UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR MEDIZINISCHE INFORMATIK

Aus dem Institut für Medizinische Informatik der Universität zu Lübeck
Direktor: Prof. Dr. rer. nat. habil. Heinz Handels

Deep Multi-Task Learning in Sensor-Based Time Series Classification

Klassifikation von Sensor-Basierten Zeitreihen mittels Deep Multi-Task Learning

Masterarbeit
im Rahmen des Studienganges Medizinische Informatik
der Universität zu Lübeck

vorgelegt von
Joshua Philip Wiedekopf, B.Sc.

ausgegeben und betreut von
Prof. Dr.-Ing. habil. Marcin Grzegorzek

mit Unterstützung von
Frédéric Li, M.Sc.

Lübeck, den 10. Januar 2020

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Joshua Wiedekopf

Lübeck, den 10. Januar 2020

Kurzfassung

Multi-Task Learning (MTL) im Bereich von tiefen neuronalen Netzen (Deep Neural Networks, DNNs) bezeichnet die Verwendung mehrerer Aufgaben, die parallel durch das Netzwerk gelernt werden sollen. Durch dieses Vorgehen sollen die latenten Repräsentationen der Eingaben plausibler werden, wodurch viele Ansätze für MTL den bisherigen Status Quo übertreffen konnten. Die Verwendung von MTL im Bereich der Zeitreihenanalyse, die für eine Vielzahl von Anwendungen im medizinischen Bereich von Belang ist, ist jedoch bislang nur in geringem Maße erforscht.

In der vorliegenden Arbeit werden drei verschiedene Ansätze, *Hard Parameter Sharing* (HPS), *Soft Parameter Sharing* (SPS) und ein hybrides Vorgehen, unter Verwendung zweier Datensätze aus den Bereichen *Menschliche Bewegungserkennung* (OPPORTUNITY) und *Emotionserkennung* (DEAP) verglichen. SPS wurde mit einem regularisierten Ansatz unter Verwendung der Tensoren-Spur untersucht, während Cross-Stitch Networks (CSNs) als hybrider Ansatz verwendet wurden. Wir zeigen, dass nicht jeder dieser Ansätze für Zeitreihen gleich gute Ergebnisse produziert. Insbesondere konnten Verbesserungen mittels SPS gar nicht und für CSNs nur für einen Datensatz gezeigt werden. Um weitere Untersuchungen zu unterstützen, legen wir unseren Quellcode offen.

Abstract

Multi-Task Learning (MTL) applied to Deep Neural Networks (DNNs) refers to techniques aiming at training a network to perform multiple tasks at once. In this way, latent representations of the input are generated that are more plausible than what is generated using classical Single-Task Learning (STL). This is evidenced by the superior performance many approaches to this concept have demonstrated when compared to STL. However, the use of MTL approaches on sensor-based time-series data, having an enormous range of potential medical applications, has not received much attention so far.

In this thesis, three different approaches, namely Hard Parameter Sharing, Soft Parameter Sharing and a hybrid approach, are compared on two datasets from the domain of Human Activity Recognition (HAR) and Emotion Recognition—OPPORTUNITY and DEAP. For SPS, an approach using the Tensor Trace Norm as a regulariser was chosen from the literature, while the hybrid approach was investigated using Cross-Stitch Networks (CSNs). We demonstrate that not every approach is equally beneficial. In particular, benefits were observed using HPS on both datasets and CSNs on only one dataset. We could not demonstrate that the chosen approach for SPS works for time-series. To aid further research in this field, our source code is made public.

Contents

List of Figures

List of Tables

Glossary

ANN Artificial Neural Network. 5

CNN Convolutional Neural Network. 7, 8, 12

CSN Cross-Stitch Network. V, 3, 51, 54

DNN Deep Neural Network. V, 1, 5, 7

EEG electroencephalography. 23

EMG electromyography. 23

EOG electrooculography. 23

GRU Gated Recurrent Unit. 9

HAR Human Activity Recognition. V, 1, 21, 28, 31

HPS Hard Parameter Sharing. V, 3, 9, 10, 12, 41, 51

LOESS Locally Estimated Scatterplot Smoothing. 40

LSTM Long Short-Term Memory. 8, 9

MLP Multi-Layer Perceptron. 7

MTL Multi-Task Learning. V, IX, 2, 9–13

ReLU Rectified Linear Unit. 7

RNN Recurrent Neural Network. 8

SPS Soft Parameter Sharing. V, 3, 9, 10, 12, 51

STL Single-Task Learning. V, 9–11

SVM Support Vector Machine. 30

TTN Tensor Trace Norm. 18, 43, 54

Chapter 1

Introduction

1.1 Motivation

Since their initial introduction, Deep Neural Networks (DNNs) have yielded substantial improvements in many Machine Learning domains over traditional, model-based ahaspproaches. In the medical domain, the effect of these networks has especially proven to be beneficial in Image Processing pipelines, which are often used to classify medical images, or to extract and estimate data from these images. In this way, they can generate information relevant for physicians to the treatment or diagnosis of the patient. DNNs have also allowed for significant progress in medical applications relying on analyses and classification of time-series data coming from body-worn sensors, such as motion sensors, electroencephalography, eye tracking and many more. These data streams can be collected in a large variety of settings and can be used to serve a number of problems. For example, motion sensors are often placed on subjects for experiments on Human Activity Recognition (HAR), which is an active field of research in which DNNs have achieved state-of-the-art performance [Li+18; Núñ+18]. Similarly, a number of publications in Pain Recognition problems use these methods with great effect [LRP17; Vij+17]. Another very interesting field of reseach on time series is that of Emotion Recognition, which is superficially similar to Pain Recognition because both pain and emotion vary greatly between subjects and over time [Gru+18]. Still, DNNs can surface latent similarities between subjects and often perform better than comparable approaches [Li+19].

One example—illustrating both the usefulness of time-series data to assist patients and a possible application of Multi-Task Learning—is an assisted living scenario, where an (elderly) subject may be outfitted with a wristband or a smartwatch. This device features accelerometers and gyroscopes to estimate their position and velocities within the home. If the patient falls and is unable to get up on their own, an alert will be triggered, so that emergency personnel is automatically dispatched to help the patient. In such a scenario, the device has to continually analyze the stream of data, paying attention to the context of the current measurements, to give a classification result whether or not the patient is in a situation where help is required. While this scenario may well be served by traditional model-based approaches, other scenarios may not be as easy to implement using such a method. Indeed, the popular Apple Watch uses a model-based system relying on hand-crafted features and static thresholds [App19], but

for the purposes of this example, we will assume that the device used in this scenario uses neural networks for processing.

While DNNs have yielded substantial benefits in many domains, there have been some where the results have not been as drastically better or have not improved as quickly as in others. A common problem is the lack of training data, because a DNN-based model needs a lot of data to extract meaningful information from the dataset. If little data is available, it is easy for the training process to get stuck in a local minimum that it can not get out of. In this way, the model performs much worse on the validation dataset than on the training set, which is called *overfitting*. This problem is well-known [Haw04], and while there are some mitigations [Sri+14], for small datasets, the best approaches often use handcrafted features and simple classification approaches, such as Support Vector Machines [Mon+18].

In the last few years, MTL has seen a an increase in popularity as a means to boost the performance of established models. The core ideas of MTL, which were popularised in 1997 in [Car97], are very simple: If a domain yields multiple *tasks* which are of relevance to the research question, training a model in parallel on these tasks may very well improve the results over training multiple models for each task in isolation. In the example above, one task may be the detection of falls, while another task may be the detection of sport activities, which is also a feature common to many smartwatches. In this application, a common system would be trained that gives both a fall detection score as well as a sport activity classification (or the lack of such an activity). This might reduce the numbers of false-positive alarms from the system, if for example the subject is in a swimming pool and purposely jumps into the water. This may generate similar movement data to a fall at home, so that the fall-detection task may report a high score. However, the sport-detection task would indicate that the subject is swimming, and because such a combination of labels in the training data will likely not be present, the system will not activate the alarm.

From the core idea of MTL we can now assume that the training of two such tasks in unison imparts a so-called *inductive bias* to the hidden features generated by our model. Hence, the model is now forced to generate not only a hidden representation of the input data that matches the single respective task, but also has to support multiple tasks in parallel, so that a hidden representation now has to be plausible for both tasks. In effect, the space of the hidden features will now be restricted to the intersection of the two subspaces for each task.

From the perspective of a human, it is very intuitive to assume that such a setup, which „applies the knowledge we have acquired by learning related tasks“ [Rud17] is beneficial compared to training tasks in isolation, because this is a very important factor in the ways humans learn. As noted in [Rud17], „a baby first learns to recognize faces and can then apply this knowledge to recognize other objects“.

1.2 Objective & Scope

This thesis aims to investigate the possible benefits of Multi-Task Learning in the domain of sensor-based time series classification, which has a wide array of potential medical applications. To this end, different MTL approaches are investigated to establish whether they aid in the classification of the problems investigated. Experiments are carried out on two datasets from different domains, Human Activity Recognition and Emotion Recognition, to be able to draw reliable conclusions from this study. While Multi-Task Learning has seen many applications in the literature already, results on time-series classification in the medical field has not yet been explored in-depth.

Because there is a very wide range of MTL methods available in the literature, experiments were restricted to three different approaches, namely Hard Parameter Sharing (HPS), Soft Parameter Sharing (SPS) and Cross-Stitch Networks (CSNs), as an approach that falls in between the other two. While HPS is a very generic approach, for each of the other two methods one, specific work from the literature [Mis+16; YH16] was chosen to implement the respective technique.

The contribution of this thesis mainly lies in the comparative evaluation of the three very different approaches. It also provides guidance for implementing MTL for time-series, for which the source code is also publicly provided[1]. This code may help other researchers, because to our knowledge no other code in the popular Deep Learning framework *Keras* for CSNs and SPS is currently available publicly. However, it is not the explicit aim of this thesis to find an architecture that surpasses the current state-of-the-art in the respective field. Rather, the evaluation of the experiments that were carried is mostly performed in a qualitative way to draw conclusions whether the three approaches, and hence Multi-Task Learning, can aid these classification problems and is thus potentially effective when applied to time-series data.

1.3 Outline

The present work is divided into five parts. In the following chapter 2, the fundamental concepts, namely Deep Learning and Multi-Task Learning, are introduced. In chapter 3, the experimental setup is described. As this thesis examines three approaches applied to two datasets, the considerations for these are explained in detail. Following this description of the setup, the results are presented in chapter 4 for each of the three approaches. They are discussed in chapter 5. That chapter also provides an outlook for possible future works.

[1] at https://github.com/LtSurgekopf/TimeSeriesMTL

Chapter 2

Fundamental Concepts

2.1 Deep Neural Networks

While Artificial Neural Networks (ANNs) are not a new idea, recent advances in hardware, software and mathematical concepts have enabled the training of very deep *Deep Neural Networks (DNNs)* featuring a large number of layers, which enabled significant advances in many domains [Ben09]. The terms ANN and DNN are now often used interchangeably. In this thesis, the term DNN will be preferred, which will be defined as a network with at least two intermediate, hidden layers.

The basic idea of these networks is based on the way neurons in living organisms interact. In essence, every DNN is a network of interconnected neurons with inputs and outputs that each apply a function with learnable parameters to $n \in \mathbb{N}$ inputs, and output $k \geq 1$ vectors $\vec{y}_k \in \mathbb{R}^\star$, which may have different shapes for complex networks. For example, in multi-label classification, the network might output confidence scores for the different labels, which may feature a different number of classes. Thus, the network can be described as some (generally highly non-linear) function $\mathcal{F}$ over some inputs $X \in \mathbb{R}^n$, giving estimated outputs $\hat{Y}$.

For the network to perform any function, it has to be trained. This process requires tuning the parameters of the component functions (i.e. the parameters of every neuron), so that it can perform the task with a minimum error. The error will be qualified using a *loss function*, taking the data, the predicted outputs, and the expected outputs as inputs, and returns a scalar indicating how different the outputs are from the expected values. By using the gradient descent algorithm, which utilises partial differentiations of this loss, the parameters can be changed in the required direction so that the loss function becomes minimal over time.

For most applications, DNNs perform *supervised* learning, i.e. the training data is labelled with the expected values for the task that is investigated. For a classification problem, every example will have at least one label attached. By training on a large number of samples, the network will then learn features from the data, so that it can reproduce the distribution of the labels in the training data and classify previously unknown data. For these supervised problems, the loss function will be defined in terms of the estimated labels and the known, actual labels in the training dataset. The choice of a loss-function is however dependent on the problem investigated. The alternative to supervised learning is *unsupervised* learning, where no labels are available. Instead, the

network will infer clusterings of the data itself. Because this clustering does not follow a known label distribution, this may not correspond to what the user expects. While unsupervised DNN approaches have many uses, they are often used for generative models or in domains where labelled data is very hard to obtain. Often, classification using unsupervised approaches is not as accurate as supervised ones [Li+18; Wan+19], and in the context of MTL, it is harder to define tasks for unlabelled datasets. Thus, in this thesis, only supervised networks were considered.

A supervised network can learn the distribution of the input data using two steps. First, the optimiser first has to perform a *forward pass* through the layers, in which it applies the functions described by the layers to the inputs to obtain estimated outputs. The parameters will generally be randomly initialised, so that the error will be very large. By partially differentiating the loss function for every parameter in the overall parameter $\boldsymbol{\Theta}$, i.e. computing the gradient $\nabla_{\boldsymbol{\Theta}}\mathcal{L}$, the optimiser can compute an update for each parameter that pushes the loss further toward zero. This process is known as *back-propagation* [RHW86], because the loss is fed backwards through the network.

In practice, this training is generally done by providing *minibatches*, small subsets of the dataset of a few samples each, to the network during training. While processing single examples in each step is also an option, the minibatches stabilise the training process and combat overfitting. This is because each sample in the minibatch will yield different gradients that have to be averaged out to obtain an update. In this way, it is harder for the network to get stuck in a local minimum. In most applications, the effect of the update is further attenuated using a hyperparameter called the learning rate. A hyperparameter is any parameter that is not changed by the optimiser and which has to be chosen empirically by the user. This learning rate weights the amount of information from the update that is applied to the current parameter. Choosing the appropriate learning rate—or the appropriate strategy of changing this hyperparameter during training—is a crucial step of any DNN method.

