Ali Jafari, Arnab Neelim Mazumder, Hasib-Al Rashid, Ashwinkumar Ganesan, Chetan Thalisetty, Morteza Hosseini, Haoran Ren, Tim Oates, and Tinoosh Mohsenin

SensorNet: An Educational Deep Neural Network Framework for Low Power Multimodal Data Classification

August 21, 2020

Springer Nature

Contents

Chapter 1
SensorNet: An Educational Neural Network Framework for Low Power Multimodal Data Classification

This chapter presents SensorNet which is a scalable and low power embedded Deep Convolutional Neural Network (DCNN), designed to classify multimodal time series signals. Time series signals generated by different sensor modalities with different sampling rates are first converted to images (2-D signals) and then DCNN is utilized to automatically learn shared features in the images and perform the classification. SensorNet: (1) is scalable as it can process different types of time series data with variety of input channels and sampling rates, (2) does not need to employ separate signal processing techniques for processing the data generated by each sensor modality, (3) does not require expert knowledge for extracting features for each sensor data, (4) makes it easy and fast to adapt to new sensor modalities with a different sampling rate, (5) achieves very high detection accuracy for different case studies, and (6) has a very efficient architecture which makes it suitable to be employed at IoT and wearable devices. A custom low power hardware architecture is also designed for the efficient deployment of SensorNet at embedded real-time systems. SensorNet performance is evaluated using three different case studies including Physical Activity Monitoring, stand-alone Tongue Drive System (sdTDS) and Stress Detection and it achieves an average detection accuracy of 98%, 96.2% and 94% for each case study, respectively.

1.1 Introduction

Time series data is a generalized form of data that is gathered in different kinds of domains from healthcare [23] where one can track a patient's vital signs (heart rate, blood pressure), to fitness and wellness where one can monitor a person's activity [34], to engines in cars and power plants [42] using sensors. Modeling and classifying time series thus has a wide range of applications. All these datasets are represented by a time series which is univariate or multivariate depending on the number of sensor modalities being measured. Multivariate (Multimodal) signals are generated by different sensors usually with different sampling frequencies such as accelerometers, magnetometers, gyroscopes and heart rate monitors.

Traditionally, time series classification problems have been solved with approaches like Dynamic Time Warping (DTW) [2, 36] and k-nearest neighbor (k-NN) [10]. These methods or a combination of them provide a benchmark for current time series classification research [4]. Different signal processing techniques such as feature extraction and classification are employed to process the data generated by each sensor modality which: 1) can lead to a long design time, 2) requires expert knowledge in designing the features, 3) requires new algorithm development and implementation if new sensors are employed, which is tedious, 4) needs extensive signal pre-processing, and 5) is unscalable for different real-time applications.

Deep Convolutional Neural Networks (DCNN) have become extremely popular over the last few years after their success during the Imagenet challenge [24]. Supervised CNNs are used to perform a large number of tasks such as object detection [24], image segmentation [11], and are combined with Recurrent Neural Networks (RNN) to generate captions for images [41] as well as to recognize speech [13]. Inspired by these developments, deep networks are applied to classify time series data, perform event detection and engineer features from the input data [14,22,25,28,32,39,40,42,43,46]. However, these solutions encounter various challenges such as: Low detection accuracy, high latency, large and power-hungry architectures when deployed at Internet of Things (IoT) and wearable devices.

In this chapter, Sensornet shown in Fig. 1.1 is proposed which is a scalable Deep Convolutional Neural Network (DCNN) designed to classify multimodal time series signals in embedded, resource-bound settings that have strict power and area budgets. SensorNet: (1) is scalable as it can process different types of time series data with variety of input channels and sampling rates. (2) does not need to employ a separate signal processing techniques for processing the data generated by each sensor modality. (3) does not require expert knowledge for extracting features for each sensor data. (4) achieves very high detection accuracy for different case studies. (5) has a very efficient architecture which makes it suitable to be deployed at low power and resource-bound embedded devices.

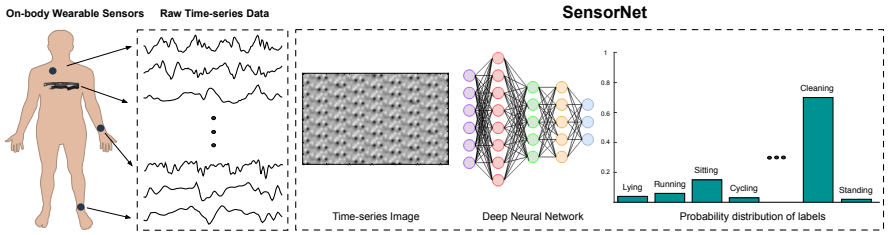

Fig. 1.1: High-level diagram of the proposed Sensornet.

1.2 Background

1.2.1 Overview

In recent years, several multimodal data classification approaches have been proposed which are discussed in this section. These approaches have been proposed for different applications such as Human Activity Recognition, Congestive Heart Failure Detection, Mobile Sensing Data Processing. Furthermore, a number of solutions on deep convolutional neural network for low power embedded settings have been proposed which will be discussed in this section.

1.2.2 Related Prior Work

1.2.2.1 Multimodal Data Classification

[40] uses standard CNN architectures like Tiled Convolutional Neural Networks by first converting a time series into an image before using a CNN to classify it. The authors convert the time series into an image using two types of representations, i.e., Grammian Angular Fields (GAF) and Markov Transition Fields (MTF). The above mentioned architectures either model each variable separately before correlating them or require preprocessing the stream into an image. In [46] authors proposed an architecture which employs a CNN per modality (variable) that processes each variable separately and then correlates them using a fully (dense) connected layer. They tested their network on two different datasets including Physical Activity Monitoring and Congestive Heart Failure Detection and they achieved a detection accuracy of 93% and 94%, respectively. [28] proposed a generic deep framework for activity recognition based on convolutional and LSTM recurrent units and evaluated their framework on the Opportunity and the Skoda datasets. They showed they achieved better accurcay results compared to baseline CNN mdoel. [39] proposed A-Wristocracy, a Deep Learning Neural Network-based framework for recognizing in-home activities of human users with wrist-worn device sensing. They validated 22 daily activities with average test accuracy of 90%. Their network consists of two

hidden layers. Their proposed network is designed specifically for their sensing system. Authors in [25] introduced a system that recognizes concurrent activities from real-world data captured by multiple sensors of different types using 7 layers CNN that extracts spatial features, followed by a long-short term memory network that extracts temporal information in the sensory data. They tested their system with three datasets. Their proposed network has 27M model weights which requires a large memory for saving on an embedded system. In [43] authors proposed DeepSense which is a deep learning framework for time series mobile sensing data Processing. DeepSense integrates convolutional and RNN to exploit and merge local interactions among similar mobile sensors and extract temporal relationships to model signal dynamics. They deployed DeepSense on Nexus5 and Intel Edison and reported latency and power consumption results. [14] proposed Ensembles of deep LSTM learners for Activity Recognition using Wearables. They tested their approach on three different datasets including Opportunity, PAMAP2 and Skoda. [22] proposed an approach for the activity recognition task that, first assembles signal sequences of accelerometers and gyroscopes into a novel activity image. Then 2D Discrete Fourier Transform is applied to the signal image and its magnitude is chosen as their activity image and as input to the DCNN. [32] proposed a Cardiologist-Level Arrhythmia Detectiona using a 34-layer CNN and they exceed the average cardiologist performance in both sensitivity and precision.

1.2.2.2 Deep Neural Networks

In recent years, Deep Neural Networks (DNNs) have become extremely popular because of their outstanding results in the areas such as computer vision [24], voice recognition [17], natural language processing [6], robotics [19] and time series data classification [46]. However, due to intensive computations and large memory requirements, implementing deep neural networks on resource limited and low power embedded platforms is challenging, specially when dealing with a network with many fully connected or convolution layers [29]. Also, performance becomes limited by memory bandwidth to access the training weights saved usually in off-the-chip memory. Several methods have been proposed to address efficient training and inference in deep neural networks [33]: Shallow networks [9], quantizing parameters [12], network binarization [7] and compressing pre-trained deep networks [15]. These methods improve the efficiency of DNNs for implementing on low-power embedded platforms, however, even by employing these methods, an off-the-chip memory is usually required to save the model weights. Moreover, power consumption is still high due to the need of accessing off-the-chip memory constantly. For wearable devices and Internet of Things (IoT) platforms, the power consumption, size and real-time requirements are even more restricted. In 2016, [8] proposed Binarized Neural Networks (BNNs) to address the previously mentioned challenges. BNN has a great compact representation of network weights and activation values compared to a standard DNN by constraining each value to either +1 or -1 (binary). During the forward pass BNNs: 1) Drastically reduce the memory requirements since the

model weights can be stored in one single bit (-1 can be stored as 0 and +1 stored as 1). 2) Eliminate the need of using off-the-chip memory in some popular networks. 3) Replace multiply operations with bit-wise operations (mostly XNOR) which improves power efficiency. Since 2016, different works [7, 33] have shown that BNNs can achieve comparable accuracies to full-precision DNNs for some popular datasets such as MNIST, CIFAR10 and ImageNet. [27,37,45] have proposed BNN hardware accelerators on CPU, GPU, FPGA and ASIC. Although BNNs are very efficient to be employed in low power and resource-limited embedded devices, they have not shown high detection accuracy for multimodal time series data classifications yet.

