of memory accesses and computations. Among the most compute intensive operations that are performed in the inference pass are the multiplications.

Today's trend is to move towards mobile and embedded devices, which have tighter

power constraints, considering that many of them are battery-powered or rely on low-power systems. Actually, for the devices that are part of the so called Internet of Things, or IoT, energy consumption is a critical design criterion.

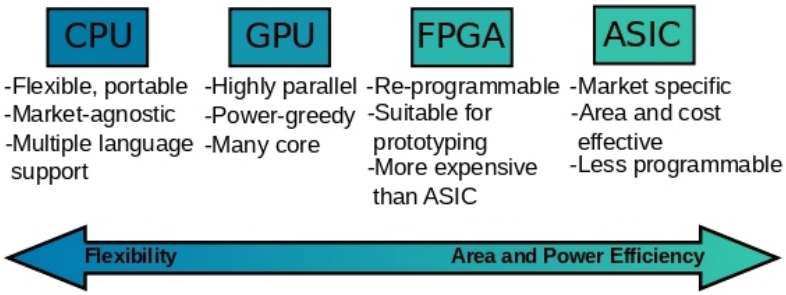


Figure 1.27: Power efficiency requirement of ASIC devices [2]

As a consequence, during the last few years, there has been a rising desire to deploy NNs on low-power mobile devices. On the other side, the trend to achieve higher accuracy has been the development of deeper networks, with a larger number of layers and parameters and this evolution hardly fits with the idea of employing neural networks for edge computing. In particular, CNNs have seen enormous advances in recent years, achieving close or even better accuracy than human level in several tasks, such as image classification. However, the significant amount of computational power used by CNNs prevents their widespread use in IoT and wearable devices.

Some optimisation methods exist to compress the models of NNs without affecting the achieved accuracy, such as network pruning. In particular, the pruning method exploits the redundancy of the parameters in NNs by setting to zero the parameters that do not affect the network accuracy.

One possible solution to deploy DNNs on edge devices is to apply a paradigm known as approximate computing, whose foundational idea is to trade quality for efficiency, at different abstraction levels.

In order to develop efficient implementations of DNNs, hardware metrics, such as area and energy, are included in the design as additional constraints beside application accuracy. The main goal is to minimise the cost in terms of area and energy, without incurring in considerable loss of accuracy, by realising a trade-off between area/energy costs and application accuracy.

Approximate computing is an attractive design technique that can be used to achieve low power, high performance and reduced circuit complexity by relaxing the accuracy requirement. Actually, the requirement of accurate results is not particularly strict for many error-tolerant applications that presents inherent error resilience, such as multimedia image processing. By relaxing the constraint of accuracy, performance and power consumption can be significantly improved using inexact computing. In many applications, such as computer vision, human perceptual limitations mitigate the effect of computational errors, so that approximate computing can be employed to improve hardware efficiency.

Approximate computing techniques can be applied at multiple levels including algorithms, software, architectures and circuits.

A consistent number of approximate designs have been proposed for addition and multiplication. Approximate division can save power and area by performing the transformation of the operands into the logarithmic domain, so that a trade-off between accuracy and hardware performance is achieved. To evaluate the accuracy of approximate designs, several error metrics are used, such as the error rate (ER), error distance (ED), mean error distance (MED) and maximum error distance (ED_{max}).

Approximate computing exploits a property of various applications referred to as error tolerance, that is the ability to accept erroneous outputs, provided the errors are within a certain application-specific threshold. Approximate techniques can take advantage of the application flexibility to reduce circuit area, delay and power consumption for a given quality constraint. The target in approximate computing is to obtain the highest computing efficiency for a given accuracy constraint or, viceversa, to achieve the highest quality for a specified efficiency requirement.

Among the applications that feature an intrinsic error-resilience property, there is a large number of applications, referred to as RMS, i.e. recognition, mining and synthesis.

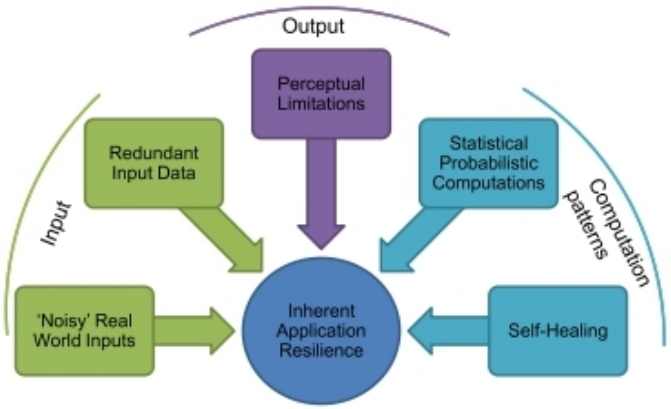


Figure 1.28: Sources of application error resilience [10]

In the big data and Internet of Things era, algorithms are meant to deliver good enough answers, quickly, at scale and with energy efficiency and approximate computing turns out to be a suitable approach to meet these competing goals.

1.6 Approximate Computing for non linear functions in DNNs

The neural network models require the use of a non-linear activation function at the output of each neuron. Without a non-linear activation function, a neural network is simply a cascade of linear algebra operations and it is unable to solve complicated non-linear problems. Therefore, several non-linear functions can be applied to the weighted sum of the inputs of a neuron, such as sigmoid, hyperbolic tangent or rectified linear unit functions.

Actually, the aforementioned functions make each neuron a non-linear unit, by mapping a large input domain to a narrow output domain. As a consequence, the network is capable of representing non-linear decision surfaces and can be profitably employed for classification tasks.

In the hardware implementation of NNs, the non-linearity is a critical factor that puts a constraint on either the occupied area or the inference time of the network.

In particular, a good indicator of the computational cost of non-linear operations is represented by the compute ratio per unit area, that is significantly lower for non-linear units than for linear operations blocks, thus revealing an inefficient area usage. As a matter of fact, non-linear operations in DNNs typically account for less than 1% of total operations, but they occupy 20% of the total area allocated to computational blocks.

An area and energy aware implementation of non-linear operations is key to improve the inference time in DNNs, especially for resource-constrained edge devices.

A variety of strategies are used to approximate the non-linear functions in neural networks, in order to obtain an efficient implementation. The main approaches to implement a non-linear function like sigmoid are Taylor series expansion, table look-up or piecewise linear approximation.

In particular, the sigmoid function can be evaluated by summing a truncated series expansion and this approach is likely to be expensive in terms of computation time and area. The function can be stored in a RAM or ROM and a table look-up scheme can be used to read the function values directly from the stored values. This technique requires large area for the implementation of the required memory. As an alternative method, piecewise linear approximation is exploited to approximate the non-linear function by using table look-up combined with linear interpolation, resulting in an efficient implementation that saves area and shows short latency time.

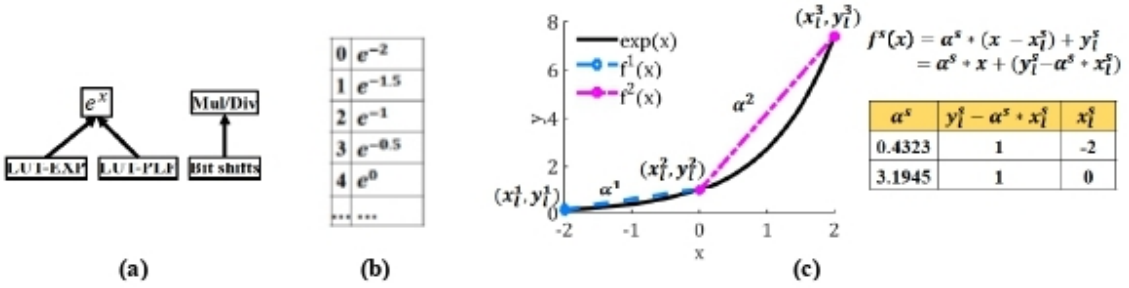


Figure 1.29: Approximation techniques for computationally-expensive operations: (a) approximate exponential and division operations, (b) look-up table, (c) piecewise linear approximation [11]

The sigmoid is a scalar-output activation function. An important non-linear vector-valued function is softmax, that is widely used in machine learning systems for multi-category classification tasks. The softmax function can be considered a generalisation of the sigmoid function that transforms a n-dimensional vector of real values into a n-dimensional vector, where each output vector entry is in the range (0, 1) and the entries add up to 1.

The exponent and division operations in softmax function are computationally expensive,

so an efficient hardware implementation of the function is required especially for edge devices. In particular, cost-efficient techniques to approximate the exponent and division calculations are needed, such as Taylor expansion, table look-up or piecewise approximation for the exponentiation and logarithmic domain transformation for the division operation.

The softmax layer is typically applied in the DNNs for multi-classification tasks. In particular, the softmax layer is placed after the last fully-connected layer in the network architecture in order to assign probabilities to multiple classes of objects.

Chapter 2

Softmax and Squash functions approximation

2.1 An Overview of the proposed Softmax and Squash function approximations

The approximations of softmax and squash are performed at the algorithmic level, by exploiting pre-existing research work and introducing innovative solutions for the computation of the two functions used in Capsule Networks, such as mathematical reformulation with powers of 2 for softmax or approximation of the Euclidean norm for squash.

The starting point of the work is to identify the softmax and squash functions in the context of the iterative routing-by-agreement algorithm, that is used to compute the activity vectors of the output capsules in the simple CapsNet architecture as it was presented in the 2017 paper "Dynamic Routing Between Capsules" [6]. First of all, the softmax function is used to determine the coupling coefficients between each primary capsule and all the capsules in the layer above, i.e. digit capsules. At the end of the algorithm, the squash function is applied to compute the activity vectors of the output capsules, whose length represents the probability that an instance of the associated class is present in the current input image fed to the CapsNet.

Mathematically, the softmax and squash functions are vector functions of several real variables, so they take as input n real numbers and return as output a vector of n real values. Each component of the input vector is processed according to the formulas (2.1) and (2.2). In particular, the number of components of the softmax and squash functions used in the Capsule Network architecture in [6] is 10 and 16 respectively. The softmax function has 10 components because the number of distinct classes is 10 in the benchmark dataset used for the image classification task, so each primary capsule is associated with 10 prediction vectors and 10 routing coefficients. On the other hand, the components of the squash function amount to 16 because the final layer (DigitCaps) has one 16-dimensional capsule per digit class, where each dimension out of 16 represents a particular feature of

the digits of that class, like stroke thickness, localized skew and width.

$$c_{ij} = \frac{e^{b_{ij}}}{\sum_{k=1}^{n} e^{b_{ik}}} \tag{2.1}$$

$$\mathbf{v}_j = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{2.2}$$

Procedure 1 Routing algorithm.

1: **procedure** ROUTING($\hat{\mathbf{u}}_{j|i}$, r, l)
2: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3: **for** r iterations **do**
4: for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$ ▷ `softmax` computes
5: for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6: for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$ ▷ `squash` computes
7: for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
return $\mathbf{v}_j$

Figure 2.1: Dynamic routing algorithm

Building upon extent research work and innovative ideas, three softmax and three squash approximations are derived by analyzing the algorithmic structure of the two functions. The key feature of each approximation is enclosed in its distinctive name.

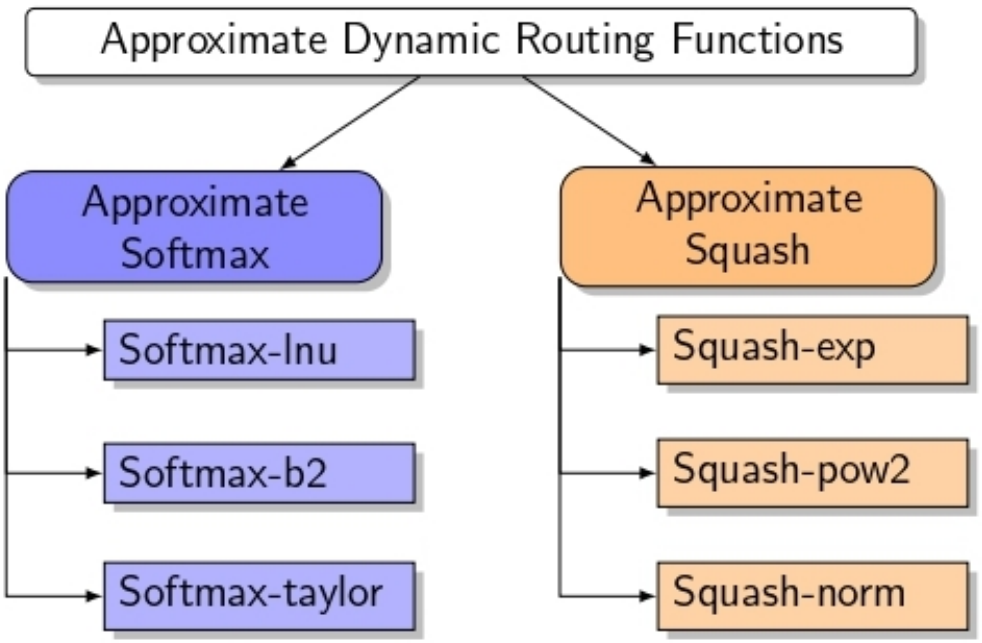
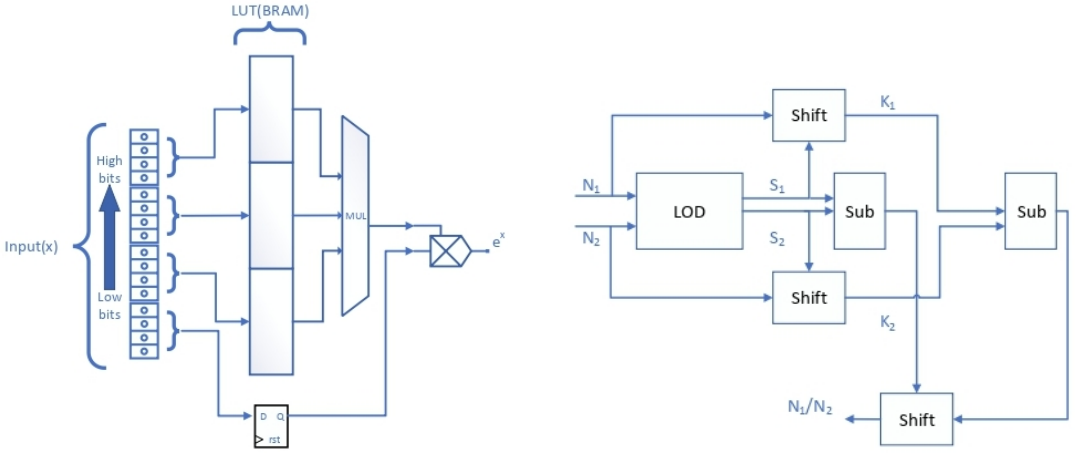


Figure 2.2: Proposed approximate softmax and squash functions

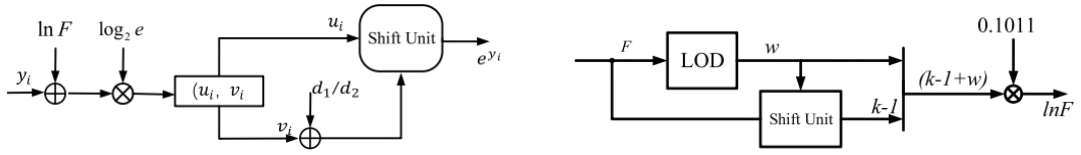
As far as softmax approximations are concerned:

- *Softmax_app_taylor* exploits look-up tables and Taylor series expansion for the exponential computation and performs division in the logarithmic domain;
- *Softmax_app_lnu* makes use of mathematical domain transformation, a natural logarithmic unit and a natural exponential unit;
- *Softmax_app_b2* builds on the previously mentioned approximation; the proposed approximation introduces the idea of modifying the mathematical expression of the softmax function by replacing natural exponentials with powers of 2 and that allows for hardware cost reduction thanks to the saving of two constant multipliers.



(a) *Exponent Unit* **(b)** *Division Unit*

Figure 2.3: Softmax-taylor architecture overview [12]

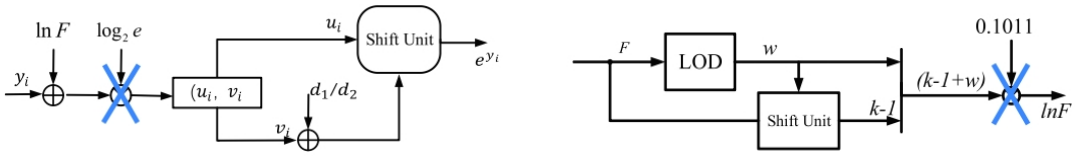


(a) *Exponent Unit* **(b)** *Natural Logarithm Unit*

Figure 2.4: Softmax-lnu architecture overview [13]

As regards the squash approximations:

- *Squash_app_exp* exploits a piecewise approximation of the squashing function with a natural exponential unit and a look-up table on two different ranges of Euclidean norm;
- *Squash_app_pow2* is a variant of the aforementioned approximation, where for values of norm in a given range the natural exponential unit is replaced with a less



(a) *Power-2 Unit* **(b)** *Log-2 Unit*

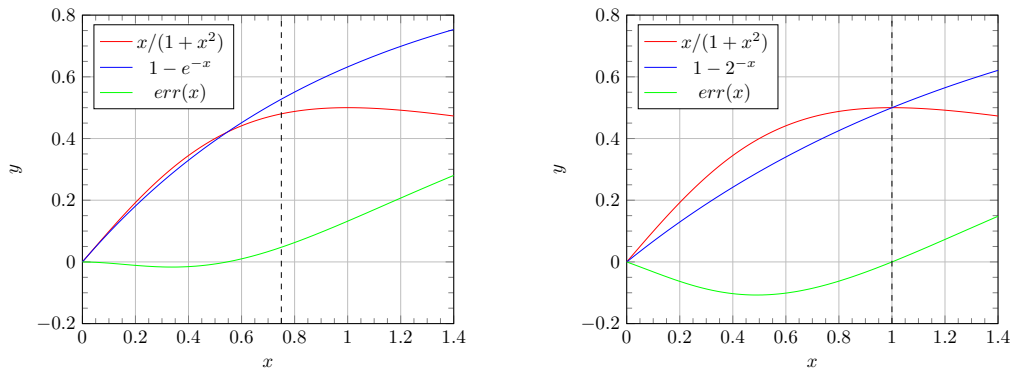
Figure 2.5: Softmax-b2 architecture overview

Softmax approximation	Hardware resources
app_taylor	EXPU (small exp-LUT, adder (+1), multiplier) Accumulator DIVU (LOD, 2 shift units, 2 subtractors, shift unit for LBC)
app_lnu	EXP Unit (subtractor, constant multiplier, shift unit) Accumulator LN Unit (LOD, shift unit, constant multiplier)
app_b2	POW2 Unit (subtractor, shift unit) Accumulator LOG2 Unit (LOD, shift unit)

Figure 2.6: Softmax units hardware resources overview

complex base-2 exponential unit, thus resulting in reduced cost of the final hardware implementation;

- *Squash_app_norm* builds on one of the several Euclidean norm approximations presented in the paper [14], i.e. the "Chaudhuri et al.'s" approximation, shown in (2.3); the proposed approximation removes the need of a square root operator for the norm computation and limits the number of multiplications to 1, regardless of the number of components, at the expense of comparisons among the component absolute values to determine the maximum.



(a) *Squash-exp fitting* **(b)** *Squash-pow2 fitting*

Figure 2.7: Squash-exp and Squash-pow2 approximations

$$D_\lambda(\mathbf{x}) = |x_{i_{max}}| + \lambda \sum_{\substack{i=1 \\ i \neq i_{max}}}^{n} |x_i| \tag{2.3}$$

Squash approximation	Hardware resources
app_exp	Norm Unit (CapsAcc) Squash unit (a EXP Unit, a subtractor, a LUT to compute the squashing coefficient and a multiplier to compute the squashed component)
app_pow2	Norm Unit (CapsAcc) Squash unit (a POW2 Unit, a subtractor, a LUT to compute the squashing coefficient and a multiplier to compute the squashed component)
app_norm	Simplified Norm Unit (removing the sqrt-LUT, limiting the number of multiplications to 1 (for D_lambda) or 2 (for D_mu_lambda and D_a_b) and introducing comparisons among absolute values) Squash-LUT (CapsAcc)

Figure 2.8: Squash units hardware resources overview

2.2 Software Implementation of approximate Softmax and Squash functions

The next step of the process is to describe the approximate softmax and squash algorithms mentioned in the previous section in a software programming language. The software implementation of the approximations is based on *Python*, an interpreted high-level programming language that allows for high productivity thanks to its user-friendly syntax and data structures. In addition to that, Python is a widely adopted programming language in the Machine Learning and Deep Learning fields of study. The main reasons behind the software description of the approximations are listed below. First of all, writing a software model allows to understand and examine in detail the algorithmic steps of each approximation. Secondly, the goal is to verify the quality of the approximate softmax and squash outputs with respect to the corresponding ideal function outputs, by performing software simulations on input values in a realistic range; the input range is derived in the case study of an image classification task performed by the CapsNet model presented in [6] on the benchmark MNIST dataset.

Following the above observations, describing the approximate softmax and squash algorithms in Python is fundamental to integrate the approximate functions in a Capsule Neural Network model described in Python itself, as it is provided by the open-source framework *Q-CapsNets* [7]. It will be possible to see how each approximation of the softmax or squash function impacts on the inference accuracy of the overall CapsNet on some benchmark datasets, in order to asses the quality of the proposed approximations in a

realistic environment. The idea is to measure the degradation in accuracy of the Capsule Network including the approximate softmax or squash function in an image classification task. In particular, the performances of the softmax and squash approximations are evaluated in 4 case studies, by exploiting two Capsule Neural Network models and two benchmark image datasets. The two used CapsNet models are Shallow CapsNet as presented in [6] and DeepCaps as proposed in [9]. Building on the preexisting Shallow CapsNet model, DeepCaps is a deep capsule network architecture that exploits the concepts of skip connections and 3D convolutions in the Capsule Network domain. As regards the datasets, the two benchmark image databases are the MNIST dataset and the Fashion-MNIST dataset, where the acronym MNIST stands for Modified National Institute of Standards and Technology. The former consists of grayscale images of handwritten digits from 0 to 9 naturally belonging to 10 classes, while the latter includes grayscale images of commercial fashion articles associated with a label from 10 classes among which T-shirt, coat and bag.

2.2.1 Software Simulation

In order to compare each approximate function to its ideal counterpart, a software simulation environment is set up. It consists of 5 Python modules, i.e. a data maker and a testbench which includes the approximate function as unit under test, an error checker and an error plotter.

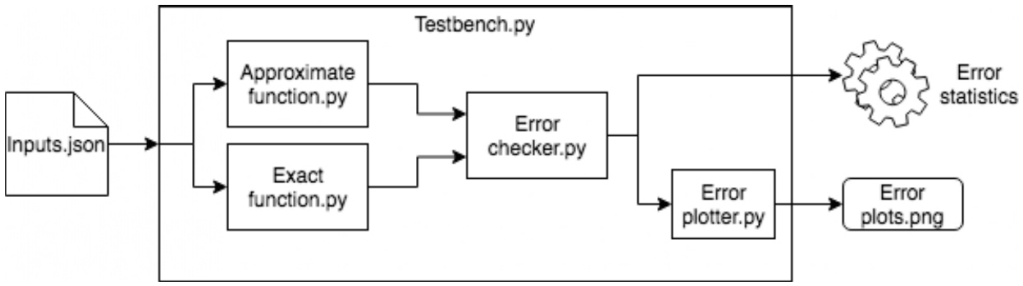


Figure 2.9: Software simulation setup

The software experiments are performed according to the procedure explained below. First of all, the data maker module creates a file in json format containing 1,000 input vectors of a given number of components with uniform distribution in a given range; specifically, the input vectors fed to the approximate softmax functions have 10 components in the range $[1e-4, 1e-2)$, while the approximate squash functions take as input vectors with 16 components in the range $[1e-2, 1e-1)$. The component number and range of the softmax and squash functions are selected to be compliant with a specific case study represented by Shallow CapsNet inference on the MNIST dataset. Secondly, the input file generated by the data maker is read by the testbench module, so that for each input vector the approximate function computes its outputs and the error checker produces 4 error metrics, by comparing the approximate results with the exact outputs of the ideal function. The 4 error metrics computed for each input vector are maximum and average component absolute errors and maximum and average component relative errors. At the

end of the input file, the testbench averages the 1,000 values of each error metric associated with the input vectors and produces 4 global error statistics in the form ($mean \pm stddev$), identified by the acronym *MED* that stands for Mean Error Distance. In addition to the aforementioned error statistics, two plots are generated by the error plotter module, i.e the absolute errors plot that shows the maximum and average component absolute errors for each multidimensional input vector and the relative errors plot that reports the maximum and average component relative errors. A variant of the 2 previously mentioned error plots is produced where the component absolute and relative errors are represented as a function of the input vector Euclidean norm, in order to highlight a possible error trend depending on the vector norm.

As far as the approximate squash functions are concerned, 6 additional error metrics are analysed for each input vector, besides the maximum and average component absolute and relative errors, i.e. the absolute and relative errors in the computation of 3 squash-specific parameters: input vector norm, squashing coefficient and output vector norm. The reason behind the additional squash error metrics is to identify the main source of the component errors (input vector norm or squashing coefficient computations), as well as to evaluate the impact of the component errors on the final class probability values, that is represented by the squash output vector norm.

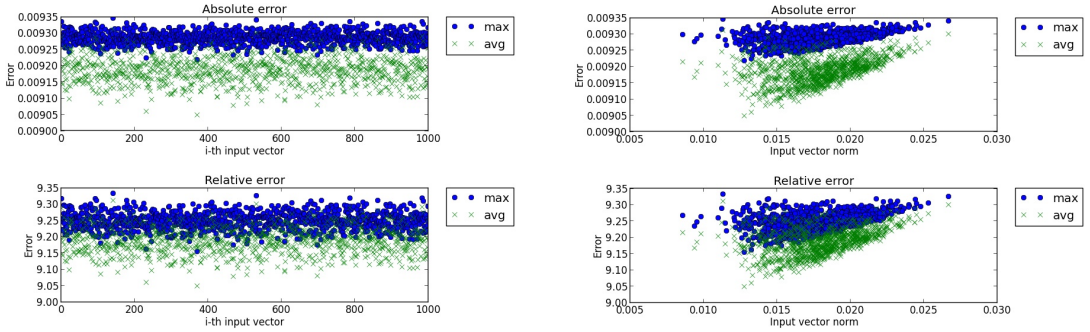
As reported in table 2.10, the Mean Error Distance of the maximum and average component relative errors on the 1,000 input vectors in the given range is about 9% for all of the softmax approximations, thus demonstrating similar accuracy degradation of the approximate softmax functions with respect to the ideal softmax. The softmax absolute and relative errors have an increasing trend with respect to the input vector norm, suggesting a dependency of the softmax approximation errors on the Euclidean norm of the input vector.

Softmax approximation	MED on mean absolute errors	MED on mean relative errors	MED on max absolute errors	MED on max relative errors
app_taylor	0.00919 ± 0.00004	9.1861 ± 0.0400	0.00929 ± 0.00002	9.2478 ± 0.0272
app_lnu	0.00924 ± 0.00003	9.2386 ± 0.0290	0.00928 ± 0.00003	9.3145 ± 0.0351
app_b2	0.00928 ± 0.00002	9.2807 ± 0.0200	0.00943 ± 0.00004	9.4676 ± 0.0479

Figure 2.10: Softmax software simulation results

On the other hand, the *MED* of the maximum and average component relative errors on the 1,000 input vectors in the given range is approximately 23% for the approximate squash_exp and squash_pow2 and only 2% for the approximate squash_norm, showing a distinct behaviour in accuracy of the approximate squash functions, due to the approximation of a different main computation, i.e. squashing coefficient for squash_exp and squash_pow2, while input vector Euclidean norm for squash_norm. As regards the dependency of the squash approximation errors on the input vector norm, it is interesting to note that squash_norm presents a slightly decreasing trend for both component absolute and relative errors, while the other two approximate squash functions show a marked

decreasing trend of component relative errors but an increasing dependency of absolute errors.