In summary, the training process minimizes an arbitrary loss function $\mathcal{L}$ for a network $\mathcal{F}$ with a parameter tensor $\boldsymbol{\Theta}$:

$$\min_{\boldsymbol{\Theta}\in\mathcal{R}}\mathcal{L}(\hat{Y}, Y) = \min_{\boldsymbol{\Theta}\in\mathcal{R}}\mathcal{L}(\mathcal{F}(X|\boldsymbol{\Theta}), Y) \tag{2.1}$$

The optimiser is a variation of the Stochastic Gradient Descent algorithm [RHW86], where the network first computes the estimate outputs and the error function for a minibatch, and then computes the derivative of this loss in respect to each parameter, which pulls the loss towards a minimum. The update rule for the parameter tensor $\boldsymbol{\Theta}$ at timestep t with a learning rate of η is [Bon18, p. 337]:

$$\boldsymbol{\Theta}^{(t+1)} = \boldsymbol{\Theta}^{(t)} - \eta\nabla_{\boldsymbol{\Theta}}\mathcal{L} \tag{2.2}$$

The following sections will present the most common types of DNNs and further concepts used in this thesis.

2.1.1 Multi-Layer Perceptrons

Multi-Layer Perceptrons (MLPs) are one the simplest types of DNNs. Their behaviour is directly inspired by the way neurons in biological organisms behave. A single perceptron has a vector of inputs $\vec{x} \in \mathbb{R}^n$, and a single output y. A perceptron with a weight $\vec{w} \in \mathbb{R}^n$ and a bias term $b \in \mathbb{R}$ will compute:

$$y(\vec{x}|\vec{w}, b) = f(x \cdot w + b), \tag{2.3}$$

Here, f is a non-linear function, called the *activation function*, which processes the weighted sum of the neuron inputs. Popular choices of the activation function include tanh and especially the Rectified Linear Unit (ReLU) [Hah+00], which has seen a lot of use in recent publications [HDR19]:

$$ReLU(x) = \max(x, 0) \tag{2.4}$$

A typical architecture of a MLP network consists of an input layer, to which at least one intermediate layer (the *latent* or *hidden* layer) is connected. The hidden layers are followed by an output layer. Every neuron in a layer is connected to every neuron in the previous layer. By means of the backpropagation algorithm, the hidden layers will generate features from the input data that represent different quantities of the input, and which are combined in deeper layers using the connections between them. These features are often difficult to interpret for humans [GAZ19; Vau96], but because each neuron in the first layer is connected to every input feature, they will be different weighted sums of these inputs. When processing images for example, some neurons might pay attention to pixels close to the midpoint, while some may only consider horizontal lines [Vau96], setting the other weights to zero.

The output layer neurons of the MLP will each output a scalar value, that is often subject to the softmax function. In classification problems, there will be one output neuron per possible class and the weighted sum computed in each neuron will most often be fed into the *softmax* function. It maps this value into the range $(0, 1)$, so that it can be interpreted as a probability. The class with the highest class probability is considered the output of the network. In the literature, MLPs are often also referred to as *dense* or *fully-connected* networks.

2.1.2 Convolutional Neural Networks

MLP networks are very powerful, but because every neuron is connected to every output in the previous layer, they feature a large number of tuneable parameters, making training of very deep networks difficult [Ben09]. Convolutional Neural Networks (CNNs) [LeC+98] work around this problem by featuring a large number of small filters, that are applied to every equally-sized patch in the input tensor. These filters are often one-or two-dimensional tensors with few pixels each. To compute the convolution, the input image is divided into a regular grid with (overlapping) patches of the same size

as the filter. The filter is then applied to every patch and the scalar product of the patch and the (scalar) result is used as the output for this midpoint. As opposed to MLP networks, the output of a CNN neuron will generally not just be a scalar number, but an arbitrarily shaped tensor that will be used as the input of the next channel.

Common choices for image processing are two-dimensional filters with a side length of 3, with a stride (step of the grid midpoints) of 1. In this way, the features obtained by the convolution of a filter can approximate common image processing operators such as edge detectors or gradient computation [Kar16]. Going back to the introductory example of fall detection using wearable sensors, one feature could detect zero-crossings in the data stream of each channel, while another filter could combine information from consecutive channels to detect simultaneous large values.

Many CNN architectures also feature *pooling* layers, which are designed to reduce the dimensionality of the output tensors by applying a suitable downsampling operator to small blocks of the input tensor. For this purpose, a maximum or an average operator is often used. By reducing the dimensionality, the risk of overfitting is mitigated by making it harder for the the network to just *store* the input data and forcing it to learn higher-level features.

2.1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are different to other types of networks because their output does not only depend on the input sample, but also on its *history*. They are thus in principle very well-suited to time-series processing. One example given in [Bon18, p. 400] is that of predicting a day's high temperature x^t from the previous day's temperature x^{t-1}. By virtue of the backpropagation algorithm, a MLP trained on a suitably large training set will predict a value that yields minimum error on average, but one that may be wildly different to the actual temperature. This value will be dependent on the effect of seasons, pressure differentials and so forth, which will affect the previous values that come before x^t, so for the network to take those into consideration, it has to be provided with some kind of memory.

This addition of memory will however make each neuron a function of both the current input value, as well as the previous values. The way this feedback loop is implemented is what differentiates approaches to RNN. One of the dominant approaches is the Long Short-Term Memory (LSTM) cell [HS97], which has enabled tremendous advances in recent years [Bon18]. This type of network not only provides a mechanism for *remembering* data, but also for *forgetting*. For this, the output of a LSTM cell depends on a cell state that is controlled by three gates, the *forget gate*, the *input gate*, and the *output gate*. These gates, which are sigmoid functions that control the amount of information that flows through them, control the cell state and thus the output of the block. The *forget* gate controls whether items from the previous state will be deleted—for example, for periodic samples because a new wave crest starts. Afterwards, new information might get added to the state using the *input* gate if the cell decides that it

contributes to the task. The ultimate output is controlled by the *output* gate that takes information from the state. Variants of this idea exist, such as the Gated Recurrent Unit (GRU) [Cho+14], which is a simplified version of the LSTM that features only two gates and no explicit state.

2.2 Multi-Task Learning

Multi-Task Learning (MTL) is based on the idea that a learning method can benefit from processing multiple different, somewhat related, tasks at the same time. Such tasks are often regression or classification tasks which are performed on the same data. They may be primary or auxiliary in nature, while some approaches even use adversarial tasks [Rud17]. For most purposes, high performance in the auxiliary tasks is not required, but they merely aim to improve the performance of the main task [Rud17].

2.2.1 MTL in Neural Networks

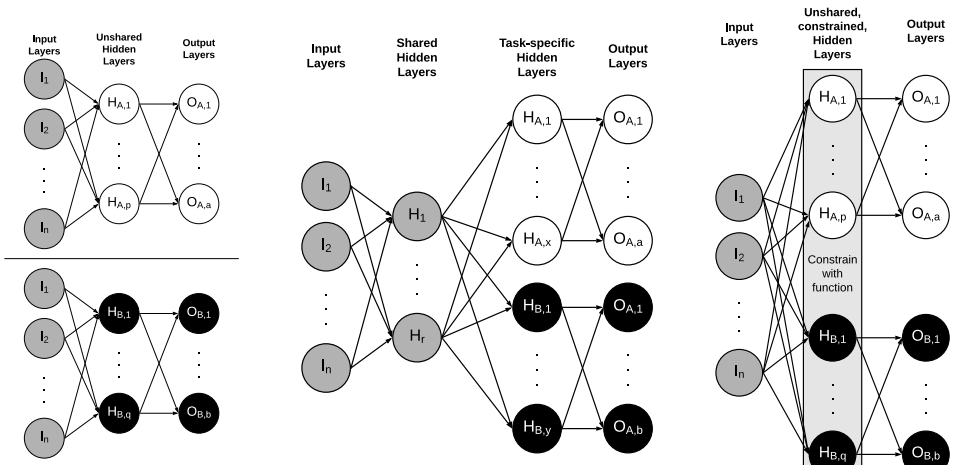
As surveyed in [Rud17], MTL in DNN models can be roughly separated into two classes, which are illustrated in Figure 2.1 on the following page.

- HPS, where the network has a number of shared layers and a number of task-specific branches, and
- SPS, where every task uses a separate network, but the weights of the networks are regularized to enforce similarity.

Of these two classes, HPS is the one that has been proposed and popularised by Rich Caruana [Car97] in the last century and has seen more use. Other approaches that fall between HPS and SPS have also been proposed recently, some of which are outlined in subsection 2.2.3.

Traditionally, most Machine Learning problems are approached using only one task, which in this thesis will be referred to as Single-Task Learning (STL). Most multi-label classification problems, where more than one label for the data has to be obtained, should be regarded as STL problems, because most neural-network based approaches share almost everything, without clear task-specific branches. Furthermore, these problems are often approached by training multiple models on the same data, which do not share anything.

For any MTL method to work, the definition of a set of tasks is mandatory. In some domains, the tasks are trivial to find, as the domain might require the data to be classified in more than one labelled dimension—e.g. for the aforementioned fall detection example, another task would be the sporting activity classifier receiving the same sensor data stream. In other domains, it may not be possible to define another equally important task, so auxiliary tasks are defined instead. For these kinds of tasks, the performance



(a) STL **(b)** HPS **(c)** SPS

Figure 2.1: A comparison of the MTL approaches HPS and SPS with STL for a two-task problem. For the sake of presentation, the architectures have been simplified by using only dense neurons. Neurons belonging to Task A are white, while neurons specific to Task B are black. Shared neurons have been coloured grey. This illustration has been inspired by [Car97, P. 19]
(a): two different tasks are trained on the same inputs, but not simultaneously
(b): the tasks are trained in unison, featuring shared (H_i) and unshared ($H_{\{A,B\},j}$) hidden layers
(c): two different sub-networks are trained simultaneously, operating on the same inputs. Hidden layers are encouraged to be similar by a regularization function

of the network is not considered, they only serve to improve performance of the main task. As pointed out by [Car97] and [Rud17], the set of tasks should be related to each other, i.e. operate on the same inputs. While other approaches may allow for distinct inputs for the tasks, this was not considered in this thesis.

In the case of HPS, most authors use an architecture that features a number of shared, feature-generating layers, on which task-specific classification layers reside. One common approach is the adaptation of an existing, proven network (or often rather a set of networks) to form a single network with sharing between tasks. When combining these networks, the sharing architecture has to be chosen in addition to the overall architecture of the network. This includes the possibility of the task-specific layers having different architectures entirely. Hence, transforming a set of well-performing STL models into a MTL model is not trivial and requires a careful choice of these further hyperparameters for sharing. Furthermore, the shared architecture will generally increase the number of adjustable parameters by increasing the number of layers compared to one STL network. This may adversely effect training times and will require more computing resources.

SPS however maintains separate networks for each task, and often, a set of well-performing networks can be re-used in a MTL context. Still, a suitable regulariser has to be defined and applied to suitably selected layers of the overall network. The regularisation function serves to enforce similarity in the parameters of each task-specific layer. Common examples of regularisers include the L2 distance norm, ex-

plored in [Duo+15], as well as the Tensor Trace Norm, which has been investigated in [YH16]. For each layer subject to the constraints of this function, a term will be added to the overall loss function, which penalises differences in the parameters of the component networks along with the loss component penalising wrongly classified training examples.

In addition to SPS and HPS, as well as the aforementioned approaches that fall in between these two—some of which will be introduced in subsection 2.2.3—, some sources also investigate the viability of algorithmically finding a suitable architecture for sharing. This can prove especially beneficial if a lot of tasks are available to augment the main task. In this case, it may not be clear which tasks actually boost performance and at which point they should be included in the architecture. To this end, the use of genetic algorithms and methods inspired by these can achieve very good results [Lu+16; Rud17]. In this thesis, this was however not investigated because such algorithms feature very long running times, may not always find a global optimum when using a greedy strategy [Rud17] and because the number of tasks was not extremely large for the datasets investigated.

2.2.2 Explaining MTL Benefits

In [Car97], a number of hypotheses, which aim to explain the observed benefits by MTL, are introduced, which are further explained in [Rud17]:

Representation Bias The network will likely assume a configuration during training where representations that all tasks prefer are generated, which improves generalisation.

Data Augmentation All tasks are inherently noisy to some degree. The noise pattern will be different for each tasks, thus the network is able to average out the noises more easily, decreasing the risk of overfitting.

Eavesdropping Some tasks may be hard to learn just from the input alone. By eavesdropping on the other tasks, the underlying relationship might be easier to model by the network.

Regularization MTL adds an inductive bias to the model, yielding a network that is less able to fit random noise.

Under these assumptions, a MTL model should be less susceptible to overfitting and perform better under the metrics imposed by the domain than a comparable STL model.

2.2.3 Prior Works on Deep MTL

MTL has been used in neural networks since the concept was popularized by Rich Caruana in his PhD thesis [Car97]. At that time, shallow MLP-based architectures with only one or two hidden layers were predominantly used, but in recent days, the

research has transitioned to modern deep networks using a large variety of different layers. For example, in the domain of CNN-based facial landmark detection, the use of additional tasks (e.g. head pose estimation and facial attribute inference) has been shown both to outperform existing methods, as well as reducing model complexity with regards to the number of calculations required [Zha+14].

Many publications that show performance improvements using MTL approaches feature innovative new ideas. One rather intuitive proposition, which falls between HPS and SPS, was introduced in [Mis+16]. There, different layers were trained for each task, similar to SPS, but the layers at same depths were not constrained by a regularizer. Instead, special *cross-stitch units* were used, which combine the output of the previous layers in a pixel-wise fashion by performing weighted matrix addition.

For example, if a cross-stitch unit is placed between the branches of a two-task network, combining the outputs of two convolutional neurons, it is parameterised by $\mathbf{\Phi} \in \mathbb{R}^{2\times 2}$. It calculates outputs for each position of the identically-shaped output tensors of the convolutions. For simplicity, it can be assumed that these tensors can be indexed using a one-dimensional index, i.e. they are flattened to a one-dimensional vector. Hence, the tensors returned by the convolutional operators can be viewed as vectors $\vec{x}^{(A)}, \vec{x}^{(B)} \in \mathbb{R}^l$.