1.3 SensorNet Architecture

1.3.1 Deep Neural Networks Overview

In most of the deep neural networks, there are large variety of layers including Convolutional, Fully-connected, Pooling, Batch normalization layers. Also, there are activation functions such as Sigmoid, Tanh, and ReLU, which can be considered as separate layers. Among the neural network layers, fully-connected and convolutional layers are often the most highly utilized and contain the majority of the complexity in the form of computation and memory requirements. A brief explanation about the most commonly used layers including their mathematical formulation and complexity requirements in terms of computation and memory is provided in the following section.

1.3.1.1 Convolutional Layers

Convolutional layers are the core building block of a convolutional neural network. The layers consist of learnable filters banks (sets), which have a small receptive field that extend through the full depth of the input. During the forward pass, each filter is convolved across the width and height of the input, computing the dot product between the entries of the filter and the input and producing a feature map of that filter. Feature maps for all filters along the depth dimension of the input data, form the full output of the convolution layer. Figure 1.5 shows convolution operation for a 3x3 image by a 2x2 filter followed by an activation function. A hardware schematic which demonstrates one single operation is also depicted. The convolutional layers use a non-linear activation function which will be discussed later.

For a 1-D input $X_{M,C_{in}}$ of length M and with input channels C_{in}, a 1-D convolutional layer with stride S, filter length F, weight $W_{C_{out},C_{in},F}$, feature maps C_{out}, an output signal $Y_{N,C_{out}}$ with length $N = 1 + \lfloor (M - F)/S \rfloor$ and output channels C_{out}, the output of a single element of a feature channel is computed by:

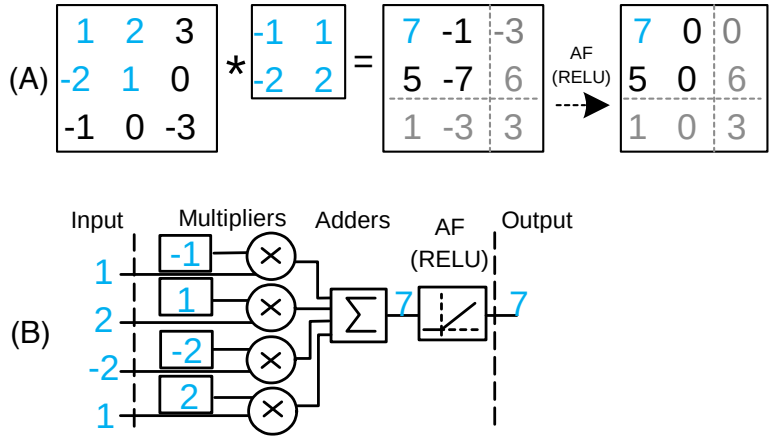

Fig. 1.2: (A) An example of convolving a 3x3 image by a 2x2 filter, (B) A hardware schematic which demonstrates one single convolution operation.

$$Y_{i,j} = \sum_{c=1}^{C_{in}} \left(\sum_{f=1}^{F} \left(X_{f+iS,c} W_{j,c,f} \right) \right) \; for \; i = 0..N-1, j = 1..C_{out} \tag{1.1}$$

The total amount of memory requirements by the layer corresponds to the number of weights for all of the filters which is $C_{out} C_{in} F$. The total computation required for the layer is $2FC_{in}C_{out}N$.

1.3.1.2 Pooling Layers

Pooling layers are usually used immediately after convolutional layers and perform dimensionality reduction. These layers also referred to as downsampling layers. What the pooling layers do is simplify the information in the output from the convolutional layer. There are different pooling layers such as max-pooling and average-pooling. Max-pooling reduces the size of the image and and also helps the network to learn abstract features in the signal by maximizing the value across the pooling window. The pooling layers are usually applied independently to each input channel. Given a 1-D input $X_{M,C_{in}}$ of length M and with C_{in} input channels, a 1-D pooling layer with stride S and pooling length P will produce an output signal $Y_{N,C_{in}}$ with length $N = 1 + \lfloor (M-P)/S \rfloor$. This layer does not requires any memory and significantly less computation compared to convolution layers because it is applied independently to each input channel.

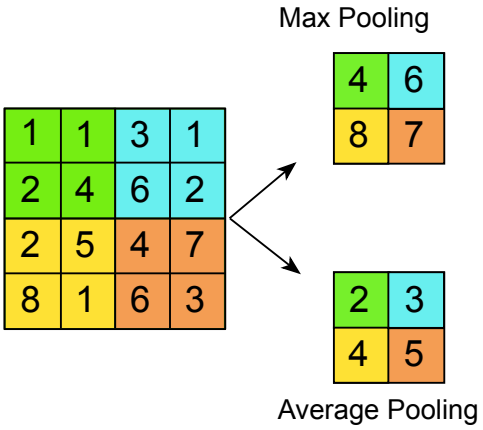

Fig. 1.3: Max-pooling and average-pooling examples with a 2x2 window and stride = 2.

1.3.1.3 Fully-connected Layers

The fully-connected layer is a traditional Multi Layer Perceptron (MLP) that connects every neuron in the previous layer to every neuron on the next layer. Their activations can thus be computed with a matrix multiplication followed by a bias offset.

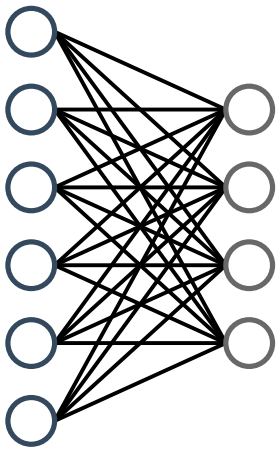

Fig. 1.4: Fully-connected Layer.

The main issue with fully-connected layers is that the layer requires significant amount of memory and computation complexity. Given a 1-D input X_M of length

M, a fully-connected layer with N neurons, weight $W_{N,M}$ and a 1-D output Y_N with length N, the output for a single neuron is computed by:

$$Y_j = \sum_{m=1}^{M} \left(X_m W_{j,m}\right) \quad for\ j = 1..N \tag{1.2}$$

The total amount of memory required for the layer corresponds to the total number of weights, NM, and the total computation is approximately $2MN$. Therefore, the memory and computation contribute equally in terms of complexity.

Usually, after several convolutional and max-pooling layers, the high-level reasoning in the neural network is performed through fully-connected layers. Also, a fully-connected layer with Softmax activation function is used in the output layer for the final classification.

1.3.1.4 Activation Functions

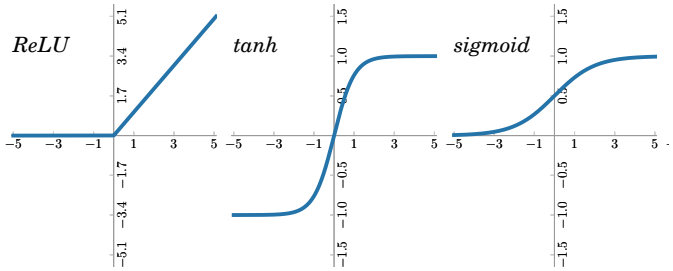

Fig. 1.5: Some common activation functions used in neural networks.

In biologically inspired neural networks, the activation function is usually an abstraction representing the rate of action potential firing of the cell. Activation functions play an important role in the Artificial Neural Network to learn and make sense of Non-linear complex functional mappings between the inputs and response variables and ability to satisfy the profound universal approximation theorem. Figure 1.5 shows some common activation functions used in the Neural Networks including Rectified Linear Unit (ReLU), Hyperbolic tangent (Tanh) and Sigmoid. Convolutional and fully-connected layers use non-linear activation functions. Recently, the most common activation functions are ReLUs which have shown to provide better performance compared to others. A ReLU is represented with the following function:

$$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

In the ReLUs, the activation is linear when the output is positive and hence does not suffer from a vanishing gradient problem. Also, ReLUs are very efficient for hardware implementation because they require few logics and operations to perform.