(a) *Errors vs i-th input vector* **(b)** *Errors vs input vector norm*

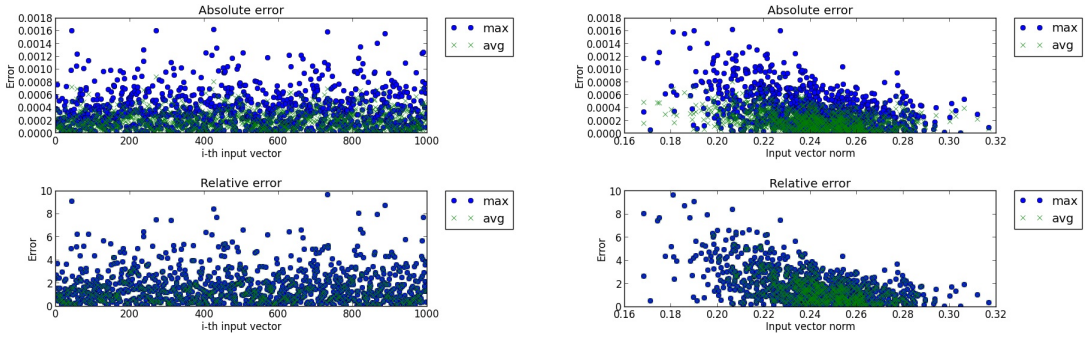
Figure 2.11: Softmax-taylor approximation error plots

Squash approximation	MED on mean absolute errors	MED on mean relative errors	MED on max absolute errors	MED on max relative errors	MED on norm_sj absolute errors	MED on norm_sj relative errors	MED on sq_coeff absolute errors	MED on sq_coeff relative errors	MED on norm_vj absolute errors	MED on norm_vj relative errors
app_exp	0.00295 ± 0.00049	23.6322 ± 0.8189	0.00508 ± 0.00045	23.6322 ± 0.8189	0.0 ± 0.0	0.0 ± 0.0	0.05361 ± 0.00288	23.6322 ± 0.8189	0.01299 ± 0.00193	23.6322 ± 0.8189
app_pow2	0.00295 ± 0.00049	23.6322 ± 0.8189	0.00508 ± 0.00045	23.6322 ± 0.8189	0.0 ± 0.0	0.0 ± 0.0	0.05361 ± 0.00288	23.6322 ± 0.8189	0.01299 ± 0.00193	23.6322 ± 0.8189
app_norm	0.00020 ± 0.00015	1.7462 ± 1.5208	0.00036 ± 0.00029	1.7462 ± 1.5208	0.0045 ± 0.0035	1.9510 ± 1.6708	0.00380 ± 0.00304	1.7462 ± 1.5208	0.00088 ± 0.00065	1.7462 ± 1.5208

Figure 2.12: Squash software simulation results

2.3 How Softmax or Squash approximations affect CapsNet model accuracy

After having evaluated the quality of the softmax and squash approximations with respect to the ideal functions, it is crucial to assess how the approximations affect the Capsule network inference accuracy on some benchmark datasets. As previously mentioned, the approximations will be tested in 4 case studies, with 2 CapsNet models and 2 image datasets, in order to evaluate their behaviour in a spectrum of possible applications. The next step is to include each softmax or squash approximation in the network layers that make use of those functions, by importing the Python module containing the coded algorithm of the approximation. Before integrating them in the whole CapsNet model, the software descriptions must be adapted to work with the *PyTorch* library and tensors, because the Q-CapsNets framework, that provides the CapsNet models and image datasets, is implemented by using the PyTorch library. For a previous design choice, the approximations



(a) *Errors vs i-th input vector* **(b)** *Errors vs input vector norm*

Figure 2.13: Squash-norm approximation error plots

were described in Python using the NumPy library and its main class ndarray. The reason behind the need of describing the approximations using software libraries that manage multidimensional tensors is that the CapsNet model processes data in the form of tensors, through each of its constituting layers.

In particular, PyTorch is an open source machine learning library used for applications such as computer vision and natural language processing. PyTorch provides two high-level features: tensor computing with acceleration via graphics processing units (GPU) and Deep neural networks with automatic differentiation system. PyTorch declares a class called Tensor (torch.Tensor) to store and perform computations on multidimensional arrays of numerical data. PyTorch Tensors are similar to NumPy arrays, but can also be operated on a CUDA-capable Nvidia GPU.

The approximate softmax or squash function replaces its exact counterpart in different layers depending on the CapsNet architecture. Specifically, in Shallow CapsNet, the approximations are inserted in the PrimaryCaps and DigitCaps layers, where the dynamic routing algorithm is performed for one and three iterations respectively. On the other hand, DeepCaps uses the squash function in the convolutional layers of capsules (Conv2DCaps) and includes both softmax and squash functions in the layers operating the dynamic routing algorithm, i.e. the deepest parallel ConvCaps layer (Conv3DCaps) and the fully-connected capsule layer at the output of the architecture.

After having included the PyTorch-based approximate softmax or squash function in the Capsule network model provided by the framework, the next step is to execute the inference pass on different datasets. To this purpose, a dedicated experimental setup is prepared, that is composed of both software and hardware components. In particular, the setup consists of the following parts: a software environment, created using the Python distribution Anaconda that allows for software package management and includes the PyTorch library; the Nvidia CUDA Toolkit, used to execute computation on a GPU; a local CUDA-capable Nvidia GPU to increase the execution speed of the inference passes.

Figure 2.14: MNIST and Fashion-MNIST 10-classes datasets

Figure 2.15: Q-CapsNet framework overview [7]

2.3.1 Q-CapsNets Framework for Inference

In the following, a description of the Q-CapsNets framework files is presented for the sake of clarity. First of all, the main Python module instantiates the CapsNet model selected by the user among the two currently supported ShallowCapsNet and DeepCaps. Each CapsNet model consists of a number of layers according to its peculiar architecture. Specifically, ShallowCapsNet is composed of three layers, i.e. Conv2d_ReLU, ConvPixelToCapsules and Capsules; on the other hand, DeepCaps includes six main layers, i.e. Conv2d_BN_ReLU, four DeepCapsBlocks and Capsules. Secondly, the framework main file loads the dataset chosen by the user on which the inference pass will be performed, by using the load_data function defined in the utils module, that in turn exploits the data_loaders module containing a specific function to load each of the available datasets. In particular, the datasets currently available are MNIST, Fashion-MNIST, CIFAR-10 and SVHN, each containing images of 10 classes. Then, the pre-trained weights are loaded for the specific model-dataset pair. At this point, the model is moved to the available GPU set by the user and the inference process starts to execute. The test function in the test_train_functions module is in charge of performing the inference pass according to the procedure reported below. First of all, the testing dataset is split into a number of batches, whose size, or number of images per batch, is defined by the user with the test-batch-size argument of the framework. Each image in a batch is labelled with its correct class and these target classes are needed to determine if the predictions made by the model for the corresponding images are correct. For a number of iterations equal to the number of batches, each batch is fed into the CapsNet model that makes a prediction for the class of each image in the batch. For a given image, the model outputs a number of probability values equal to the number of classes in the dataset and the class prediction for the current image is the class

corresponding to the highest probability value. The number of correct predictions per batch is counted and after the processing of all the dataset batches, the fraction of correct class predictions over the total number of images in the dataset is computed, in the form of a percentage accuracy. At the same time, the total loss for an input image is computed as the sum of the margin losses of the output capsules, as explained in [6]. Moreover, an average total loss is produced for each batch and a final average total loss is computed over all the batches at the end of the inference pass.

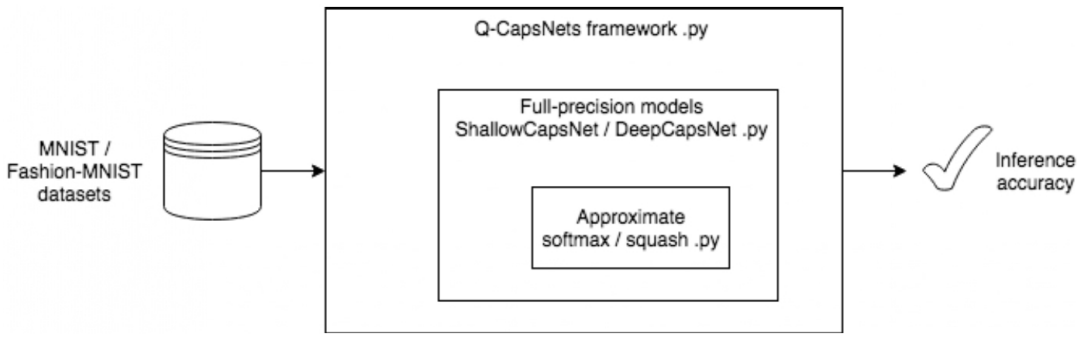


Figure 2.16: Inference pass

The Q-CapsNets framework can be run by command line, setting all the necessary arguments. The framework parameters allow to select CapsNet model, dataset, pre-trained weights path, model parameters including pixel size and number of channels of the input images, number of classes in the dataset and dimension of the output capsules, as well as GPU number and batch size to be used for the inference pass. By tweaking the framework arguments, 4 inference passes are performed to analyse the performance of the softmax or squash approximations in 4 case studies, i.e. ShallowCapsNet on MNIST and Fashion-MNIST and DeepCaps on the same two datasets. The results of the software experiments in the Capsule network domain are reported in table 2.17.

FP	ShallowCapsNet on MNIST	ShallowCapsNet on Fashion-MNIST	DeepCapsNet on MNIST	DeepCapsNet on Fashion-MNIST
Exact Functions	99.67	92.79	99.75	95.08
softmax_lnu	99.65	92.81	99.75	95.09
softmax_b2	99.68	92.73	99.74	95.04
softmax_taylor	99.68	92.77	99.74	95.08
squash_exp	99.64	92.09	99.70	94.57
squash_pow2	99.59	90.68	99.70	94.59
squash_norm	99.62	92.91	99.69	94.64

Figure 2.17: Inference accuracy results

2.3.2 Inference Accuracy Results

In the presence of the approximate softmax functions, the degradation in inference accuracy is negligible in all the case studies. Looking at the data in table 2.17, the highest loss in accuracy with respect to the full-precision value is shown by the approximate softmax_b2 in ShallowCapsNet for Fashion-MNIST dataset, where the use of the approximate softmax function causes the CapsNet to misclassify 6 extra images with respect to the full-precision case.

As regards the squash approximations, the accuracy loss is negligible in the case of inference on the MNIST dataset with both CapsNet models ShallowCapsNet and DeepCaps. Actually, the worst-case loss is reported by the approximate squash_pow2 in the case study ShallowCapsNet for MNIST with 8 extra wrong predictions with respect to the same model and dataset with exact squash function. On the other hand, the inference accuracy degradation becomes significant with Fashion-MNIST dataset, which includes images with a richer information content than MNIST dataset. In particular, in the case of DeepCaps for Fashion-MNIST the three squash approximations have similar performances, causing the CapsNet inference accuracy to decrease by about 0.5% with respect to the full-precision case. On the other side, in ShallowCapsNet for Fashion-MNIST the approximate squash functions impact differently on the inference accuracy, in the sense that there is one squash approximation, squash_pow2, that performs much worse than the others, causing an accuracy loss of about 2%, i.e. 200 extra misclassified images with respect to the model with exact squash function.

In the following paragraph, the arithmetic error of the approximate softmax or squash functions, as reported in tables 2.10 and 2.12, is compared to the accuracy loss in the CapsNet inference, as shown in table 2.17, in order to find out possible trends and dependencies.

As regards the softmax approximations, the similarity of behaviour in arithmetic errors with respect to the exact softmax function corresponds to comparable CapsNet inference accuracy values. In the case of the squash approximations, squash_exp and squash_pow2 have a larger arithmetic error than squash_norm and this trend is found also in the CapsNet inference accuracy losses of the three squash approximations in the case study of ShallowCapsNet for Fashion-MNIST, where the accuracy degradation is more relevant.

2.4 Quantization of CapsNets models and approximate functions

The next step is to quantise the approximate softmax and squash functions and test them in quantised CapsNet models. Quantising means lowering the numerical precision of data involved in a computation, in order to be compliant with a specific number representation. In particular, in this work a signed fixed-point number representation is selected, so that each floating point number is associated to a number in a given range and with a given precision.

The main reasons behind the quantisation of data involved in the softmax and squash approximations are explained below. Up to this stage of the work, the approximate functions

are described in the Python programming language and simulated in order to evaluate the quality of each approximation, both locally with respect to the corresponding ideal function and in the context of a CapsNet model, to assess the effect of the approximations on the number of correct predictions made by the network in an image classification task on a given dataset.

One of the next objectives of this work is to implement the approximate functions in hardware, so that it will be possible to make a comparison between the softmax or squash approximations in terms of relevant metrics in the hardware domain, i.e. area, power and delay. In view of the hardware implementation, it is convenient to make the current software description of the functions as close as possible to the hardware description.

First of all, experimentation at the software level in the CapsNet domain allows to determine the range of the input and output data involved in the softmax and squash functions and useful hints about the datapath bitwidth are obtained from the derived ranges. Secondly, by using the derived input ranges, quantisation is applied at the software level in the relevant algorithmic steps of the approximate functions, in order to emulate the internal dataflow of the future hardware implementations. The first step is performed by making inference simulations in the 4 case studies and plotting tensor values in order to find the numerical range of the input and output data of the softmax and squash functions, used in the CapsNet models operating on the image datasets. Additional information

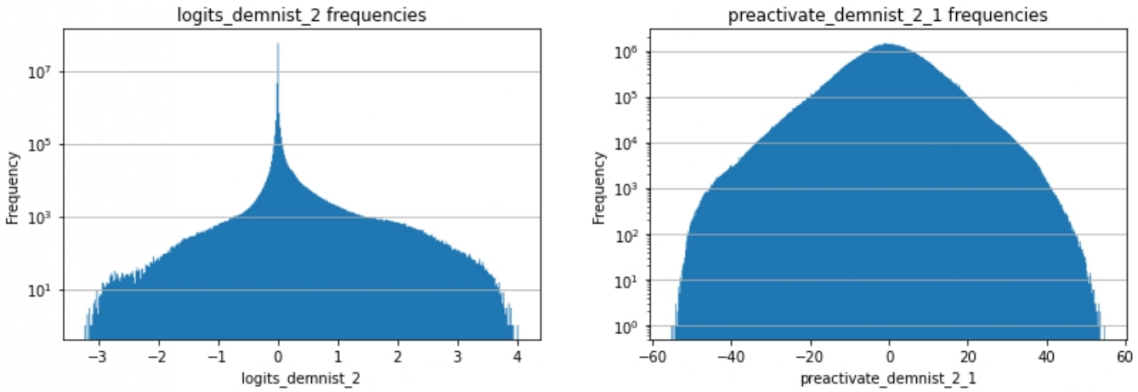


Figure 2.18: Frequency plots of softmax and squash inputs

about the bitwidth of the signed fixed-point numbers at the input of the softmax and squash functions are given by the quantisation of the whole CapsNet model as explained in the next section. The second step is performed by propagating the data through the arithmetic operations all the way through the approximate function algorithm and lowering the data numerical precision depending on the expected numerical range at the output of the various computations.

To sum up, the quantisation of the software functions allows to infer the datapath bitwidths and to write a software model that mimics the future hardware implementation of the functions, so that the accuracy results obtained with the quantised software model can be considered representative of those that would be produced by the corresponding hardware implementation. To get even more realistic hardware-aware results, the CapsNet

model is quantised by exploiting the peculiar feature of the Q-CapsNets framework, in order to emulate the context of a hardware accelerator for Capsule networks.

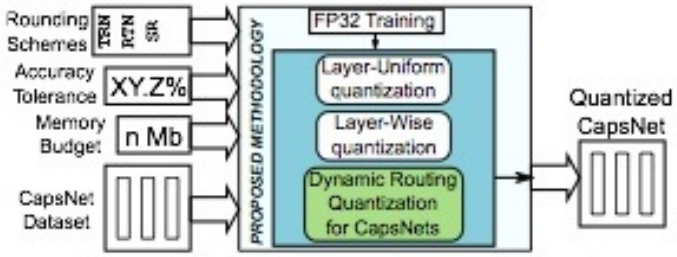


Figure 2.19: Q-CapsNet framework for model quantization [7]

SHALLOWCAPSNET						
MNIST -0.4%						
INT	1	2	3			
FRAC W	11	10	9			
FRAC A	11	11	11			
FRAC DR		11	5			
FMNIST -0.4%						
INT	2	4	3			
FRAC W	11	10	9			
FRAC A	12	12	12			
FRAC DR		12	8			
DEEPCAPS						
MNIST -0.4%						
INT	4	6	7	6	2	2
FRAC W	11	10	9	8	7	7
FRAC A	9	9	9	9	9	9
FRAC DR					3	3
FMNIST -0.4%						
INT	5	5	6	6	1	3
FRAC W	11	10	9	8	7	7
FRAC A	8	8	8	8	8	8
FRAC DR					4	4

Figure 2.20: Quantized CapsNet models

2.4.1 Q-CapsNets Quantization Algorithm

The quantisation of a CapsNet model is performed by means of a multi-step quantisation algorithm defined by the Q-CapsNets framework. The quantisation algorithm must be provided with the CapsNet architecture to be quantised for a given test dataset, a rounding

method to be used to quantise the data, a tolerance on the accuracy loss of the quantised network with respect to the full-precision model and a memory budget that can be used for the storage of the quantised weights. Quantising weights and activations allows to find an efficient trade-off between the classification accuracy of the model and the cost of the model in terms of memory usage and energy consumption during inference computations. The procedure followed to quantise a given CapsNet is described in the following paragraph.

At the first stage, weights and activations are quantised in a layer-uniform way, i.e. the number of bits of their fractional part is progressively reduced, consuming part of the accuracy tolerance. Secondly, an additional reduction in the numerical precision of weights is performed in a layer-wise manner, that is weights in the final layers of the network are quantised more than those in the first layers, as it can be observed from the decreasing trend in the number of fractional bits of weights in figure 2.20. At the next and last step, data involved in the dynamic routing algorithm are quantised more aggressively than other activations, with a lower wordlength, as it can be seen in figure 2.20. Specifically, the more aggressive quantisation is performed on the input data of the softmax and squash functions, in order to reduce the energy consumption of the computationally-expensive dynamic routing functions, due to their inherent complexity and iterative operations. It is interesting to note that softmax or squash functions input data can tolerate a more aggressive quantisation because they are updated iteratively and can adapt to quantisation more easily than other activations. To sum up, weights, activations and dynamic routing data are converted to a fixed-point arithmetic and the corresponding bitwidhs are reported in figure 2.20, where values are shown for each layer of two CapsNet models on two datasets, with accuracy tolerance 0.4% and truncation rounding scheme.

2.4.2 Quantized Inference Accuracy Results

At the software level, inference accuracy values are obtained by testing the quantised CapsNet model including the quantised approximate softmax or squash function.

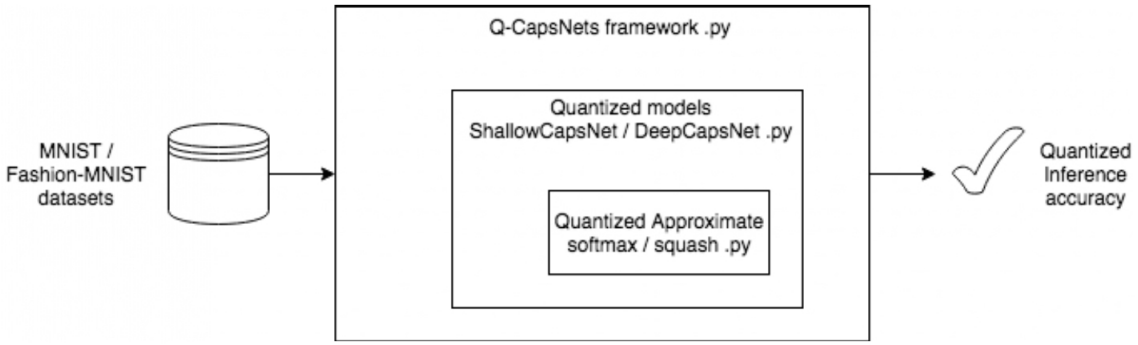


Figure 2.21: Inference pass with quantized models and functions

The results collected from multiple simulations in 4 case studies with softmax or squash approximation are reported in table 2.22.

The first row of table 2.22 shows the inference accuracy values of the two quantised CapsNet models for two datasets with exact softmax and squash functions, as a reference

QUANT	ShallowCapsNet on MNIST	ShallowCapsNet on Fashion-MNIST	DeepCapsNet on MNIST	DeepCapsNet on Fashion-MNIST
Exact Functions	99.44	92.42	99.35	94.69
softmax_lnu	99.46	92.37	99.42	94.71
softmax_b2	99.49	92.33	99.33	94.64
softmax_taylor	99.42	92.47	99.41	94.69
squash_exp	99.18	91.32	98.79	94.76
squash_pow2	99.00	89.05	98.58	94.62
squash_norm	99.26	92.51	99.23	94.70

Figure 2.22: Quantized inference accuracy results

to highlight the possible accuracy loss caused by the softmax or squash approximations. In particular, the bitwidth of the model weights, activations and dynamic routing functions inputs are those reported in figure 2.20, for each case study.

As regards the quantised softmax approximations, the inference accuracy degradation of the quantised CapsNet is negligible in all the cases, with respect to the accuracy values with exact softmax function reported in the first row. The highest accuracy loss is given by the approximate softmax_b2 in ShallowCapsNet for Fashion-MNIST, with 9 extra misclassified images with respect to the quantised CapsNet model with exact softmax function.

In the presence of squash approximations, the accuracy loss is negligible only in the case of inference of DeepCaps on Fashion-MNIST, where the worst-case loss is reported by the model when using squash_pow2 approximation and corresponds to 7 extra wrong predictions with respect to the same model with exact squash function. On the other hand, the inference accuracy degradation becomes relevant in the other three case studies. In particular, squash_norm performs better than the other two approximations, with a limited worst-case accuracy loss of about 0.2% in ShallowCapsNet for MNIST. On the other side, squash_pow2 records the worst inference accuracy results in all the cases, by causing a worst-case accuracy loss of approximately 3.5% in ShallowCapsNet for Fashion-MNIST, i.e. 350 extra misclassified images with respect to the same quantised model with exact squash function.

It is interesting to note that the similarity in arithmetic error of the approximate softmax functions shown in table 2.10 persists in the quantised capsule network domain, where the inference accuracy values of CapsNet models using the softmax approximations are comparable in the various case studies. In the same way, the trend of the arithmetic error of approximate squash functions is found also in the CapsNet inference accuracy with squash approximations, where squash_exp and squash_pow2 have higher accuracy losses than squash_norm.

Chapter 3

Architecture design and implementation

3.1 Design Flow of approximate Softmax and Squash processing units

At this stage, the approximate softmax and squash functions have been evaluated, in order to assess the quality of the approximations in terms of two metrics: arithmetic accuracy with respect to the exact functions and CapsNet inference accuracy loss with respect to the same CapsNet model with the exact function. In the capsule network domain, the evaluation has been performed in 4 case studies with two CapsNet models and two datasets, both with full-precision and quantised approximate functions and CapsNet models. The inference accuracy results obtained with the quantised approximations and CapsNet models can be regarded as representative of the performance of the future hardware implementation of the approximate functions in the context of a hardware accelerator for CapsNet inference.

The next step is to implement in hardware the approximate softmax and squash functions, in order to compare the softmax or squash approximations in terms of relevant hardware metrics, such as occupied chip area, power consumption and maximum path delay. Aiming at the hardware-aware comparisons between the proposed softmax or squash approximations, a digital design procedure is followed that is summarised in the following paragraph and explained in detail in the following sections.

First of all, the datapath and control unit of the approximate softmax and squash processing units are described by using a data flow block diagram for the datapath and an algorithmic state machine chart for the control unit. Secondly, each architecture is implemented in Register Transfer Level with a VHDL-based description of both datapath and control unit. Then, the VHDL model of each processing unit is simulated by means of a VHDL testbench structure with input and output files, in order to verify the functionality of the RTL architecture and check the VHDL model outputs against the corresponding Python model results. The next step is to synthesise the complete architecture of each processing unit in a 45 nm open-source technology library, by using the typical ASIC design flow, in order to get the precise area, power and timing performance of each design. As

a last step, functional and timing post-synthesis validation are performed to assess the functionality of the gate-level netlist generated by the logic synthesis process.

3.2 Approximate Softmax Architectures

3.2.1 Softmax-lnu

The fundamental idea behind the approximate softmax_lnu architecture is derived from the paper [13], where a hardware implementation of the softmax function is presented for deep neural networks. To get to the proposed architecture, a mathematical transformation of the softmax function is performed by using natural logarithm and exponent operations, which is reported in the formula $f(x_i) = \exp\left(\ln\left(e^{x_i} / \sum_{j=1}^{n} e^{x_j}\right)\right) = \exp\left(x_i - \ln\left(\sum_{j=1}^{n} e^{x_j}\right)\right)$. Firstly, the logarithmic operation allows to eliminate the need of a complex division operator and perform division by using a simpler operation, i.e. subtraction. Secondly, the exponentiation involved in the mathematical transformation is needed to convert the softmax results back from the logarithmic domain. The idea is to exploit the proposed domain transformation to obtain an efficient hardware implementation of the softmax function. The complete architecture of the softmax function consists of three main computational units: a natural exponential unit, an accumulator and a natural logarithmic unit. In particular, the exponential unit performs the exponentiation of the softmax inputs, the accumulator sums up the computed exponentials and the natural logarithmic unit computes the natural logarithm of the sum, that is needed to perform the division involved in the softmax in the logarithmic domain. Other simple units are included in the proposed softmax architecture. First of all, a maximum finder unit determines the maximum softmax input needed for later computations. Secondly, a subtractor is used to perform two operations at two different steps of the algorithm: (1) scaling of the softmax inputs to non-positive values, in order to limit the numerical range of the exponentiation results in (0, 1] and (2) division between each exponential and the exponentials sum in the logarithmic domain. Finally, a generic modulo counter and an encoding unit are exploited to allow the softmax function to work with a variable number of inputs. In particular, the architecture is able to process 10, 32 and 128 inputs, in order to be compliant with two capsule network models, ShallowCapsNet and DeepCaps for two datasets, MNIST and Fashion-MNIST. Specifically, the softmax function used by ShallowCapsNet and DeepCaps in their final capsule layer takes as input arrays of 10 components, corresponding to the 10 classes in the testing datasets. In the ShallowCapsNet model, PrimaryCaps layer, the softmax function uses 32 inputs, related to the 32 channels of 8-dimensional capsules in the layer, while in the DeepCaps architecture, Conv3DCaps layer, the number of softmax input components is equal to 128, related to 32 output channels and 2×2 output feature maps in the layer. To sum up, the softmax architecture is designed to work properly in the capsule network domain, in 4 case studies, with two CapsNet models and two datasets.

At the algorithmic level, the exponential calculation is performed by exploiting a local mathematical transformation of the exponential function, which is shown in $e^{y_i} = 2^{y_i \cdot \log_2 e} = 2^{u_i + v_i} = 2^{u_i} \cdot 2^{v_i}$. In particular, the input of the exponent function is multiplied by the constant $\log_2 e$ and the result of the multiplication is split into its integer and

fractional part, u_i and v_i. At this point, the calculation is simplified in the following way: (1) 2^{v_i}, with v_i limited in the range [0, 1), is computed by using the linear fitting function $1 + v_i$; (2) the multiplication by 2^{u_i} is implemented as a simple shift operation of a fixed-point number in the hardware implementation. Due to the fact that the softmax inputs are scaled to non-positive values, the arithmetic shift operation is performed to the right direction by a number of positions equal to the absolute value of the integer part, u_i. At the architectural level, the exponential unit is composed of a constant multiplier to perform the multiplication by $\log_2 e$, a bus expander to implement the $1 + v_i$ operation and a shift unit to shift $1 + v_i$ to the right direction by a number of positions equal to the magnitude of u_i and compute the exponential result.

As regards the natural logarithm calculation, a mathematical transformation is performed as reported in $\ln F = \ln 2 \cdot \log_2 F = \ln 2 \cdot (w + \log_2 k)$. First of all, the change-of-base formula is applied to the natural logarithm and the operation is written as base-2 logarithm multiplied by a constant $\ln 2$. Secondly, the input of the natural logarithm is expressed as $2^w \cdot k$, where w in an integer number and k is in the range [1, 2), so that the base-2 logarithm can be computed as $w + \log_2 k$. Finally, the calculation of the natural logarithm is simplified as in $\ln F = \ln 2 \cdot (w + k - 1)$ by exploiting the linear fitting function $k - 1$ to compute $\log_2 k$, with k in [1, 2).