For any position i in these unrolled tensors, the function computed by the Cross-Stitch unit will be:

$$\begin{aligned} CS_{\mathbf{\Phi}}(x_i) = \mathbf{\Phi} \cdot x_i &= \begin{bmatrix} \alpha_{A,A} & \alpha_{A,B} \\ \alpha_{B,A} & \alpha_{B,B} \end{bmatrix} \cdot \begin{bmatrix} x_i^{(A)} \\ x_i^{(B)} \end{bmatrix} \\ &= \begin{bmatrix} \alpha_{A,A} \cdot x_i^{(A)} + \alpha_{A,B} \cdot x_i^{(B)} \\ \alpha_{B,A} \cdot x_i^{(A)} + \alpha_{B,B} \cdot x_i^{(B)} \end{bmatrix} \end{aligned} \tag{2.5}$$

Hence, the value $\alpha_{A,A}$ would weight the amount of information from the CNN block for the first task that should be retained for task A, while $\alpha_{A,B}$ will weight the amount of information from task B that should be shared for task A. This calculation is carried out for every position in the tensor output of the previous layers, which thus have to have identical shapes.

This idea is further explored in [Rud+17] in the *Sluice networks* they proposed, where the cross-stitch units only operate on one part of the output for each task. In addition, not all layer outputs are used equally for prediction, which is enforced by a parameter β for each layer. These architectures are said to be able to *learn what to share* to a greater extent then the other approaches introduced above. This is also an important feature of other MTL methods, some of which use adaptive algorithms to determine which tasks to use and where to share information automatically in an iterative process. However, for the purposes of this thesis, these approaches (some being tallied in [Rud17]) are beyond the scope of what could be investigated. In particular, while Sluice networks are a very interesting approach, they are similar enough in principle to

Cross-Stitch networks that from a high performance of the latter, it could be inferred that the former would also perform well. Furthermore, no Keras code for Sluice networks was publicly available, so that time constraints did not allow for an independent implementation.

Chapter 3

Experimental Setup

In this thesis, experiments have been carried out on two datasets, one of which in the domain of Human Activity Recognition, the other in the domain of Emotion Recognition. While there are special considerations for each dataset, the core framework of this thesis follows the same basic ideas and architectures for both applications.

In this chapter, the general procedures are examined in section 3.1. Afterwards, the datasets are introduced and their individual preprocessing discussed. Moreover, details regarding the implementation (i.e. Deep Learning framework and hardware specifications) are featured in section 3.3.

3.1 Concepts and Architectures

3.1.1 Single-Task Learning Baselines

As mentioned previously, the main objective of this thesis was not put on outperforming current state-of-the-art approaches. Instead, the focus of this work should be on establishing whether existing approaches can easily be improved upon by using MTL techniques and slight modifications to the existing architectures.

Because not every team of researchers evaluates their approaches using exactly the same protocol, a suitable STL baseline not only had to be determined using extensive literature reviews, but that method also had to be trained and evaluated using the same methodology as the MTL methods.

It would certainly be an option to use the methods that have the best reported results as of writing as the baseline. However, the baseline serves another role: the MTL approaches evaluated are defined based on the architecture of the baseline, to evaluate the ease of transformation of a STL model to MTL. Since MTL does incur an overhead in computational cost, especially using SPS methods, consideration had to be given to the cost of the baseline. Hence, baselines which were relatively cheap, but still yielded acceptable performance, were chosen if there were more expensive options that did not achieve much better performance. This may also have the effect of providing a bigger margin for improvement using MTL, which makes it easier to detect significant differences. Because the selection of the STL baseline network is ultimately very

dataset-specific, the architecture of the networks used will be introduced below, after the introduction of the datasets.

3.1.2 Hard Parameter Sharing

As aforementioned, if a set of tasks is to be learned using a set of very similar, task-specific networks, transforming this set into a MTL network using Hard Parameter Sharing is rather simple in principle. A decision has to be made which common layers of the network are to be shared. Hence, a network of the desired architecture has to be created, using the predefined architecture from the base networks. Then, the task-specific branches have to be added on top of this set of shared feature layers. This architecture is illustrated in Figure 3.1.

During training, all of these branches should be made to contribute to the overall loss function, so that their performance can be taken into account. An easy way to achieve this goal is simply defining a loss function for each output tensor, which may use supervised or even unsupervised learning, and then define the overall loss function as a (weighted) addition of these component loss functions.

For a three-task network with the tasks $T = \{A, B, C\}$ with ground truth data Y and estimated outputs $\hat{Y}$, the loss function is given below w.l.o.g:

$$\begin{aligned} \mathcal{L}(\hat{Y}, Y, X) &= \mathcal{L}\left(\left(\hat{Y}_A, \hat{Y}_B, \hat{Y}_C\right), (Y_A, Y_B, Y_C), X\right) \\ &= \sum_{t \in T} \omega_t \mathcal{L}_t(\hat{Y}_t, Y_t, X) \end{aligned} \tag{3.1}$$

If for example, the tasks A and B were classification tasks, $\mathcal{L}_{A,B}$ would be chosen as some kind of accuracy loss function, such as the categorical cross-entropy. For a unsupervised task C, which might be supposed to learn the function of an autoencoder, the component loss function would be a distance norm between the input image X and the estimated image $\hat{Y}_C$. An autoencoder is used to reduce the dimensionality of the inputs by passing it through a set of (convolutional) layers, so that it can extract the most important pieces of information. Training is done by adding an expanding part that reconstructs an estimation of the input from this information.

Having thus defined the loss as a linear combination of the component losses, we obtain a set of hyperparameters, i.e. ω_t with $t \in T$, that weight the influence of the component loss functions, which may restrict the influence of a task. Hence, if only one tasks output matters for the application, the task can be prioritised by attributing a higher weight to it. The architecture defined thus is illustrated in a simplistic fashion in Figure 3.1.

Because all experiments in this thesis were carried out using classification tasks, the component loss function for each task was the *categorical cross-entropy*. This function is the dominant loss function [Bon18] in supervised multi-class classification, where each

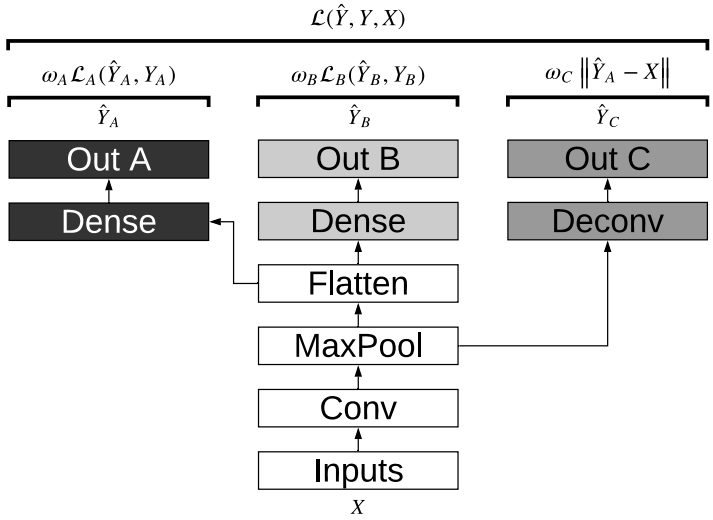


Figure 3.1: Exemplary architecture for a tree-task HPS convolutional network, as described by Equation (3.1). This architecture is grossly simplified in comparison to currently used methods for these types of tasks and would certainly not perform well. Task-specific layers are in shades of gray, while shared layers are white.

sample is assigned a single label from a set with $\lambda \geq 2$ elements. In those scenarios, the labelling is usually done using a *one-hot* encoding scheme, i.e. the label is given as a unity vector $\vec{v} \in \{0, 1\}^{\lambda}$ with a single component equal to one. The network is designed so that it gives scores to each class. The highest-scoring output will then be taken as the prediction of the network. To calculate the loss, we apply [Bon18]:

$$CE(\hat{Y}, Y) = -\sum_{i=1}^{n} y_i * \log \hat{y}_i \tag{3.2}$$

This function is such a popular choice for classification because it makes the network learn a distribution as similar as possible to the input data by minimising the Kullback-Leibler-divergence (KL-divergence) of these distributions [Bon18], which measures the dissimilarity between these distributions. It can be shown that this measure is equal to the sum of the entropy of the data-generating distribution and the cross-entropy. Hence, minimising the cross-entropy also minimises the KL-divergence. A small KL-divergence means that the distribution learned by the model is very similar to the distribution of the data [Bon18], yielding low error.

The implementation of HPS for this thesis follows the architecture described in [Rud17], which stems from the ideas of [Car97]. By using functionalities native to the Python Deep Learning Framework *Keras*, training such a network was possible with very little additional effort. More information on the choice of framework and the advantages conferred is given in Section 3.3 on page 34.

3.1.3 Soft Parameter Sharing

The method of Soft Parameter Sharing implemented for this work follows the implementation in [YH16]. While they share code on GitHub, they only distribute TensorFlow code, which while compatible with the other code written for this thesis (which also runs on TensorFlow), had to be adapted to be properly used.

The basic idea of this method is that every task is processed by a task-specific set of layers. These component models follow the same or at least a very similar architecture. Then, the parameters of every set of similar layers is put under a norm, which is minimised to force the parameters to be similar. For this, the collection of parameters is stacked to a single tensor, for which the Tensor Trace Norm (TTN) is calculated. This norm should restrict the rank of the parameters matrix so that simple solutions will be preferred over more complex ones.

The TTN is an extension of the mathematical concept of the trace of two-dimensional matrices to higher-dimensional tensors. The trace is the sum along the main diagonal of a matrix and equal to the sum of the eigenvalues of this matrix. For higher order tensors, the trace norm is equivalent to the sum of the singular values, which in turn are a generalisation of the eigendecomposition of two-dimensional matrices. To calculate this norm, the tensors that represent the parameters of the layers put under this norm have to be converted to a two-dimensional representation. This step is also referred to as *matriciation* or *flattening*. This step can be done in multiple ways, of which Tucker's approach [Tuc66] was chosen as the only one evaluated in this work. This unrolling decomposes the tensor into a small core tensor and a set of matrices, for which the trace norm can be calculated. For this approach, the parameter tensors should be unrolled to a matrix where the rows index the tasks.

To calculate the TTN for a m-way tensor with the m dimensions $D_1, D_2, \ldots, D_m$, where the last dimension D_m is equal to the number of tasks T, we need to calculate the m mode-i tensor flattenings using Tucker's method. The mode-i flattening of a tensor $\mathcal{W}$ is designated by $\mathcal{W}_{(i)} \in \mathbb{R}^{D_1 D_2 \ldots D_{m-1} \times T}$; while the trace norm of a matrix $\mathbf{W}$ is designated by $\|\mathbf{W}\|_\star$.

In [YH16], they also feature a weight parameter γ_i for every mode-i flattening that controls the trade-off between the cross-entropy classification loss and the trace loss terms. Because they set all γ_i to the same value of 0.01 in their experiments, we too simplified the formula to a form with only a single weight γ:

$$\|\mathcal{W}\|_{Tucker} = \sum_{i=1}^{m} \gamma \|\mathcal{W}_{(i)}\|_\star \tag{3.3}$$

To use the TTN, it is added to the categorical cross-entropy loss function that is used to penalise the network for wrongly-classified samples. This is illustrated in Listing A.1 in Appendix A.4 and Equation 3.4. Because Keras accepts custom loss functions, as long as they accept tensor inputs for the truth and output values from the network and return a scalar tensor, the function returned by `loss_trace_norm` can be used without

any modifications. The gradient for this loss, which is required for optimisation, can be automatically determined by Keras' backend, TensorFlow.

While [YH16] define two more variants of this function, based on other flattenings of the parameters, they did not report significant advantages by using either approach. Thus, only the default choice using Tucker's method was evaluated in this work.

The overall loss function of a network with tasks T and a base classification loss function $\mathcal{L}_C$:

$$\mathcal{L}_{SPS}(\hat{Y}, Y, \mathcal{W}, X) = \omega_{TTN}\|\mathcal{W}\|_{Tucker} + \sum_{t \in T} \omega_t \mathcal{L}_C(\hat{Y}, Y, X) \tag{3.4}$$

Because the Tensor Trace Norm is non-differentiable, the sub-gradient [Van19] is used for gradient descent. It can be derived as $\frac{\partial \|\mathbf{X}\|_\star}{\partial \mathbf{X}} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1/2} = \mathbf{U}\mathbf{V}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are calculated using Singular Value Decomposition [VR17; YH16].

3.1.4 Cross-Stitch Networks

This method was introduced in [Mis+16] and uses a very intuitive approach to combine information from multiple tasks. The fundamental idea was already explored in subsection 2.2.3. As with SPS, every task is processed using an own set of layers. Between some layers, special operators with scalar parameter for every pair-wise combination of tasks weight the amount of information to include from the second task in the further processing of the first task. This is illustrated in Figure 3.2.

There is no code publicly available for [Mis+16] and no suitable reference implementation could be found, so that the method had to be implemented from scratch using TensorFlow functions. Because the implementation should be able to slot neatly into the existing network, it was implemented as a subclass of `Layer`. Each output of the cross-stitch unit is a different linear combination of the inputs, which made it possible to use a number of scalar multiplications, which were then summed up, for each task

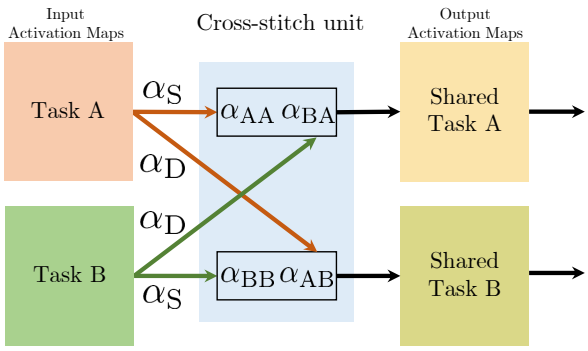


Figure 3.2: Operation of an cross-stitch unit for a two-task network, from [Mis+16].

combination. Because TensorFlow uses a computation graph, it was able to calculate an optimal strategy to compute the required operations and did not need to run through the two `for` loops present in the layer definition for every call of the function. Further information on the implementation is provided in Appendix A.3.

3.2 Data used

For this thesis, the above concepts were explored using two different datasets. In the following section, these datasets are first introduced, and the required preprocessing discussed. Furthermore, a literature search has been carried out to find a suitable baseline for comparison.