1.3.2 Signal Preprocessing

Consider a given time series that consists of M modalities/variables with same or different sampling frequency. Prior to training, each variable is independently normalized using the $l2$ norm. To generate an image from the normalized variables, a sliding window of size W and step-size S is passed through all variables, creating a set of images of shape $1 \times W \times M$ (single channel image). The label associated with this image depends on the dataset. The datasets used to test the network in this paper contain a label for every time step. Since a single label is assigned to each image, the label of the current time step is taken as the label of the image (and the label that needs to be predicted subsequently while testing). A given image generated at time-step I_t has the prior states of each variable from $(t - W + 1)...t$. Thus, the network can look back W prior states of each variable and given the current state of each variable, predicts the label.

1.3.3 Neural Network Architecture

Fig. 1.6 shows SensorNet architecture. It consists of 5 convolutional layers, 1 fully connected and a softmax layer that is equivalent in size to the number of class labels (depending on the case study). In the pre-processing stage, SensorNet takes the input time series data and fuses them into images. Then, the images are passed into the convolutional layers and some features which are shared across multiple modalities are generated using a set of local filters. Then, these features are fed into the fully-connected and the softmax layers. SensorNet architecture including number of layers, number of filters and filter shapes for each layer are chosen based on an extensive hyperparameter optimization process which will be discussed in details in Section 1.5.

First, second, third, fourth and fifth convolutional layers contain 32, 16, 16, 8 and 8 filter sets, respectively. The convolution filters have a height of either M or 1, because it's assumed that there are no spatial correlations between the variables. Also, the ordering of variables prior to generating images doesn't affect the ability of the network to perform classification. A filter of height M or 1 remains unaffected by the ordering of the variables. Therefore, the filter size for the first convolutional layer is $M \times 5$ where M is number of input modalities. For other layers, filter shape of 1×5 is chosen.

Max-pooling is applied thrice, once after the second convolutional layer, then after the fourth convolutional layer and the last one after the fifth convolutional layer. A max-norm regularization of 1 is used to constrain the final activation output. The

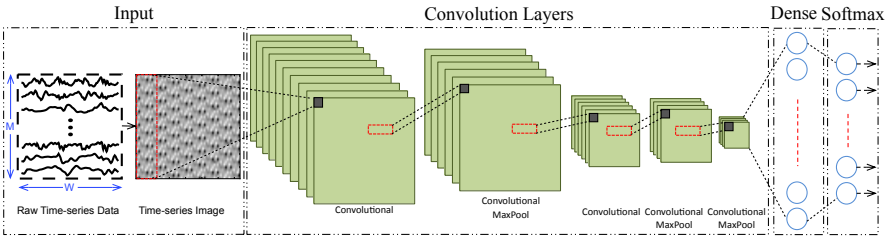

Fig. 1.6: The proposed Sensornet architecture which consists of Convolutional, Fully-connected (Dense) and Softmax layers.

pooling size for all max-pooling layers is 1×2. Once the convolution operations have been performed, the image is flattened into a single vector so that a fully connected layer can be added.

Algorithm 1: Train *SensorNet* to predict labels (actions)

Input: The Network N as defined in Figure 1.6. An input dataset D of size k $(d_1..d_k)$ sampled from various sensors with each point having M attributes.
Output: Predict the class label l_i for a datapoint d_i
 # Consider the training batch size to be b, the learning rate LR and reshape() changes the shape of the tensor.
 # W is the size of the sliding window.
 # epochs is the number of epochs for which the model is trained.
 # For categorical crossentropy refer to Eq.1.3
# x_{train} is a list of images and y_{train} has the expected labels.
for $i \leftarrow W$ ***to*** D **do**
    # After reshape the tensor is of shape (1, W, M)
    x^i_{train} = reshape($d_{i-W+1}..d_i$)
    $y^i_{train} = l_i$
# Train the model.
for $e \leftarrow 1$ ***to*** *epochs* **do**
    for $j \leftarrow 1$ ***to*** $\lfloor \frac{D}{b} \rfloor$ **do**
        batch = $x_{train}[j * b : (j+1) * b]$
        y_{pred} = *forwardpass(N, batch)*
        loss = L(y_{pred}, $y_{train}[j * b : (j+1) * b]$)
        g = backwardpass(loss)
        gradientupdate(g)
return N

Two fully connected layers are employed in SensorNet which the first one has a size of 64 nodes the second one has a size equivalent to the number of class labels with Softmax activation. All the layers of the network have their weights initialized from a normal distribution. A learning rate of 0.0001 is used to train the network. Rectified Linear Unit (ReLU) is used as activation functions for all the layers. The network

is trained using backpropagation and optimized using RMSprop [16]. Categorical crossentropy is used as the loss function. Following is the loss function:

$$L(y_p, y_a) = \frac{-1}{N} * \sum_{i=1}^{N} [y_a^i log y_p^i - (1 - y_a^i) log(1 - y_p^i)] \tag{1.3}$$

where, y_p is the predicted label and y_a is the expected label.

As shown in algorithm 1, there are three main functions defined:

- **Reshape:** This function reshapes a given tensor into another form. We transform a 2D matrix to a 3D tensor with the first axis as 1 representing a single channel.
- **Forwardpass:** It is a single complete processing of the input image to predict the label (for the given dataset).
- **Backwardpass:** It computes the gradient of weights with respect to the loss function (required to perform gradient descent).
- **GradientUpdate:** This function updates the weights using the gradient and learning rate that is defined.

1.4 SensorNet Evaluation using Three Case Studies

Table 1.1: Information for three different case studies including Physical Activity Monitoring, sdTDS and Stress Detection

Application	# of Activity labels	Sensors Position	Sampling Rate (Hz)	# of Subjects	# of Channels
Physical Activity	12	Chest & Arm & Ankle	100 & 9	8	40
sdTDS	12	Headset	50	2	24
Stress Detection	4	Wrist & Finger	8 & 1	20	7

SensorNet is evaluated using three real-world case studies including Physical Activity Monitoring [35], stand-alone dual-mode Tongue Drive System (sdTDS) [20] and Stress Detection [3] and in depth analysis and experimental results are provided.

The information for all the case studies are shown in Table 1.1. As it can be seen from the Table, the sampling rates of the sensors for each case study are different in range of 1 Hz to 100 Hz. Also, the sensors are placed in variety of spots on human body including Chest, Arm, Ankle, Head and hand fingers. The number of channels for each case study refers to the number of input time series signal which are received simultaneously by SensorNet. Physical Activity Monitoring, sdTDS and Stress Detection case studies can be considered to generate large, medium and small size datasets.

For all the case studies, SensorNet is trained using Keras [5] with the TensorFlow as backend on a NVIDIA 1070 GPU with 1664 cores, 1050 MHz clock speed and

8 GB RAM. Models are trained in a fully-supervised way, backpropagating the gradients from the Softmax layer through to the convolutional layers.

1.4.1 Case study 1: Physical Activity Monitoring

1.4.1.1 Dataset

Physical Activity Monitoring dataset (PAMAP2) [35] records 12 physical activities performed by 9 subjects. The physical activities are, for instance: 'standing', 'walking', 'lying' and 'sitting'. Three IMUs (inertial measurement units) and one heart rate monitor are placed on chest, arm and ankle to record the data. The sampling frequency of the IMU sensors is 100 Hz and the heart rate monitor sensor has a sampling frequency of 9 Hz. In total, the dataset includes 52 channels of data but 40 channels are valid according to [35]. Also, out of 9 subjects the data of 8 subjects are used, as subject 9 has a very small number of samples.

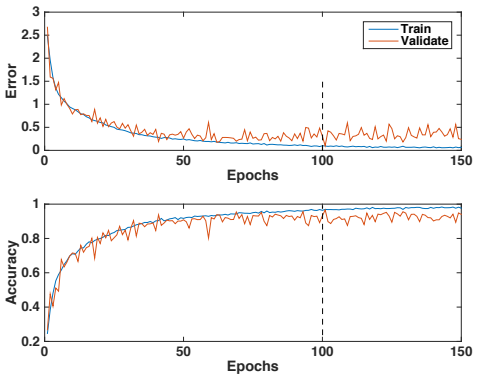

Fig. 1.7: Error and accuracy of the training and validation sets for Physical Activity Monitoring case study over 150 epochs. The vertical dashed line indicates the determined epoch.

1.4.1.2 Experiment Setup and Results

As we mentioned in Section 1.3.1, SensorNet utilizes 5 convolutional layers, followed by 2 fully-connected layers. First convolutional layer has 32 filter sets and each filter size is 40×5. Other convolutional layers have 16, 16, 8 and 8 filter sets with a size of 1×5. For this experiment, 80%, 10% and 10% of the entire data for each subject is chosen randomly as the training, validation and testing set, respectively. To determine the number of required epochs for the training, we train SensorNet

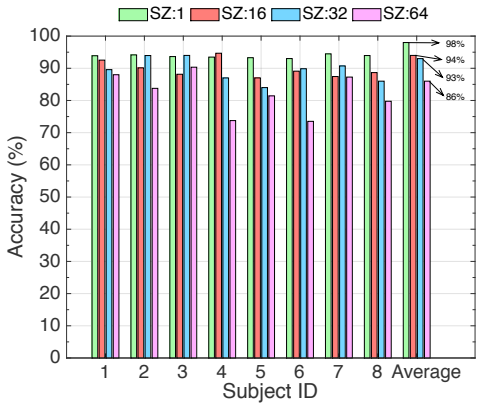

Fig. 1.8: Comparison of SensorNet classification accuracy for Physical Activity Monitoring case study. The results are for different subjects with a sliding window of size 64 samples and step-size (SZ) of 1-16-32-64.

for 150 epochs and plot validation and training loss and accuracy results. As is shown in Fig. 1.7, after 100 epochs the validation loss and accuracy are stable and satisfactory. Therefore, for all the experiments for this dataset we train SensorNet with 100 epochs.