In the hardware implementation, the natural logarithm unit consists of 4 main components: 1) a leading one detector, to compute w by determining the highest bit '1' position in the fixed-point input number of the logarithm; 2) a right shift unit, to calculate k by shifting the logarithm input to the right direction by w positions; 3) a bus interconnection, to generate the base-2 logarithm of the input by aligning the LOD output w and the fractional part of the shift unit output k; 4) a constant multiplier to compute the natural logarithm from the calculated base-2 logarithm. Due to the fact that the scaled softmax inputs are non-positive, the natural logarithm input, i.e. the sum of the exponentials of the scaled softmax inputs, is greater or equal than 1, implying that w is a non-negative value.

In the overall architecture, the use of registers is optimised, i.e. the same group of input registers is used in two different algorithmic steps, as suggested by the multiplexer placed at the beginning of the registers chain. At the first stage, the registers load the softmax inputs, while the maximum input is being searched and the number of inputs is being counted. After the scaling of the inputs by their maximum value, the same registers store the scaled inputs, that will be needed in a later subtraction operation to compute the softmax division in the logarithmic domain.

In the following section, a step-by-step description of the dataflow in the proposed softmax architecture is reported, by highlighting the states transitioned by the processing unit. Initially, the registers and counter in the datapath are set to zero by a synchronous reset signal in the reset state, after the input reset signal activation. State transitions are triggered by the rising edge of the clock signal. At the next active clock edge after the input reset signal deactivation, the processing unit moves to an idle state, where it waits for available data at its input ports. Valuable data is placed on the input data bus DIN and signalled by the input signal VIN. At the next clock tick, the processing unit begins to load the available data in the internal registers, thanks to the Mealy control signals triggered by the VIN signal activation. At each clock tick following the VIN active transition, three

main operations are performed by the processing unit. Besides loading the input data in the internal registers, a maximum finder unit looks for the maximum input value, by loading the first available input into a register and then replacing the content of the register only when signalled to do so by a comparator, that determines if each of the next input data is larger than the current maximum value stored in the register. Moreover, a 7-bit counter counts the number of input values loaded by the architecture, as it is required to be able to process a different number of input components.

The VIN deactivation signals the end of available data on the input data bus and triggers the stop of the data loading, maximum search and count of number of inputs. As regards the counting operation, at the next active clock edge after the VIN low transition, the counter output value representing the number of loaded inputs is stored in a register, as it is needed as a counter terminal count for later computations and the counter is then reset. The processing unit transitions to a state referred to as acc, where the inputs are scaled by subtracting the maximum value from them, the exponentials of the scaled inputs are computed by the exponential unit and the sum of the exponentials is accumulated in a sum register. Depending on the number of inputs previously counted, an encoding unit selects the correct group of registers where the inputs are stored. As each input is processed, the inputs move along the chain of registers to reach the head of the chain and the scaled inputs are inserted in the tail of the chain for later computations. Hence, in the acc state, at each active clock edge, an input is processed and each input register loads the content of the one preceding it, with the register in the tail of the chain that loads the scaled version of the current processed input. As soon as the sum of the exponentials is calculated, the processing unit moves to a state where the natural logarithm of the sum is computed by the logarithmic unit and stored in a dedicated register. In the last state, the computed natural logarithm of the exponentials sum is subtracted from each scaled input to perform the softmax division in the logarithmic domain. The logarithmic result is then converted to the correct softmax output result by the exponential unit and stored in an output register. When all the softmax outputs have been produced, the processing unit resets the sum register and performs a transition to the idle state, ready to load and process new input data.

As regards the bitwidth of input and output data of the softmax architecture, the softmax inputs representation is <3; 8> and the softmax outputs are represented with <1; 12>. The number of integer and fractional bits of the input and output data are determined by exploiting the quantisation information in figure 2.20, focusing on the last two layers and on the rows labelled as int, frac a and frac dr, in each case study. Frac dr corresponds to the fractional part of the softmax or squash inputs in a given layer of a given case study, while frac a represents the fractional part of the softmax or squash functions outputs, among other types of activations.

The worst-case number of fractional bits of the softmax inputs is equal to 8, in ShallowCapsNet for Fashion-MNIST, DigitCaps layer, where the softmax is performed for a number of iterations equal to three. The number of integer bits, 3, is derived from the worst-case range of the softmax inputs, that is [-4, 4), as reported in figure 2.18.

As regards the output data bitwidth, the fractional bits number, 12, is derived from the case study ShallowCapsNet for Fashion-MNIST, in DigitCaps layer, while the number

of integer bits, 1, is related to the numerical range of a generic softmax function output, that is by definition in (0, 1) for a number of components larger than 1.

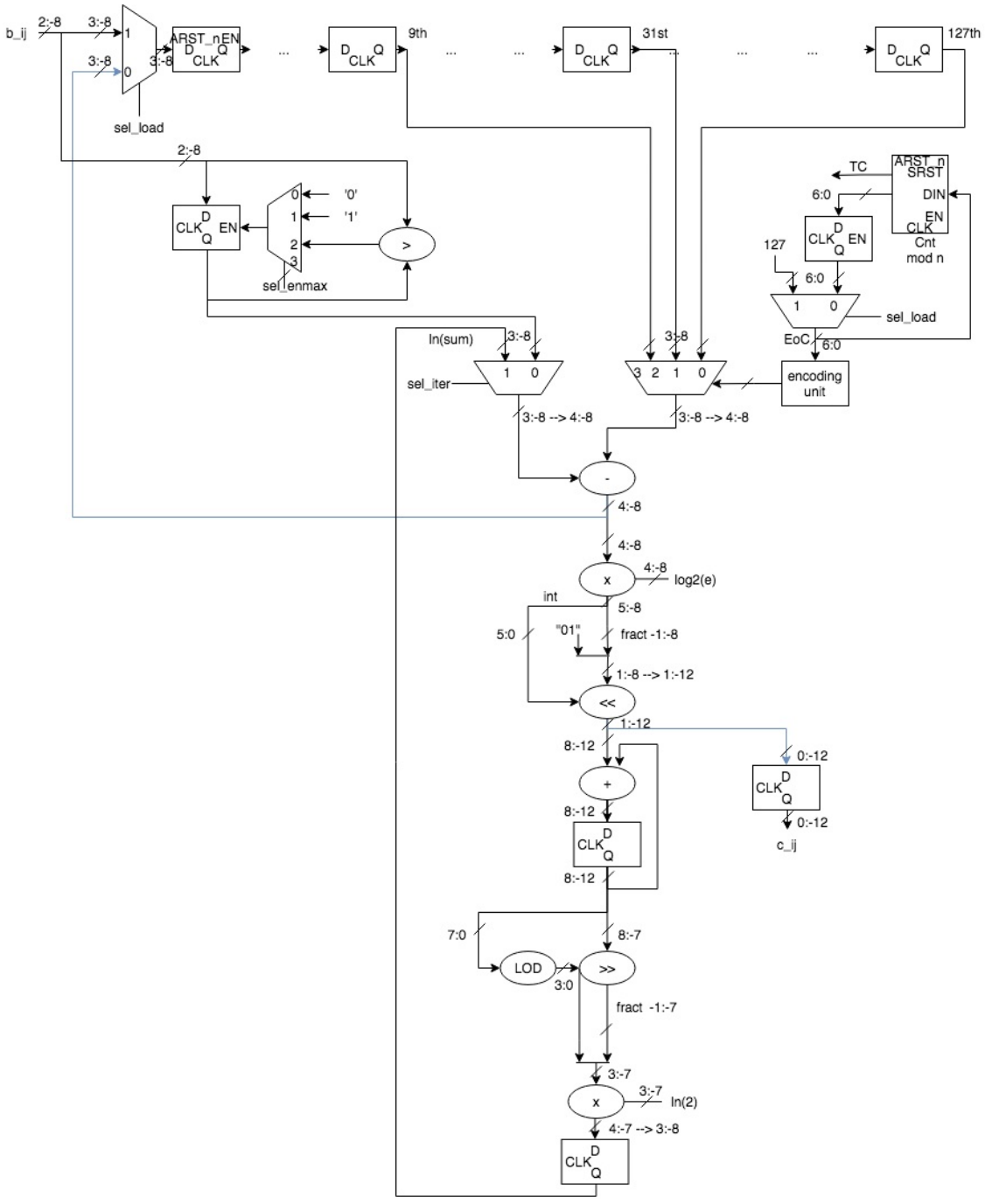


Figure 3.1: Softmax-lnu datapath

3.2.2 Softmax-b2

The approximate softmax_b2 function builds on the approximate softmax-lnu algorithm that is presented in the previous section. The softmax_b2 approximation introduces the concept of modifying the mathematical expression of the softmax function, from $e^{x_i} / \sum_{k=1}^{n} e^{x_k}$ to $2^{x_i} / \sum_{k=1}^{n} 2^{x_k}$, where powers of 2 are used in place of natural exponentials. The new formula allows for a reduction of the algorithmic strength because 2^x is easier to compute than e^x, that is it requires a lower number of components in the hardware implementation.

The arithmetic error of the softmax_b2 approximation is comparable to that of the approximate softmax_lnu, as it is demonstrated by the results reported in table 2.10, obtained with software simulations in a specific case study. The same similarity in behaviour is found in the capsule network domain, as shown by the inference accuracy values in table 2.22. Therefore, the proposed approximation allows for a complexity reduction of the hardware implementation, while causing a limited inference accuracy loss with respect to the softmax_lnu approximation, with worst-case accuracy difference equal to 0.09% in the case study DeepCaps for MNIST. It is interesting to note that the modified mathematical expression can be used as a softmax-like function, i.e. a probabilistic version of the argmax function, which returns 1 for the highest input value and 0 for all the other values.

At the architectural level, the implementation of the softmax_lnu described in the previous section is modified, in order to compute the softmax-like function $2^{x_i} / \sum_{k=1}^{n} 2^{x_k}$. Actually, the mathematical transformation performed to calculate $2^{x_i} / \sum_{k=1}^{n} 2^{x_k}$ involves power of 2 and base-2 logarithm operations, as in $\text{pow2}\left(\log_2\left(2^{x_i} / \sum_{j=1}^{n} 2^{x_j}\right)\right) = \text{pow2}\left(x_i - \log_2\left(\sum_{j=1}^{n} 2^{x_j}\right)\right)$. The modification of the hardware implementation consists of the removal of two constant multipliers used in the natural exponential unit and natural logarithmic unit, so that the two units compute the power of 2 and base-2 logarithm operations needed by the new approximation. Specifically, in the exponential unit, the preliminary multiplication by the constant $\log_2 e$ is eliminated to calculate 2^{x_i} and in the logarithmic unit, the final multiplier by the constant $\ln 2$ is removed to compute $\log_2 \sum_{i=1}^{n} 2^{x_i}$, where x_i is the i-th scaled input of the softmax architecture. Overall, the complete architecture is composed of three main units: a power of 2 unit, an accumulator and a base-2 logarithm unit. A detailed description of the architecture topology is reported in the previous section. Here, it is convenient to note that the processing unit still computes the scaling of the inputs by subtracting the maximum value from each input, in order to keep the numerical range of the power of 2 operation limited to (0, 1]. Moreover, as mentioned for the softmax_lnu approximation, the softmax_b2 architecture is able to process a variable number of inputs to be used in the capsule network domain, with two CapsNet models, ShallowCapsNet and DeepCaps, for two datasets, MNIST and Fashion-MNIST.

As regards the state machine diagram, the states transitioned by the processing unit during its execution are generally equivalent to those used for the approximation in the previous section. It is important to note that the calculations performed by the softmax_b2 architecture in the state acc are scaling of the inputs, power of 2 of the scaled inputs and accumulation of the base-2 powers. Then, in the state log2_acc, the processing unit computes the base-2 logarithm of the sum of base-2 powers, that is required to perform

the division between powers of 2 of the scaled inputs and sum of powers in the logarithmic domain.

Due to the fact that the softmax scaled inputs are non-positive and that 2^{x_i} is larger than e^{x_i} per non-positive inputs, the exponential unit output bitwidth can be reduced from <2; 12> in the softmax_lnu architecture to <2; 8> in the current softmax_b2 implementation. The number of integer bits, 2, is required to represent the maximum exponential output, 1, for a null scaled input, in a signed fixed-point representation. On the other hand, the number of fractional bits, 12 and 8, allows to represent the smallest exponential output for the smallest scaled input, -8, that is e^{-8} and 2^{-8}, respectively.

3.2.3 Softmax-taylor

The basic concept of the approximate softmax_taylor function is obtained from the paper in [12], where an approximate softmax design is proposed for deep neural networks. The goal is to approximate the hardware-expensive exponent and division operations in the softmax function algorithm, in order to get an efficient hardware architecture.

The exponent calculation is simplified by using the mathematical transformation $e^{x_i} = e^{a+b+c} = e^a \cdot e^b \cdot (1+c)$, that exploits the basic property of the exponent function and the first-order polynomial approximation of the exponent based on the Taylor expansion method. Specifically, the input number of the exponent function is divided into three parts: the exponent of the first two parts is computed by using an exponential function corresponding to each part domain, while the exponent of the third smaller part is approximated by using the Taylor expansion formula truncated at the first order. The final exponent value of the input is obtained by multiplying the three computed exponent contributions.

As regards the softmax division operation, a mathematical manipulation involving power of 2 and base-2 logarithm is performed, which is shown in $\text{pow2}\left(\log_2\left(e^{x_i} / \sum_{j=1}^{n} e^{x_j}\right)\right) = \text{pow2}\left(\log_2\left(e^{x_i}\right) - \log_2\left(\sum_{j=1}^{n} e^{x_j}\right)\right)$. The input of each base-2 logarithm can be expressed as $2^S \cdot F$, where F is in [1, 2) and S is an integer number, so that the previous expression is simplified in $\text{pow2}\left(S_1 + \log_2 F_1 - (S_2 + \log_2 F_2)\right)$. The computation can be further simplified, by using the linear fitting function $F - 1$ to compute the base-2 logarithm, so that the formula becomes $\text{pow2}\left(S_1 + F_1 - (S_2 + F_2)\right)$. To compute the final softmax output, the input of the power-2 operation is split into its integer and fractional part, u_i and v_i, such that $2^{u_i+v_i} = 2^{u_i} \cdot 2^{v_i}$ and the computation is performed in two steps: (1) 2^{v_i} is computed by using the linear fitting function $1 + v_i$, with v_i in [0, 1); (2) the multiplication by 2^{u_i} is implemented by a shift operation of the fixed-point number in the hardware implementation.

At the architecture level, the proposed approximate softmax design is composed of three main units: an exponent unit to compute the exponential of the inputs, an accumulator to sum up the exponentials and a division unit to calculate the division between each exponential and the exponentials sum in the logarithmic domain.

The exponent unit consists of two look-up tables, a specific bus interconnection and a multiplier. The fixed-point input of the exponent unit is divided into three parts, i.e. integer, most significant fractional and least significant fractional part. The exponents of

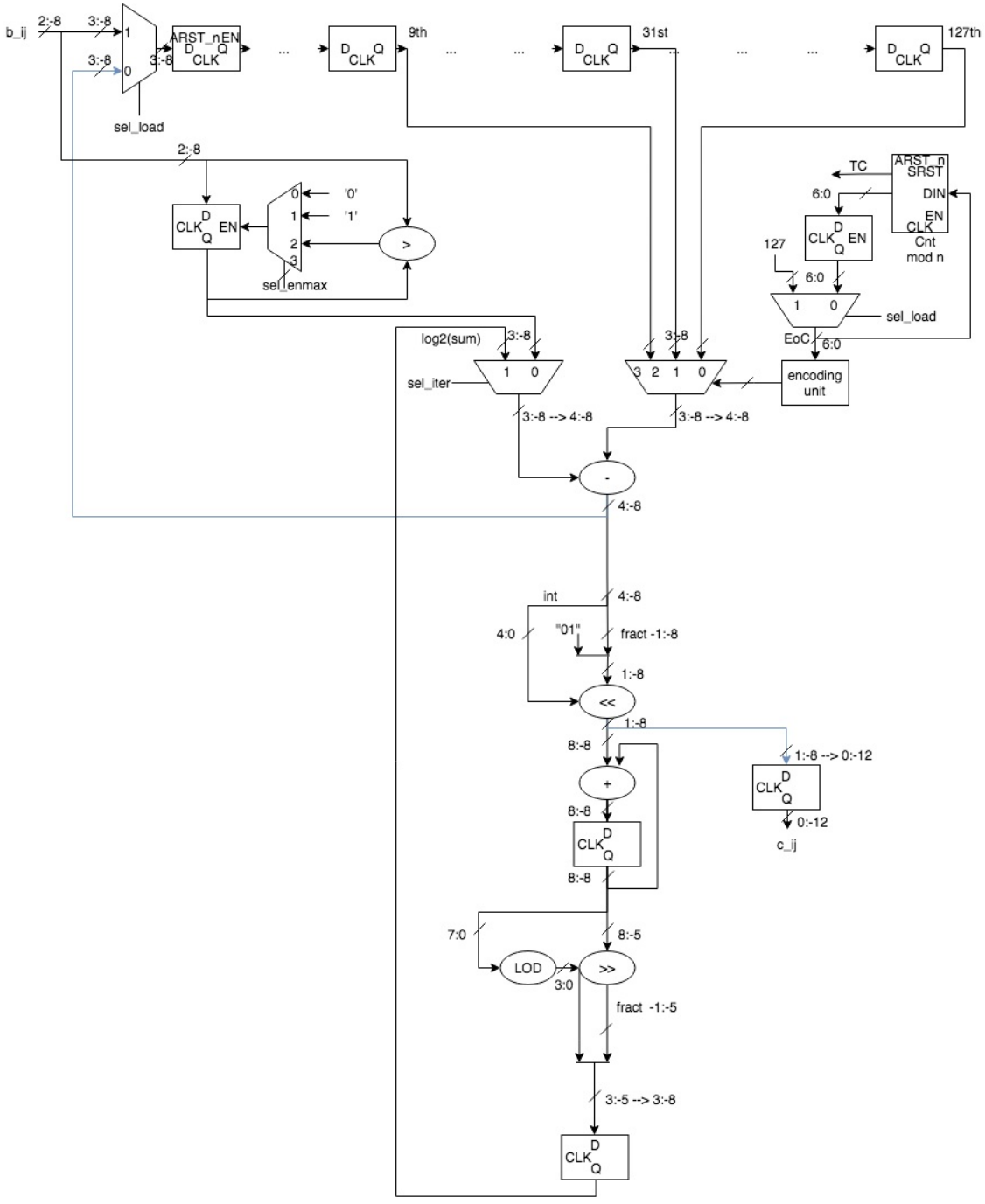


Figure 3.2: Softmax-b2 datapath

the integer and most significant fractional part are computed by using two LUTs, that implement the exponent function over a precise input domain with a given output resolution. On the other hand, the exponent of the least significant fractional part is calculated by performing the $1 + x$ operation by means of a bus alignment. The final exponent value can be computed by iteratively multiplying the three exponent contributions with a single multiplier. In order to keep limited the numerical range of the exponent result, the input of the exponent unit is scaled by subtracting the maximum input value from it.

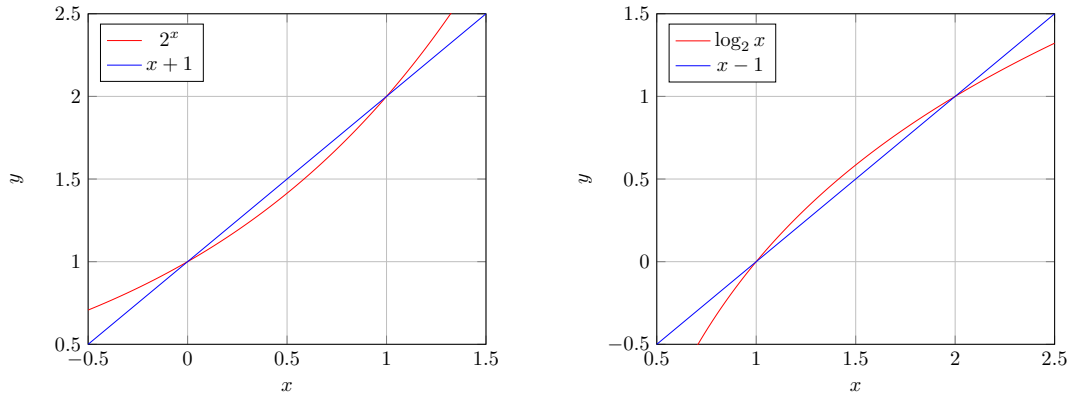
As regards the division operation, the division unit is composed of two base-2 logarithmic units, a subtraction unit and a base-2 exponential unit. The two logarithm units are used to compute the base-2 logarithm of the dividend and divisor involved in the softmax division. Specifically, each logarithmic unit consists of a leading one detector and a shift unit, so that the base-2 logarithm integer and fractional part are given by the LOD output and the fractional part of the shift unit output, respectively. The shift unit for the dividend performs a shift in the left direction by a number of positions equal to the corresponding LOD output, because the dividend is an exponential of a non-positive scaled input, so it is in the range (0, 1]. On the other hand, the divisor is shifted in the right direction by a number of positions given by the related LOD output, since the divisor is the sum of the exponents of the non-positive scaled inputs, so it is greater or equal than 1. As regards the subtraction unit, the logarithms of the dividend and divisor are subtracted to perform the softmax division in the logarithmic domain. In particular, the integer and fractional parts of the logarithms are subtracted by using two different subtractors to get the integer and fractional part of the resulting logarithm. An additional subtractor is needed to obtain the correct integer part of the resulting logarithm in the case that the fractional parts subtractor produces a negative result in the range (-1, 0). Finally, the base-2 exponent unit is exploited to implement the logarithmic conversion to get the final softmax output. It consists of a bus interconnection and a shift unit. The former performs 2^{v_i} as $1 + v_i$, where v_i is the fractional part of the computed resulting logarithm, while the latter is used to multiply $1 + v_i$ by 2^{u_i}, where u_i is the integer part of the resulting logarithm. The shift unit operates a shift of its input to the right direction by a number of positions equal to the absolute value of the integer part of the resulting logarithm, u_i. Actually, the computed logarithm corresponds to the logarithm of the softmax division result $e^{x_i} / \sum_{k=1}^{n} e^{x_k}$, so the logarithm integer part is a non-positive number, since $e^{x_i} / \sum_{k=1}^{n} e^{x_k}$ is in the range (0, 1].

In the proposed architecture, other simple units are included to execute the softmax computation. Firstly, a maximum finder unit composed of a register and a comparator, and a subtractor are used to perform the scaling of the softmax inputs. Secondly, a generic modulo counter counts the number of inputs and allows the architecture to process a variable number of input softmax components, in order to be compliant with the capsule network domain, as explained in the previous section. In addition to that, a modulo-2 counter is used in the exponent unit to manage the multiplication of the three exponent contributions, by using a single multiplier in an iterative way. Actually, the modulo-2 counter value allows to route the correct data to the inputs of the multiplier in two iterations, by selecting the inputs of two multiplexers.

In the proposed design, the input registers are used in two algorithmic steps, for the loading of the softmax inputs and the exponentials of the scaled inputs. The input

registers usage optimisation determines the datawidth of the registers as <3; 12> in a signed fixed-point representation. In particular, 3 integer bits are required to represent the softmax input values in the range [-4, 4) and 12 fractional bits are derived from the numerical resolution of the exponentials of the scaled inputs.

In the following section, a description of the state machine diagram of the proposed approximate softmax processing unit is reported. After the input reset signal activation, the registers and counters are set to zero and at the next active clock edge, the processing unit transitions to the idle state, ready to accept valuable input data. As soon as input data is placed on the input data bus, the architecture starts loading input values into the input registers, computing the maximum value and counting the number of loaded inputs. After the deactivation of the VIN signal, at the next clock tick, the current number of inputs is stored in a register for later computations and the processing unit performs a transition to the mpy_state. At this stage, each input is scaled by subtracting the maximum input value from it and the exponent of the scaled input is computed, by iteratively multiplying three exponent contributions with a single multiplier in two clock ticks. Specifically, at the first clock tick, a partial exponent result is stored in the multiplier register and at the second clock tick, the exponent computation is completed. Hence, the processing unit moves to the acc_state, where the computed exponent is accumulated in a sum register and stored in the tail of the input registers chain. The iterative process of input scaling, exponent computation and exponent accumulation and storage is repeated for each softmax input value, so that all the exponentials and their sum are calculated. Finally, in the output state, the processing unit performs the division between each computed exponential and the sum of exponentials in the logarithmic domain, thus producing the final softmax outputs. At the end of the processing, the sum register is reset and the processing unit goes back to the idle state, ready to begin a new softmax computation.