3.2.1 Datasets Used

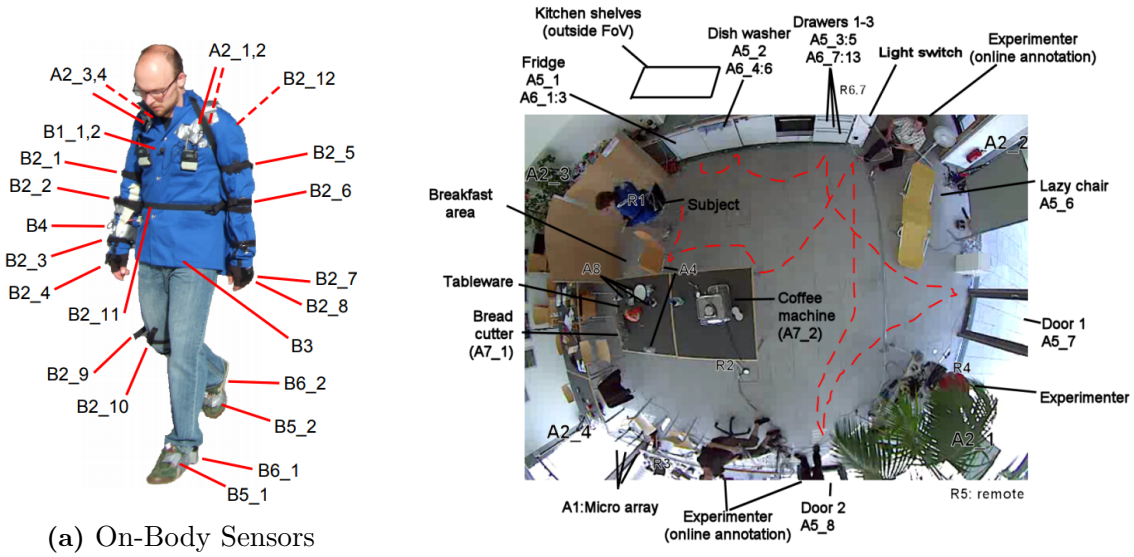
OPPORTUNITY

OPPORTUNITY [Rog+10a], which was first introduced in 2010 by researchers at the ETH Zürich and other institutions, is a dataset suitable for sensor-based HAR. The name stems from the fact that the collection was done to facilitate *opportunistic* HAR. This means that a classifier for a certain activity should not be overly reliant on specific sensors or sensor placements, but should opportunistically make use of what is available to be resistant to (intermittent) losses of modalities. Thus, the datasets consists of a collection of labelled runs where a subject performs specific activity in a highly sensor-rich environment [Rog+10a; Rog+10b]. The most important reference of what is featured in the dataset is given in [Rog+10a], whence the following description was compiled.

The environment simulates a single-room apartment with a bed, a kitchenette, and a table. To allow for using opportunistic classification, a large number of networked sensors were used, so that opportunistic scenarios can be obtained by using only (possibly random) subsets of the available streams. The laboratory setup and sensor placement is illustrated in Figure 3.3. The sensors used fall into two big classes:

Body-Worn The subject was instrumented with acceleration sensors, inertial measurement units (which also give position indications), relative positioning systems (giving the distance of the hand to the body), as well as microphones. In total, 133 sensor channels were recorded on the body.

Environmental Also, the environment itself was instrumented with accelerometers, reed switches (sensing the opening and closing of doors and drawers), as well as cameras and ultra-wide-band localization systems. In total, 109 such measurements are available.



(a) On-Body Sensors

(b) Laboratory Setup

Figure 3.3: On-body sensor placement (a) and laboratory setup (b) for OPPORTUNITY, from [Rog+10a]

Consequently, the amount of data in each run of the subjects through the scripted routines is very high. The data from the different sensor channels were synchronized in post-processing and subsampled to approximately 30 Hz. Afterwards, the data was labelled manually in five (or rather seven) different tracks:

Locomotion Whether the subject is *standing*, *walking*, *sitting* or *lying*

HL_Activity A high-level activity description, one of *Relaxing*, *Coffee time*, *Early Morning*, *Cleanup*, *Sandwich Time*

LL_Left_Arm & LL_Right_Arm What the respective arm is doing, e.g. *open*, *sip* or *move*

LL_Left_Arm_Object & LL_Right_Arm_Object Which object the respective arm interacted with, e.g. *Door1*, *Cup* or *Fridge*

ML_Both_Arms A composite of the previous two channels, e.g. *Open Door 1* or *Drink from Cup*

Most publications report results for the `ML_Both_Arms` channel, which is indeed recommended by the OPPORTUNITY experimenters [Rog+12].

One important feature of the dataset is that every label channel may also assume a NULL value, if none of the possible choices apply. Similarly, not all possible combinations are present in the *ML_Both_Hands* channel, as the mostly right-handed subjects were encouraged to interact with their environment in a natural way and thus mainly used their right hand. Moreover, since capturing such a large number of sensors simultaneously is very challenging, some data points are missing. This packet loss can stem from connection issues from occlusion, empty batteries or a crowded spectrum due to the high number of sensors, most operating in the 2.4 GHz and 5 GHz bands. Still,

the authors estimate that only about 2.5 % of packets were lost, while most data losses were very short.

The openly accessible dataset consists of four sessions, each with six recorded runs. Of these six runs, the first five were *ADL runs*, where the subject performed a scripted sequence of *Activities of Daily Living*. The first run for each subject was accompanied by an instructor. The sixth run was a *Drill run*, where the subject performed 20 repetitions of a shorter sequence. While data from twelve subjects was captured, only four subjects are available from the University of California Machine Learning Repository[1], which is likely to the substantial amount of work involved with labelling all captured data. Still, since a ADL run took 15 to 20 min and a Drill run took about 20 to 35 min, there are at least 155 min of recordings at 30 Hz available.

DEAP

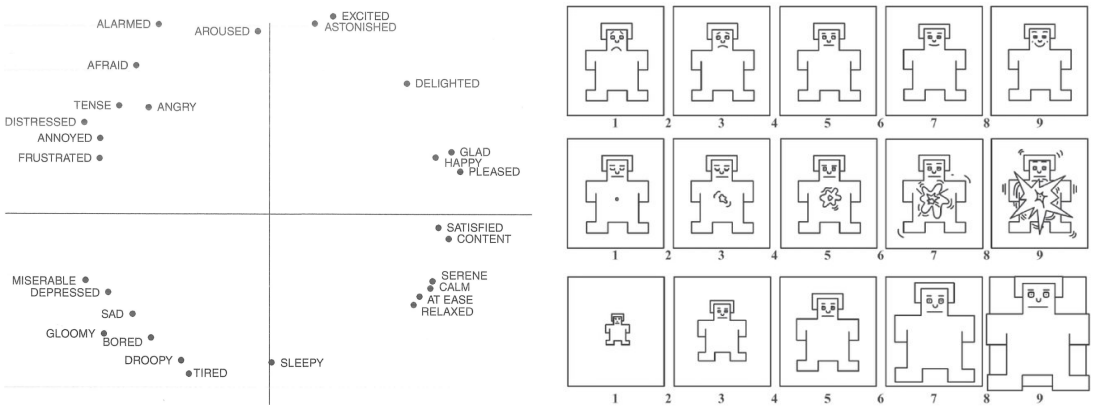
DEAP was recorded in a project by researchers from the University of Twente, University of Geneva, and of the Queen Mary University of London, to allow for the analysis of emotions using physiological signals [Koe+11]. During the acquisition, subjects were presented with 40 one-minute long music video clips each, and asked to rate the emotion these clips elicited in them. This was done while the subjects' brain waves were recorded using 32-channel electroencephalography (EEG). Furthermore, electrooculography (EOG) and electromyography (EMG) were used to capture the movement of facial muscles and the eyeballs, while the conductivity of the subject's skin, their respiration and their skin temperature were also recorded using further sensors. In total, 40 sensor channels are available, of which 32 stem from the EEG.

While watching these clips, the subjects gave ratings to indicate what emotions the video introduced. Because the concept of *emotions* is a very active research field in psychology, this was not a trivial task. In general, two competing meta-theories dominate in psychological research [SK18].

Some theories believe that there are some *basic emotions*, which mix to form composite emotions. These may be, for example, *happiness, sadness, anger, fear, disgust* and *surprise*, which were proposed in [Ekm+87]. Other sets of basic emotions were proposed, but this set is the best-known [SK18].

The other set of theories decompose emotions into a number of axes, which do not correspond to basic emotions, but characterise some qualities of the emotions themselves instead. For example, according to the *core affect* theory in [Rus03], emotions can be arranged in a two-dimensional space with the axes *arousal* (the level of excitement associated to the emotion) and *valence* (also known as *pleasantness*). Thus, „using this model, we can describe excitement as a combination of pleasure and high arousal, contentment as a combination of pleasure and low arousal, and so forth“ [SK18]. This 2D space is also illustrated in Figure 3.4(a). A commonly-used third dimension is that of

[1] at https://archive.ics.uci.edu/ml/datasets/opportunity+activity+recognition



(a) Core Affect model

(b) SAM Mannequins

Figure 3.4: (a): The *core affect* model of emotions, with the two axes *valence* on the x-axis and *arousal* on the y-axis; from [Rus03], via [SK18].
(b): Self-Assessment Mannequins [BL94] as used in DEAP for rating the current emotions on three axes. From top to bottom, the scales rate valence, arousal and dominance.

dominance, being notably used in [MR74], which rates whether the emotions make the subject feel in control or submissive. For example, *Angry* and *Afraid* are close to each other in the 2D space in the low arousal, low-valence quadrant, but the first is dominant, while the second is submissive [SK18]. There are different theories in this area that feature different axes or restrict the space spanned by these dimensions. However, the two-axes model with *valence* and *arousal* is one of the most common. In addition to the term *core affect* model [Rus03], the term *circumplex model* is also used in emotion classification publications. It was introduced in [Rus80] and features the important restriction that all emotions lie within a unity circle around the coordinate origin. Thus, if the point $(0, 0)$ in the space spanned by these two axes is an emotion with neutral valence v and arousal a, all emotions will have satisfy $v^2 + a^2 \leq 1$. This restriction was not enforced during the data acquisition for DEAP, however.

Furthermore, a third school of thought combines ideas from the aforementioned approaches, which is known as the *component process model* [Sch+84]. Alltogether, these three meta-theories exist simultaneously and in general, psychological studies use one of these options as the basis of their work; while it can not be ruled out that each of these theories is „right in a way and we just have to figure out how they all fit together“ [SK18]. The DEAP team has decided to measure emotions using a *core affect* model, where subjects were asked to assign a score in the range of $[1, 9]$ to the valence, arousal and dominance scores. This rating was done using the SAM mannequins proposed in [BL94] and illustrated in Figure 3.4(b). Subjects could click anywhere in the range and were not restricted to integer values. Furthermore, the subject was also asked how familiar they are with the video (integer between 1 and 5), and how much they liked the clip (float between 1 and 9, indicated using upward- and downward-facing thumb icons).

Because the ratings were done by the subjects themselves, who might not have been familiar with the circumplex model, the ratings for similar emotions will inevitably

vary between subjects and might also be different for the 40 different videos. Thus, trying to predict the exact values of the labels, e.g. using a regression model, usually does not make much sense in this field. Instead, because the emotions classified by Russell [Rus03] all roughly fall into four distinct quadrants of high and low valence and arousal, the problems are generally restated as two n-class classification problems. While $n = 2$ is the most common choice, $n = 3$ is also an option, which better captures emotions such as *sleepy* and *aroused*, which in Figure 3.4(a) fall right on the y-axis with a neutral value for *valence*.

In the literature, emotion classification using the two-dimensional core affect model is usually carried out in a single-task learning fashion, where two networks are trained for each task. Because it is reasonable to assume that the arousal and valence dimension of emotions are correlated [JSP19; ZZ18], it is a natural idea to evaluate the possible benefits of applying MTL ideas to this domain.

3.2.2 Preprocessing

In general, every dataset that serves as the input to a Machine Learning method has to be prepared to allow effective processing. Aside from creating minibatches, which serve to stabilize the training process by asking the network to classify e.g. 50 examples at once and taking the average of the gradients for the entire batch, time-series data generally has to be segmented.

For the purposes of this thesis, a time series is defined as a set of one-dimensional signals that are synchronized to a common time base. By segmenting the data stream, we obtain shorter pieces from the base signal, which is advantageous, if not essential, due to memory constraints. This is because the longer each sample is, the more parameters are required for dense layers processing these large tensors, thus increasing the model parameters considerably. Furthermore, even convolutional layers, which use much less parameters, also produce larger outputs for larger inputs. Because of the very widely-used practice of using graphics cards in the training machines for training acceleration, large network sizes can drastically affect training time. For most commercially available graphics cards, only about 6 to 8 GiB of graphics card memory is available. While modern ML frameworks can cope with networks bigger than the amount of graphics memory available by performing iterative calculations, loading only the currently-required layers at a time into memory, this incurs a substantial training time penalty. Also, the batch data has to be transferred from the system RAM to the graphics RAM before it is needed for optimal performance (which is known as pre-fetching), which ties up the system busses longer for larger batch sizes. Using graphics processors for network training is possible due to APIs provided by the graphics card manufacturer allowing for general-purpose computation on the cards due to their excellent performance when calculating large-scale simultaneous matrix-matrix and matrix-vector multiplication.

By splitting every experiment in the dataset into a number of data frames, we not only reduce the size of the batch, but may also be able to reduce the number of parameters

required to process this data. Furthermore, the splitting also serves as a generally very effective data augmentation strategy: for DEAP, 32 subjects are available, each performing 40 experiments for a total of 1280. Because we need to set some experiments aside for validation (a common ratio is 20 % validation samples), we would only train using 1024 of these samples. This is not a large number and will almost certainly result in significant overfitting, where the network becomes too dependent on the training dataset and performs worse over time on the validation dataset. With such a small number of training samples, a suitably large network might even reach almost perfect accurracy by just memorising the input data. For reference, MNIST [LeC+98], a very popular baseline dataset for classification, has a training-testing split of 60000 training and 10000 testing samples; while the ImageNet dataset [Den+09] has about 14 million images in total.

If the DEAP dataset were to be segmented into three-second non-overlapping time windows, the number of samples available would increase by a factor of 20, since the music videos are 60 s long, which gives a equally much larger number of training batches. While the partitioning of DEAP is mostly required for data augmentation, OPPORTUNITY experiments are much longer at multiple minutes, though sampled only at 32 Hz. Segmentation is thus absolutely essential for feasible training on OPPORTUNITY, because of memory and time constraints.