After training SensorNet, we evaluate the trained model to determine the detection accuracy. Fig. 1.8 shows the classification accuracy of SensorNet for the Physical Activity Monitoring case study for different subjects with a sliding window of size 64 samples and step-size of 1-16-32-64. As can be seen from the figure, all subjects with step-size 1 achieve a high detection accuracy. However, as the step-size increases from 1 to 64 the detection accuracy decreases. The average accuracy of all subjects with step-sizes of 1, 16, 32 and 64 are 98%, 94%, 93% and 86%, respectively.

1.4.2 Case study 2: Stand-alone Dual-mode Tongue Drive System (sdTDS)

1.4.2.1 sdTDS Overview and Experimental Setup

In [20], we proposed and developed stand-alone Tongue Drive System (sTDS) which is a wireless wearable headset and individuals with severe disabilities can use it to potentially control their environment such as computer, smartphone and wheelchair using their voluntary tongue movements [21]. In this work, in order to expand the functionality of sTDS, we propose a stand-alone dual-mode Tongue Drive System (sdTDS) by adding head movements detection to the previous version. Fig. 1.9 shows sdTDS prototype which includes a local processor, four magnetic and acceleration sensors, a BLE transceiver, a battery and a magnetic tracer which is glued to the user's tongue. Two magnetic and acceleration sensors are placed on each side of the

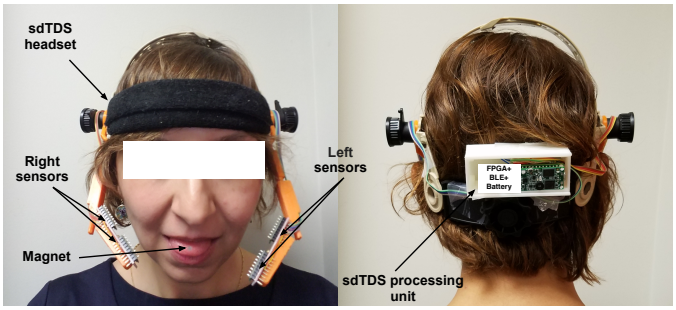

Fig. 1.9: sdTDS prototype placed on a headset which includes a FPGA, four acceleration and magnetic sensors, a Bluetooth low energy transceiver, a battery and a magnetic tracer which is glued to the user's tongue.

headset and the processor is placed in a box at backside of the headset. The box is also used for placing a battery and it's weight is around 0.14 lb. The box is designed using 3D printing technology. In order to generate user-defined commands, user should move his/her tongue to 7 specific teeth or move his/her head to 5 different directions. The raw data generated by 4 magnetometers and accelerometers are transferred into a FPGA processor where the entire signal processing including feature extraction and classification is performed by SensorNet and 12 different user-defined commands can be generated.

1.4.2.2 Experiment Results

Several different data sets are captured using sdTDS for training and testing purpose. sdTDS generates 24 channels of time series data that corresponds to tongue and head movements. As it was mentioned in Section 1.3.1, SensorNet utilizes 5 convolutional layers, followed by 2 fully-connected layers. For the sdTDS, first convolutional layer has 32 filter banks and each filter size is 24×5. Other convolutional layers have 16, 16, 8 and 8 filter banks with a size of 1×5. For this experiment, 80%, 10% and 10% of the entire data for each trivial is chosen randomly as the training, validation and testing sets, respectively. We train SensorNet for 100 epochs.

After training SensorNet using sdTDS dataset, we evaluate the trained model to determine the detection accuracy. Based on previous experiments, we train and test the sdTDS with a sliding window of size 64 samples and step-size of 1, as the step-size of 1 gives better detection accuracy, consistently. For sdTDS case study, SensorNet detection accuracy for tongue and head movements detection is approximately 96.2%.

1.4.3 Case study 3: Stress Detection

1.4.3.1 Dataset

This database contains non-EEG physiological signals used to infer the neurological status including physical stress, cognitive stress, emotional stress and relaxation of 20 subjects. The dataset was collected using non-invasive wrist worn biosensors. A wrist worn Affectiva collects electrodermal activity (EDA), temperature and acceleration (3D); and a Nonin 3150 wireless wristOx2 collects heart rate (HR) and arterial oxygen level (SpO2) data [3]. Therefore, in total the dataset includes 7 channels of data. The sampling frequency of wrist worn Affectiva is 8 Hz and wristOx2 has a sampling frequency of 1 Hz.

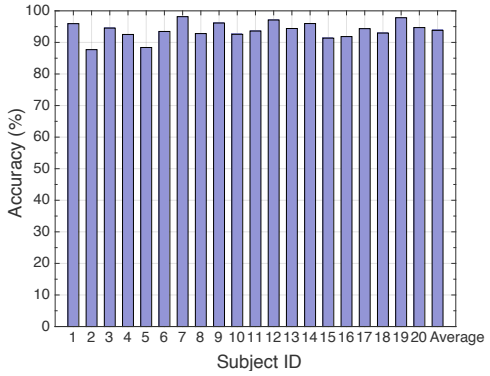

Fig. 1.10: Comparison of SensorNet classification accuracy for Stress Detection case study. The results are for different subjects with a sliding window of size 64 samples and step-size (SZ) of 1-16-32-64.

1.4.3.2 Experiment Setup and Results

As it was discussed in Section 1.3.1, SensorNet utilizes 5 convolutional layers, followed by 2 fully-connected layers. First convolutional layer has 32 filter sets and each filter size is 7×5. Other convolutional layers have 16, 16, 8 and 8 filter sets with a size of 1×5. Similar to Physical Activity Monitoring case study, for this experiment 80%, 10% and 10% of the entire data for each subject is chosen randomly as the training, validation and testing set, respectively. To determine the number of required epochs for the training, we train SensorNet for 150 epochs and plot validation and training loss and accuracy results. After 100 epochs the validation loss and accuracy are stable and satisfactory. Therefore, for all the experiments for this dataset we train SensorNet with 100 epochs. Fig. 1.10 shows the classification accuracy of SensorNet for Stress Detection case study for 20 different subjects. As is shown in the figure,

most of the subjects have a high detection accuracy more than 90%. The average accuracy of all 20 subjects is approximately 94%.

1.5 SensorNet Optimization and Complexity Reduction

In traditional convolution layer if the input is of size $D_f \times D_f \times M$ and N is the number of filters applied to this input of size $D_k \times D_k \times M$ then output of this layer without zero padding applied is of size $D_p \times D_p \times M$. If the stride for the convolution is S then D_p is determined by the following equation:

$$D_p = \frac{D_f - D_k}{S} + 1 \tag{1.4}$$

In this layer the filter convolves over the input by performing element wise multiplication and summing all the values. A very important note is that depth of the filter is always same as depth of the input given to this layer. The computational cost for traditional convolution layer is $M \times D_k^2 \times D_p^2 \times N$ [18].

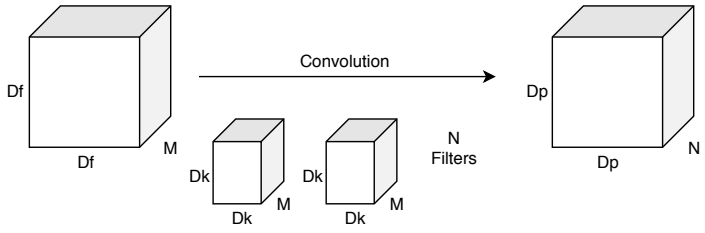

Fig. 1.11: Traditional convolution layer with input shape of $D_f \times D_f \times M$ and output shape of $D_p \times D_p \times N$.

The specified equations in Table 1.2 provide the number of parameters and computations for one forward pass in terms of traditional convolution and fully-connected layers. For convolutional layers the number of parameters and computations is dependent upon the stride which is determined by equation 1.4. The input to the fully-connected layer is flattened hence, it becomes a one-dimensional vector where the computations depend on the number of filters or neurons.