(a) *Power-2 fitting in* $[0, 1]$ **(b)** *Log-2 fitting in* $[1, 2]$

Figure 3.3: Linear fitting in softmax approximations

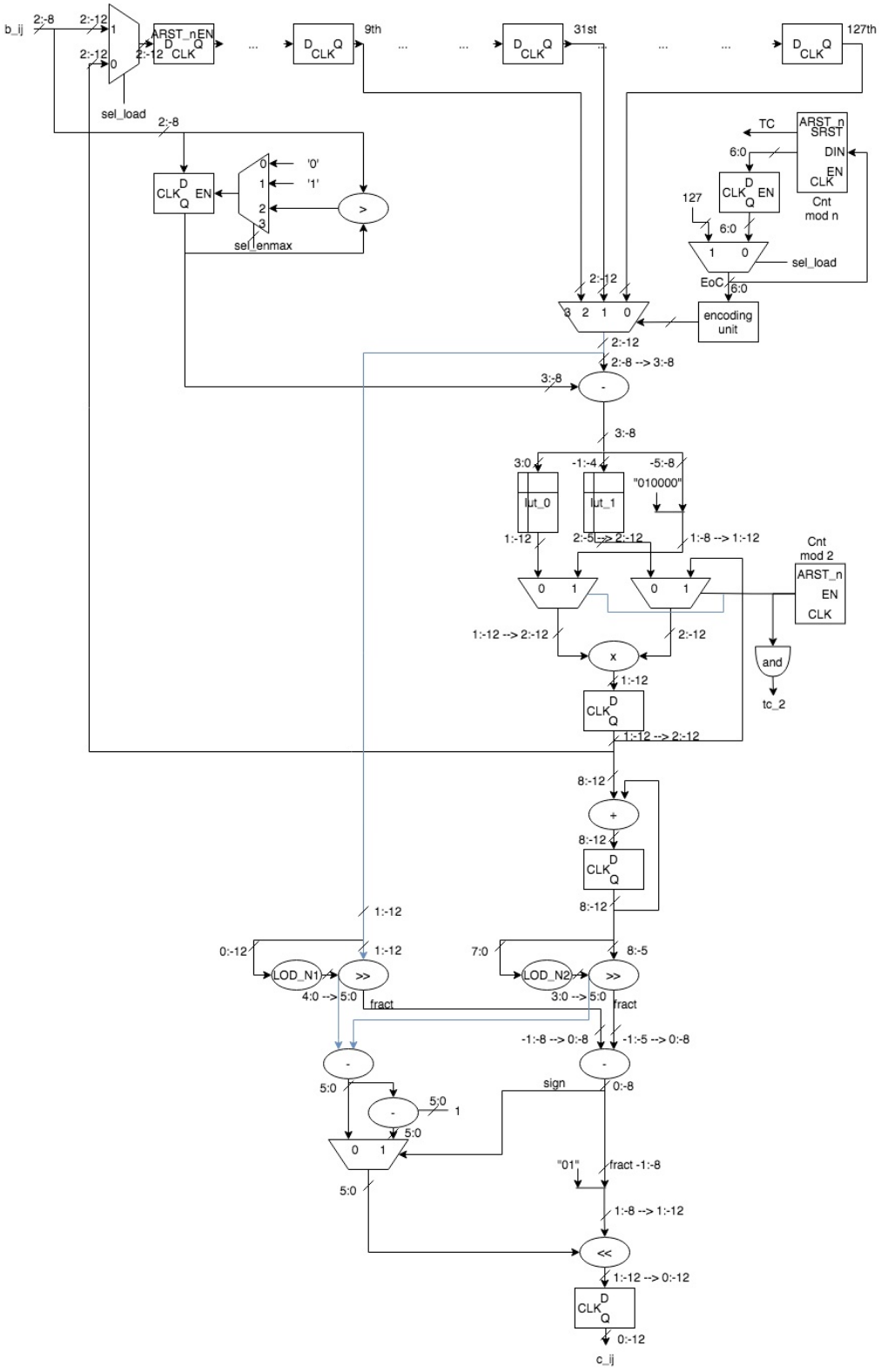
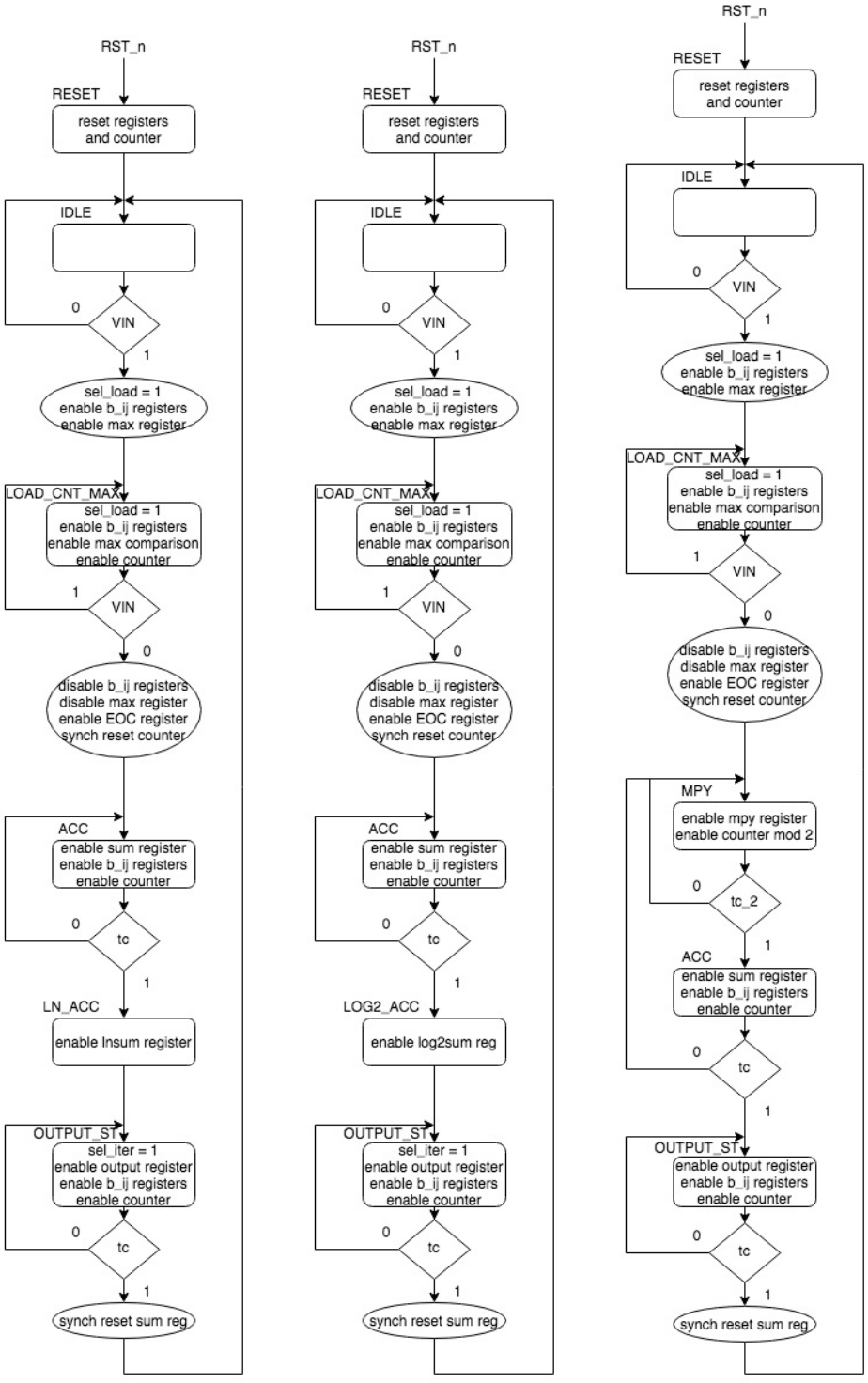


Figure 3.4: Softmax-taylor datapath



(a) *Softmax-lnu* **(b)** *Softmax-b2* **(c)** *Softmax-taylor*

Figure 3.5: Softmax control units

3.3 Approximate Squash Architectures

The squash function involves two main operations: the computation of the norm of the squash input vector and the calculation of the squashing coefficient, that multiplies each input component to produce the squashed output component. In the following section, three approximate squash functions are proposed. One of the presented squash approximations focuses on the norm computation, by approximating the Euclidean norm. On the other hand, the remaining two approximations introduce innovative solutions to approximate the squashing coefficient involved in the squash function.

3.3.1 Squash-exp

At the algorithmic level, the approximate squash_exp function exploits an original piecewise approximation of the squashing function, $x/(1+x^2)$, in two ranges of input vector norm, x. In particular, the squashing function is approximated by the function $1-e^{-x}$ in the range [0, 0.75) and by a direct-value look-up method in the range [0.75, max), as it is shown in figure 2.7 a. For the intermediate norm value, 0.75, the approximation error of $1-e^{-x}$ is about 10% of the exact squashing function value. In order to produce the squashed outputs, the input vector components are multiplied by the approximate squashing function previously described. In the proposed squash_exp approximation, the Euclidean norm is computed as the square root of the sum of the squared input vector components. Specifically, the square root operation is implemented by a direct input-output mapping, over two different ranges of squared norm values.

At the architecture level, the proposed design consists of two main units, i.e. the norm unit and the squashing unit. The norm unit computes the Euclidean norm of the input vector and it is composed of a multiplier to compute the square of the inputs, an accumulator to sum up the squared inputs and a square root unit to perform the square root of the sum of the squared inputs. In order to compute the sum of squares in the Euclidean norm of the input vector, each input is squared by a multiplier with equal inputs and then accumulated in a sum register. At this point, the square root of the sum of squared inputs is computed by the square root unit, that is composed of two look-up tables, a comparator and a multiplexer. The two look-up tables implement the square root function in two different ranges and the comparator selects one of the two LUT outputs with the aid of a multiplexer, depending on the range of the value on which the square root operation is performed. Specifically, one look-up table implements the square root function in the range $[3e-5, 4)$ with 2^{-7} step and the other one covers the range $[4, 5776)$ with 2^4 step. Keeping in mind that the approximate squash function should be used in a capsule network domain, the aforementioned numerical range of the square root unit input is derived from the range of possible norm values, that is experimentally obtained in the various case studies with two CapsNet models and two datasets, i.e. $(5.5e-3, 76)$.

The squashing unit computes the squash function outputs and is composed of a squashing coefficient approximation unit, that implements the piecewise approximation of the squashing function and a multiplier, that multiplies the approximate squashing coefficient by the input components to get the final outputs. The squashing coefficient

unit consists of two hardware computational branches. The first branch computes the approximate squashing function value in the range [0, 0.75), by exploiting a specific component to compute the negative norm value, $-x$, a natural exponential unit to calculate e^{-x} and a subtractor to obtain the approximate $1 - e^{-x}$ squashing function value. The second branch evaluates the squashing function in the norm range [0.75, 76), by using a LUT with step 2^{-1}. The correct squashing coefficient value is selected by a multiplexer and a comparator, depending on the input norm value. Finally, the input components are squashed by multiplying them to the computed squashing coefficient.

The proposed approximate squash architecture is able to process a variable number of input components, in order to be compliant with the capsule network domain, in the case studies with two CapsNet models and two datasets. In particular, the supported numbers of squash inputs are 4, 8, 16 and 32. The squash function in the ShallowCapsNet model takes as input 8 and 16 components in the PrimaryCaps and DigitCaps layer respectively, due to the capsule dimension in each layer. In the DeepCaps model, the squash function requires 4, 8 and 32 input components. Specifically, 32 components are required in the final capsule layer, where 32-dimensional output capsules are used, 4 components are processed in the convolutional layers of cell 1 and 8 components in the convolutional layers of cells 2, 3 and 4.

In order to allow the architecture to process a variable number of squash input components, a 7-bit generic modulo counter is employed to count the number of loaded inputs. Then the right group of input registers is selected by a multiplexer and an encoding unit, depending on the number of squash components to process.

In the proposed architecture, the input data bitwidth is <7; 9> in a signed fixed-point representation. The number of integer bits, 7, is required to represent the range of squash inputs in the case study DeepCaps for MNIST, in the convolutional layers of cell 2, as it is reported in figures 2.18 and 2.20. On the other hand, the number of fractional bits is 9, in order to provide the numerical resolution of squash inputs in the convolutional layers of all the cells in DeepCaps for MNIST.

As regards the output data bitwidth, the outputs of the squash function are represented by <1; 12>. Actually, the squash outputs require 1 integer bit because the numerical range of the squash function is (-1, 1), while the number of fractional bits used to represent the squash outputs is 12, as it is derived from the case study ShallowCapsNet for Fashion-MNIST in the PrimaryCaps and DigitCaps layers.

In the proposed architecture, the address saturation block at the input of a look-up table is used to map a given range of input addresses to the same LUT value, in order to saturate the look-up table output when the input is larger than a given threshold. In particular, in the second LUT of the square root unit, the table output is saturated for inputs larger than $5776 = 361 \cdot 2^4$ and in the LUT of the squashing unit, the saturation of the output is performed for inputs larger than $76 = 152 \cdot 2^{-1}$.

In the following paragraph, a description of the state machine diagram of the squash_exp processing unit is reported. After the activation of the input reset signal, the processing unit registers and counter are set to zero. At the next active clock edge, the processing unit transitions to the idle state, where it waits for input data. Available data on the input data bus is signalled by the input signal VIN. At the clock tick following the VIN activation, the

processing unit performs a transitions to the load_acc state, where it executes three main operations: (1) loading the inputs in the internal registers; (2) counting the number of loaded inputs; (3) square and accumulate operation of each input value for the Euclidean norm computation. The end of the input data is signalled by the VIN deactivation and the processing unit stops loading inputs, counting inputs and accumulating squared inputs. At the next clock tick after the VIN deactivation, the counter value is stored in a register for later computations, the counter is reset and the processing unit moves to the output state. In the output state, the processing unit performs the square root operation to compute the norm of the input vector, then calculates the approximate squashing coefficient and finally, multiplies each input component by the computed squashing coefficient to get the final squash results. After the generation of all the squash output vector components, the sum register of the norm unit is reset and the processing unit goes back to the idle state, ready to begin a new squash computation.

3.3.2 Squash-pow2

From the algorithmic point of view, the approximate squash_pow2 function focuses on the approximate computing of the squashing coefficient, by exploiting a piecewise approximation in two ranges of norm values. In particular, the squashing function is approximated by the function $1 - 2^{-x}$ in the range [0, 1) and by a direct-value look-up method in the range [1, max), as it is shown in figure 2.7 b. In the norm value 0.5, the non linear function $1 - 2^{-x}$ introduces a worst-case approximation error equal to 25% the exact squashing function value. The proposed squash_pow2 approximation can be considered as a variant of the squash_exp approximation, described in the previous section. In particular, the natural exponential computation, used in the squash_exp approximation, is replaced by a base-2 exponential operation in the current approximate squash function, in order to allow for a reduction of the complexity of the future hardware implementation. The hardware cost reduction is obtained at the expense of a higher worst-case approximation error of the squashing coefficient in the lower range of the piecewise approximation. As in the squash_exp algorithm, in the current squash_pow2 approximation, the norm is computed as an Euclidean norm, by implementing the square root operation with an input-output mapping method. At the last step of the algorithm, the squash outputs are computed by multiplying each squash input by the approximate squashing coefficient.

The architecture of the proposed squash_pow2 approximation is based on the implementation presented in the previous section. Two main modifications are introduced in the squashing unit. First of all, the constant multiplier by $\log_2 e$ is removed from the left branch of the squashing unit, in order to implement a base-2 exponential unit. Secondly, the selection between the two hardware branches of the squashing unit is performed by a comparator that takes as input the norm value and the new intermediate value 1, separating the two approximation ranges of the squashing coefficient. It is important to note that the architecture is still able to process a variable number of input components, that is 4, 8, 16 or 32 input values, so that to be used in the capsule network domain.

The state machine diagram of the squash_pow2 processing unit has the same topology of that of the squash_exp approximation, described in detail in the previous section. It is

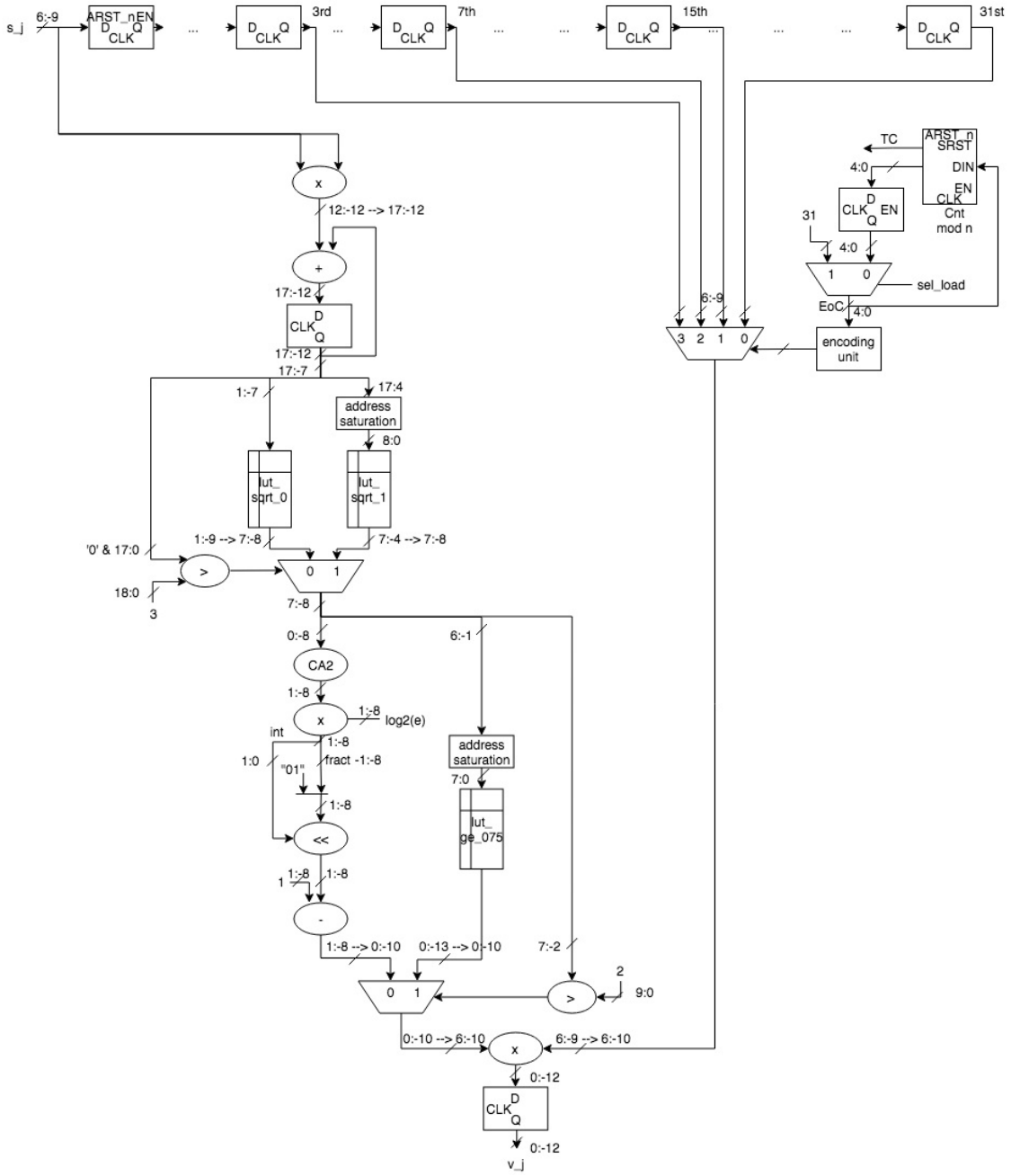


Figure 3.6: Squash-exp datapath

important to observe that, in the output state, the processing unit performs the square root operation, the squashing coefficient computation and the multiplication of the squash inputs by the computed squashing coefficient. In particular, in the squashing function left branch, a base-2 exponential operation is performed to approximate the squashing coefficient with $1 - 2^{-x}$ in the norm range [0, 1).

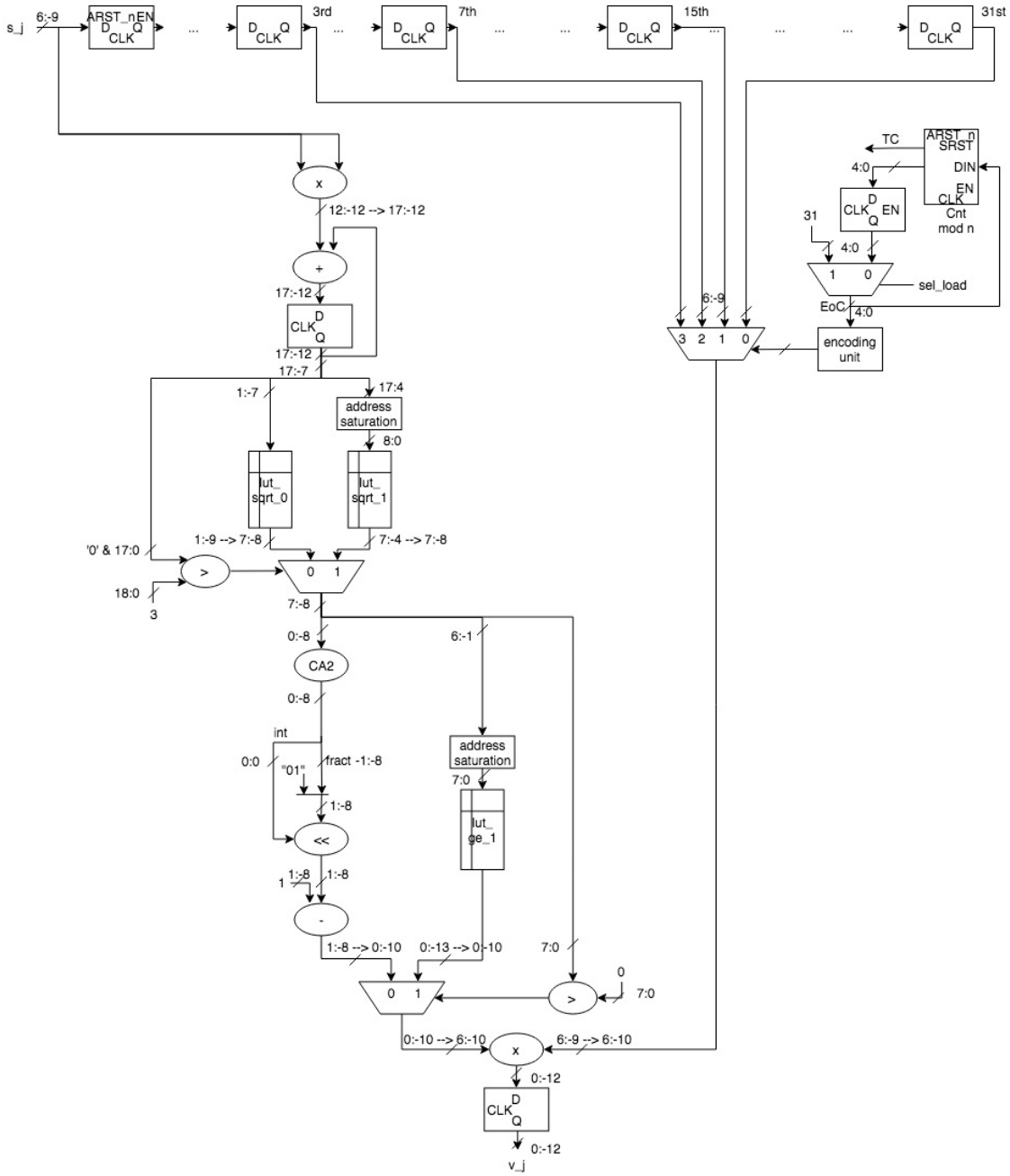


Figure 3.7: Squash-pow2 datapath

3.3.3 Squash-norm

The main focus of the squash_norm approximation is to approximate the Euclidean norm involved in the squash function. The Euclidean norm of the squash input vector is approximated by using the Chaudhuri et al.'s approximation, presented in the research paper [14] and shown in (3.1).

$$D_\lambda(\mathbf{x}) = |x_{i_{max}}| + \lambda \sum_{\substack{i=1 \\ i \neq i_{max}}}^{n} |x_i| \tag{3.1}$$

In an equivalent way, the approximation can be expressed as a linear combination of D_∞ and D_1, $D_\lambda(\mathbf{x}) = (1-\lambda)\, D_\infty(\mathbf{x}) + \lambda\, D_1(\mathbf{x})$, where D_∞ is the chessboard metric, $D_\infty(\mathbf{x}) = |x_{i_{max}}|$, and D_1 is the city-block norm, $D_1(\mathbf{x}) = \sum_{i=1}^{n} |x_i|$. By comparing the selected approximation and the Euclidean norm formula, $\|\mathbf{x}\|_2 = \sqrt{(\sum_{i=1}^{n} {x_i}^2)}$, it is clear that the goal of the approximation is to reduce the high computational cost of the Euclidean norm computation, due to the multiple multiplications required to square each input component and the square root operation. In place of the repeated multiplications and the square root calculation, the Chaudhuri's norm approximation introduces two main operations: (1) the computation of the absolute values of the vector components; (2) the comparisons among absolute values to determine the maximum absolute component. Beside avoiding the square root and the square of the input components, an additional advantage of the norm approximation is to limit the number of required multiplications to a fixed number, 1, i.e. the multiplication of the sum of absolute values by the parameter λ, regardless of the number of vector components, n. Moreover, the number of additions involved in the approximation is still equal to $n-1$, as in the exact Euclidean norm computation. The parameter λ used in the approximation depends on the number of input vector components and the λ optimal values are derived from the research paper [15]. It is interesting to note that error introduced by the norm approximation increases with the number of input components, as reported in [14], while the λ parameter decreases with n. The squashing coefficient is computed by using a direct-value look-up method in two different ranges of norm values, as explained in the following section. Finally, the squash output results are obtained by multiplying each input by the calculated squashing coefficient.

At the architecture level, the proposed design is composed of two main units, i.e. the norm unit to compute the approximate Euclidean norm of the squash input vector and the squashing unit to calculate the squashing coefficient and the squash output vector components.

The norm unit computes the Chaudhuri et al.'s approximate norm according to the formula (3.1). A specific hardware component is used to compute the absolute values of the input components. Then, the absolute components are accumulated in a sum register and a maximum finder unit, consisting of a register and a comparator, determines the largest absolute component. To get the second sum term of the Chaudhuri's approximation, the maximum absolute component is subtracted from the sum of n absolute values and the result of the subtractor is scaled by the parameter λ by using a multiplier. The final approximate norm value is obtained by adding the maximum absolute value to the output

of the λ multiplier. The right value of λ is selected by using a multiplexer, depending on the number of components of the squash input vector.

The squashing unit consists of two look-up tables, a comparator and a multiplier. The two look-up tables are used to compute the squashing coefficient in two different ranges of norm values. In particular, the first LUT implements the squashing function in the range $[5.5e-3, 0.5)$ with step 2^{-6}, while the second LUT covers the range $[0.5, 76)$ with step 2^{-1}. The correct look-up table output is selected by using a multiplexer and a comparator, that compares the norm value to the intermediate value, 0.5. Finally, a multiplier is employed to scale each squash input by the computed squashing coefficient and obtain the squash output vector components. The proposed architecture is able to process a variable number of input components, to be compliant with the capsule network domain in 4 case studies, with two CapsNet models and two datasets. Specifically, a modulo generic counter is included to count the number of squash inputs. Depending on the number of components of the squash input vector, an encoding unit selects the right group of registers where the inputs are stored and the specific λ value to be used in the norm computation for the current number of inputs.

In the following section, the state machine diagram of the squash_norm processing unit is described. After the activation of the input reset signal, the registers and counter are set to zero. At the next active clock edge, the processing unit moves to the idle state, waiting for input data. As soon as data is available and signalled by the VIN signal activation, the processing unit transitions to the load_acc_max state, where it performs 5 main operations: (1) load the inputs for the following squash outputs computation; (2) compute the absolute value of each input data; (3) accumulate the absolute values; (4) search the maximum absolute component; (5) count the number of loaded inputs. At the end of the input data, the VIN signal is deactivated and the processing unit stops loading inputs, accumulating absolute values, looking for the maximum value and counting the number of loaded inputs. At the clock tick after the VIN deactivation, the counter value is stored in a register and the counter is reset. The processing unit moves to the output state, where the Chaudhuri's norm computation is completed, the squashing coefficient is calculated and the squash output vector components are obtained, by multiplying each input by the squashing coefficient and stored in an output register. Finally, after the generation of all the squash outputs, the sum register in the norm unit is reset and the processing unit goes back to the idle state, ready to perform a new squash function computation.