Still, segmenting is not a trivial task. The length of the time windows has to be appropriately chosen and a decision whether to use overlapping data frames has to be made. Furthermore, if the labelling of the experiments changes over time, a suitable strategy for label fusion has to be developed.. These strategies and values are usually chosen using a rule of thumb or during preliminary experiments.

In the following, the preprocessing steps specific to each dataset are discussed.

OPPORTUNITY

The version of OPPORTUNITY used is distributed in 24 data files, which are plain-text files consisting of a large number of rows, each corresponding to a single point in time, with 250 whitespace-separated columns of integer numbers. Also included are two text files which serve as a key to the channels as well as the labels.

Every sample has a timestamp in the first column. The following 133 channels correspond to the on-body measurements, which are followed by the 109 environmental channels. In the last seven channels, the labels are encoded.

The overall processing consisted of two steps, where almost all operations were carried out using `numpy` functionality:

Clean-Up Because of the aforementioned dropouts of some sensors, some large data frames of measurements are missing (given as `NaN`). Some channels having a large loss of data were removed altogether in this step, so that only 107 labels are available for the network. This selection was done by [Li+18] and carried over

in this thesis. `NaN` values still present were replaced with the previous values. Additionally, labels and data were written to different files for each experiment.

Windowing Afterwards, the data was segmented into two-second long data frames (which are 64 samples) by applying a sliding-window approach with a stride of $\sigma = 2$. The labels were selected using majority voting, where the label was selected that occurred the most. Because the `numpy` function `argmax` was used for that purpose, the label with the smallest identifier was chosen in case of ties. This again uses a version of the preprocessing scripts provided in [Li+18] which was modified so that all labels are written to the files, as opposed to only the `ML_Both_Hands` channel used there.

The preprocessing steps employed for this thesis follow those described in [Li+18][2] While most of the preprocessing done by Li et al. was carried out in the same fashion for this thesis, there has been one very significant difference with these scripts. While examining the signal from a subset of sensors using two-dimensional plots, a number of large spikes could be observed. These are illustrated in Figure 3.5. Even from this limited selection of channels it is evident that the spikes are not correlated in time, which holds for larger selections of sensors as well. If those spikes corresponded to actual events (which would likely be something like the subject falling), there should be similarly drastic movements in the other axes of the respective accelerometer. These spikes drastically increase the dynamic range of the data, which may adversely affect training if the network becomes somewhat reliant on these spikes.

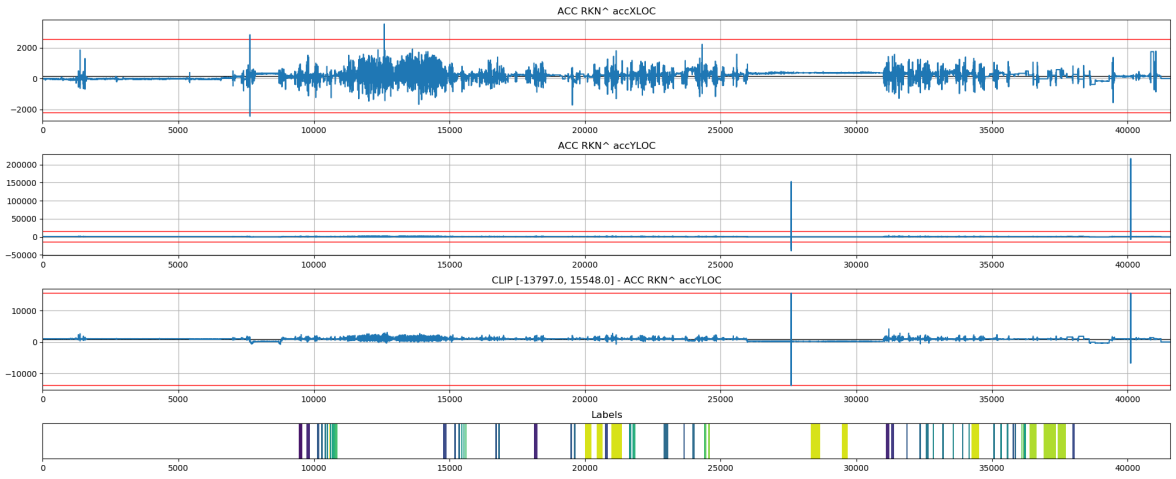


Figure 3.5: Spikes observed in some channels of the OPPORTUNITY dataset. Two channels are shown with the second one exhibiting major spikes. The red lines delineate the 10σ region around the mean of the channel (indicated by a black line). The third row shows the second channel, after the values were clipped to the region delimited by the red lines. Below, the labels (`ML_Both_Hands`) are indicated. Colored bars correspond to stretches of the data that are have non-NULL labels. They show that the spikes occurred during periods of unlabelled activity and indicate that they are noise rather than a specific activity.

[2] using the provided source code at https://www.info.kindai.ac.jp/~shirahama/sensors/

Because of the apparent unconnected nature of the spikes across different channels, it is reasonable to assume that those are artefacts of the data collection algorithms and do not correspond to actual events, especially since they are labelled as `NULL`. They were thus removed by calculating both the mean μ and the standard deviation σ of the signal for each channel independently, so that each channel could be restricted to the range $\mu \pm \tau\sigma$ for a $\tau \in \mathcal{R}$. Because the signals are not normally distributed, an ordinarily rather large value of $\tau = 10$ was selected that did not entirely eliminate the spikes, but merely restricted them to a more conservative range.

While it may be hard to quantify any possible benefit from this procedure, it was observed during training that the performance of a network on clipped data did not decrease compared to the unclipped data. Interestingly, to the best of our knowledge, this quality of the dataset has not yet been observed in the community, even though OPPORTUNITY has become one of the main benchmark datasets for HAR methods. While the ultimate consequences are hard to estimate, it is very curious that no one has written about this yet.

DEAP

Because DEAP is already provided in a preprocessed format, the files did not have to be prepared as intensively anymore. Still, some crucial steps were carried out here.

The main problem with using DEAP is that emotions are inherently difficult to measure. For example, not every subject may feel the same emotion after watching the same clip; if they don't like the artist, they may feel differently compared to when they listen to a favourite song. Even though the subjects will most certainly have tried to reduce the influence of their personal preference, the results will likely have been entirely different if they had never listened to music before.

An even more serious issue is the reliance of the dataset on electroencephalography, which measures the activation of neurons near the electrode applied to the subject's skull. As pointed out in [SK18, p. 29], non-invasive EEG is not an ideal approach for measuring the activity associated with induced emotions, because „much of the neural activity in emotions occurs much deeper in the brain [than on the scalp]". Short of implanting electrodes in deeper regions of the subjects' brains, such studies will always only be able to capture small activations. Also, brain waves are not that well understood and hard to interpret, while it also has been understood that the patterns present in the data are subject-specific and have a low information content.

Overall, this is problematic because it makes the EEG data very subject-specific. There are thus three distinct strategies in assigning data to training and testing subsets:

Subject-Specific The network is trained on data from all subjects, but some experiments, or just some samples, are withheld from the training datasets to form

a testing dataset, while the rest is used for training. It can thus learn patterns for every subject. By training the network on different training-validation-splits, the performance should not vary drastically, because the network learns specific patterns for all subjects available.

Subject-Independent On the other hand, the data from some subjects may be withheld entirely, so that the network has to classify patterns it may never have seen. This architecture is more suited to finding universal features working for any subject, but the performance is highly dependent on which subjects it was trained on. This is a much harder problem than the subject-specific problem, but may have more real-world applications if it worked for any person.

Single-Subject Strictly speaking, this is a variation on the subject-specific scenario, where data for only a single subject is available. In this case, withholding data from an entire experiment to form the testing dataset results in the network possibly never having seen an elicited emotion. Such a scenario would require a very large amount of data to avoid the model memorising the training data, which would lead to overfitting.

Of these three strategies, the single-subject case was not considered for examination because of the relatively small number of experiments per subject available in DEAP. Even though 40 experiments were carried out per subject, significant data augmentation strategies would be required to generate the needed amount. This is because patterns in the brain waves have rather low frequencies, so that there is a natural lower bound of around 1 s for splitting each experiment into data frames [Can+15]. Since each experiments lasts 60 seconds after removing the pre-trial baseline (more on that below), by splitting the data into even 0.5 s long data frames, we would only get $40 * 120 = 4800$ chunks. Setting 20 % (960) samples aside for validation, we only have 3840 data frames for training.

While the subject-independent case is also very relevant, it is also much harder for a network to learn. This might make it difficult to find a suitable baseline and to detect improvements using MTL. Hence, the subject-dependent case was prioritised over the subject-independent case, which is also more often used in the literature [LS14].

Preliminary experiments were carried out both for subject-dependent and independent classifiers in order to find a suitable baseline and to select a suitable preprocessing strategy. At first, the sliding window size was selected as three seconds, after removing the three-second pre-trial baseline. This gave 20 samples per experiment for a total number of 25600 samples. 20 % (5120) of samples were reserved for testing in both scenarios, so that training was done on 20480 samples. For each experiments, the first three seconds, i.e. 384 samples were removed, because they are reported to be a pre-trial baseline without a stimulus. They are thus not immediately relevant for classification.

While this window size was shown to be in the optimal range for Support Vector Machine (SVM)-based classification on DEAP [Can+15], it became evident this window size was selected too large, resulting in significant overfitting. This was visible in

the way the graph of the loss during training decreased steadily, while the loss curve for validation, which was carried out after every epoch, remained stable on average. In some cases, it even increased with the number of iterations calculated. As an intermediate step, overlapping three-second windows were segmented with an overlap of 1.5 s, giving 39 frames per experiment. Because this did not increase performance significantly, a much more aggressive strategy with a window length of 1 s with an overlap of 0.5 s was evaluated next. In combination with a testing split of 33 %, this proved to be the most effective choice of parameters evaluated and was used for the training of both the baseline and all MTL models.

Apart from segmentation, supervised time series pre-processing methods also need to assign labels to each sample. For DEAP, there is only one label attached to each experiment, which is in contrast to OPPORTUNITY, where labels change over time. Hence, every data frame was assigned the same combination of labels that was assigned to the recording it was extracted from. However, because emotion self-assessment is also highly subject-specific, so that the same emotion may be rated differently in the three-dimensional *valence-arousal-dominance* space by different subjects, the labels were transformed to a classification problem. In the literature, the most common choice for this is to use a two-class problem for every dimension [LS14; Yin+17]. This was adopted for this thesis, where every label on the arousal, valence and dominance axes (which were in the range $[1, 9]$) greater than or equal to 4.5 was mapped to a *high* value. For liking (in the range $[1, 5]$), this boundary was selected as 2.5. Hence, every label channel was transformed to a binary classification problem.

3.2.3 The Search for a Baseline

The aim of this thesis is not necessarily to institute a model with state-of-the-art performance, but merely to demonstrate that Multi-Task Learning, when applied to different problems, can be an effective means of improving the results of an existing model. To this end, a baseline for each dataset that could be feasibly adapted to any of the MTL techniques evaluated had to be constructed. As with most other methods, these baselines were selected using a set of preliminary experiments and use ideas from relevant literature. Simple methods that perform close to the state-of-the-art were preferred over more complex ones, even though those often perform better. This was done both to ensure that training remains possible on the available computing hardware and to keep the time required for training relatively short. Because MTL methods add overhead in computation time and memory requirements (especially SPS methods), this was required in order to be able to test a variety of different architectures.

From the assumptions underlying MTL methods, namely that MTL imparts an inductive bias and leads to more plausible latent representation, it is reasonable to assume that MTL applied to more complex architectures will have similar effects to those observed on the simpler architectures.

OPPORTUNITY

Human Activity Recognition (HAR) is a popular field in machine learning and ubiquitous computing, because the sensors required have become very cheap and both easy to use and to setup in the last decades. Consequently, there is a multitude of written material available on the classification of these types of datasets. OPPORTUNITY has specifically become a benchmark dataset for HAR purposes. The most influential publication for this work was [Li+18], which compared a number of feature learning approaches for HAR on OPPORTUNITY. The preprocessing pipeline that was ultimately employed was also modelled after that publication and the publicly available source code served as a base for the exploration of MTL.

Besides DNN approaches, the authors also compared Code-Book approaches [SG17] and hand-crafted features, which were not considered for this work, because hand-crafted features performed consistently worse than deep learning. While Code-Book also evaluated positively, it is not trivial to adapt this approach to MTL, especially because it is very sensitive to the choice of the hyperparameters.

The types of Deep Learning models evaluated in [Li+18] spanned all those described in section 2.1, while also combining CNNs with LSTM networks in hybrid fashions. In addition to these three concepts, Autoencoders were also used.