In this section, the impact of changing the following parameters or configurations on SensorNet performance is specifically explored: *1) Number of convolutional layers, 2) Number of filters, 3) Filter shapes, 4) Input zero-padding, and 5) Activation functions.*

Table 1.2: Parameter and Operations Calculation for Convolution and Fully-Connected Layers. Here, D_f, D_k, D_p, M and N represent input height/width, filter height/width, output height/width, number of input channels and number of filters respectively.

Layers	Convolution	Fully-Connected
Input Height	D_f	D_f
Input Width	D_f	*1*
# Input Channels	M	*1*
# Filters	N	N
Filter Height	D_k	
Filter Width	D_k	
Output Height	D_p	N
Output Width	D_p	*1*
# Parameters	$(M \times D_k^2 \times N) + N$	$(D_f \times N) + N$
# Computations	$M \times D_k^2 \times D_p^2 \times N$	$D_f \times N$

1.5.1 Number of Convolutional Layers

In this experiment, six SensorNet configurations with an increasing number of convolutional layers, for the three different case studies was compared. These 6 configurations are depicted in Fig. 1.12. The comparison has been made in terms of detection accuracy, number of convolutional operations, number of parameters (model weights) and memory requirements. Fig. 1.13-A, 1.13-B, 1.13-C and 1.13-D depict the impact of increasing the number of convolutional layers on the number of model parameters and memory requirements. As is shown in the figures, by increasing the number of convolutional layers, the number of model parameters and memory requirements decrease which is desired. The reason that three max-pooling layers after the convolutional layers is used comes from the intuition that by adding more convolutional layers the size of the time series images shrink and the fully-connected layer needs to process less number of data and thus requires less memory. Fig. 1.13-E shows the impact of increasing the number of convolutional layers on detection accuracy. As it can be seen from the figure, if the neural network is too shallow high-level features can not be learned, therefore the detection accuracy is low. However, the results show that, by increasing the number of convolutional layers detection accuracy increases but up to 5 convolutional layers. After that, for Activity Monitoring and sdTDS case studies the accuracy improve slightly but for the Stress Detection reduces because the useful features may be filtered out during the convolutional and max-pooling processes. Also, by adding additional convolutional layer the number of operations to finish a classification task increases slightly which is shown in Fig. 1.13-F. This analysis results show that SensorNet with 5 convolutional layers is the best candidate with regards to detection accuracy, number of convolutional operations and memory requirements.

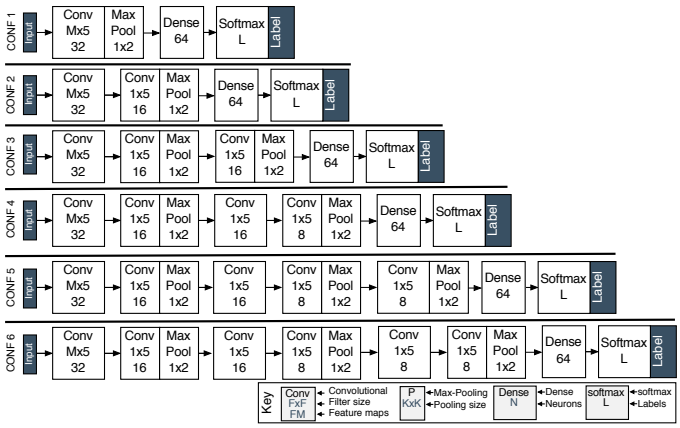

Fig. 1.12: Comparison of six different SensorNet configurations. M and L are the number of input data channels and labels for different case studies, respectively.

1.5.2 Number of Filters

The number of filters (weights) are another important hyperparameter for implementing SensorNet on low power and resource-limited embedded devices because the number of model weights affect the memory requirements and also the number of required computations to finish a classification task. The number of required computations has a direct effect on energy consumption. In this experiment, the number of convolutional layers was fixed (5 layers) and the number of filters for each layer is increased as shown in Fig. 1.14. The goal of this experiment is to find the impact of the number of filters on the detection accuracy, number of convolutional operations, number of parameters (model weights) and memory requirements for Physical Activity Monitoring, sdTDS and Stress Detection case studies. Therefore, SensorNet is trained and tested using four different configurations with different number of filter sizes. Fig. 1.15 shows a comparison of number of required parameters (model weights) for different trained models. Model weights includes the parameters for convolution, fully-connected and softmax layers. As is shown in the figure, as the number of filters for each layer is increased, the detection accuracy improves. However, the number of operations, memory requirements and the number of model parameters increase which is not desire for hardware implementation in a resource limited embedded platform.

Based on the results, SensorNet with different filter sets achieves similar detection accuracies, but Set 4 needs lower number of parameters and requires smaller memory compared to other filter sets and therefore is chosen to be implemented on hardware.

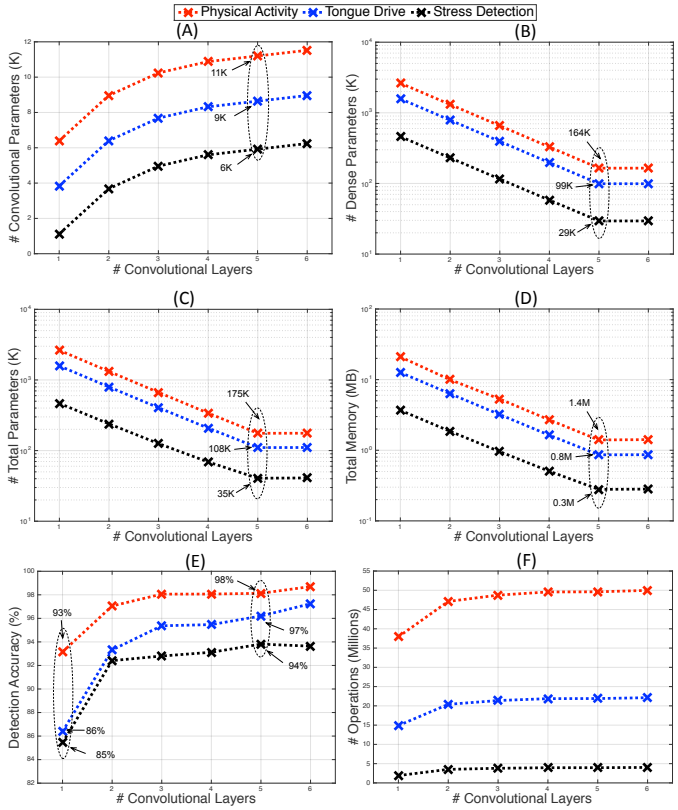

Fig. 1.13: Impact of increasing number of convolutional layers on memory requirements, detection accuracy and number of operations of SensorNet, for three different applications.

1.5.3 Filters Shapes

Another important parameter for implementing SensorNet on low power embedded platforms is the filter shape. As it was explained in Section 1.3.1 the idea is to generate some shared features across different input modalities. Therefore, the filters with size $M \times 5$, where M is the number of input modalities is chosen, for the first convolutional layer. For other convolutional layers the filters are 1×5. By employing this size of filter without zero padding the outputs of the first layer are 1-D vectors and the following layers also will be 1-D vectors. This improves the memory requirements on an embedded platform drastically; because the feature maps are 1-D signal which compared to an image is much smaller. Also, smaller number of model weights are needed as the dense layer takes 1-D vectors rather than images. Furthermore, it reduces the the number of operations which this affects directly

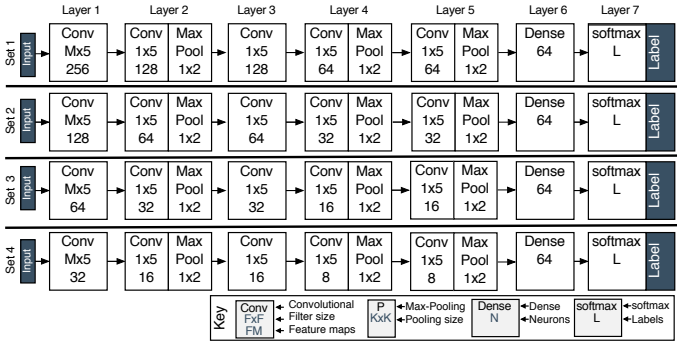

Fig. 1.14: SensorNet configurations with four different filter sets. Number of filters for convolutional layers are doubled for each filter set. M and L are the number of data channels and labels for different case studies, respectively.

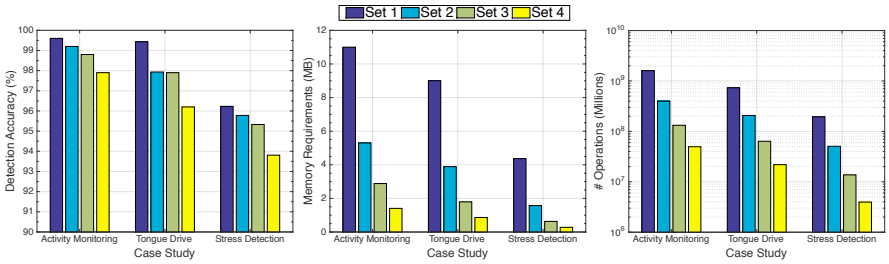

Fig. 1.15: Comparison of four different SensorNet configurations in terms of detection accuracy, memory requirements and number of operations. By adding additional model weights to each layer, the computation and memory grow dramatically with only modest improvement in detection accuracy.

on energy consumption of the framework when is implemented on an embedded platform.