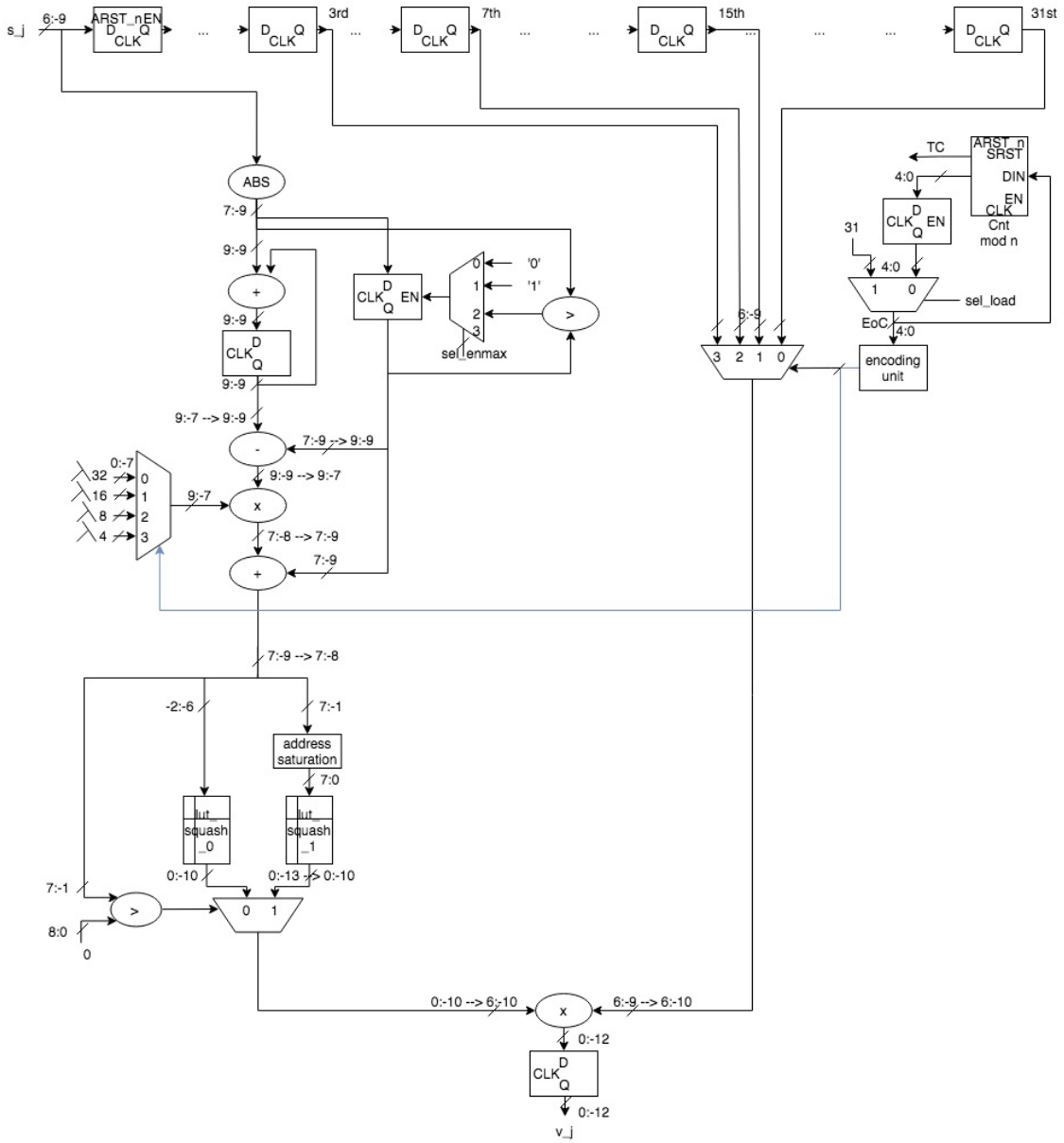
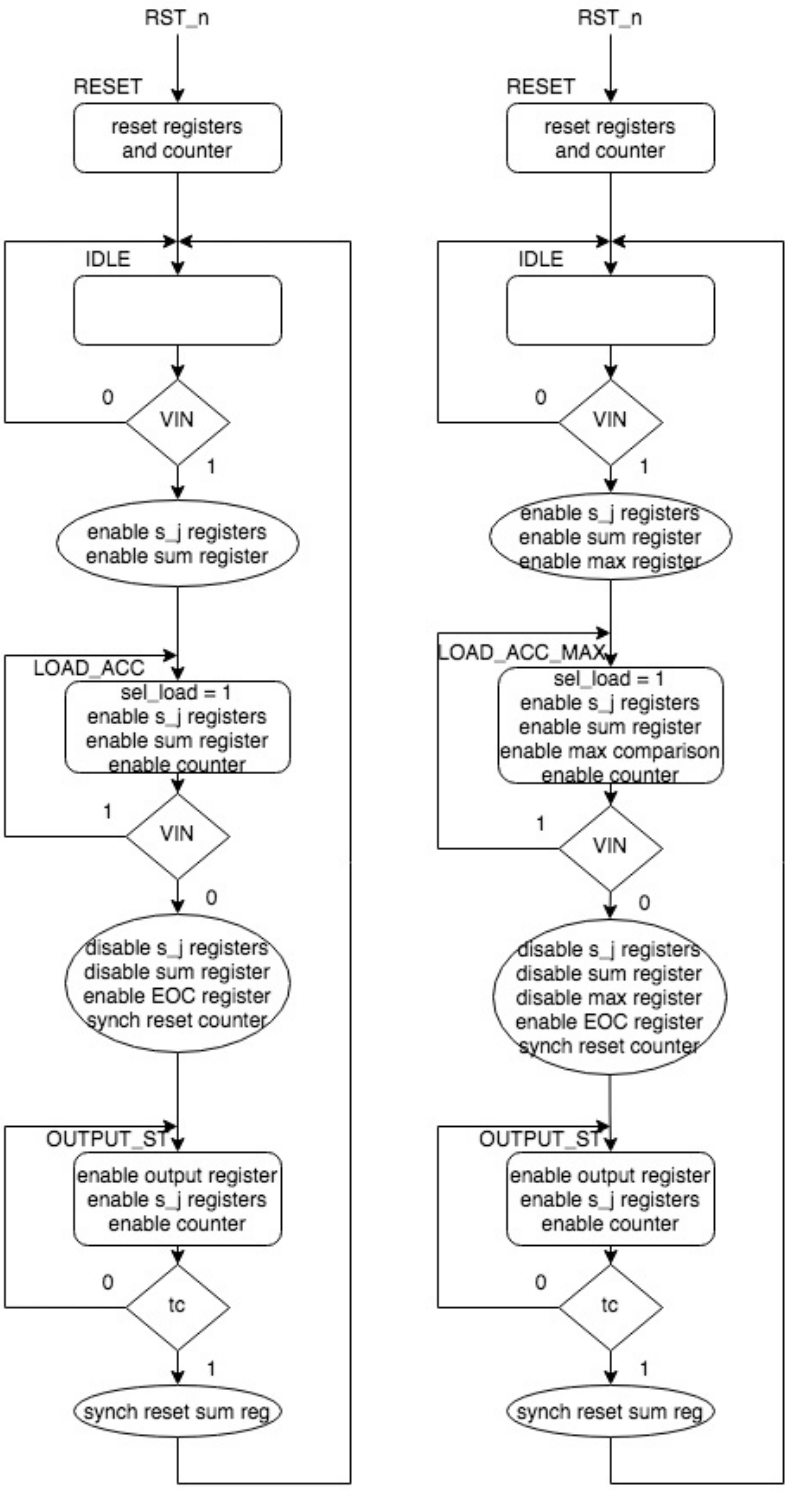


Figure 3.8: Squash-norm datapath



(a) *Squash-exp et Squash-pow2* **(b)** *Squash-norm*

Figure 3.9: Squash control units

3.4 RTL implementation and functional simulation

The next step is to describe in VHDL the proposed approximate softmax and squash processing units. The VHDL model of each processing unit is composed of a datapath and a control unit.

3.4.1 VHDL models of approximate Softmax and Squash processing units

The top-level entity of each processing unit is identified by the name of the softmax or squash approximation and includes the datapath and control unit as interconnected components in a structural architecture. The interface of the top-level module consists of four input signals and two output signals. The input signals are the reset signal, clock signal, valid input data vin signal and input data bus din. As output signals, the processing unit uses the output data bus dout and valid output data vout signal.

As regards the interconnection between datapath and control unit, the datapath sends status signals to the control unit and the control unit provides control signals to the datapath. In the proposed designs, the status signals are terminal count signals of the counters used in the datapath, while the control signals are represented by enable and synchronous reset signals of registers and counters or selection signals of multiplexers.

The datapath VHDL module takes as input the clock signal, input data bus din and control signals and produces as output the status signals, output data bus dout and valid output data vout signal. As regards the architecture, the datapath is described in a structural way, by instantiating the components that represent the various hardware units composing the datapath itself. The VHDL description of each component is behavioural and is included in separate source files. The components instantiated in the datapath can be classified into a number of groups: registers and counters, computational units (adder, subtractor, multiplier, comparator), look-up tables, multiplexers and specific units, like leading one detector and encoding unit. Moreover, the unary minus, absolute value and shift operations are performed by exploiting the corresponding functions in the *numeric_std* package of the ieee library, that are `-`, `abs` and `srl` or `sll`.

As regards the data type used in the VHDL description of the processing units, the numeric type *signed* is exploited, as defined in the package *numeric_std*. The chosen data type is suitable for use with logic synthesis tools. In the design, a signed fixed-point number representation is adopted to represent both positive and negative numbers, in a given range and with a specific numerical resolution. In order to ease the VHDL modelling phase in the early stages, a signed fixed-point numeric type, called *sfixed*, is used as defined in the special package *fixed_pkg*, that allows for an easy and immediate identification of the integer and fractional part of the represented signed fixed-point numbers.

In the proposed designs, the control unit VHDL module takes as input the reset signal, clock signal, valid input data vin signal and datapath status signals. As output signals, the control unit produces the control signals connected to the datapath components. The VHDL description of the control unit is based on three process statements, i.e. the state transitions process, next sate network process and output network process. In particular,

the state transitions process is sensitive to the reset and clock signals and allows the processing unit to perform transitions between states. The next state process is in charge of computing the next state from the current state and the control unit input signals. Finally, the output process generates the control signals in each state, possibly depending on some control unit inputs. In the proposed processing units, a Mealy finite state machine is used, where the output signals depend not only on the current state, but also on some input signals of the FSM. The motivation behind the choice of a Mealy finite state machine is that the use of mealy outputs allows for a reduction of the overall number of states of the FSM. As a drawback, a Mealy finite state machine is not glitch-free, so glitches on the inputs may lead to unwanted outputs signals transitions.

3.4.2 Logic Simulation Workflow

The next step in the process is to perform the functional simulation of the VHDL model of each processing unit. The goal is to verify that the RTL architecture described in VHDL implements correctly the approximate softmax or squash algorithm described in Python. In order to asses the functionality of each proposed design, the VHDL model outputs are compared to the expected results given by the Python model. Specifically, the quantised Python description of the approximation algorithm is used as a term of comparison for the VHDL model. The VHDL model outputs are obtained by exploiting a VHDL-based testbench, which allows to perform the functional testing of each processing unit. On the other hand, the expected results are determined by executing a simulation of the Python program describing the approximate softmax or squash algorithm, implemented by the VHDL architecture under test. Both the VHDL and Python model outputs are stored into a dedicated text file for later processing and comparison of the results. To perform the VHDL and Python simulations, input test vectors are generated by a Python program and applied to both the VHDL and Python model of the proposed designs. In order to compare the values in the aforementioned two output files, a Python script is used, that allows to compute the error distance between each VHDL model output and the corresponding expected result. The computed error statistics are stored in a log file for visual comparison.

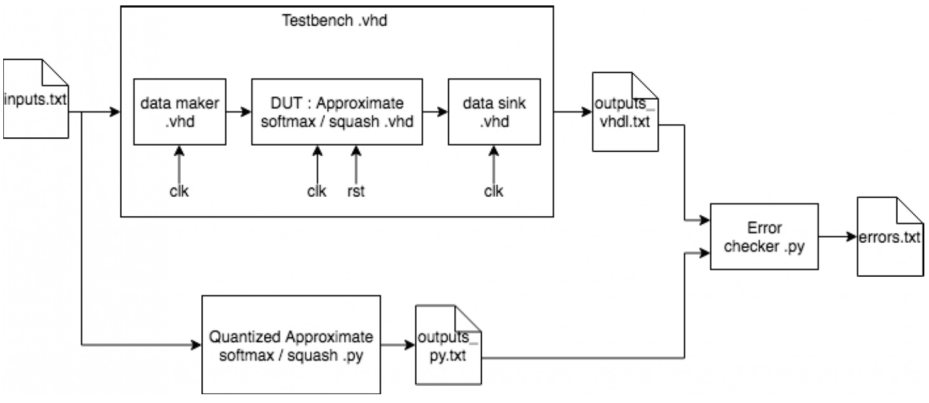


Figure 3.10: Logic simulation setup

In the following section, a detailed description of the simulation flow is reported. The simulation is performed by means of four main modules. A Python program, inputs_maker.py, generates the input test vectors to be applied to the approximate softmax and squash designs and saves them to a text file, inputs.txt. For the simulation of the approximate softmax designs, 500 input test vectors are generated with a number of components equal to 10, 32 or 128. The components of the test vectors are generated with an uniform distribution in the numerical range [-4, 4). On the other hand, in the case of the approximate squash designs, the number of input test vectors is 500 with 4, 8, 16 or 32 components having a uniform distribution in the range [-64, 64).

The Python quantised model of the approximate softmax or squash processing units is simulated by executing the corresponding .py file. In particular, the Python model reads the inputs test vectors from the inputs.txt file, performs the specific approximation algorithm of the softmax or squash function and writes the output vectors results to a text file, outputs_py.txt. Before using the input vector components, the Python model performs a quantisation of the floating-point component values read from the inputs.txt file into fixed-point numbers, by using a truncation rounding scheme. The specific numerical resolution used for the approximate softmax designs is 2^{-8}, while the approximate squash designs input vector components are quantised with a resolution equal to 2^{-9}.

The simulation of the VHDL model is performed by means of a VHDL testbench structure. The testbench module includes four components, that are properly interconnected to allow for the testing of the RTL design. The four components involved in the VHDL testbench are the unit under test, the clock and reset signals generator, the data maker and the data sink module. The unit under test component is the VHDL model of the processing unit, as it is described in the previous section. The clock and reset generator generates the clock and reset signals, that are needed by the processing unit. The clock signal is required also by the data maker and data sink modules, in order to synchronise their specific operations to the timing behaviour of the processing unit. The clock and reset generation block has an input signal, end_sim, that is used to deactivate the clock signal at the end of the simulation, that is at the end of the input data. The data maker is in charge of applying the input data to the unit under test. In order to properly generate the input data to the UUT, the values in the inputs.txt file are pre-processed, so that each floating-point value is converted to the integer value corresponding to a fixed-point representation with a given numerical resolution, according to the formula $\lfloor x \cdot 2^{NF} \rfloor$. The data maker module reads the pre-processed input values and converts each integer value to a signed vector of the specified size, as required by the UUT data bus din. At each active clock edge, the data maker places an input value on the input data bus din of the UUT and asserts the valid input vin signal. After having applied n components to the UUT, the data maker deactivates the vin signal of the unit under test and waits the processing unit to finish the computation of the first output vector. As soon as the processing unit ends the computation of the first vector, it deactivates its vout signal and after the deactivation of the vout signal, the data maker starts applying new input data to the UUT. This process is repeated until the end of the input file. After the last input vector is processed by the processing unit, the data maker asserts the end_sim signal to deactivate the clock signal and stop the processing. Actually, the processing unit computes each output vector

component and asserts the vout signal for each valid output component. At each clock tick following an output component computation, the data sink module performs a file writing operation. In particular, the data sink module reads the output signed vector from the output data bus dout of the UUT, converts it to a fixed-point number with a given numerical resolution, 12, and stores the output value in a text file, outputs_vhdl.txt.

The last step of the simulation flow is to compare the VHDL model outputs to the Python model expected results, i.e. to make a comparison between the values stored in the two output text files, outputs_vhdl.txt and outputs_py.txt. A Python program is used to compute the error distance between the VHDL simulation and Python simulation results. The program, called error_checker.py, reads the two files and writes the error distances in a text file, errors.txt, for visual analysis. The computed error metrics are described in the following paragraph. For each output vector, 8 error metrics are computed. The error metrics can be classified in component-wise and vector-wise. The component-wise metrics are 4: absolute component error and its magnitude, relative component error and its magnitude. As regards the metrics related to the whole vector, other 4 metrics are calculated, i.e. average absolute component error magnitude, maximum absolute component error magnitude and average and maximum relative component error magnitude. By visually analysing the content of the errors file produced by the simulation flow, it is possible to verify the functionality of the VHDL model of each approximate softmax or squash processing unit.

Component absolute error	Component absolute error magnitude	Maximum component absolute error magnitude	Average component absolute error magnitude	Component relative error	Component relative error magnitude	Maximum component relative error magnitude	Average component relative error magnitude
0.000001	0.000001	0.000001	0.000000	0.001280	0.001280	0.001280	0.000707
-0.000000	0.000000	0.000001	0.000000	-0.000237	0.000237	0.001280	0.000707
-0.000001	0.000001	0.000001	0.000000	-0.000914	0.000914	0.001280	0.000707
0.000001	0.000001	0.000001	0.000000	0.001280	0.001280	0.001280	0.000707
0.000000	0.000000	0.000001	0.000000	0.001280	0.001280	0.001280	0.000707
0.000001	0.000001	0.000001	0.000000	0.000711	0.000711	0.001280	0.000707
0.000000	0.000000	0.000001	0.000000	0.001280	0.001280	0.001280	0.000707
-0.000000	0.000000	0.000001	0.000000	-0.000090	0.000090	0.001280	0.000707
0.000000	0.000000	0.000001	0.000000	0.000000	0.000000	0.001280	0.000707
0.000000	0.000000	0.000001	0.000000	0.000000	0.000000	0.001280	0.000707

Figure 3.11: Errors.txt file sample for Softmax-b2

By analysing the log file associated to each processing unit, it is clear that the designed RTL architectures implement properly the approximate softmax or squash algorithms. For each approximate design, the mean error distances of the VHDL outputs with respect to the Python quantised model outputs are reported in tables 3.12 and 3.13. The 4 mean error distances are computed by calculating the average value of each of the 4 vector-wise error metrics over the total number of output vectors.

After the functional verification of the VHDL models, the logic synthesis of the approximate softmax and squash units is performed, by using the Synopsys Design Compiler.

Softmax approximation	MED on mean absolute errors	MED on mean relative errors	MED on max absolute errors	MED on max relative errors
softmax-lnu	(2.5 ± 0.5) e-07	0.0098 ± 0.0060	(4.5 ± 0.4) e-07	0.0454 ± 0.0197
softmax-b2	(2.5 ± 0.6) e-07	0.0016 ± 0.0007	(4.9 ± 0.6) e-07	0.0057 ± 0.0017
softmax-taylor	(2.2 ± 0.5) e-07	0.0091 ± 0.0059	(4.6 ± 0.5) e-07	0.0440 ± 0.0209

Figure 3.12: Softmax logic simulation results

Squash approximation	MED on mean absolute errors	MED on mean relative errors	MED on max absolute errors	MED on max relative errors
squash-exp	(2.5 ± 0.5) e-07	0.0003 ± 0.0007	(4.4 ± 0.5) e-07	0.0016 ± 0.0054
squash-pow2	(2.5 ± 0.5) e-07	0.0003 ± 0.0007	(4.4 ± 0.5) e-07	0.0016 ± 0.0054
squash-norm	(2.5 ± 0.5) e-07	0.0003 ± 0.0007	(4.4 ± 0.5) e-07	0.0016 ± 0.0054

Figure 3.13: Squash logic simulation results

3.5 Logic Synthesis

The goal of the logic synthesis process is to obtain a netlist that implements the proposed design by using standard logic gates of a selected technology library. Based on the logic synthesis results, detailed information about the hardware implementation of each design is derived. In particular, each design is characterised by three main hardware metrics, i.e. area usage, power consumption and maximum path delay. By analysing the obtained hardware information, the approximate softmax or squash designs are compared in terms of their hardware complexity and performance. The goal of the comparative analysis of the approximate softmax or squash designs in terms of hardware metrics is to explore the possible trade-offs between the hardware cost of each design and the capsule network inference accuracy loss caused by the softmax or squash function approximation. In the following section, a step-by-step description of the logic synthesis process is reported. The target technology library used to synthesise the RTL architectures is the 45 nm Nangate Open Cell Library, one of the most used libraries for academic research.

The logic synthesis process takes as input the source files containing the VHDL description of the design and generates as output the gate-level netlist for the design, described in Verilog and the text reports file. The netlist is an implementation made of library cells and the report files are used to characterise the synthesised architecture in terms of area, power and delay. The logic synthesis is performed by executing a Tcl script, start_syn.tcl, based on the Tool command language supported by the Design Compiler. Before starting the synthesis by means of the `compile` command, the design constraints are set to direct the optimisation of the design. In particular, the specified constraints are mainly timing specifications, i.e. the clock signal period and uncertainty, the delay on input and output ports relative to the clock signal and the capacitive load on the output

ports of the design, that is set equal to the capacitance of the input pin of a library buffer cell with drive strength 4. Finally, the `compile` command implements a combination of library cells that best meets the requirements specified for the design.

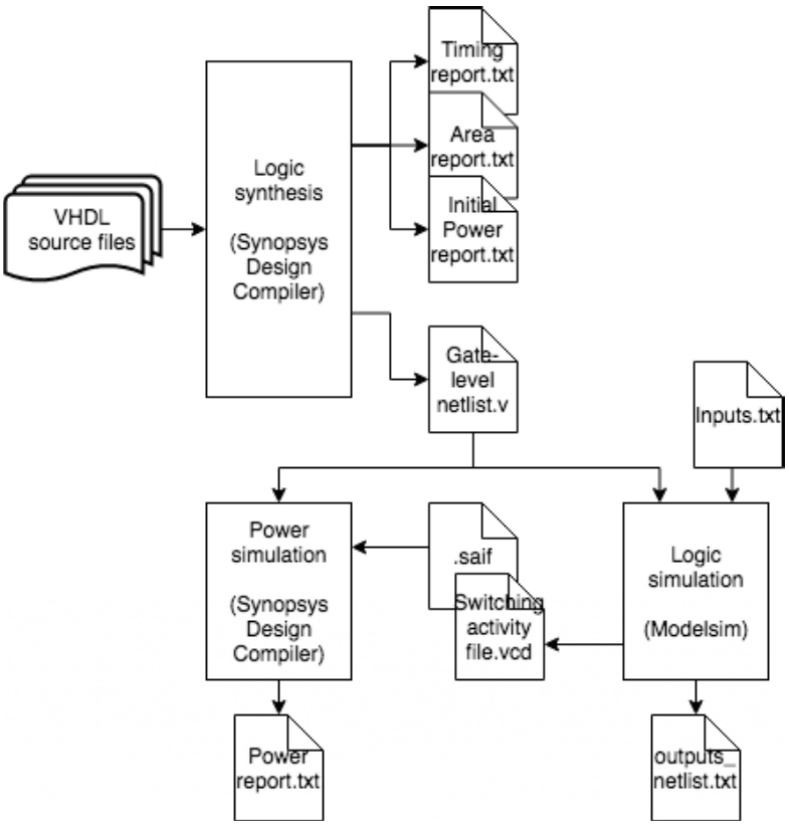


Figure 3.14: Logic synthesis workflow

3.5.1 Description of Synthesis Reports

In the following section, a description of the area, power and timing report files generated by the synthesis process is provided.

The area report provides the design size and cell counts. As regards the design size, the report file shows the total cell area of the design and the area breakdown in combinational and non combinational area contributions. In terms of cell counts, the total number of cells and the number of combinational and sequential cells are reported. The measurement units of the area values are square micron. Hence, the area report provides the area usage of the design and enables area breakdown analysis.

As regards the power report, it provides detailed information about the power consumed by the design. In particular, the power report shows the total power dissipated by the design and the three different type power contributions of the total power value. Actually, the total power is composed of the cell leakage power and the total dynamic power, that

is in turn split into cell internal power and net switching power. In addition to the aforementioned power values, a power breakdown of each contribution is provided. In particular, for each power type and the total power, the amount of power consumed by registers and combinational cells is reported. The measurement units of the leakage power are nW, while the dynamic power units are uW. It is important to note that the power analysis provided by the power report previously described is an initial estimate, because it is based purely on a statistical estimation of the nets switching activity.

As regards the timing analysis of the design, a timing report is generated that provides information about the maximum delay path in the design, i.e. the critical path for the design timing performance. In particular, based on the clock period, clock uncertainty and library setup time, the data required time is computed. Then, the data arrival time is estimated as the total delay of the critical path, by computing the incremental delay through each logic gate. Finally, the data arrival time is compared to the data required time, by calculating the difference between the two values, called slack. A positive slack means that the timing requirements are met by the design. The time units are nanoseconds. In addition to the aforementioned time values, the timing reports shows the starting point and end point, or net, of the critical path through the design, as well as the nets of the components that lie on the path itself. Since the goal of the synthesis process is to allow for a comparison of the multiple approximate softmax or squash designs, the alternative designs have equal clock period, i.e. 10 ns. The equal clock period choice enables a fair comparison since all the designs are running at the same cycle time.

3.5.2 Power consumption Experimental Analysis

In order to get a more accurate power report of the design, a power simulation is performed by using the Design Compiler. The goal of the power simulation is to obtain a power report of the design based on the experimental estimation of the nets switching activity, other than the statistical estimation previously adopted. At this step, the power analysis is realised by using the SAIF back-annotation process, that consists of two main parts.

First of all, the logic simulation of the Verilog netlist generated by the synthesiser is performed, by applying realistic input test vectors in a text file to obtain the experimental nets switching activity. A VHDL testbench is used as explained in the previous chapter and the compiled Verilog codes of the library cells in the netlist are linked. In particular, the simulation computes the nets switching activity by exploiting the information about the delays of the netlist, that are read from a .sdf file previously generated by the synthesis process. The HDL simulator writes the switching activities of the nodes of the unit under test in a .vcd file, for later use by the power analyser. Then, the vcd file is converted into a standard .saif file by the synthesiser, in order to enable the automatic annotation of the design nodes.

The second part of the process is performed by the Design Compiler: the experimental switching activities of the netlist nodes are set by reading the saif file and the power analysis based on the annotated switching activities is performed, to generate a reasonably accurate power report of the design. The clock period specified to the design compiler, to perform the power analysis properly, is equal to the cycle time used in the HDL simulation.

3.5.3 Post-synthesis netlist Validation

The last step in the design flow is to validate the gate-level netlist produced by the logic synthesis process. To perform the validation, the VHDL testbench is used to apply the input test vectors and record the gate-level model outputs, as described in the previous chapter for the VHDL model logic simulation. In particular, the timing validation of the gate-level netlist is performed, by linking the sdf delay file of the netlist and the pre-compiled Verilog codes of the library cells used in the netlist.

The gate-level model outputs are compared to the pre-synthesis VHDL model results, by automatic comparison of the corresponding text output files. To complete the validation process, the gate-level model outputs are checked against the Python quantised model results, as it is shown in figure 3.15. By observing that the Verilog gate-level model outputs coincide with the pre-synthesis VHDL model outputs, the log text file errors.txt can be derived as explained in the previous chapter. By analysing the validation results, it is possible to conclude that the synthesised gate-level netlist of each approximate softmax or squash design implements properly the corresponding approximation algorithm.

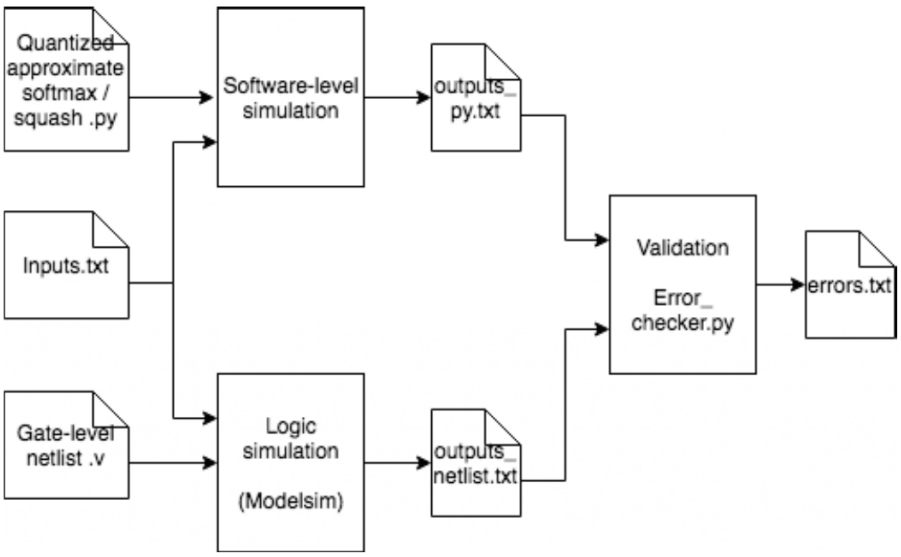


Figure 3.15: Post-synthesis netlist validation

3.6 Synthesis results

The logic synthesis of the approximate softmax and squash designs allows to characterise each architecture in terms of the three hardware metrics, area usage, power consumption and maximum path delay. In particular, the synthesis results for each softmax or squash processing unit are reported in this section.

3.6.1 Area usage

As regards the area characterisation of the designs, two main charts are reported. The bar chart shows the total chip area occupied by the design in square micron, with two area contributions given by combinational and sequential cells. The fractions of total area occupied by combinational and sequential cells are shown in a pie chart with percent values. For the softmax designs, the fraction of total area occupied by sequential cells is larger than the fraction corresponding to combinational cells. On the contrary, in the squash designs, combinational logic uses more area than sequential logic. The two different area breakdowns reflect the fact that in the softmax architectures a larger number of registers is used, while in the squash designs the higher combinational area contribution is due to limited number of registers and use of area-expensive look-up tables.

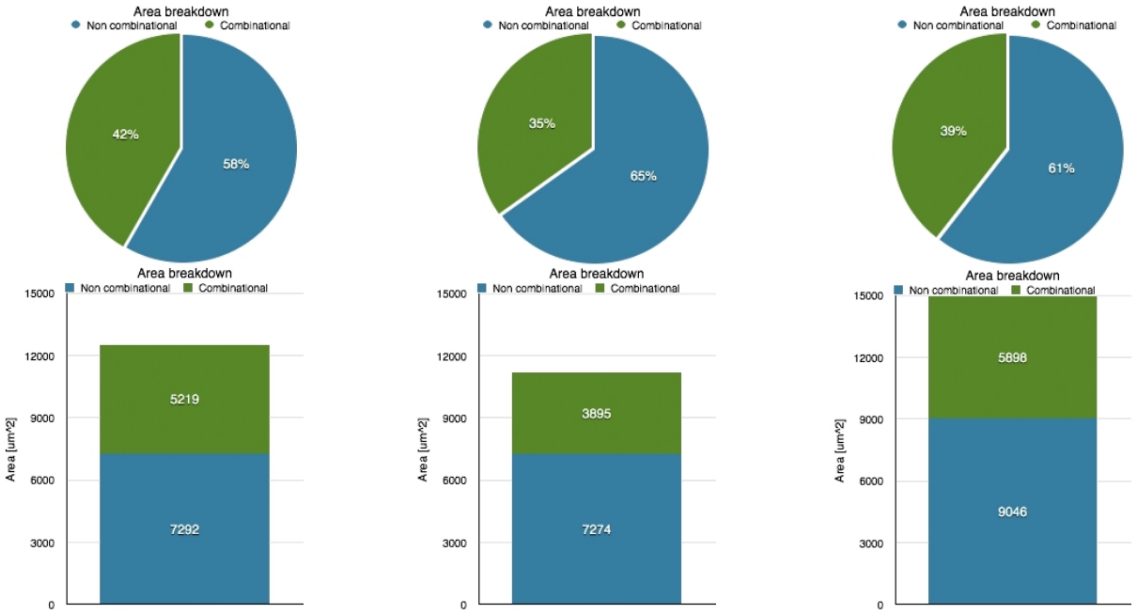


Figure 3.16: Softmax Area results (lnu – b2 – taylor)

3.6.2 Timing performance

As regards the timing characterisation, the bar chart reports the delay of the design critical path in nanoseconds. Specifically, for the longest path, the data arrival time and the slack with respect to the cycle time are shown. Actually, the effective slack is lower due to the library setup time and clock uncertainty constraint. The critical path of each softmax or squash design is highlighted in appendix A.