Model performance was considered using three metrics that were calculated using a validation split of approximately 1/4. Those metrics were the accuracy along with the weighted and average F_1 scores. In the following, Y and $\hat{Y}$ designate the set of ground-truth and estimated labels respectively. Both sets contain N elements y_i and $\hat{y}_i$, which stem from $\mathcal{R}^l$, where l is the number of possible labels for the current channel. The ground-truth values are assumed to be in a one-hot form, i.e. exactly one element in each y_i will be equal to 1, while other elements are 0. The accuracy is then defined as:

$$\epsilon(a, b) = \begin{cases} 1 & a = b \\ 0 & a \neq b \end{cases} \tag{3.5}$$

$$accuracy(Y, \hat{Y}) = \sum_{i=1}^{N} \frac{\epsilon(\arg\max \hat{y}_i, \arg\max y_i)}{N} \tag{3.6}$$

The F_1 score is a special case of the F_β score, which is a weighted mean of precision and recall. These metrics are defined in terms of the different types of errors a network makes. For a binary classification problem with one *positive* and one *negative* classify, a false positive (FP) is an error where the network falsely classifies the sample as belonging to the *positive* class, while it actually belongs to the *negative* class. In the same way, false negatives (FN) can be defined. The goal of a binary classifier is to correctly label each positive example with a positive label (true positive, TP) and each negative example with a negative label (true negative, TN). The accuracy of a classifier can be summarised with the two metrics $recall = \frac{TP}{TP+FN}$, which calculates

the ratio of correctly classified positive examples versus all positive examples, and $precision = \frac{TP}{TP+FP}$, which is the ratio of correctly predicted positive examples against all examples predicted as positive. For a diagnostic test in medicine, a precision of 1 signifies that the test does not classify healthy persons as ill, while a recall of 1 means that it does not miss patients suffering from the condition. While ideally, both of these measures should be close to 1, it is hard to satisfy both. Hence, the F_β measure combines those two measures into one. By changing β, the tradeoff between the two metrics can be controlled based on the metric that is more important for the particular application:

$$F_\beta(Y, \hat{Y}) = (1 + \beta^2) \cdot \frac{precision(Y, \hat{Y}) \cdot recall(Y, \hat{Y})}{\beta^2 \cdot precision(Y, \hat{Y}) + recall(Y, \hat{Y})} \tag{3.7}$$

$$= \frac{(1 + \beta^2) \cdot TP}{(1 + \beta^2) \cdot TP + \beta^2 \cdot FN + FP} \tag{3.8}$$

For a multi-class classification problem, this can be adapted by defining FP, FN, TP and TN as one class versus all others. Hence, for a l-class classification problem, l values for these metrics will be calculated. For the class 1, this means that the true positives correspond to all samples that were actually an instance of class 1 and classified as such, while the false negatives were samples that are also instances of class 1, but classified as any other class. There are now three different ways of calculating a single F_{beta} value from these l sets of values:

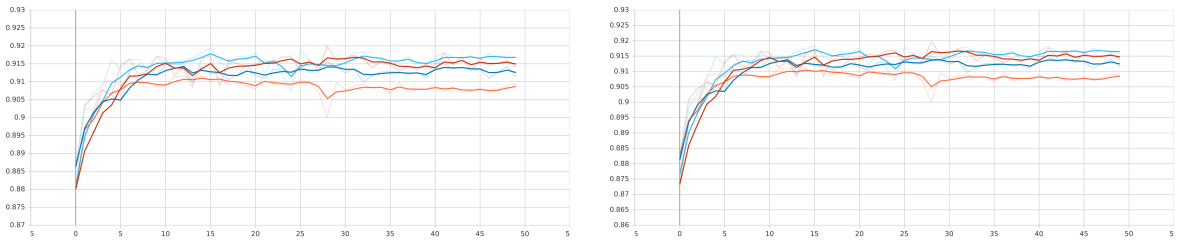
Macro/Average Calculate the F_β independently for each label and average over the number of possible labels

Weighted Calculate the F_β independently for each label, but weight each result by the number of examples in each class while averaging, which takes class imbalance into account

Micro Aggregate the number of true positives and the number of predicted and true positives over all classes and calculate a single F_β

When using Keras, arbitrary metrics can be calculated independently to the loss function, which do not contribute to the training process. This setup of training using the categorical cross-entropy loss and validation using the average F_1 metric is a very common practice for classification problems and was adopted in the evaluation on OPPORTUNITY for this work. With $\beta = 1$, the F_1 measure assigns equal importance to precision and recall.

Of the networks considered for use as a baseline (MLP, CNN, LSTM and Hybrid-CNN-LSTM), the Hybrid model was reported to perform the best in all three of the metrics introduced above in the experiments performed in [Li+18]. This result could be confirmed experimentally using the provided scripts, which were used to train both TF1 and TF2 implementations (see below). However, by adding dropout units to the CNN architecture, its performance could be improved dramatically by reducing overfitting.



(a) Validation Accuracy **(b)** Validation F_1

Figure 3.6: Training graphs for the evaluated STL baselines. The performance of CNN without dropout is orange, the pure LSTM without CNN is dark blue, the hybrid CNN with LSTM is red and the CNN with dropout is light blue. The figures where generated by TensorBoard and show the metrics reported by the Keras-TF1 implementation.

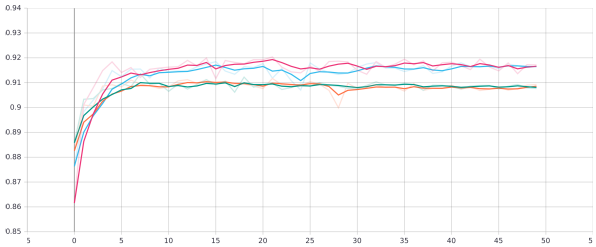


Figure 3.7: Investigation of spike removal by training identical models on unclipped and clipped data. Shown are the training curves (F_1) for the CNN with dropout in light blue without clipping and pink with clipping, while the CNN without dropout is shown in orange and green respectively.

This was also advantageous because the training of LSTM networks takes much longer than a CNN. Hence, the CNN with dropout was chosen as the baseline to not only allow for rapid iteration, but also to keep the number of parameters manageable. It is assumed that an improvement using a CNN architecture would translate to a similar improvement when using a different base. Training graphs are shown in Figure 3.6, while the architecture of the CNN with dropout that was chosen as a baseline is shown in Figure C.1(a) and Table C.1 in the appendix.

During this step, the effect of spike removal, as described in section 3.2.2, was also investigated. This was done by comparing the CNN model with and without dropout to a model of the same architecture that was trained on data where spikes outside the $\mu \pm 10\sigma$ tunnel were clipped to the bound of that range. This is shown in Figure 3.7. It is clearly visible that the overall performance of neither network was affected, but it can be argued that the despiked data allowed for a smoother approach to the limit, especially for the model without dropout, already being more susceptible to overfitting.

DEAP

The architecture that was chosen for DEAP is ultimately very similar to the one selected for OPPORTUNITY. In contrast to most methods in the literature, it uses the full 40 channels available in each experiment and does not rely on feature extraction

methods applied before training. Instead, they process the data before feeding it into a neural network. Oftentimes, only the EEG data is considered and will be divided into different frequency bands using a suitable processing method. These methods include both the Short-Time and the Fast Fourier Transform [Yin+17; ZZL17], as well as wavelet-based approaches [KSK18], which all can extract frequency information from the time-domain information in the dataset. Furthermore, some approaches eschew both time-domain and frequency-domain-based approaches and use statistical features [Tri+17].

The approach reported in [Tri+17] was examined in-depth for its feasibility before deciding on using the raw data. Using their method, each experiment was transformed to a set of 101 hand-crafted features. For this, each experiment was divided into approximately ten six-second long data frames, for which they calculated nine statistical measures: mean and median, minimum and maximum and the difference of these, standard deviation and variance, along with the skew and kurtosis. By calculating these values both for each of the ten data frames and the entire experiment, they obtained 99 features. To these, they added the ID of the subject and of the experiment. This approach did however yield a significantly smaller number of samples available, and the STL network described in their paper overfitted significantly. This is in contrast to the results reported in the paper, where the authors reported state-of-the-art performance, and the approach was therefore not used for the MTL experiments.

Based on the failure of the aforementioned method, the use of other feature extraction methods was considered. However, we have established empirically that using the unprocessed data directly using CNN networks can also prove effective. In this way, the optimisation process can establish filters that approximate some of these other features, while others may generate entirely different features. To this end, we evaluated a convolutional network without recurrent elements as a possible baseline, which performed much better than the former approach. The architecture of this baseline is also displayed in Figure C.1(b) and Table C.2.

During the selection of the baseline, we observed that the way the data is shuffled has a dramatic influence of the performance of the network. For `numpy`-based input data, this shuffling is mostly done by reading the entire dataset into a number of arrays and shuffling these in their entirety. However, because another approach to providing the input data was pursued, which only read data into memory when required, this was to be avoided during training. To shuffle data, the TensorFlow-based datasets provide a buffer. When the dataset is first used, this shuffle buffer of size b will be filled from the first b elements of the dataset. Whenever a sample is requested, a random element from the buffer will be returned and replaced by the next item in the dataset. The size of this shuffle buffer had great effect on the performance of the network, so that we opted to shuffle the entire dataset during preprocessing (cf. section A.2). This ensured a high-quality shuffle, identical to traditional pipelines.

As with OPPORTUNITY, we initially trained our models using the *categorical cross-entropy* loss function, but did observe that performance of the network increased when training using sparse labellings, where the data is not one-hot encoded. This required

the use of the respective loss and metric function variants, the *sparse categorical cross-entropy* and the *sparse categorical accuracy* respectively.

3.3 Technical Setup

While there are many different software frameworks for training Artificial Neural Networks, *Keras* [Cho+15] has emerged as a dominant player in the last years. This project defines a API for easy creation of networks and is able to utilise a number of different back-ends for matrix/vector/tensor arithmetic. This is in contrast to some other frameworks that implement the needed arithmetic themselves. Furthermore, it is easy to set up using pre-built binary python packages, as opposed to a distribution in source-code form that has to be compiled manually. The same applies for a rich ecosystem of other data science packages that are available for the Python programming language.

While Keras is able to use different back-ends (TensorFlow [Mar+15], CNTK [SA16], MXNet [Che+15] and Theano [AlR+16]), the dominant [Hal18] choice is TensorFlow. Indeed, because Keras is both a Python package able to run on these four frameworks, and an API specification, the TensorFlow package implements the Keras API in the `tensorflow.keras` package while being mostly compatible to the framework-agnostic implementation. In doing this, TensorFlow is able to offer specific optimizations that would not be possible in the agnostic implementation.

The TensorFlow-specific implementation of the API was ultimately used because of some of this unique functionality. Most Keras models use data that is encoded in multi-dimensional arrays using functionality from the `numpy` package. By passing a large array to the Keras `model.fit` function, along with an integer batch size, the framework batches the data appropriately. This is of course very convenient, because the data is often gathered and processed using Python scripts anyway, where such computations are almost invariably performed with `numpy` functionality. However, this comes at a significant cost: Numpy arrays have to be loaded into memory, because they are linear (1D) arrays that can be indexed with multi-dimensional coordinates at their foundation. Thus, every value of the array has to be loaded into the RAM of the training machine. While training is often performed on higher-spec PCs, the *overhead* incurred by this can impact performance negatively.

Training for this thesis was carried out on the machine whose specifications are summarised in Table 3.1. Originally, while the agnostic implementation `keras==2.2.4` was used in conjunction with `tensorflow-gpu==1.14.0`, the main system memory was exhausted on the machine while training on OPPORTUNITY, because no Swap space was available at first. After adding the Swap, however, the model and data still filled up the entire 32 GiB of combined memory over time and required a lot of swapping, leading to long runtimes and the possibility of system crashes. While this did not happen with every network, only with those that featured a lot of parameters, this had the result of some architectures being effectively untrainable using the available

Hardware	CPU	*AMD FX-8120* 8 cores, 8 threads 3100 MHz (4000 MHz boost)
	RAM	16 GiB DDR3 4 x *Kingston KHX1600C9D3/4* 667 MHz 16 GiB Swap on 7200 min^{-1} HDD
	GPU	*Asus Turbo GeForce GTX1060* 6 GiB GDDR5 1280 CUDA cores
	SSD	*SanDisk SDSSDA24* for OS and data
Software	Operating System	Linux Mint 19.2 *Tina* Linux kernel 4.15.x
	Python environment	Anaconda with Python 3.6.9
	TensorFlow version	`tensorflow-gpu==2.0.0`
	Graphics driver	proprietary Nvidia 410.104
	CUDA version	10.0.130 with CuDNN 7.6.5

Table 3.1: Specifications of the training machine. All version numbers, except for the Linux kernel and TensorFlow, were kept constant during development. TensorFlow 2 was used while still in development, after the release of the first beta version, and was upgraded from the beta version 2.0.0b1 to the stable version 2.0.0 via the Release Candidate 2.0.0rc1. As of writing, no stable version newer than 2.0.0 has been released.

hardware and chosen setup. Hence, the very recent version 2 of TensorFlow, which reproduces the API defined by Keras as an easy-to-use front-end to the low-level tensor arithmetic, was chosen due to its support for the `tf.dataset` format. These datasets greatly reduce the amount of memory required for training. In Appendix A, further considerations with regards to this format and the required processing, are outlined. Furthermore, a list of the files included in the source code distribution[3], and guidance on how to use this code, is provided in Appendix B.

[3]which is freely available at https://github.com/LtSurgekopf/TimeSeriesMTL

Chapter 4

Results

For both datasets, we trained networks for each approach to Multi-Task Learning. The architectures of these networks were derived from the baselines introduced in the previous chapter, against which the MTL approaches were compared. The effectiveness of each method was mainly investigated graphically by plotting the relevant metrics over the number of epochs. In this chapter, the evaluation methods will be introduced prior to the experiments performed.

4.1 Evaluation Procedure

In many papers using Deep Learning, the results are often reported by comparing a single accuracy score against reported scores in the literature. This may not tell the whole story, because the training of the network is a stochastic process, so that one score may not convey all information required for a proper evaluation of the method. For example, by using MTL, a network might reach the same maximum performance as the STL baseline, but might require a significantly smaller number of epochs to reach it. Furthermore, because MTL is thought to be a regulariser which imparts an inductive bias onto the model, MTL approaches might result in a smoother approach to the limit. Both of these characteristics can only be conveyed graphically. We therefore base our evaluation mainly on graphs plotting the evolution of the relevant metrics during the training process. In addition, a comparative evaluation by comparing metrics of the best-performing models for each approach and dataset is also provided in section 4.5.

To generate these graphics, several elements had to come together. Firstly, the metrics, i.e. the (sparse) categorical accuracy, along with the F_1 measure on OPPORTUNITY, had to be calculated after every epoch using a testing dataset. By writing these to a suitable file format, graphics can be generated which compare the current approach with others. The easiest way to generate these training curves is certainly using TensorBoard, which is a web-based tool distributed independently of TensorFlow that is able to display a large array of information about a network. Besides being able to visualise scalar metrics, it can also demonstrate the TensorFlow computation graph, profile the computation to find bottlenecks, visualise embeddings and much more. By providing the Keras `model` with a callback, the relevant data can be written into a log file after every epoch, which includes the metrics for training and testing. TensorBoard can then

be started in a directory with multiple of these log files, so that different approaches can easily be compared visually.

This does however rely on the metrics reported by Keras, which does work very well for accuracy metrics. Because the F_1 metric was not available in Keras by default, however, a custom implementation had to be created. While theoretically, it should have been possible to provide a user-defined function taking estimated and true labels as inputs to calculate the F_1, this did not in practise return plausible results when using TensorFlow 2. This is due to the way Keras averages the metrics over the entire dataset, which could not be altered. For OPPORTUNITY, the metrics were therefore calculated independently of Keras in a format that is incompatible with TensorBoard.