In this experiment the filters shapes are changed for the first convolutional layer to $M \times 5$, 5×5, 3×3 and 1×5 for Physical Activity Monitoring, sdTDS and Stress Detection case studies. Based on the results, filter size of $M \times 5$ gives better detection accuracy compared to 5×5, 3×3 and 1×5 filter sizes, as is shown in Fig. 1.16. Also, another interesting finding is that, for the dataset with more number of input channels, choosing $M \times 5$ filter size give better accuracy compared to smaller datasets. Because, the small size filters can cover most of the input channels in the smaller dataset but not in the dataset with many input channels.

1.5.4 Zero-padding

In this experiment the impact of input data zero-padding in the first convolutional layer on detection accuracy is explored, for Physical Activity Monitoring, dTDS and

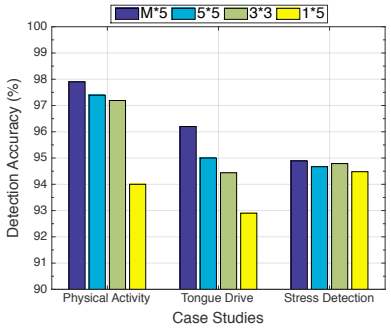

Fig. 1.16: Comparison of SensorNet detection accuracy for four different filter shapes.

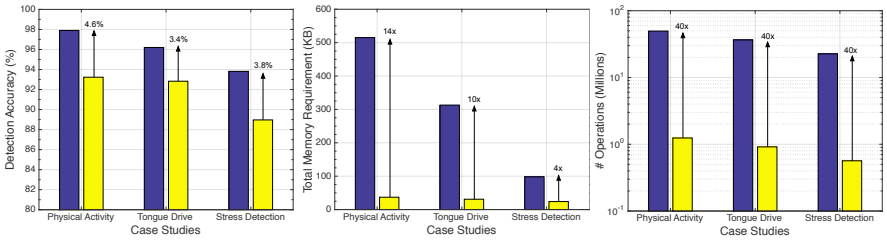

Fig. 1.17: Impact of zero-padding on SensorNet detection accuracy, memory requirements and number of computations, for Physical Activity Monitoring, sdTDS and Stress Detection case studies.

Stress Detection case studies. Input zero-padding makes the output of the convolutional layer to be similar or same as the input to the layer. Based on the results shown in Fig. 1.17, zero-padding the input data helps with accuracy, although it increases the number of parameters and memory requirement. As it can be seen from the figure, by applying the zero-padding, the detection Accuracy increases by 4.6%, 3.4% and 3.8% for Physical Activity Monitoring, sdTDS and Stress Detection case studies, respectively. However, total memory requirements and number of operations are increased 9×, 40×, on average which will affect the power consumption negatively as well. Therefore, SensorNet without zero-padding was implemented on the hardware.

1.5.5 Activation Functions

In Section 1.3.1, it was mentioned that Rectified Linear Unit (ReLU) activation functions is efficient because it requires few operations to perform. Therefore, it reduces the hardware complexity on a hardware embedded setting. In all the convolutional layers used in this chapter ReLU is employed as the activation function. Typically Sigmoid is used as the activation function for the fully-connected layer. However,

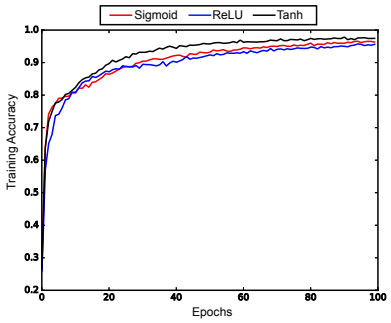

Fig. 1.18: Impact of different activation functions including Sigmoid, Tanh and ReLU in the fully-connected layer on SensorNet accuracy.

Sigmoid introduces hardware complexity to the design which is not desired. Thus, in this section, SensorNet detection accuracy by employing different activation functions in the fully-connected layer is explored. In this experiment, the SensorNet for stand-alone dual-mode Tongue Drive System case study using ReLU as the activation function for all the convolutional layers and using three activation functions including Sigmoid, Tanh and ReLU for the fully-connected layer was trained. The performance results in terms of training accuracy during 100 epochs is shown in Fig. 1.18. As it can be seen from the figure, SensorNet using any of Sigmoid, Tanh and ReLU activation functions achieves similar accuracies eventually and using different activations function does not affect what SensorNet can learn. Therefore, ReLU is chosen as the activation function for all the layers because it has less hardware complexity compared to other activation functions and achieves comparable training and testing accuracy.

1.6 SensorNet Hardware Architecture Design

Implementing hardware architecture for SensorNet faces several challenges such as computational model implementation, efficient parallelism and managing memory transfers. Following are the objective for the hardware architecture design: consumes minimal power, meets latency requirement of an application, occupies small area, needs to be fully reconfigurable and requires low memory. Also, the design constraints require SensorNet hardware architecture to be reconfigurable, because different applications have different requirements. Parameters such as filter shapes, number of filters in the convolutional layers, sizes of the fully-connected and softmax layers are configurable.
Fig. 1.19 depicts SensorNet hardware architecture with implementation details. This architecture is designed based on Algorithm 1 which was depicted and explained

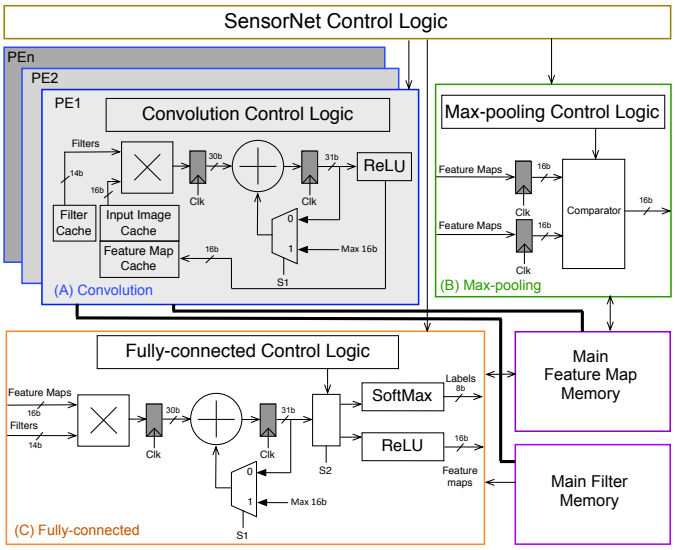

Fig. 1.19: Block diagram of SensorNet hardware architecture which includes convolution, max-pooling and fully-connected blocks and also a top-level state-machine which controls all the blocks. PE refers to convolution Processing Engine (PE)

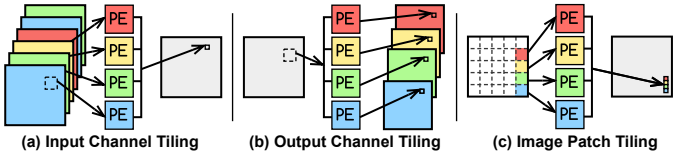

Fig. 1.20: Comparison of different parallel tiling techniques for convolutional layers. Output channel tiling has the least communication contention and inter-core dependency.

in Section 1.3.1. The main components of SensorNet on hardware consists of the following:

(A) Convolutional Performs convolutional layer operations. Also, this block includes ReLU activation logic. PE refers to convolution Processing Engine (PE).

(B) Max-pooling Performs max-pooling operations.

(C) Fully-connected Performs fully-connected layer operations. The fully-connected block includes ReLU and Softmax activations functions. ReLU will be used as the activation for the first fully-connected layer and Softmax will be used for the last fully-connected layer and will perform classification task.

Fig. 1.19-A shows convolution block. As is shown, convolution block contains one multiplier, one adder/subtractor, one cache for saving filters, input feature maps and output feature maps, a multiplexers, a few registers, and a state machine block. When the convolution operations are done for all the input feature maps, the output

feature maps will be saved into the main feature map memory. The input data coming from the sensors are 16-bit two's complement. Also, the filters are considered to be 16-bit two's complement which will be discussed in sub-section **??**. After performing the convolution, the data will pass to ReLU activation function. The output of ReLU is truncated to 16 bits and saved in feature map memory. An offline training is performed to obtain model weights using keras. The model weights are converted to fixed-point format and are represented by 16 bits. The floating-point arithmetic is complex and requires more area, therefore use of fixed point arithmetic will avoid complex multipliers. Fig. 1.19-B shows the max-pool block which contains some registers and a comparator. The input to the max-pool is feature maps data, which is formed by convolution block. After max-pooling operations finish, the results will be saved into the main feature map memory. 1.19-C shows the fully-connected block. As is shown, the architecture consists of a serial dot product engine, a dynamic sorting logic for the Softmax activation function, ReLU logic and a state machine for controlling all sub-blocks. Depends on the layer either ReLU or Softmax can be used. After finishing computations for the fully-connected layers the results will be saved into the main feature memory.