3.6.3 Power consumption

The power analysis results of the back-annotation process are reported by using three main charts. The first bar chart shows the total power consumed by the design, that is broken

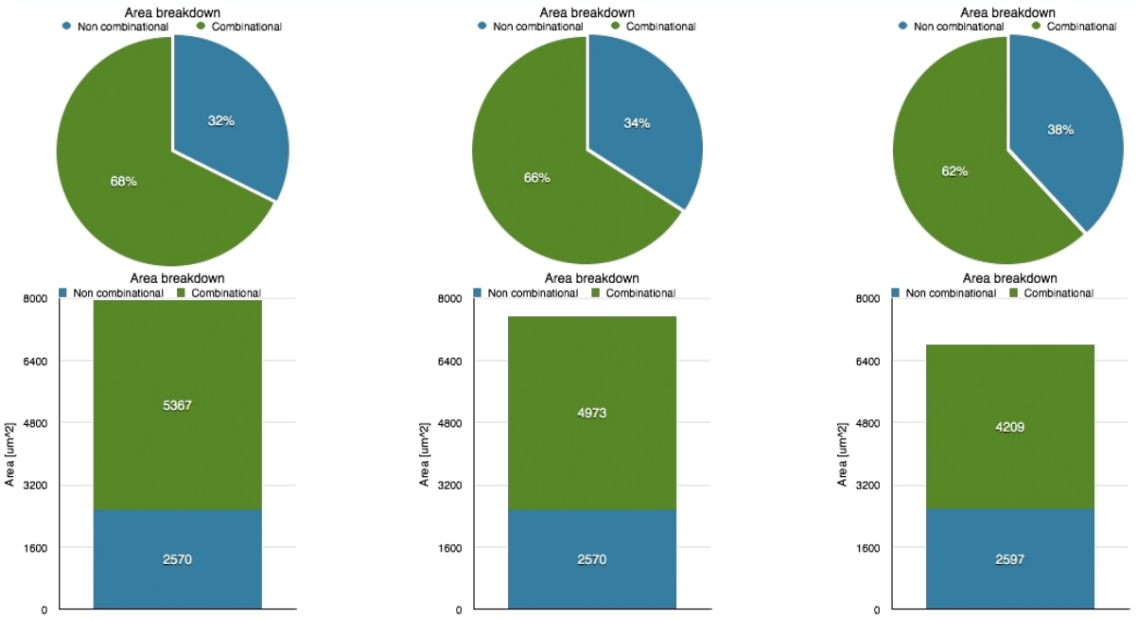


Figure 3.17: Squash Area results (exp – pow2 – norm)

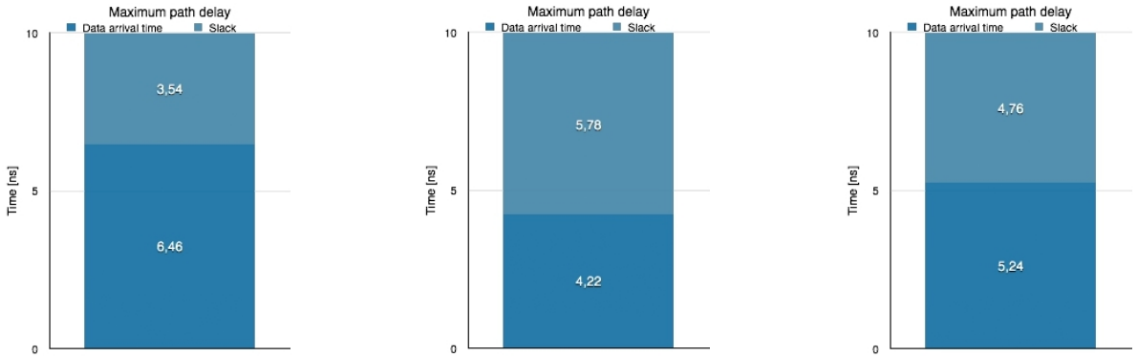


Figure 3.18: Softmax Timing results (lnu – b2 – taylor)

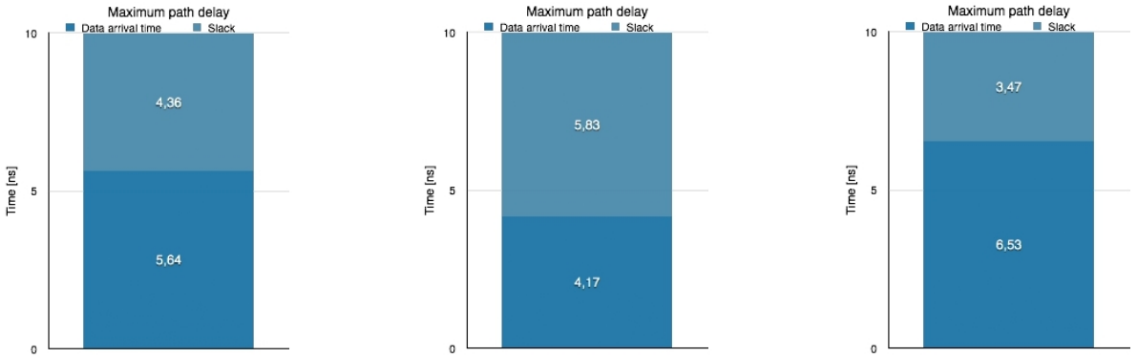


Figure 3.19: Squash Timing results (exp – pow2 – norm)

into internal, switching and leakage power. In the second bar chart, the power is also broken down into power consumed by registers and combinational logic. In particular, the total power breakdown in sequential and combinational cells is reported in a pie chart with percent values. For the softmax or squash designs, it is interesting to note that the power breakdown trend reflects the area breakdown. Specifically, in the softmax designs, registers consume more power and more area than combinational logic (60% to 40% of total area or power). On the other hand, in the squash designs, combinational cells use more power and occupy more area than sequential cells (70% to 30% of total area or power).

Concerning the power contributions, internal and switching power are forms of dynamic power, while leakage power is a form of static power. Cell internal power is the short-circuit power, related to the fact that CMOS transistors are arranged in pull-up and pull-down networks that can be both in conduction. It depends on transistor size, input transition time and output load capacitance of the cell. Net switching power is the power dissipated by the charging and discharging of the load capacitance at the output of each cell. Finally, cell leakage power is the power due to the transistor sub-threshold leakage. It si interesting to note that in the softmax and squash designs, the dynamic power is more significant than the static power (90% to 10% of total power).

In the following section, the comparative analysis of the softmax or squash processing units is reported. First of all, the softmax or squash designs are compared in terms of the three hardware metrics, area, power and delay, computed by the logic synthesis process. Secondly, a detailed comparison of the softmax or squash units is performed in terms of capsule network inference accuracy loss in 4 case studies, that is caused by the proposed softmax or squash function approximations. Finally, both the hardware characterisation and inference accuracy results are analysed in order to explore possible trade-offs between hardware complexity and classification accuracy of the proposed approximate softmax or squash designs.

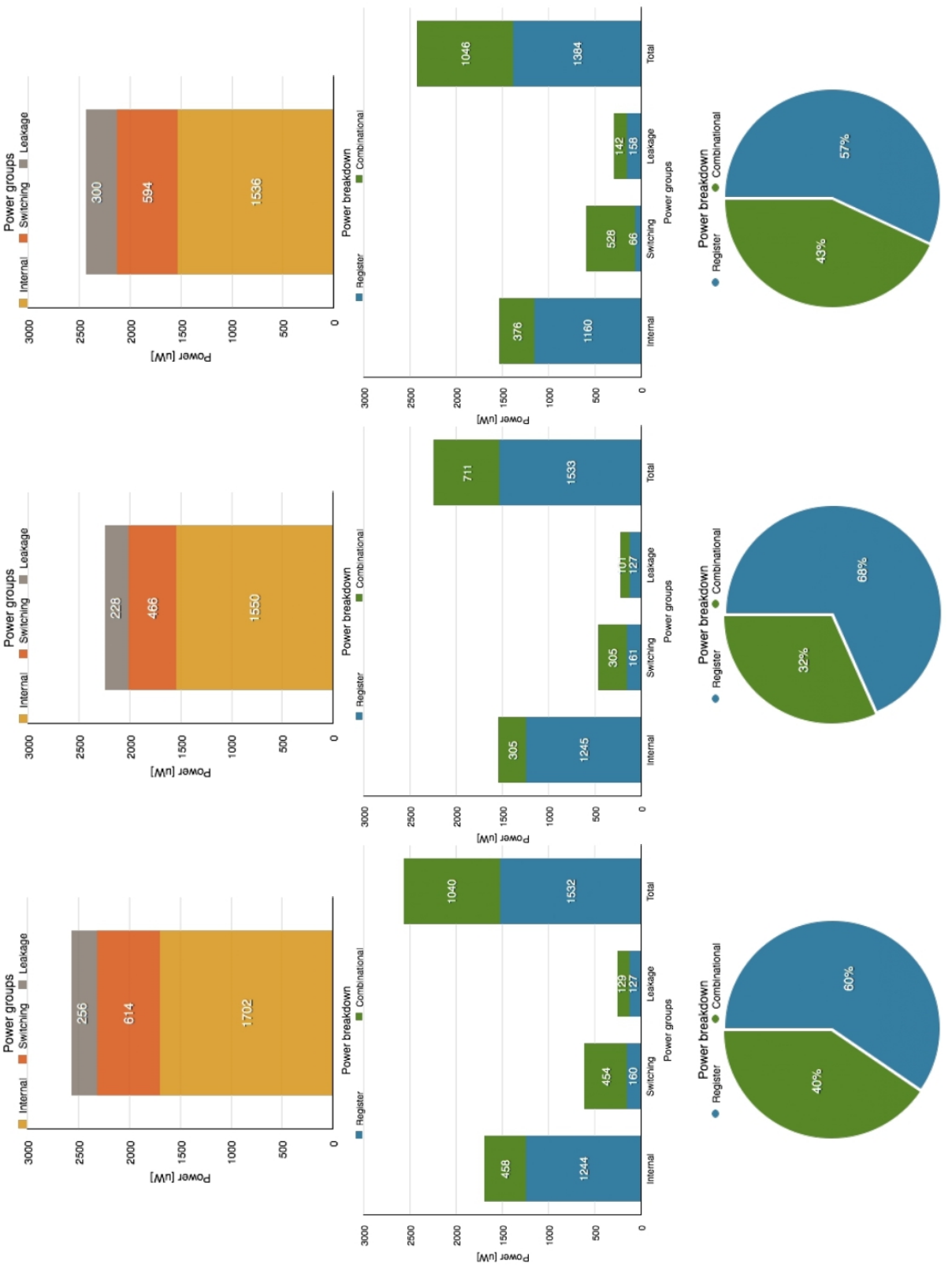


Figure 3.20: Softmax Power results (lnu – b2 – taylor)

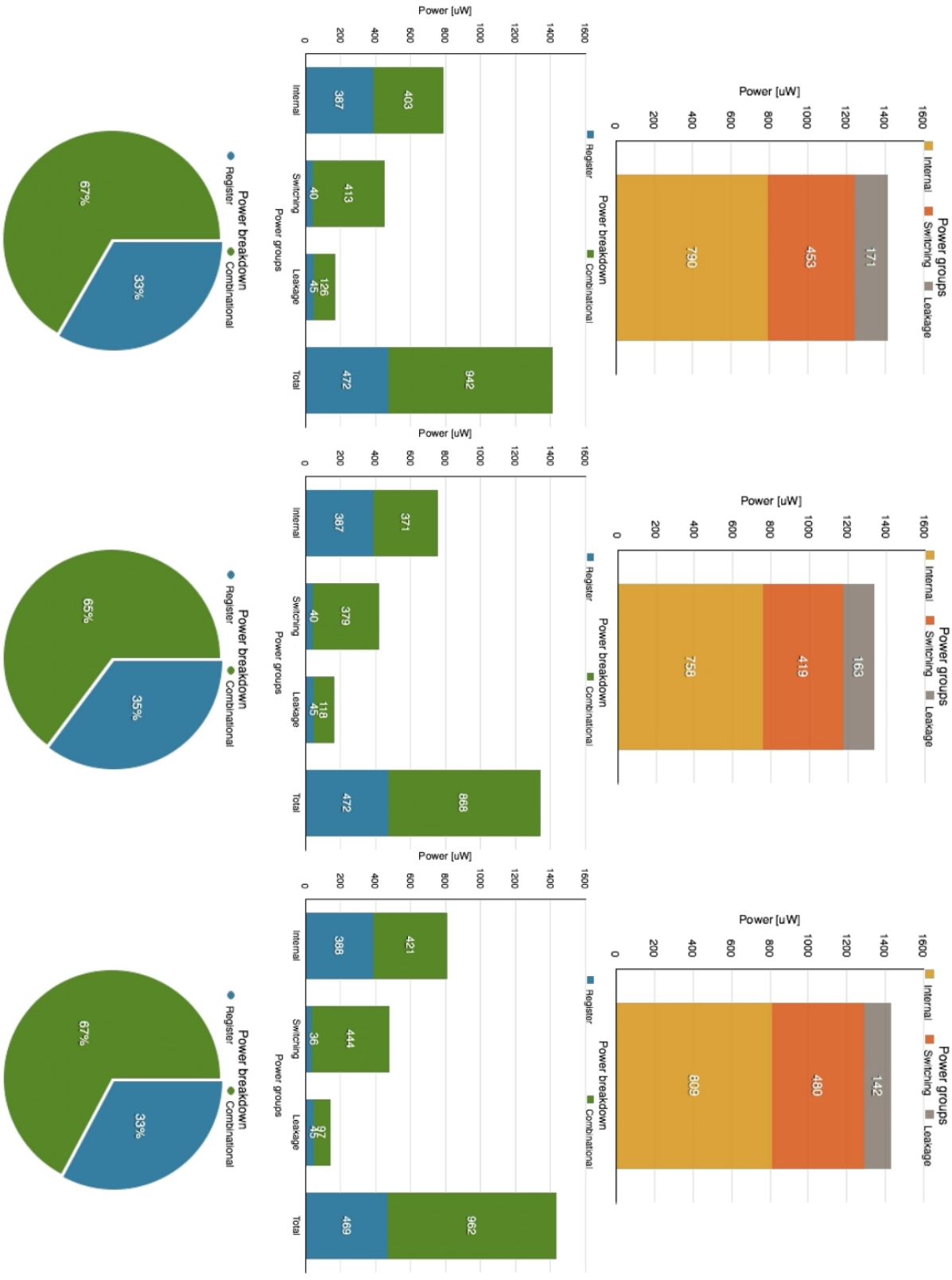


Figure 3.21: Squash Power results (exp−pow2−norm)

3.7 Comparative Analysis of Softmax Approximations

3.7.1 Comparison by Hardware metrics

As regards the softmax units, the comparative analysis in terms of hardware metrics is shown in the three bar charts reported in figure 3.22. The first bar chart reports the total area occupied by each softmax design; the second bar chart shows the total power consumed by the softmax units and the third chart reports the data arrival time in the critical path of each softmax design. By analysing the three bar charts, the three approximate softmax processing units, Softmax_lnu, Softmax_b2 and Softmax_taylor, are compared to find out how the softmax designs differ in terms of area, power and critical path delay.

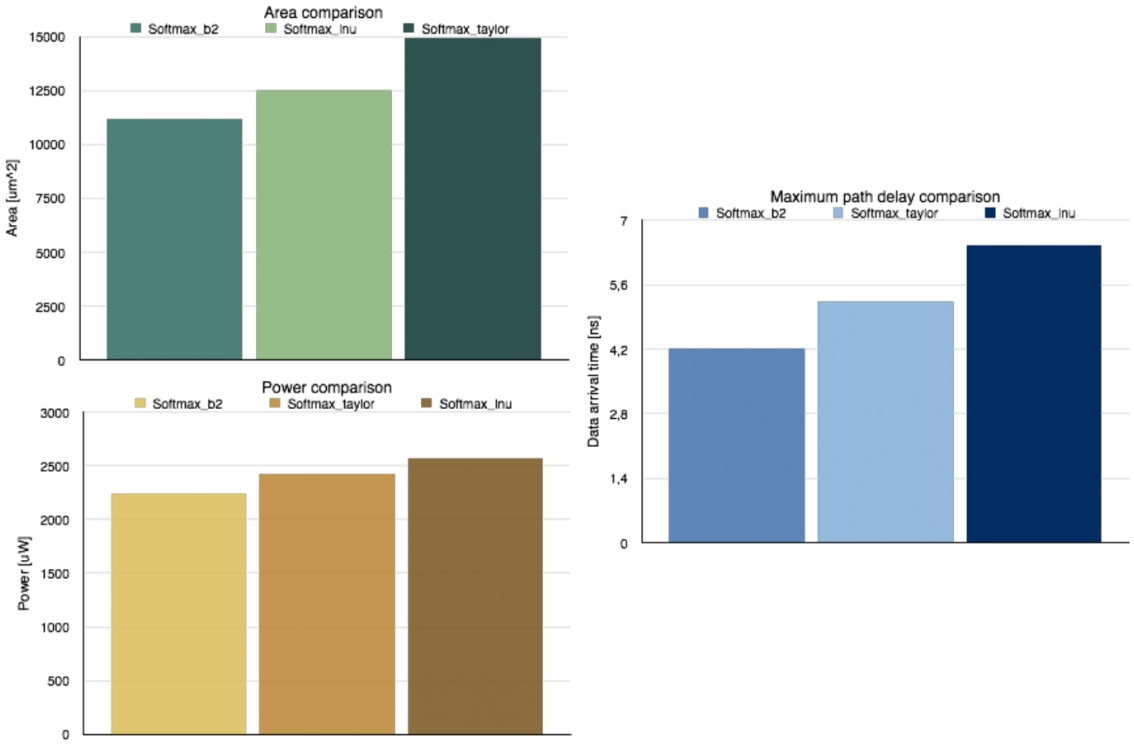


Figure 3.22: Softmax Hardware metrics comparisons

As regards the area and power comparisons, the following observations can be made.

- Softmax_b2 occupies less area (-11%) and dissipates less power (-13%) than Softmax_lnu thanks to the removal of 2 constant multipliers, i.e. $\log_2 e$ and $\ln 2$ multipliers in the exponential and logarithmic units, and reduced data bitwidth in the accumulator.

- Softmax_taylor uses more area (+20%) than Softmax_lnu because of the usage of two look-up tables for the exponent computation, additional arithmetic resources (subtractors, shifter) and wider input registers.

- Softmax_lnu consumes more power (+6%) than Softmax_taylor because of higher internal power (+10%) in both combinational and sequential cells and higher switching power (+3%) in registers.

- Softmax_taylor uses more power (+8%) than Softmax-b2 due to larger switching power (+28%) in combinational units and leakage power (+30%) in both combinational and sequential cells.

By looking at the maximum path delay comparison, it is possible to note that Softmax_b2 and Softmax_taylor have a smaller maximum path delay than Softmax_lnu (-35% and -20%), thanks to the absence of a constant multiplier by $\log_2 e$ and the use of fast table look-ups, respectively. Actually, in Softmax_b2 critical path the $\log_2 e$ multiplier is removed with respect to Softmax_lnu, while in Softmax_taylor longest path the look-up table access is faster than the shift and addition operations used in Softmax_lnu critical path.

3.7.2 Comparison by CapsNet accuracy

The three approximate softmax units are compared in terms of the CapsNet inference accuracy in 4 case studies. The image classification task is performed for two image datasets by using two CapsNet models. In particular, the accuracy results are reported in two bar charts. The first chart refers to the MNIST dataset and the second chart corresponds to the Fashion-MNIST dataset. In each chart, the inference accuracy reached by the CapsNet model using the three approximate softmax functions is reported for two CapsNet models, i.e. ShallowCapsNet and DeepCaps. The reported accuracy results are obtained by performing inference steps with the quantised CapsNet models including the quantised approximate softmax functions.

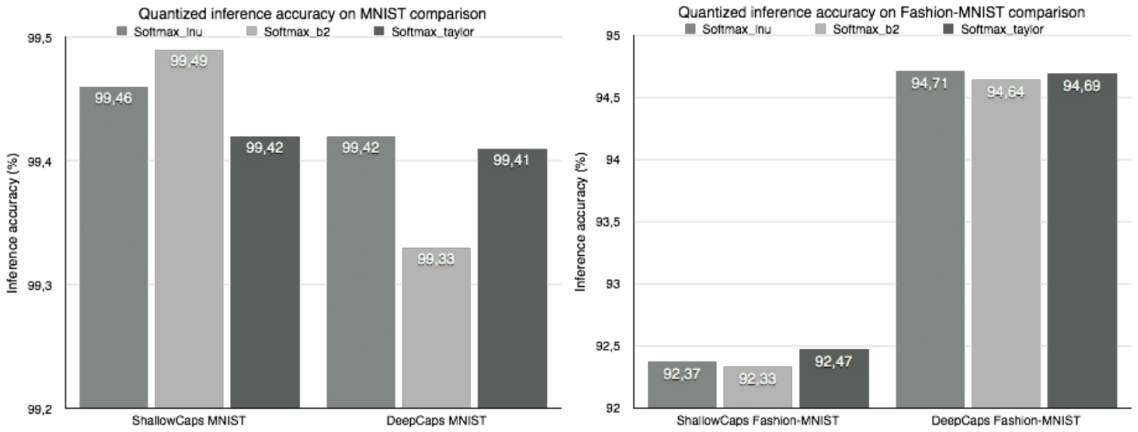


Figure 3.23: Softmax Inference Accuracy comparisons - 1

In figure 3.24, the 4 case studies are analysed in detail by showing the inference accuracy and accuracy loss for the three approximate softmax units. The accuracy loss is evaluated

with respect to the full-precision inference accuracy, that is obtained with full-precision CapsNet model and exact softmax functions.

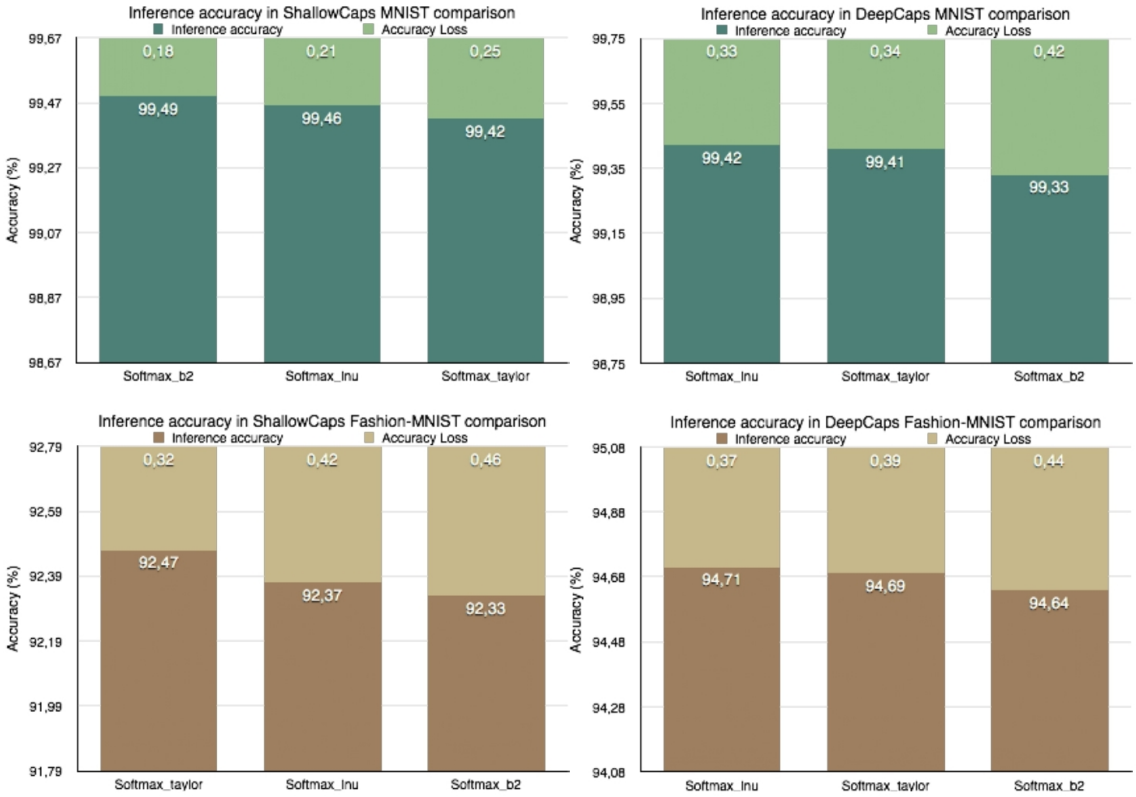


Figure 3.24: Softmax Inference Accuracy comparisons - 2

By analysing the accuracy results, the following observations can be made.

- CapsNet inference accuracy Loss is kept within a 0.5% margin relative to the full-precision accuracy in all the case studies, i.e. the CapsNet misclassifies at most 50 extra images with respect to the full-precision case.

- Accuracy losses across the approximate softmax units are quite similar in each of the case studies, with a maximum accuracy difference in the range 0.07%-0.14%.

- Softmax_lnu performs slightly better than Softmax_taylor (with accuracy difference in the range 0.01%-0.04%) in all cases except for Shallow Fashion-MNIST, where Softmax_taylor outperforms Softmax_lnu by 0.1% accuracy difference.

- Softmax_b2 has the highest accuracy Loss in all cases except for Shallow MNIST, with worst-case accuracy Loss equal to 0.46% in Shallow Fashion-MNIST (46 extra misclassified images).

3.7.3 Exploration of cost-accuracy trade-offs

For each of the 4 case studies, the CapsNet inference accuracy values with the three softmax approximations are reported as a function of two alternative hardware metrics, i.e. area and power of the approximate softmax architecture. The approximate softmax processing units are identified by circular icons in the accuracy vs. area plots in figure 3.25 and by triangular icons in the accuracy vs. power plots in figure 3.26. A specific point in the accuracy-area and accuracy-power plots is associated to each of the approximate softmax designs. By exploiting the aforementioned plots, it is possible to analyse the relationship between the classification accuracy of a CapsNet model using the softmax approximation and the hardware cost of the architecture that implements the approximate softmax function. The following observations are inferred from the 4 accuracy-area and 4 accuracy-power plots.

- Softmax_b2 consumes less area (-11% and -25%) and power (-13% and -8%) than Softmax_lnu and Softmax_taylor, but as a drawback it has the highest accuracy Loss in all cases except for Shallow MNIST, with worst-case accuracy Loss (0.46%) and worst-case accuracy difference with respect to the best unit (0.14%) in Shallow Fashion-MNIST.

- Softmax_lnu and Softmax_taylor have similar accuracy Losses (accuracy difference in range 0.01%-0.04%) in all cases except for Shallow Fashion-MNIST, where Softmax_taylor has a better accuracy by 0.1% at the expense of more area occupied (+20%) and with the benefit of less power consumed (-6%) than Softmax_lnu.

- Overall, a trade-off between area/power cost of the approximate softmax unit and CapsNet inference accuracy is better demonstrated in the case study Shallow Fashion-MNIST, where the accuracy values are farther apart (highest maximum accuracy difference 0.14%) than in other cases: Softmax_taylor has a higher accuracy (+0.14%) than Softmax_b2 at the expense of an additional cost in area (+35%) and power (+8%).