By leveraging callbacks, a method is called after every epoch that calculates a confusion matrix by classifying the entire testing dataset. From this confusion matrix, the precision and recall, and hence the F_β score, can be calculated. To ensure the correctness of the metrics calculated in this fashion, the implementation from the Python machine learning library `sklearn`, which is often used for reporting F_β in publications, was copied. Using this approach, it could be avoided to store the entire testing dataset in memory, while still using every available experiment in this subset. To generate graphics from this data, the statistical programming language R with the package `ggplot` was then used to generate the needed visualisations, which compare different approaches.

Because of the stochastic training process, the point clouds for each method mostly followed clearly visible trend lines, but exhibited a large amount of jitter from these curves. To be able to illustrate the trend, the curves were smoothed using the Locally Estimated Scatterplot Smoothing (LOESS) algorithm [Cle79], which combines linear least squares with nonlinear regression by approximating multiple polynomial functions to subsets of the data. In this way, it can approximate non-linear functions very accurately, if enough data is available.

To further illustrate the performance of each model, the last 10 (OPPORTUNITY) or 25 (DEAP) epochs for each training run are plotted against each other. Ideally, each network should have approached a limit after the previously-defined number of iterations, so that the relevant metric remains mostly constant. Hence, the point clouds for these ten epochs were regressed using a linear model. The comparative tables were manually generated from the same data, using the last 10 epochs of every model for both datasets.

To ensure a consistent presentation, the results for DEAP also have been processed with the same scripts as OPPORTUNITY to generate similar graphics, instead of using TensorBoard directly, which was not possible for OPPORTUNITY.

Because of space constraints, not every visualisation that was generated is printed in this thesis. Further graphs are available at the GitHub page.

4.2 Hard Parameter Sharing

4.2.1 OPPORTUNITY

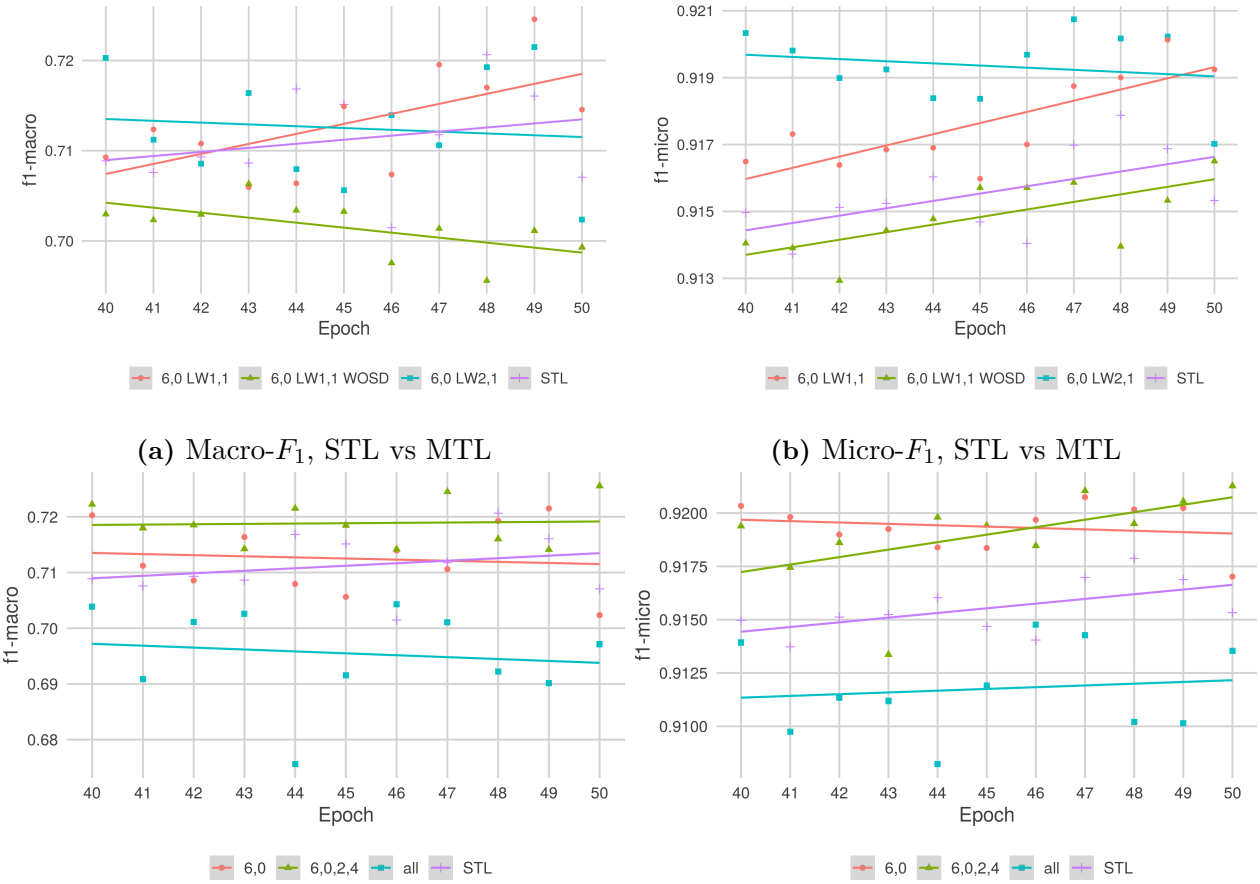
Experiments on HPS were carried out on OPPORTUNITY first. For this dataset, data from seven tasks was available. In the following analyses, these will be referred to using numeric identifiers in the order they are present in the respective data files. Hence, the main task, *ML_Both_Arms* (also abbreviated by `ML_B`) is referenced with the identifier 6.

The first experiments that were carried out used a two-task network, where the classification of the main task is aided by the *LL_Locomotion* channel (abbreviated by `LL_L`, with a numeric identifier of 0). This channel states whether the subject is in motion or stationary and can have 5 different labels, along with the `NULL` label (cf. section 3.2.1). In these experiments, the baseline introduced in section 3.2.3 was compared with the architecture shown in Figure C.2(a) and Table C.3. This architecture is very similar to the baseline, but features two branches after the initial dense blocks. These feature-specific branches have less parameters than the shared layers and serve the purpose of learning features specific to the respective tasks.

For the two-task networks, the influence of two parameters were principally investigated. First, the loss weights ω_t (cf. Equation 3.1) are meta-parameters that assign each task an influence for the overall loss function. Because the `ML_B` task was prioritised for OPPORTUNITY, it was suspected that assigning a higher weight to this task might result in better classification accuracy. Second, the amount of shared layers that is required was explored.

To transform the baseline into a HPS network, the task-specific layers were put onto the existing shared layers without any modification to the underlying network. This MTL network features a very large number of fully-connected units both in the shared and unshared parts, which might have hurt performance. Hence, a model without any shared fully-connected layers (i.e. the layers *dense*, *dense_1* and *dense_2*, as well as the following dropout units, were removed) was also trained and compared with the two-task models. This architecture also featured a dropout layer after the first task-specific fully-connected layers (i.e. *{ML_Both, LL_Locomotion}_0*) with a rate of 0.4.

The results of these experiments are summarised graphically in Figure 4.1(a) and (b). It should be noted that the linear models can not approximate the stochastic nature of the point cloud distribution and only carry limited trend information. While MTL performs much better than STL with respect to the micro-F_1 score, it performs roughly similar to STL with respect to the macro-F_1 score, which is the prevalent metric in the literature. The hypothesis that the shared fully-connected layers hurt performance could however be disproved, since the curve indicating the results of that experiment in green in Figure 4.1(a) and Figure 4.1(b) is at a much lower level than STL and the other MTL models. It can however not be ruled out that using a smaller number of shared fully-connected neurons might actually improve results again.



(a) Macro-F_1, STL vs MTL

(b) Micro-F_1, STL vs MTL

(c) Macro-F_1, task count

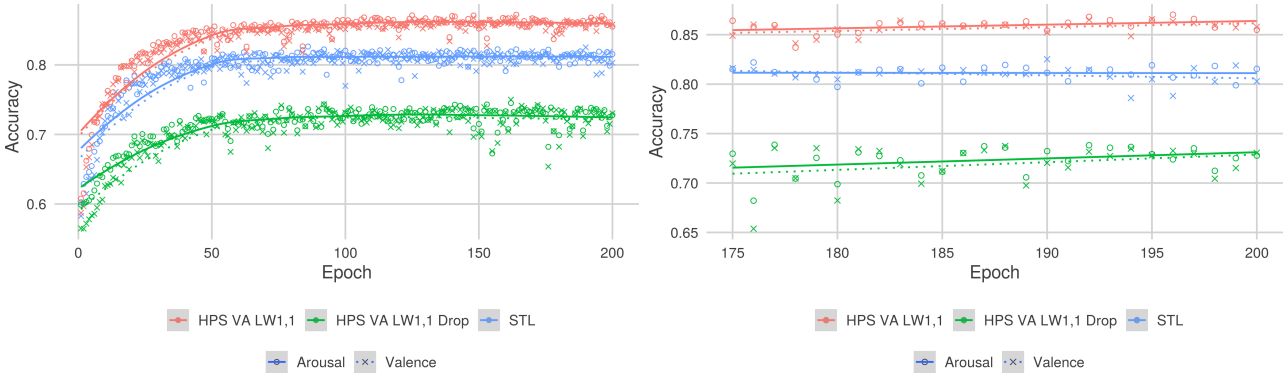
(d) Micro-F_1, task count

Figure 4.1: Results for HPS on OPPORTUNITY using two-task networks. The test performance for the last ten epochs is shown. Abbreviations: integers indicate which tasks were considered (6: `ML_B`, 0: `LL_L`, 2: `LL_Left_Arm`, 4: `LL_Right_Arm`); "LWm,n": a loss weight of m was chosen for the first task, n for the second, "WOSD": *without shared dense layers*

In further experiments on OPPORTUNITY, the effect of using more tasks to aid *ML_B* classification was investigated. Because not every possible task combination using the six available auxiliary channels could be tried, the experiments were restricted to two- and four-task networks, along with an architecture incorporating all seven tasks. Results for these experiments are shown in Figure 4.1(c) and (d). All MTL models were trained with a loss weight of 2 for the main task and 1 for the others. The use of three auxiliary tasks (*LL_Locomotion* and *LL_{Left, Right}_Arm*) did indeed improve results over STL with respect to the macro-F_1 metric. However, adding more tasks did actually decrease performance significantly.

4.2.2 DEAP

Similar experiments were carried out using the DEAP dataset. However, the results will not be directly comparable to those on OPPORTUNITY, since both the *arousal*



(a) STL vs HPS, all epochs

(b) STL vs HPS, last 25 epochs

Figure 4.2: Results on HPS for DEAP. (a): performance over all epochs for two-task MTL (red and green) and individually trained STL (blue); (b): the last 25 epochs for the same models. Abbreviations: *VA*: Valence and Arousal; *LWm,n* a loss weight of m for valence and n for arousal; *Drop* the model featured two dropout units with a rate of 0.5 in each of the two heads.

and the *valence* tasks are equally important to extract meaningful information from the data. Hence, both classification performance scores have to be considered, as opposed to only the score for *ML_B* prediction. Furthermore, results for binary emotion classification on DEAP and other datasets are generally reported using accuracy scores, while OPPORTUNITY was evaluated in terms of the F_1 score.

The results are illustrated in Figure 4.2 using the model summarised in Figure C.2(b) and Table C.4. Two similar HPS models were tested—one featuring dropout units and one without these—against independently trained STL networks. It is clearly visible that the baseline classified both channels at around the same level and was clearly surpassed by the MTL model without dropout for both *arousal* and *valence*. The model with dropout, which featured two dropout units in each head, after the layers *{valence, arousal}_{0,1}* with a rate of 0.4 performed much worse than the network without such units. This is in contrast to OPPORTUNITY, where adding dropout units significantly improved performance for the baseline. For all three architectures, the rate at which they converged is roughly similar, while the accuracy is mostly stable in the final 25 epochs at one level for both tasks.

4.3 Soft Parameter Sharing

For soft parameter sharing, multiple task-specific networks that were fundamentally very similar to the baselines described in subsection 3.2.3 were trained in parallel utilising the same inputs, while the corresponding convolutional networks were put under a norm. This function, selected to be an instance of the Tensor Trace Norm (TTN), served to enforce similarity between the parameters of these corresponding neurons by restricting the rank. This step is conjectured to be an adaptive strategy for sharing between tasks [YH16].

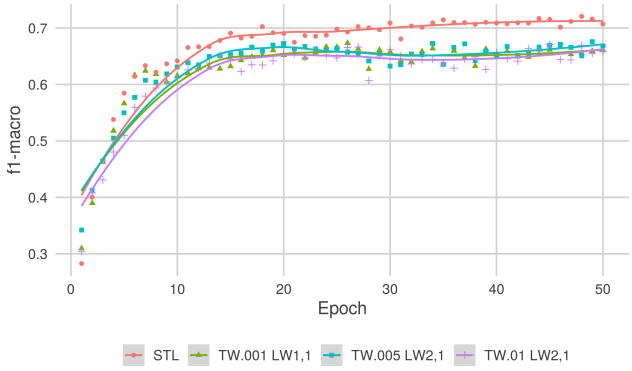
4.3.1 OPPORTUNITY

For OPPORTUNITY, the architecture employed is very similar to the one illustrated in Figure C.1(a). Experiments were carried out only using two-task networks with the tasks `ML_B` and `LL_L`, because SPS networks have a greatly increased amount of parameters compared with HPS. This makes training of bigger networks infeasible on the available hardware. The main hyperparameter this method added for evaluation was the weight of the TTN loss, ω_{TTN} in Equation 3.4. While it is possible to assign different weights to the trace norm of every parameter, the original authors chose a single value of $\omega_{TTN} = 0.001$ in their public code, and added this weighted result to the Mean Squared Error. This value was used as a starting point for our experiments. Besides this value, we also evaluated other choices for this parameter, both larger and smaller, along with assigning different loss weights for the two tasks. Results for these experiments are illustrated in Figure 4.3(a) and (b). Unfortunately, all SPS models performed much worse than the STL baseline on this dataset. This includes other choices for the loss weights and ω_{TTN}, which are not shown for clarity. It is however visible that the weight of the TTN does have an influence on the performance of the network: if it is chosen too small or too large, it performs even worse. This is shown by the cyan curve, with a medium value of 0.005 for this weight, outperforming the other two SPS models. The green curve corresponds to the level chosen in [YH16], which was too small for this dataset.