1.6.1 Exploiting Efficient Parallelism

Scalability is one of the key features of the proposed SensorNet on hardware. Therefore, SensorNet hardware architecture was designed to be configured to perform convolution operation in parallel if it is needed specially for fast applications. In deep convolutional neural networks, convolutional layers dominate the computation complexity and consequently affects the latency and throughput. Therefore, for the applications with many input modalities or the applications that need to issue a command very fast, efficient forms of parallelism that exist within convolutional layers must be exploited. In [29] three main forms of parallelism methods were explored, that can be employed in convolutional layers. The basic process for the three tiling methods are shown in Fig. 1.20. The first method, referred as input channel tiling, convolves multiple input feature channels concurrently for a given feature map. The second method, output channel tiling, performs convolution across multiple output channels for a given input channel, simultaneously. The third method, referred as image patch tiling, breaks a given input feature channel into patches and perform convolution on the patches concurrently. [44] analyzed these three tiling methods using the rooftop model to determine what method provides the best throughput in FPGA fabric. Their findings confirm that output channel tiling provides the best form of parallelism when taking into account I/O memory bandwidth and computational load using the computation to communication (CTC) ratio. Therefore, output channel tiling due to minimal dependency among the parallel cores and minimal communication contention is primarily explored here.

1.7 Resources

The CNN architecture explored in this chapter has been constructed based on several tools and libraries. In this section, some of the tools have been categorized:

Numpy Library Numpy arrays are standard representations of numerical data in relation to the Python language. Arrays of such structure ensure efficiency and timing economy in terms of large computations. [38] introduces the numpy array and illustrates how to coordinate this with other libraries.

Pandas Library Pandas is a Python library designed to work with structured datasets allowing ease of manipulation and computation of the data. The work in [26] provides detail design and features of Pandas which serves as as strong complement to the existing Python stack.

TensorFlow TensorFlow is a machine learning module utilized in a variety of environments depending on the application in focus. This maps the dataflow graph neurons across many machines in a cluster including CPU, GPU and TPU (Tensor Processing Units). The flexible architecture of TensorFlow makes it suitable to be deployed in training and inference tasks of deep neural networks. TensorFlow as a large scale machine learning tool was introduced and delineated in [1].

Keras API Keras is a deep learning API (Application Programming Interface) written in Python. Keras allows the swift compilation of deep learning layers and complements the machine learning platform of TensorFlow. The combination of Keras API and TensorFlow ensures efficient low level tensor operations, scaling computation to many devices and precise computing of the model gradients.

Scikit-learn Library This library provides modular representations of machine learning algorithms for supervised and unsupervised learning. Along with packages designated to efficiently implement deep learning model, Scikit-learn also allows functions and building blocks designed for data structure manipulation. [30] gives a walkthrough regarding the features, packages and documentation of the library to enhance ease of use, performance and consistency.

Jupyter Notebook Jupyter notebook is an open source software designed for interactive data science computing mainly across the Python programming language. It started from the 2014 IPyhton project detailed in [31]. This was developed in the open on GitHub and is a free to use platform for algorithmic processes.

1.8 Conclusion

Sensornet is a low power embedded deep convolutional neural network for multimodal time series signal classification. First, the time series data generated by different sensors are converted to images (2-D signals) and then a single DCNN is employed to learn features over multiple modalities and perform the classification task. SensorNet is scalable as it can process different types of time series data with variety of input channels and sampling rates. Also, it does not need to employ separate signal processing techniques for processing the data generated by each sensor modality. Furthermore, there is no need of expert knowledge for extracting features for each sensor data. Moreover, SensorNet has a very efficient architecture which makes it suitable to be employed at IoT and wearable devices. The proposed solution was tested using three real-world case studies including Physical Activity Monitoring, dual-mode Tongue Drive system and Stress detection and based on the results, it achieved an average classification accuracy of 98%, 96.2%, 94% for each application, respectively. A custom low power hardware architecture is also designed for the efficient deployment of SensorNet at embedded real-time systems.

Problems

1.1 Loading the Dataset
Download the dataset (PAMAP2) for the physical activity case study from this link https://archive.ics.uci.edu/ml/datasets/PAMAP2+Physical+Activity+Monitoring.
Read in the dataset as .numpy files or in pandas dataframe and report the following -
(1) Number of samples for each subject.
(2) Perform normalization of the data for each subject.

1.2 Creating the Window Frames
One of the fundamental steps of preparing the time-series dataset for inference is to create windows from the stream of data as described in Article 1.3.2. Build a function that employs striding to create window images from the datasets while taking into account of the sampling frequency of the sensors.

1.3 Generating One Hot Encoded Labels
In deep neural networks involving activity classification the loss function used is most often categorical crossentropy. In order to use this loss function the labels themselves must be categorized. One hot encoding is one such way to create categorical labels. Build a function that can take in any stream of labeled data and generate one hot vectors from it.

1.4 Obtaining Customized Training, Validation and Test Sets
Usually, the input data fed to the model is segmented into training, validation and test set for convenience of functionality. Create a function that can read in the whole data stream and create segments of 70% training, 10% validation and 20% test sets.

1.5 Building the Model
The SensorNet architecture is illutrated in Figure 1.6. Using the Keras libraries perform the following -
(1) Build the SensorNet model with the number of filters and parameters defined as in Article 1.3.3.
(2) Summarize the model architecture in terms of parameters.
(3) Find out the total number of computations for the architecture generated in (1).

1.6 Compiling the Model
Use the model built in problem 1.5 and change the activation functions to tanh. Now, compile the model with 32 batch size, 'Adam' optimizer, 'categorical_crossentropy' loss and 'accuracy' as the metric for 150 epochs.

1.1 The solution to problem 1.1 is revealed here.

Python Code -

```
import numpy as np
from sklearn.preprocessing import normalize

data = np.load("Enter Your Dataset Directory Here")
data_label = np.load("Enter Your Label Directory Here").astype(int)

data = normalize(data[:,1:41], axis=0)
print(data.shape)
label = data_label
print(label.shape)
```

End of Code*

Subject 1 has 408095 samples and 53 channels
Subject 2 has 447064 samples and 53 channels
Subject 3 has 252897 samples and 53 channels
Subject 4 has 329640 samples and 53 channels
Subject 5 has 374847 samples and 53 channels
Subject 6 has 361881 samples and 53 channels
Subject 7 has 313663 samples and 53 channels

Only the first 40 channels are valid for the PAMAP2 dataset.
Hence, during normalization only the first 40 channels are selected.

1.2 The solution to problem 1.2 is revealed here.

Python Code -

```
def Windowing(Window_size,step_size,data,label):
    window_labels = np.zeros([0, label.shape[-1]])
    t_data = np.transpose(data)
    shape = (int((t_data.shape[-1] - Window_size)/step_size) + 1,)\\
     + t_data.shape[:-1] + (Window_size, )
    strides = (t_data.strides[-1] * step_size,) + t_data.strides
    window_images = np.lib.stride_tricks.as_strided(t_data, shape=shape,\\
    strides=strides)
    window_labels = np.append(window_labels, label[Window_size - 1: \\
```

```
    len(label):step_size,:],axis=0)
    return window_images, window_labels

Window_size = 64
step_size = 64
[data,label] = Windowing(Window_size,step_size,data,label)
```

End of Code*

1.3 The solution to problem 1.3 is revealed here.

Python Code -

```
from sklearn.preprocessing import OneHotEncoder

def onehot(labels):
    enc = OneHotEncoder()
    enc.fit(np.unique(labels).reshape(-1, 1))
    labels = enc.transform(labels.reshape(-1, 1)).toarray()
    num_classes = labels.shape[1]
    print("Total Number of classes: " + str(num_classes))

    return labels

label = onehot(label)
print(label.shape)
```