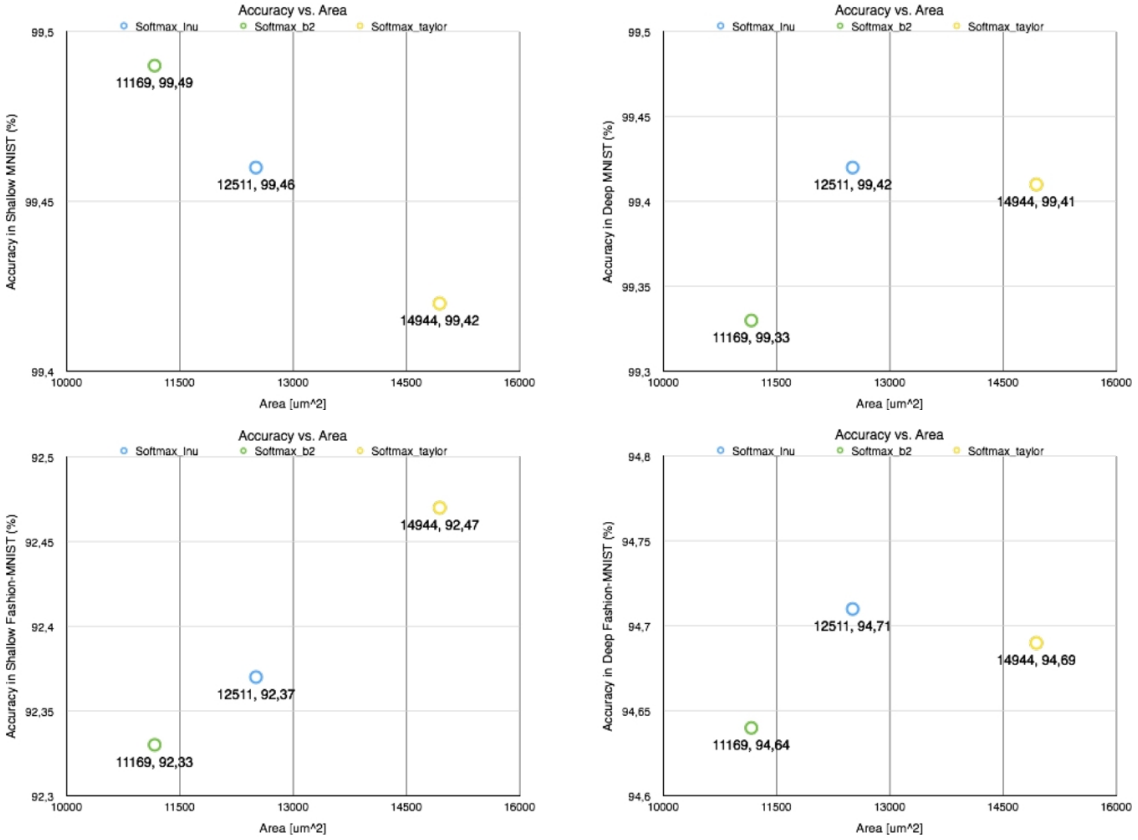


Figure 3.25: Softmax Inference Accuracy vs Area plots

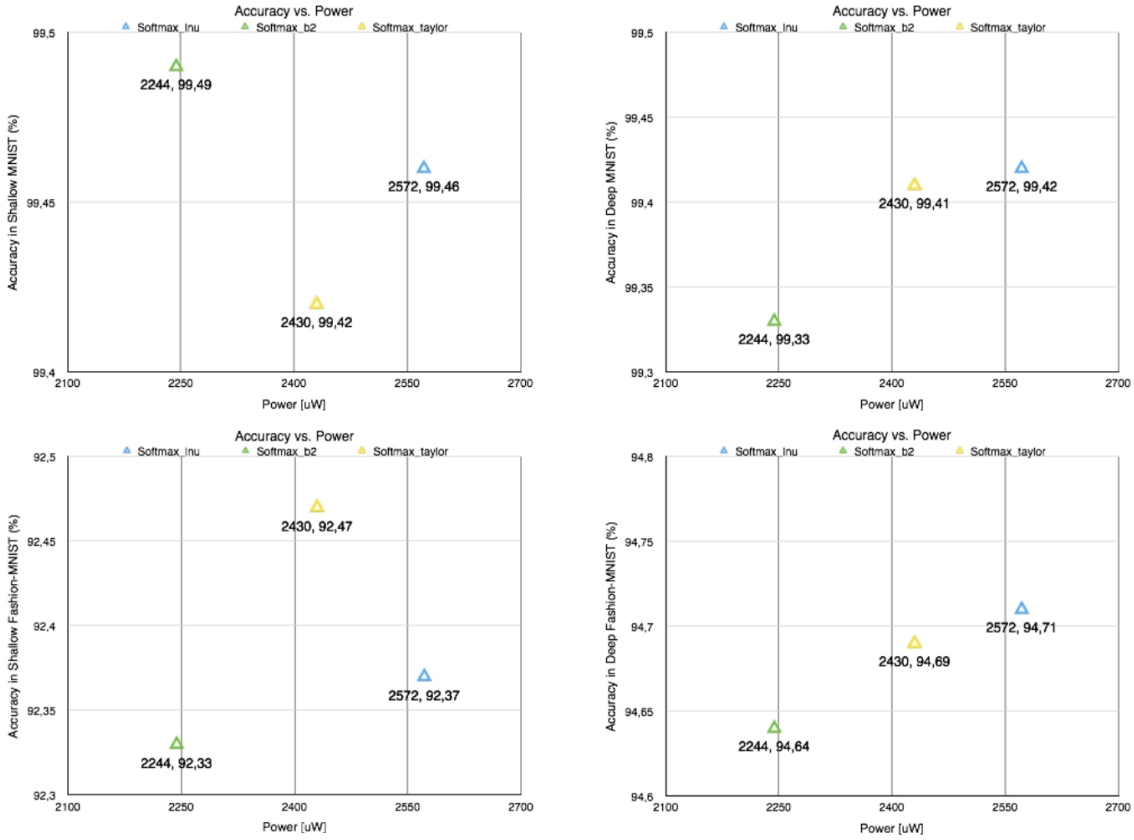


Figure 3.26: Softmax Inference Accuracy vs Power plots

3.8 Comparative Analysis of Squash Approximations

3.8.1 Comparison by Hardware metrics

As regards the squash units, the comparative analysis in terms of hardware metrics is shown in the three bar charts reported in figures 3.27. The first bar chart reports the total area occupied by each squash design; the second bar chart shows the total power consumed by the squash units and the third chart reports the data arrival time in the critical path of each squash design. By analysing the three bar charts, the three approximate squash processing units, Squash_exp, Squash_pow2 and Squash_norm, are compared to find out how the squash designs differ in terms of area, power and critical path delay.

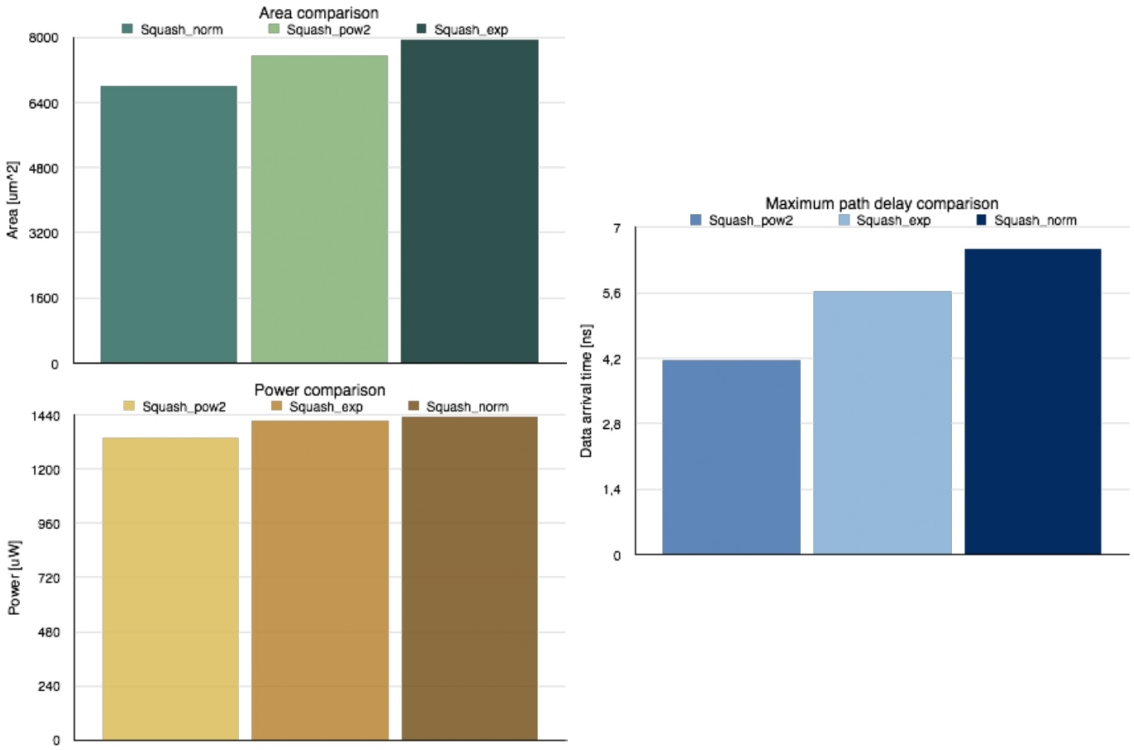


Figure 3.27: Squash Hardware metrics comparisons

As regards the area and power comparisons, the following observations can be made.

- Squash_norm uses less area (-13% and -8%) than both Squash_exp and Squash-pow2, thanks mainly to the usage of fewer and smaller look-up tables. Actually, Squash_norm employs two look-up tables, with 32 and 152 entries, for the computation of the squashing coefficient, but any LUT is used to calculate the norm.

- Squash_norm consumes slightly more power (+1% and +7%) than both Squash_exp and Squash_pow2 because of higher switching power (+6% and +15%) and internal power (+2% and +7%) in combinational cells.

- Squash_pow2 consumes less area (-5%) and power (-5%) than Squash-exp thanks to the removal of a constant multiplier by $\log_2 e$.

By looking at the maximum path delay comparison, it is possible to note that Squash_pow2 has a lower maximum path delay (-25%) than Squash-exp thanks to the absence of a constant multiplier $\log_2 e$, while Squash_norm has a higher maximum path delay (+15%) than Squash_exp because of wider arithmetic units. Actually, in Squash_pow2 the critical path starts at the clocked input port din and ends at the input of the accumulator register, while in Squash_exp the longest combinational path lies between the accumulator register and the output register. In Squash_norm, the critical path still goes from the accumulator register to the output register, but the arithmetic units along the path have wider bitwidths than those in the critical path of Squash_exp.

3.8.2 Comparison by CapsNet accuracy

The three approximate squash units are compared in terms of the CapsNet inference accuracy in 4 case studies. The image classification task is performed for two image datasets by using two CapsNet models. In particular, the accuracy results are reported in two bar charts. The first chart refers to the MNIST dataset and the the second chart corresponds to the Fashion-MNIST dataset. In each chart, the inference accuracy reached by the CapsNet model using the three approximate squash functions is reported for two CapsNet models, i.e. ShallowCapsNet and DeepCaps. The reported accuracy results are obtained by performing inference steps with the quantised CapsNet models including the quantised approximate squash functions.

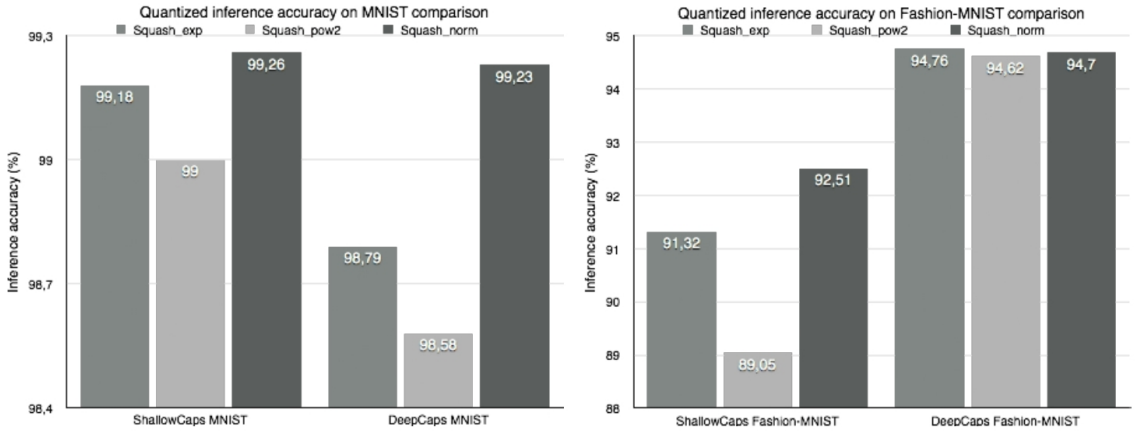


Figure 3.28: Squash Inference Accuracy comparisons - 1

In figure 3.29, the 4 case studies are analysed in detail by showing the inference accuracy and accuracy loss for the three approximate squash units. The accuracy loss is evaluated with respect to the full-precision inference accuracy, that is obtained with full-precision CapsNet model and exact squash functions.

By analysing the accuracy results, the following observations can be made.

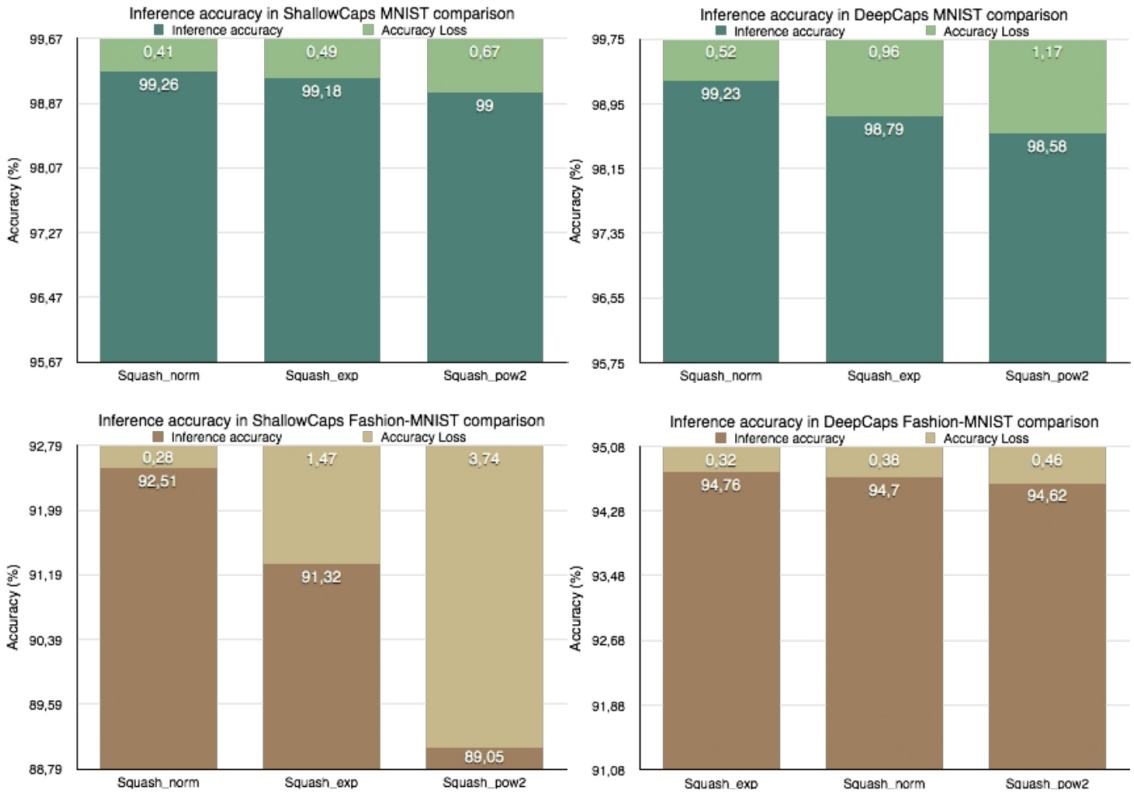


Figure 3.29: Squash Inference Accuracy comparisons - 2

- CapsNet inference accuracy Loss is kept within the 0.5% margin in the Deep Fashion-MNIST case study, while the margin is violated by accuracy Loss with Squash_pow2 in Shallow MNIST (0.67%) and accuracy Loss with Squash_exp and Squash_pow2 in Deep MNIST (0.96% and 1.17%) and Shallow Fashion-MNIST (1.47% and 3.74%).

- Accuracy losses across the approximate squash units are quite similar in Shallow MNIST and Deep Fashion-MNIST (maximum accuracy difference equal to 0.26% and 0.14%), while quite different in Deep MNIST and Shallow Fashion-MNIST (maximum accuracy difference equal to 0.65% and 3.46%).

- Squash_norm performs better than Squash_exp (accuracy difference in range 0.08%-1.19%) in all cases except for Deep Fashion-MNIST, where Squash_exp slightly outperforms Squash_norm by a tiny 0.06% accuracy difference.

- Squash_pow2 has the highest accuracy Loss in all cases, with worst-case accuracy Loss equal to 3.74% in Shallow Fashion-MNIST (374 extra misclassified images with respect to the full-precision case).

3.8.3 Exploration of cost-accuracy trade-offs

For each of the 4 case studies, the CapsNet inference accuracy values with the three squash approximations are reported as a function of two alternative hardware metrics, i.e. area and power of the approximate squash architecture. The approximate squash processing units are identified by circular icons in the accuracy vs. area plots in figure 3.30 and by triangular icons in the accuracy vs. power plots in figure 3.31. A specific point in the accuracy-area and accuracy-power plots is associated to each of the approximate squash designs. By exploiting the aforementioned plots, it is possible to analyse the relationship between the classification accuracy of a CapsNet model using the squash approximation and the hardware cost of the architecture that implements the approximate squash function. The following observations are inferred from the 4 accuracy-area and 4 accuracy-power plots.

- In spite of consuming less area (-13% and -8%) and only slightly more power (+1% and +7%) than the other squash units, Squash_norm has the lowest accuracy Loss in all cases except for Deep Fashion-MNIST, with worst-case accuracy Loss equal to 0.52% in Deep MNIST and accuracy difference with respect to the second-best unit in the range 0.08%-1.19%.
- Squash_pow2 consumes less area (-5%) and power (-5%) than Squash-exp at the expense of higher accuracy Loss in all cases (accuracy difference in the range 0.14%-2.27%), with worst-case accuracy Loss equal to 3.74% in Shallow Fashion-MNIST and accuracy difference with respect to the case best unit in the range 0.14%-3.46%.
- A trade-off between area/power cost of the approximate squash unit and CapsNet inference accuracy is demonstrated with Squash_exp and Squash-pow2 in all the case studies: Squash_pow2 achieves lower cost in area usage and power consumption than Squash_exp, in exchange for higher accuracy Loss.

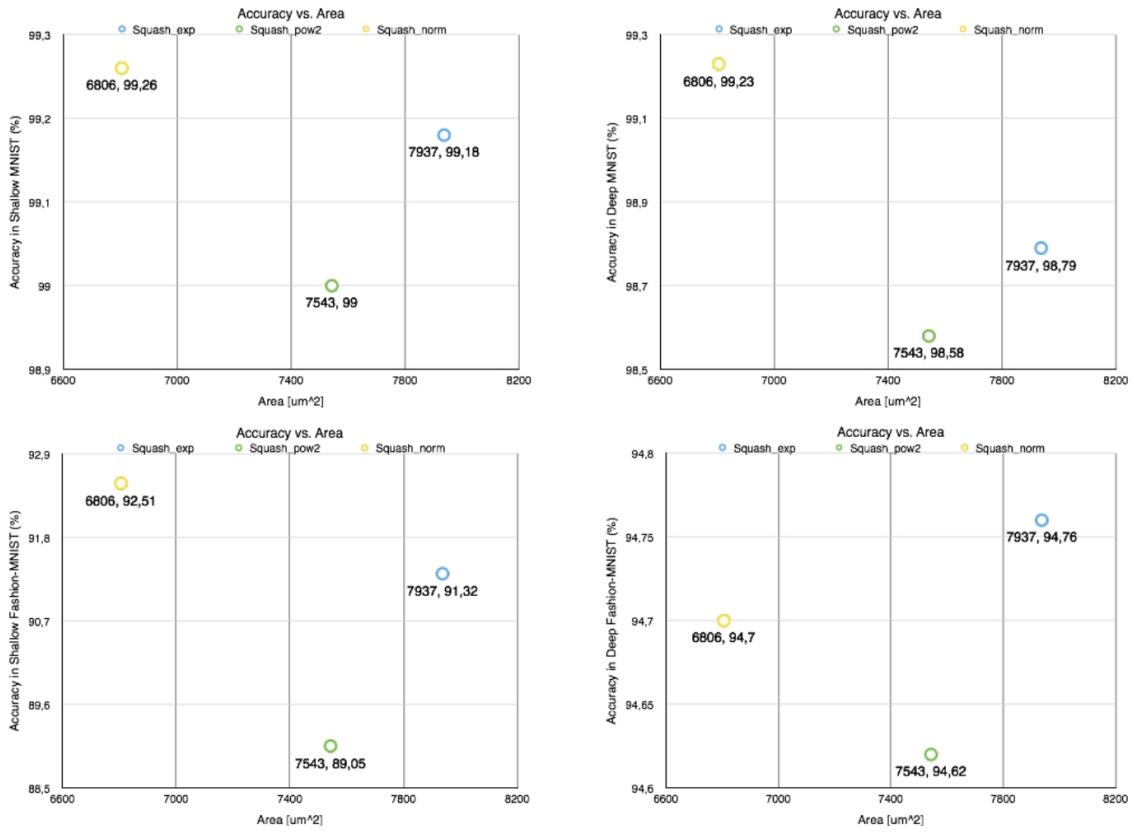


Figure 3.30: Squash Inference Accuracy vs Area plots

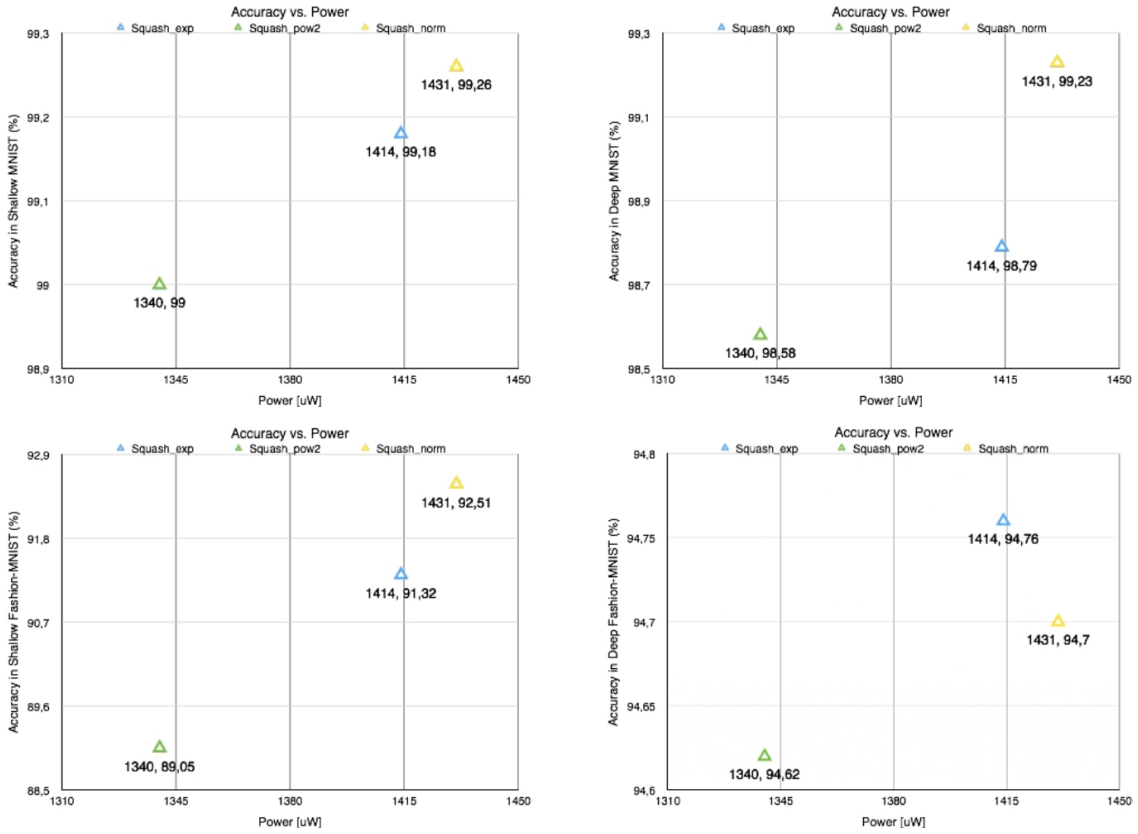


Figure 3.31: Squash Inference Accuracy vs Power plots

Chapter 4

Conclusions and future works

This is one of the first works where multiple approximate softmax and squash techniques are proposed and compared, with the primary goal of exploring trade-offs between the hardware complexity of the approximate architectures and the capsule network inference accuracy loss due to the dynamic routing function approximations.

As regards the *approximate softmax units*, the following final comments are made.

Softmax-b2 turns out to be the best option in terms of hardware implementation metrics, area, power and critical path delay, in spite of being the worst approximate softmax technique in loss of CapsNet inference accuracy, in all cases except for Shallow MNIST, where the three softmax approximations produce similar inference accuracy results.

Overall, Softmax-taylor is considered the best solution in terms of accuracy loss. Actually, the Softmax-taylor technique outperforms the other softmax methods in Shallow Fashion-MNIST, in spite of producing similar accuracy results in Shallow MNIST and DeepCaps case studies. The performance in accuracy is obtained at the expense of the worst area metric and intermediate power and delay data.

Finally, Softmax-lnu is characterised by the worst power and delay metrics and intermediate area usage. It performs similar to Softmax-taylor in all cases except for Shallow Fashion-MNIST, where it incurs in a higher accuracy loss than the Softmax-taylor technique.

Regarding the *approximate squash units*, the following final observations are reported.

Squash-norm is the best solution in terms of CapsNet inference accuracy loss in all cases, except for DeepCaps Fashion-MNIST, where it performs similar to Squash-exp. The good performance in accuracy is obtained at the expense of the worst power and delay metrics, but with the benefit of the best area metric.

Squash-pow2 turns out to be the worst technique in terms of accuracy loss in all the considered cases. On the other side, it is the best option in terms of hardware implementation metrics, power and delay and it has intermediate area usage.

Finally, Squash-exp is characterised by accuracy values similar to Squash-norm in two cases Shallow MNIST and DeepCaps Fashion-MNIST, but significantly worse than Squash-norm in the other two cases. In exchange for reduced accuracy performance with respect to Squash-norm, Squash-exp has the benefit of intermediate power and critical path delay metrics. As a major drawback, it is the worst technique in terms of area usage.

As regards the possible future works, three main innovative paths are outlined at different stages of the design work.

In the description of the approximate softmax and squash architectures, the pipelining optimisation method may be applied by adding pipeline stages on the critical path of the designs, in order to explore a more aggressive clock period constraint and improve the timing performance of the designs. Currently, the approximate softmax and squash processing units have been evaluated for a cycle time equal to 10 ns, i.e. a clock frequency equal to 100 MHz.

At the logic synthesis stage, the approximate softmax and squash designs may be synthesised by using a commercial technology library. The reason behind the use of a commercial technology library for the design synthesis is to obtain a gate-level netlist consisting of standard-cells that correspond to a real technological process and can be potentially fabricated. UMC 65 nm and TSMC 90 nm are two of the possible commercial digital standard-cells libraries that can be exploited for the design synthesis in place of the currently used academic library, i.e. the Nangate 45 nm Open Cell Library. Moreover, by using different technology libraries the design synthesis can be experimented in multiple technology nodes. Focusing on the synthesis process, the quality of the synthesis results with respect to the hardware metrics area, power and delay may be improved by using the `compile_ultra` command in the synthesiser Synopsys Design Compiler. As a matter of fact, the currently used `compile` command does not perform many optimizations. The synthesiser also includes the `compile_ultra` command which does many more optimizations and will likely produce higher quality of synthesis results. The use of the `compile_ultra` command may require a more robust design of the approximate softmax and squash architectures since the beginning, by keeping in mind that multiple optimisation steps will be executed by the synthesiser to generate the gate-level netlist for the design.

After the logic synthesis phase, a place and route software may be used to perform the area, power and timing analysis of the designs similar to what was done by using the synthesiser. During the place-and-route process, the standard-cells used in the post-synthesis netlist will be placed in the cell area on the silicon die and the connections among the cells will be performed using the available metal layers in the routing phase. The post-place-and-route results will be more accurate than the preliminary post-synthesis results and will allow a fairer comparison of the approximate softmax and squash designs in terms of the three hardware metrics. In particular, the area metric may change due to an optimisation in the cells positioning during the layout phase. Secondly, in the timing analysis the interconnection delay will be considered in the data arrival time and slack computation for timing violation check, due to the parasitic resistance and capacitance values of each routed metal wire. Finally, the power analysis after the place and route phase will take into account the parasitic capacitance of the interconnections to estimate the dynamic power contribution.