End of Code*

1.4 The solution to problem 1.4 is revealed here.

Python Code -

```
def dataset_segmentation(data,label):
    ratio_t = 0.7
    ratio_v = 0.1

    valid_d = data[int(ratio_t * len(data)): int((ratio_t + ratio_v) * len(data)),:]
    valid_l = label[int(ratio_t * len(data)): int((ratio_t + ratio_v) * len(data)),:]
    print('Validation set shape:')
    print(valid_d.shape)
    print('Validation set label shape:')
    print(valid_l.shape)

    test_d = data[int((ratio_t + ratio_v) * len(data)): len(data),:]
    testl= label[int((ratio_t + ratio_v) * len(data)): len(data),:]
    print('Test set shape:')
    print(test_d.shape)
    print('Test set label shape:')
```

```
    print(test_l.shape)

    train_d = data[0:int(ratio_t * len(data)),:]
    train_l = label[0:int(ratio_t * len(label)),:]
    print('Training set shape:')
    print(train_d.shape)
    print('Training set label shape:')
    print(train_l.shape)

    return train_d,train_l,valid_d,valid_l,test_d,test_l

print('Dataset Configurations')
[train_d,train_l,valid_d,valid_l,test_d,test_l]=dataset_segmentation(data,label)
```

End of Code*

1.5 The solution to problem 1.5 is revealed here.

Python Code -

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Flatten, MaxPooling2D

model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(32, (40, 5), activation='relu', input_shape=
(40,64,1),data_format='channels_last'))
model.add(tf.keras.layers.Conv2D(16, (1, 5), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(1, 2),strides=(1,2)))
model.add(tf.keras.layers.Conv2D(16, (1, 5), activation='relu'))
model.add(tf.keras.layers.Conv2D(8, (1, 5), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D((1,2),strides=(1,2)))
model.add(tf.keras.layers.Conv2D(8, (1, 5), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D((1,2),strides=(1,2)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(13, activation='softmax'))
```

End of Code*

The input shape is (40,64,1) with 40 multimodal information, 64 window size and 1 channel. The last fully-connected layer has the softmax activation to facilitate the 13 labels in the datasets.

In Table 1.3 the maxpool and flatten layers do not contain any trainable parameter because there are no computations inside these layers.

1.6 The solution to problem 1.6 is revealed here.

Table 1.3: Parameter and Computation Calculation for the Model in Problem 1.5

Layers	#Params	#Comps
Conv_1	6432	12288000
Conv_2	2576	1146880
Maxpool_1	0	0
Conv_3	1296	491520
Conv_4	648	102400
Maxpool_2	0	0
Conv_5	328	15360
Maxpool_3	0	0
Flatten	0	0
Dense_1	1600	1536
Dense_2	845	832
Total	13725	14046528

Python Code -

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Flatten, MaxPooling2D

model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(32, (40, 5), activation='tanh', input_shape=
(40,64,1),data_format='channels_last'))
model.add(tf.keras.layers.Conv2D(16, (1, 5), activation='tanh'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(1, 2),strides=(1,2)))
model.add(tf.keras.layers.Conv2D(16, (1, 5), activation='tanh'))
model.add(tf.keras.layers.Conv2D(8, (1, 5), activation='tanh'))
model.add(tf.keras.layers.MaxPooling2D((1,2),strides=(1,2)))
model.add(tf.keras.layers.Conv2D(8, (1, 5), activation='tanh'))
model.add(tf.keras.layers.MaxPooling2D((1,2),strides=(1,2)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(64, activation='tanh'))
model.add(tf.keras.layers.Dense(13, activation='softmax'))

model.compile(optimizer='Adam', loss='sparse_categorical_crossentropy', \\
metrics=['accuracy'])
model.fit(train_d, train_l, epochs=150, batch_size = 32)
```

End of Code*

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.
2. Batista, G. E., Wang, X., and Keogh, E. J. A complexity-invariant distance measure for time series. In *SDM* (2011), vol. 11, SIAM, pp. 699–710.
3. Birjandtalab, J., et al. A non-eeg biosignals dataset for assessment and visualization of neurological status. In *Signal Processing Systems (SiPS), IEEE International Workshop on* (2016), IEEE, pp. 110–114.
4. Chen, Y., Keogh, E., Hu, B., Begum, N., Bagnall, A., Mueen, A., and Batista, G. The ucr time series classification archive, July 2015.
5. Chollet, F., et al. Keras, 2015.
6. Collobert, R., and Weston, J. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning* (2008), ACM, pp. 160–167.
7. Courbariaux, M., et al. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems* (2015), pp. 3123–3131.
8. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).
9. Dauphin, Y. N., and Bengio, Y. Big neural networks waste capacity. *arXiv preprint arXiv:1301.3583* (2013).
10. Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., and Keogh, E. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1542–1552.
11. Girshick, R., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2014).
12. Gong, Y., et al. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115* (2014).
13. Graves, A., Mohamed, A., and Hinton, G. E. Speech recognition with deep recurrent neural networks. *CoRR abs/1303.5778* (2013).
14. Guan, Y., and Ploetz, T. Ensembles of deep lstm learners for activity recognition using wearables. *arXiv preprint arXiv:1703.09370* (2017).
15. Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
16. Hinton, G., et al. Neural networks for machine learning-lecture 6a-overview of mini-batch gradient descent.
17. Hinton, G., et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine 29*, 6 (2012), 82–97.
18. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
19. Hwu, T., et al. A self-driving robot using deep convolutional neural networks on neuromorphic hardware. In *Neural Networks (IJCNN), 2017 International Joint Conference on* (2017), IEEE, pp. 635–641.
20. Jafari, A., et al. A low-power wearable stand-alone tongue drive system for people with severe disabilities. *IEEE transactions on biomedical circuits and systems 12*, 1 (2018), 58–67.

21. Jafari, A., Ghovanloo, M., and Mohsenin, T. An embedded fpga accelerator for a standalone dual-mode assistive device. *Memory (Kb) 101* (2017), 102.
22. Jiang, W., and Yin, Z. Human activity recognition using wearable sensors by deep convolutional neural networks. In *Proceedings of the 23rd ACM international conference on Multimedia*.
23. Kampouraki, A., Manis, G., and Nikou, C. Heartbeat time series classification with support vector machines. *IEEE Transactions on Information Technology in Biomedicine 13*, 4.
24. Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.
25. Li, X., et al. Concurrent activity recognition with multimodal cnn-lstm structure. *arXiv preprint arXiv:1702.01638* (2017).
26. Mckinney, W. pandas: a foundational python library for data analysis and statistics. *Python High Performance Science Computer* (01 2011).
27. Nurvitadhi, E., et al. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *Field-Programmable Technology (FPT), 2016 International Conference on* (2016), IEEE.
28. Ordóñez, F. J., and Roggen, D. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors 16*, 1 (2016), 115.
29. Page, A., et al. Sparcnet: A hardware accelerator for efficient deployment of sparse convolutional networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* (2017).
30. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research 12*, 85 (2011), 2825–2830.
31. Pérez, F., and Granger, B. E. IPython: a system for interactive scientific computing. *Computing in Science and Engineering 9*, 3 (May 2007), 21–29.
32. Rajpurkar, P., Hannun, A. Y., Haghpanahi, M., Bourn, C., and Ng, A. Y. Cardiologist-level arrhythmia detection with convolutional neural networks. *arXiv preprint arXiv:1707.01836* (2017).
33. Rastegari, M., et al. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision* (2016), Springer, pp. 525–542.
34. Reiss, A., and Stricker, D. Introducing a modular activity monitoring system. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society* (2011), IEEE, pp. 5621–5624.
35. Reiss, A., and Stricker, D. Introducing a new benchmarked dataset for activity monitoring. In *Wearable Computers (ISWC), 2012 16th International Symposium on* (2012), IEEE, pp. 108–109.
36. Seto, S., Zhang, W., and Zhou, Y. Multivariate time series classification using dynamic time warping template selection for human activity recognition. In *Computational Intelligence, 2015 IEEE Symposium Series on* (2015), IEEE, pp. 1399–1406.
37. Umuroglu, Y., et al. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the SIGDA* (2017), ACM.
38. van der Walt, S., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *CoRR abs/1102.1523* (2011).
39. Vepakomma, P., et al. A-wristocracy: Deep learning on wrist-worn sensing for recognition of user complex activities. In *Wearable and Implantable Body Sensor Networks (BSN), 2015 IEEE 12th International Conference on* (2015), IEEE, pp. 1–6.
40. Wang, Z., and Oates, T. Encoding time series as images for visual inspection and classification using tiled convolutional neural networks. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015).

41. Xu, K., Ba, J., Kiros, R., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044* (2015).
42. Yan, W., and Yu, L. On accurate and reliable anomaly detection for gas turbine combustors: A deep learning approach. In *Proceedings of the Annual Conference of the Prognostics and Health Management Society* (2015).
43. Yao, S., et al. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web* (2017), pp. 351–360.
44. Zhang, C., et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2015), FPGA '15, ACM, pp. 161–170.
45. Zhao, R., et al. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2017), ACM, pp. 15–24.
46. Zheng, Y., Liu, Q., Chen, E., Ge, Y., and Zhao, J. L. Time series classification using multi-channels deep convolutional neural networks. In *International Conference on Web-Age Information Management* (2014), Springer, pp. 298–310.