In addition to the three possible paths described above, future works may include the use of a different quantisation method for quantising both the CapsNet models and the approximate softmax and squash functions data. Moreover, the designed approximate softmax and squash architectures will have to implement in hardware the selected rounding scheme. The motivation behind the use of a different quantisation method is to explore the

sensitivity of both CapsNet inference accuracy and hardware metrics of the approximate softmax and squash designs to the quantisation scheme. Round-to-nearest half-up and stochastic rounding are two possible rounding schemes that can be selected in place of the currently used truncation method.

Appendix A

Critical path of Softmax and Squash architectures

A.1 Softmax architectures

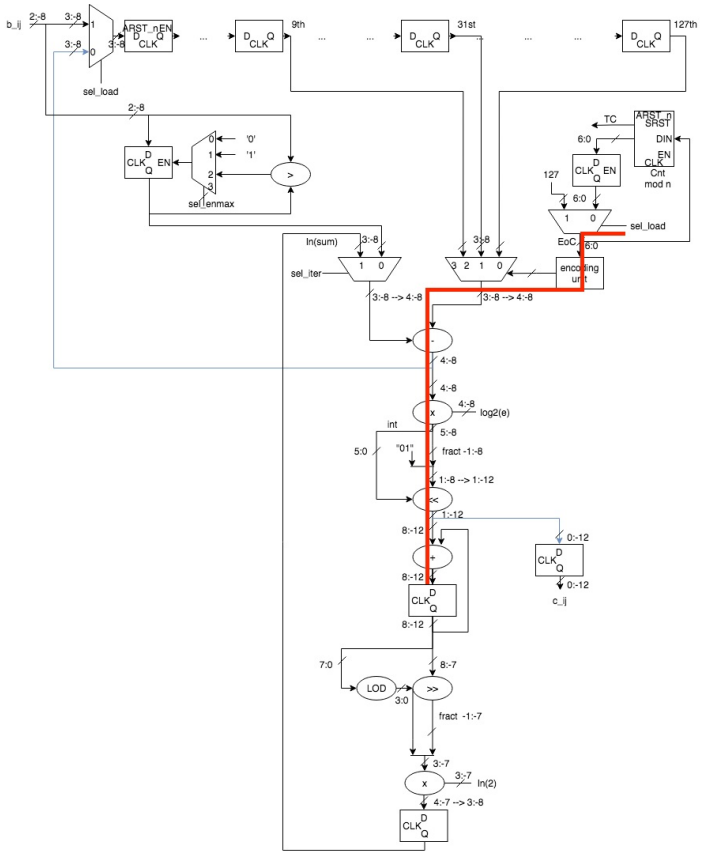


Figure A.1: Softmax-lnu datapath

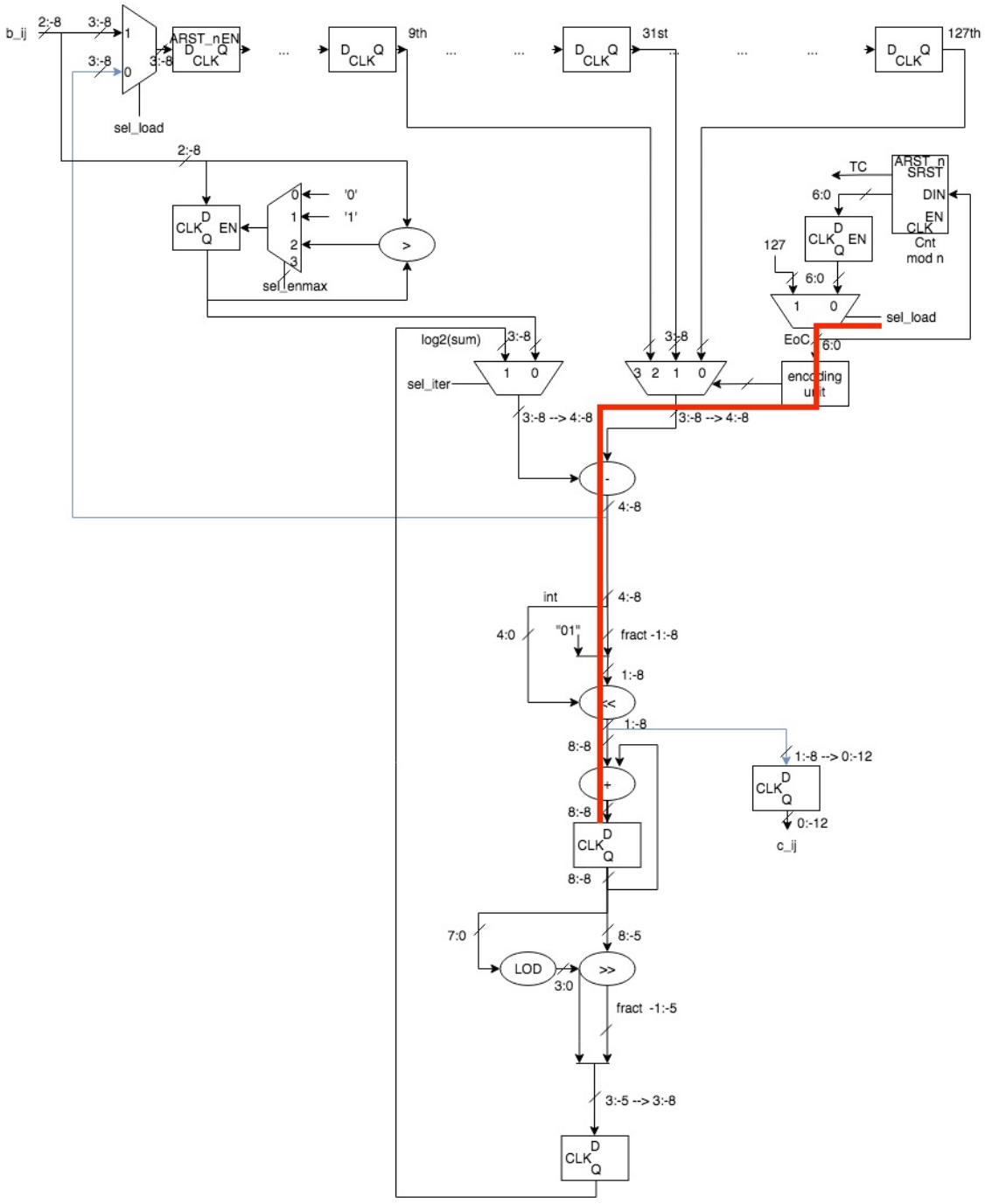


Figure A.2: Softmax-b2 datapath

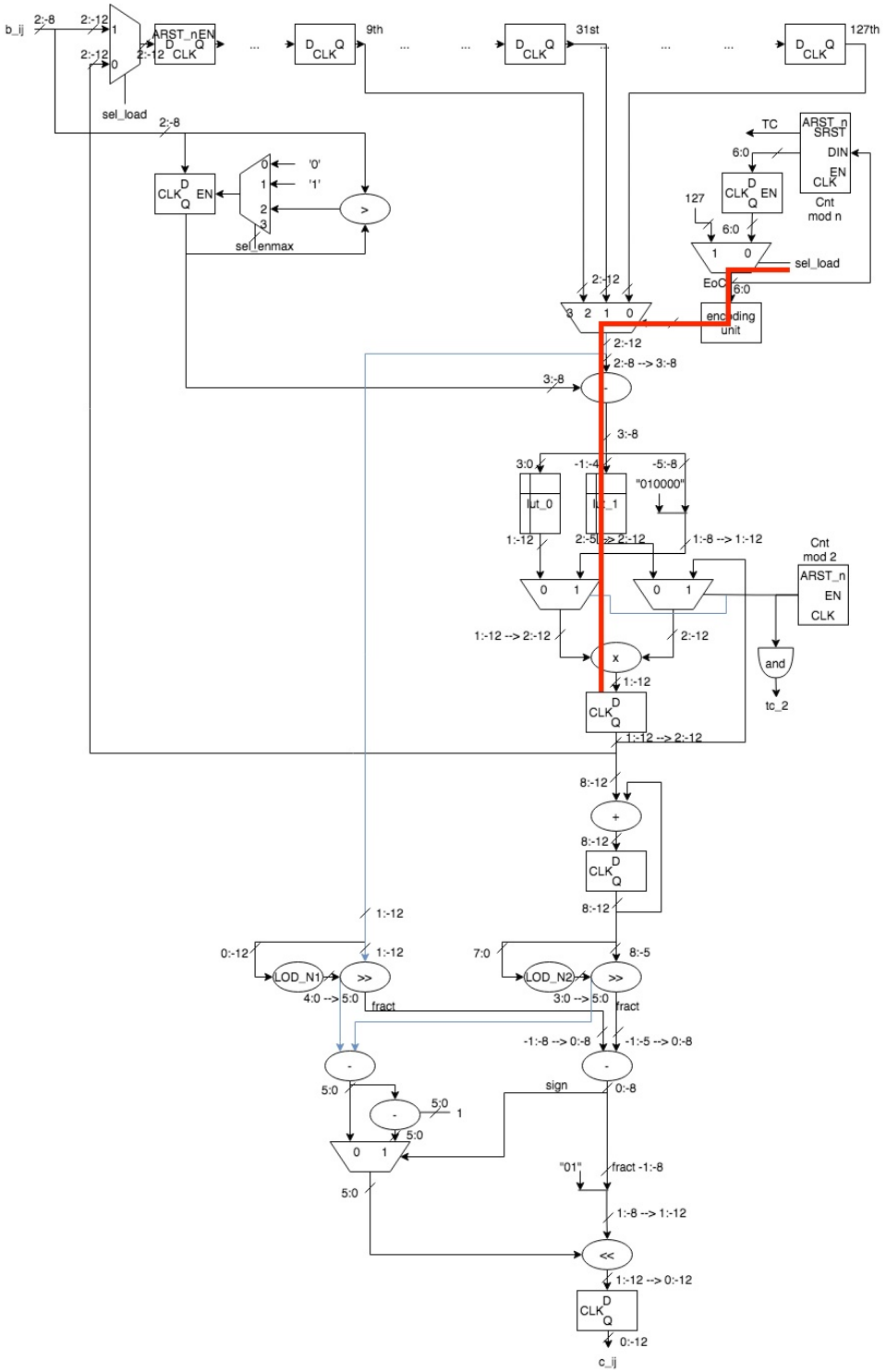


Figure A.3: Softmax-taylor datapath

A.2 Squash architectures

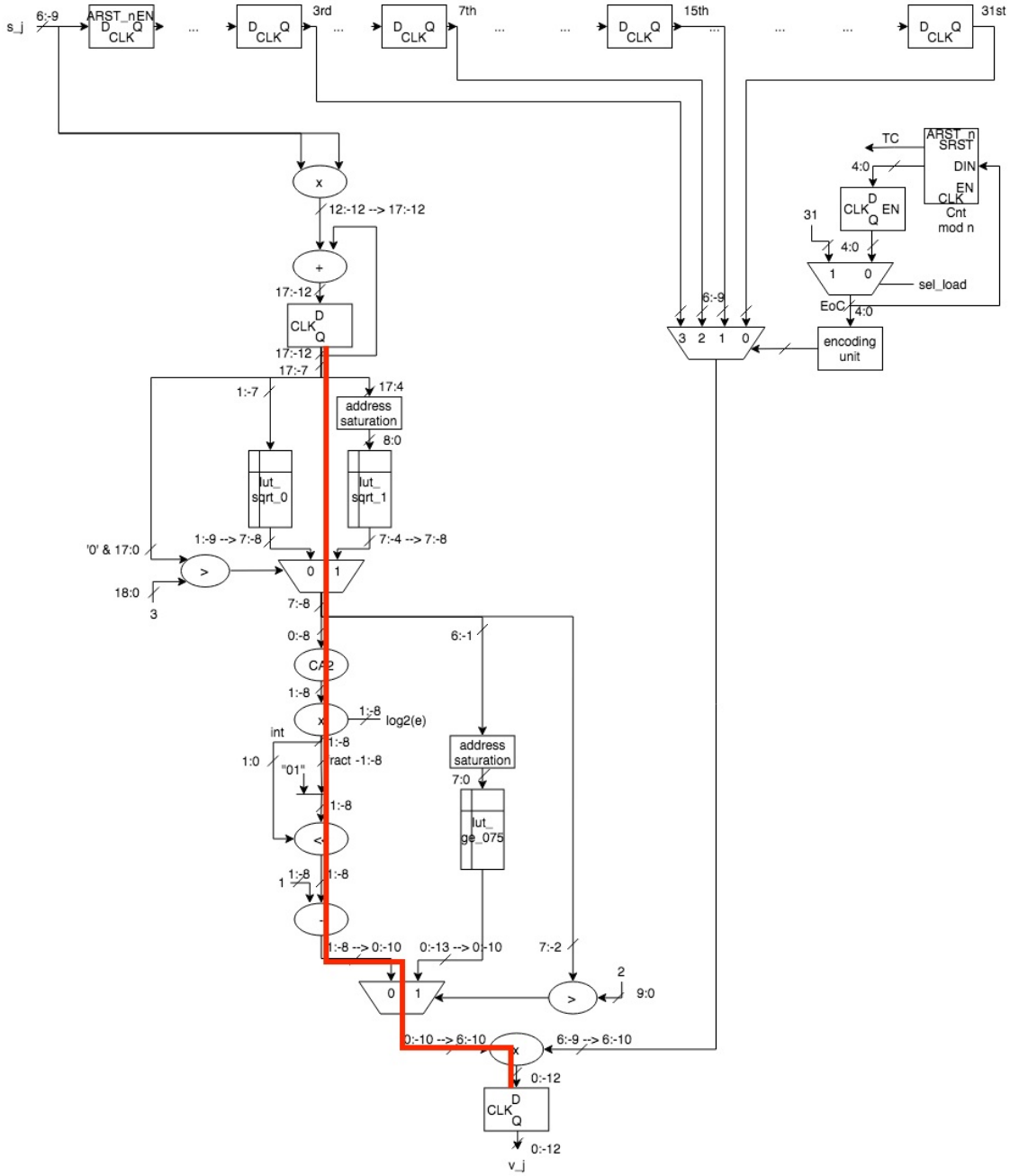


Figure A.4: Squash-exp datapath

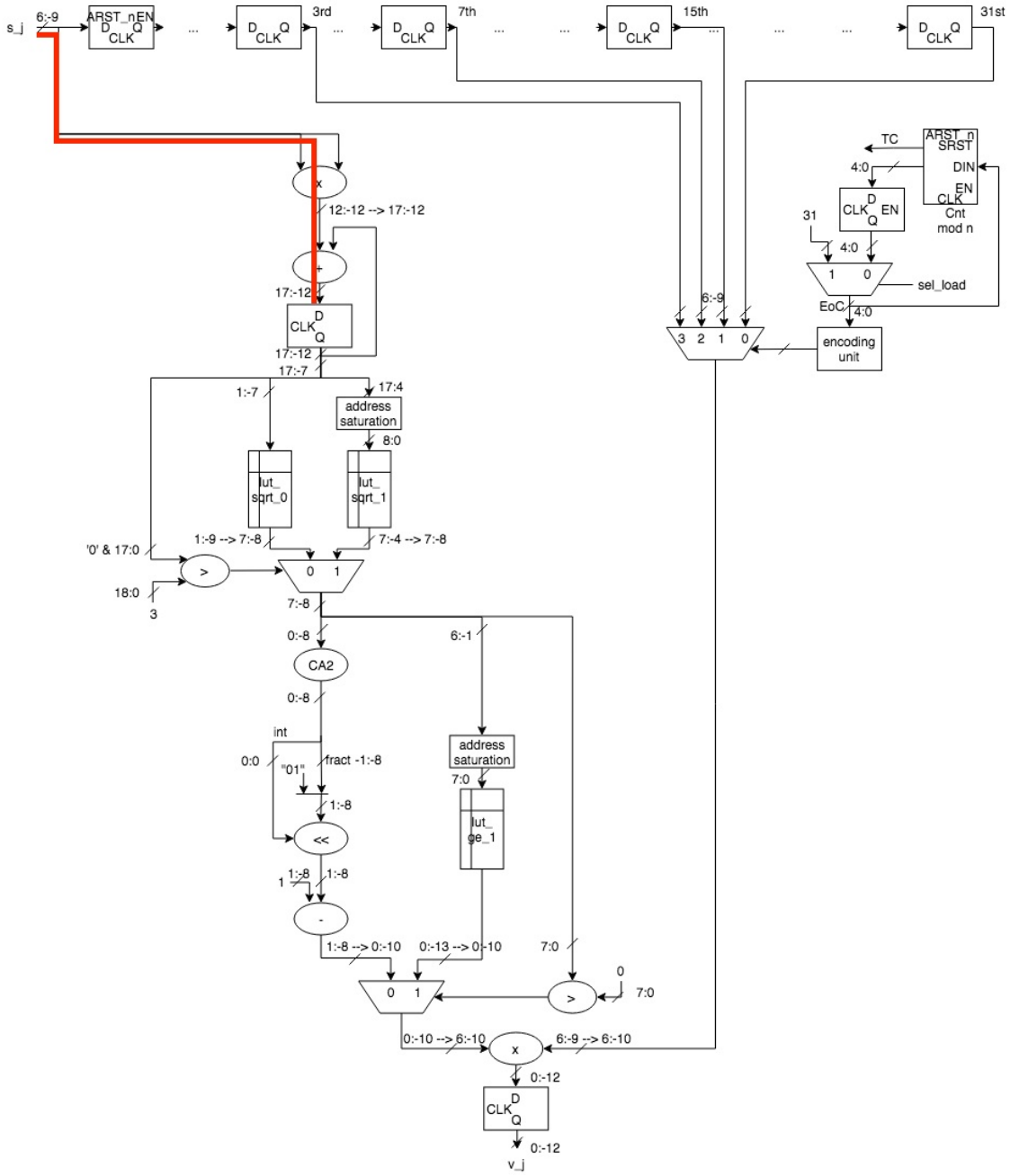


Figure A.5: Squash-pow2 datapath

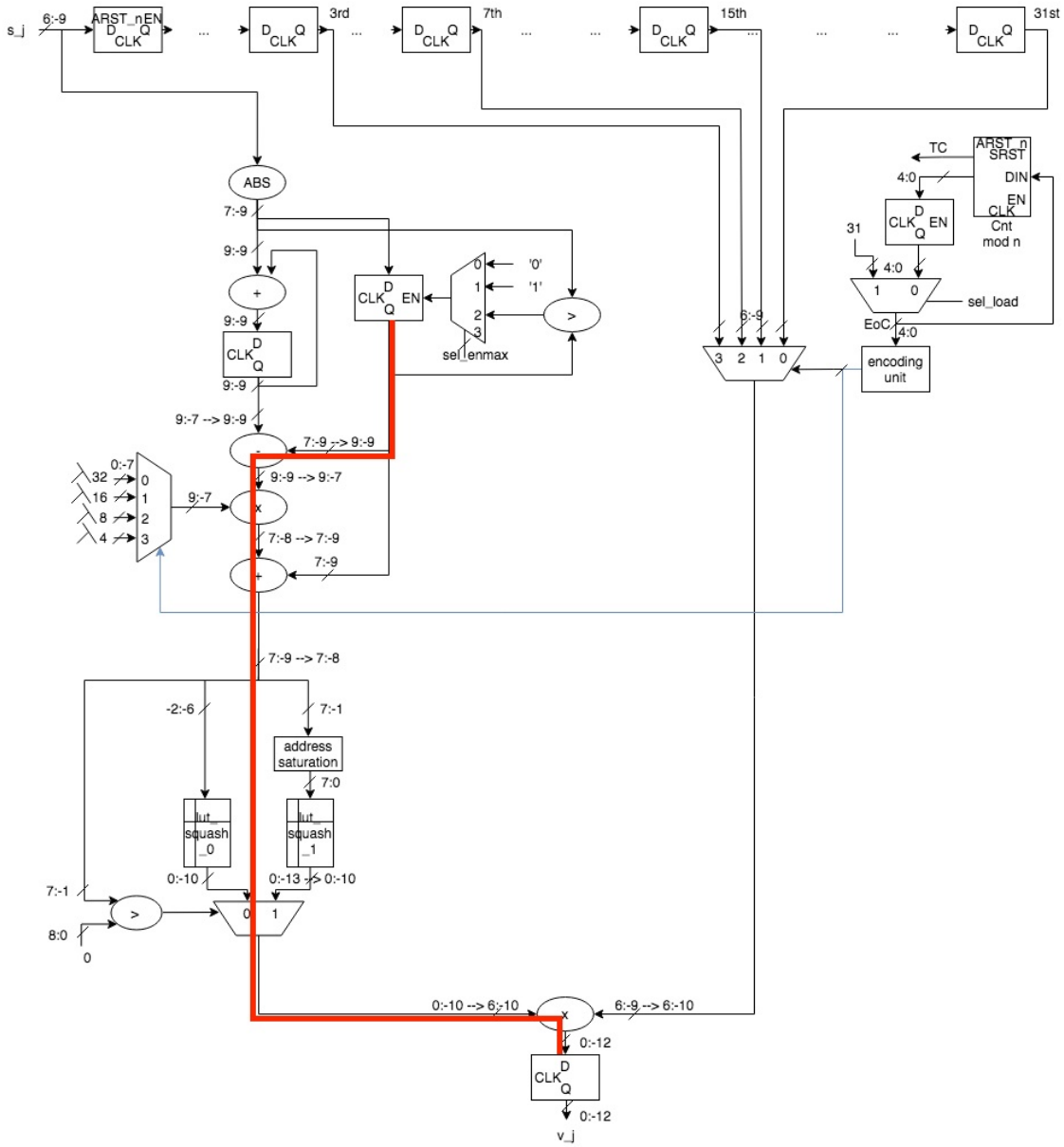


Figure A.6: Squash-norm datapath

Bibliography

[1] Maurizio Capra et al. “An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks”. In: *Future Internet* (2020).

[2] Maurizio Capra et al. “Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead”. In: *IEEE Access* 8 (2020), pp. 225134–225180.

[3] Vivienne Sze et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[4] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE*. 1998.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Communications of the ACM*. Vol. 60. 6. 2017, pp. 84–90.

[6] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. “Dynamic Routing Between Capsules”. In: *31st Conference on Neural Information Processing Systems*. 2017.

[7] Alberto Marchisio et al. “Q-CapsNets: A Specialized Framework for Quantizing Capsule Networks”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6.

[8] Alberto Marchisio et al. “FasTrCaps: An Integrated Framework for Fast yet Accurate Training of Capsule Networks”. In: *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020, pp. 1–8.

[9] Jathushan Rajasegaran et al. “DeepCaps: Going Deeper With Capsule Networks”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 10717–10725.

[10] Vinay K. Chippa et al. “Analysis and characterization of inherent application resilience for approximate computing”. In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–9.

[11] Xue Geng et al. “Hardware-aware Softmax Approximation for Deep Neural Networks”. In: *2018 Asian Conference on Computer Vision (ACCV)*. 2018, pp. 107–122.

[12] Yue Gao, Weiqiang Liu, and Fabrizio Lombardi. “Design and Implementation of an Approximate Softmax Layer for Deep Neural Networks”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5.

[13] Meiqi Wang et al. “A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning”. In: *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2018, pp. 223–226.

[14] M. Emre Celebi, Fatih Celiker, and Hassan A. Kingravi. “On Euclidean norm approximations”. In: *Elsevier* (2010).

[15] Frank Rhodes. “On the metrics of Chaudhuri, Murthy and Chaudhuri”. In: *Elsevier* (1994).

[16] François Chollet. *Deep Learning with Python*. Manning, 2018.

[17] Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O’Reilly, 2017.

[18] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[19] AI Index Steering Committee. *The AI Index 2019 Annual Report*. Human-Centered AI Institute, 2019.

[20] Janardan Misra and Indranil Saha. “Artificial neural networks in hardware: A survey of two decades of progress”. In: *Elsevier* (2010).

[21] Alexander Amini. *Introduction to Deep Learning*. Lecture 6.S191. MIT, 2020.

[22] Alexander Amini. *Deep Computer Vision*. Lecture 6.S191. MIT, 2020.

[23] Srihari Sargur. *Convolutional Networks: Overview*. Lecture CSE676. University at Buffalo, 2020.

[24] Sargur Srihari. *Capsule Networks*. Lecture CSE676. University at Buffalo, 2020.

[25] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[26] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature*. Vol. 521. 2015, pp. 436–444.

[27] P. Murtagh and A. C. Tsoi. “Implementation issues of sigmoid function and its derivative for VLSI digital neural networks”. In: *IEE Proceedings*. Vol. 139. 3. 1992.

[28] Ge Liangwei, Chen Song, and Yoshimura Takeshi. “High-speed, pipelined implementation of squashing functions in neural networks”. In: *9th International Conference on Solid-State and Integrated-Circuit Technology*. 2008, pp. 2204–2207.

[29] Vladimir Havel and Karel Vlcek. “Computation of a nonlinear squashing function in digital neural networks”. In: *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*. 2008, pp. 1–4.

[30] Alberto Marchisio, Muhammad A. Hanif, and Muhammad Shafique. “CapsAcc: An Efficient Hardware Accelerator for CapsuleNets with Data Reuse”. In: *2019 Design, Automation and Test in Europe Conference Exhibition (DATE)*. 2019, pp. 964–967.

[31] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 367–379.

[32] Song Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 243–254.

[33] Yu-Hsin Chen et al. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[34] Geoffrey E. Hinton, Alex Krizhevsky, and Sida D. Wang. “Transforming auto-encoders”. In: *International Conference on Artificial Neural Networks*. 2011, pp. 44–51.

[35] Geoffrey Hinton, Sara Sabour, and Nicholas Frosst. “Matrix Capsules with EM Routing”. In: *ICLR*. 2018.

[36] Bo Yuan. “Efficient hardware architecture of softmax layer in deep neural network”. In: *2016 29th IEEE International System-on-Chip Conference (SOCC)*. 2016, pp. 323–326.

[37] Qiwei Sun et al. “A High Speed SoftMax VLSI Architecture Based on Basic-Split”. In: *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. 2018, pp. 1–3.

[38] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research*. 2014.

[39] Edgar Xi, Selina Bing, and Yang Jin. *Capsule Network Performance on Complex Data*. 2017. arXiv: `1712.03480 [stat.ML]`.

[40] Eric Matthes. *Python Crash Course*. No Starch Press, 2016.

[41] Mark Lutz. *Learning Python*. O’Reilly, 2009.

[42] John N. Mitchell. “Computer Multiplication and Division Using Binary Logarithms”. In: *IRE Transactions on electronic computers*. 1962.

[43] Jie Han and Michael Orshansky. “Approximate computing: An emerging paradigm for energy-efficient design”. In: *2013 18th IEEE European Test Symposium (ETS)*. 2013, pp. 1–6.

[44] Jinghang Liang, Jie Han, and Fabrizio Lombardi. “New Metrics for the Reliability of Approximate and Probabilistic Adders”. In: *IEEE Transactions on Computers* 62.9 (2013), pp. 1760–1771.

[45] Vinay K. Chippa et al. “Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency”. In: *Design Automation Conference*. 2010, pp. 555–560.

[46] Yi Wu et al. “An Efficient Method for Calculating the Error Statistics of Block-Based Approximate Adders”. In: *IEEE Transactions on Computers* 68.1 (2019), pp. 21–38.

[47] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: `1512.03385 [cs.CV]`.

[48] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with PyTorch*. Manning, 2020.

[49] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[50] Alberto Marchisio et al. "NASCaps: A Framework for Neural Architecture Search to Optimize the Accuracy and Hardware Efficiency of Convolutional Capsule Networks". In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9.

[51] H. Amin, K. M. Curtis, and B. R. Hayes-Gill. "Piecewise linear approximation applied to nonlinear function of a neural network". In: *IEE Proceedings*. Vol. 144. 6. 1997.

[52] Peter Nilsson et al. "Hardware implementation of the exponential function using Taylor series". In: *2014 NORCHIP*. 2014, pp. 1–4.

[53] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. "Approximate Computing: A Survey". In: *IEEE Design Test* 33.1 (2016), pp. 8–22.