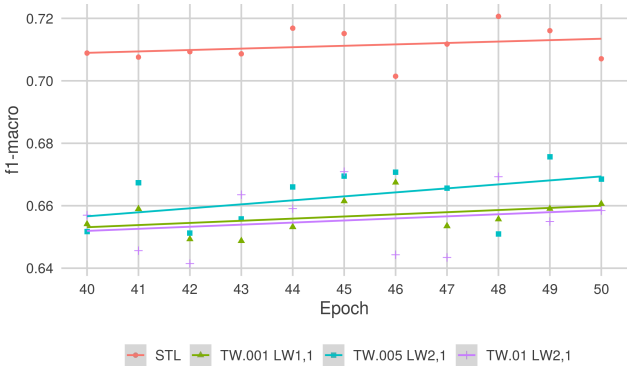
4.3.2 DEAP

Again, the experiments on DEAP were similar to the ones carried out on OPPORTUNITY. The established baseline was modified so that the TTN of each of the convolutional layers was added to the norm with a total weight ω_{TTN}. Since the use of different loss weights for the two tasks did not make sense for DEAP, as the tasks are considered equally relevant, only the optimisation of the TTN weights was investigated. This is shown in Figure 4.3(c) and (d). Unfortunately, this approach did again perform significantly worse than STL on the dataset. One very interesting effect that defies explanation is that valence was classified much better by the SPS models than arousal, while the STL and HPS models achieved approximately equal performance on this label. This was also the case for the model with $\omega_{TTN} = 0.001$ (blue). Training for this model was aborted after 78 epochs because the performance of both models with respect to the sparse categorical cross-accuracy remained very stable at below 60 %, while both the training and testing loss remained at about 1.363 with only extremely small fluctuations. The model with $\omega_{TTN} = 0.0001$ (green) was for 100 epochs with exactly the same configuration and performed much better than the other SPS model. Still, STL reached a much higher performance after 100 epochs.

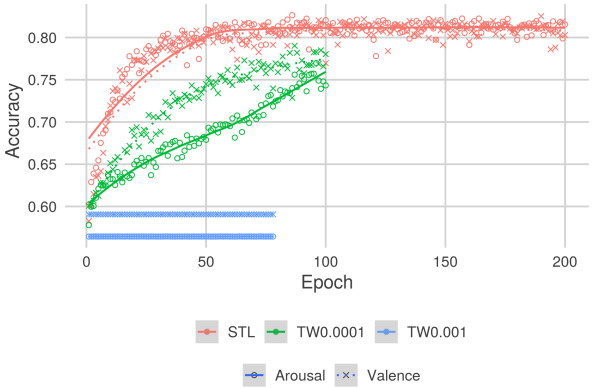
The reason the green model was only trained for 100 epochs lies in the extremely slow calculations required for training an SPS model using the TTN proposed in [YH16]. For the two STL baselines, training took approximately 5 h each for 200 epochs, while the 100 epochs for the green SPS model required over 22 h. Training for the full 200



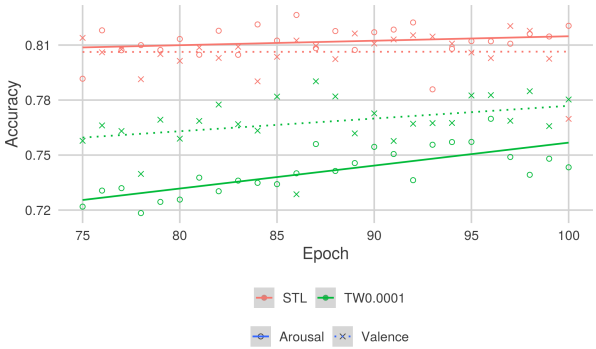
(a) OPPORTUNITY, all epochs



(b) OPPORTUNITY, last 10 epochs



(c) DEAP, all epochs



(d) DEAP, epochs 25 to 50

Figure 4.3: Results for SPS on OPPORTUNITY (top row) and DEAP (bottom row). For DEAP, the SPS networks were not trained for the full 200 epochs. The model *TW0.0001* (green) was trained for 100 epochs, while training on the blue model *TW0.001* was aborted after 78 epochs. Shown in (d) are the epochs 75 to 100, which were available for all both the model with *TW0.0001* and the STL baseline. The abbreviation *TW* refers to the weight ω_{TTN}, while *LW* refers to the weights assigned to tasks `ML_B` and `LL_L`.

epochs would thus have run for about than two full days. Since the results for SPS on OPPORTUNITY were already rather bad and the performance on DEAP after the 50 epochs is still significantly poorer than the baseline, it is reasonable to assume that it would not have performed much better than STL when provided with enough time 200 epochs. While training of SPS models on OPPORTUNITY also proved slow, with SPS models training for double the time, this was not an undue burden, since 50 epochs proved more than adequate to assess the performance. For STL, training took about 2 h, while SPS required up to 4 h.

The slow calculation in question is most likely the Singular Value Decomposition that is used during the approximation of the gradient of this TTN. While TTN is essentially not-differentiable [YH16], they provided an iterative approximation of the gradient which required this SVD, a rather expensive operation in $\mathcal{O}(m \cdot n \cdot \min(m, n))$ for a matrix of size $m \times n$ [VR17], which will be very large for the amount of parameters used for the network. Furthermore, this calculation has to be carried for every sample that is processed by the network during training.

4.4 Cross-Stitch Networks

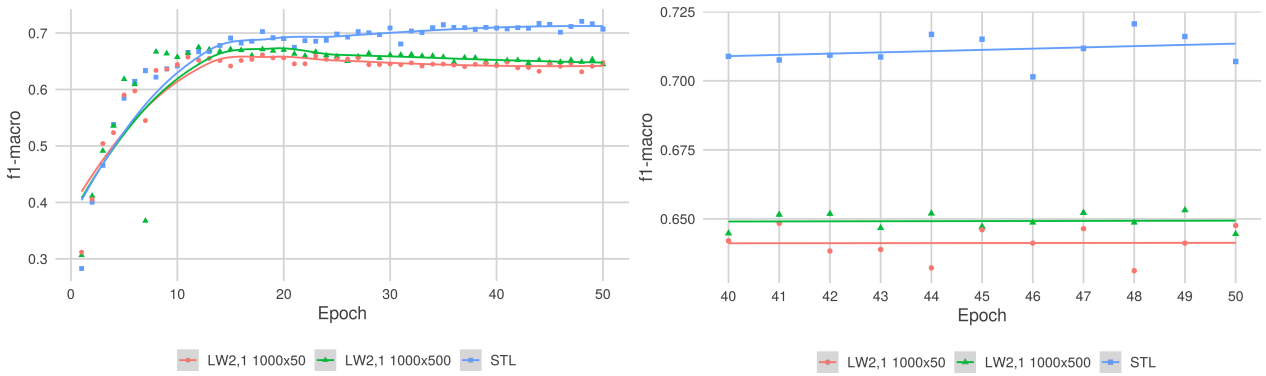
Experiments using this architecture were carried out last for both datasets. Like SPS, these networks were implemented starting from the single-task baseline, using two tasks due to memory constraints. Cross-task sharing was allowed by placing cross-stitch units between each pair of convolutional layers at the same depth, which controlled the flow of information between these layers. The two weighted sums of the inputs served as the outputs for both branches of the task-specific sub-models. In this way, the optimisation process can decide how much additional information each task requires.

Each cross-stitch unit features a 2×2 parameter that is tuned during training for this purpose. For the models examined in this study, these parameters were extracted from the trained model files to be visualised in the form of heatmaps. This parameter is relatively easy for humans to interpret. This is in contrast to many other parameters of Deep Learning models, such as the values of convolutional filters, which be very challenging to understand [GAZ19]

4.4.1 OPPORTUNITY

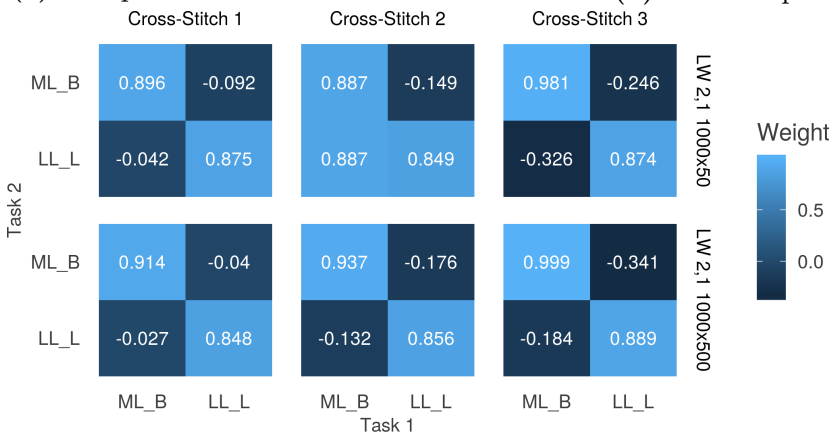
The architecture of the CSN employed for OPPORTUNITY is shown in Figure C.4(a). Cross-Stitch units were inserted after every pooling block and provided the sole inputs for the successive layers. The remaining architecture of the model was almost identical to the one of the SPS model, featuring three convolutional blocks, followed by two fully-connected layers before the output layer in each branch. The number of fully-connected units was varied in the trials on this approach: for both models, the first layer (*dense_0_{ML_Both, LL_Locomotion}*) had 1000 units, while the second layer featured 50 or 500 units respectively. The results of these experiments are shown in Figure 4.4. Again, the results are disappointing, as CSNs were not able to outperform the single-task baseline in any of the three metrics, but performed worse within a significant margin. It is however visible that the model using 500 units in the second fully-connected layer reached a very stable maximum, i.e. the multi-task nature of the network likely stabilised the training process somewhat by imparting an inductive bias.

The examination of the cross-stitch parameters for this model reveals that most of the information the two branches required for classification remained rather task-specific, because the values on the main diagonal are rather large, while the antidiagonal elements, which allow for sharing, are mostly small. Still, some sharing was allowed during training, especially in the last cross-stitch unit, as the antidiagonal elements are not close to zero.



(a) All epochs

(b) Last 10 epochs

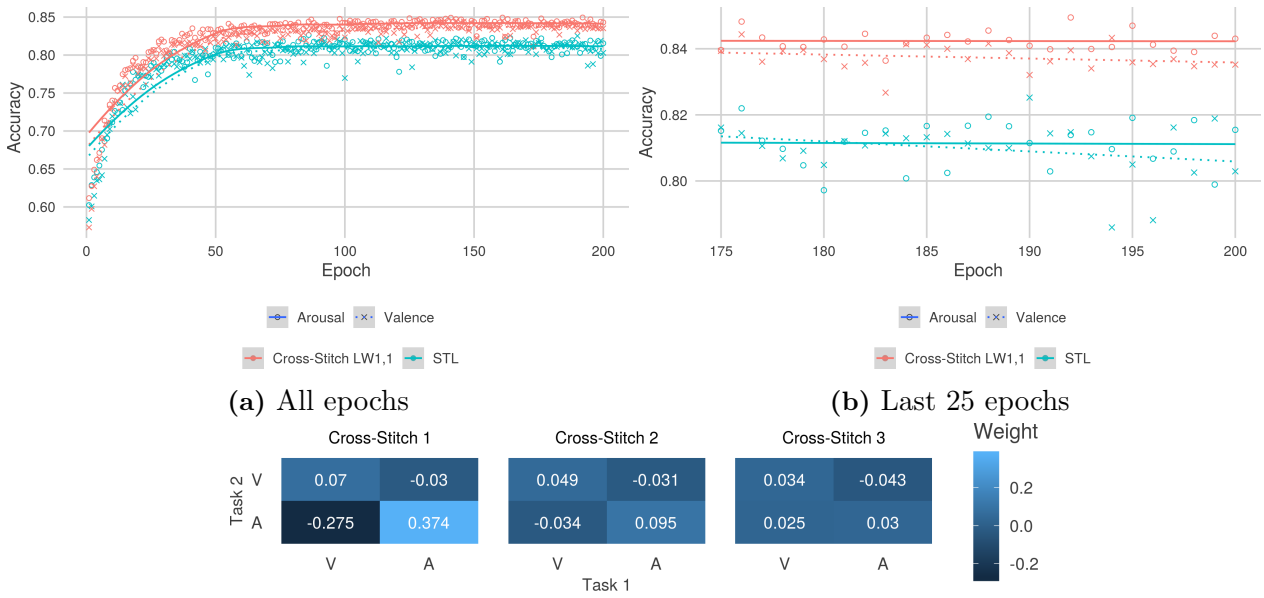


(c) Parameters of the Cross-Stitch units for OPPORTUNITY.

Figure 4.4: Results for all epochs (a) and for the last ten epochs (b) and Cross-Stitch parameters (c) for Cross-Stitch Networks on OPPORTUNITY. In the heatmap, lighter shades correspond to larger, positive values. The elements on the main diagonal are used for same-task pairings, while sharing occurs along the antidiagonal. Abbreviations: `ML_B`, `LL_L`: tasks; *LWm,n*: loss weights for first and second task; $p \times q$: number of units in the first and second fully-connected unit in each head

4.4.2 DEAP

Cross-Stitch Networks were also evaluated on DEAP. The architecture used for this experiment is shown in Figure C.4(b). Unlike the models trained on OPPORTUNITY, CSNs outperformed the STL model by a wide margin, while also exhibiting a smaller variance than the STL model. This can likely be attributed to the MTL nature of the approach, where the interaction of the two tasks stabilises the training process. The parameter distribution for this model is different from those for models trained on OPPORTUNITY, since the first layer has very different values for the same-task combinations, while the second and third cross-stitch units all fall within the same range. A possible explanation is that the information contained in the deeper layers of the two sub-networks is very similar in structure, so that a lot is shared between the two branches. It has to be noted that the parameters of different cross-stitch units can not be compared against another, because their individual magnitude is mostly irrelevant. If all parameters are close to each other, sharing is still carried out to a large degree.



(a) All epochs

(b) Last 25 epochs

(c) Cross-Stitch Parameters

Figure 4.5: Results for all epochs (a) and the last 25 epochs (b), along with Cross-Stitch parameters (c) for Cross-Stitch Networks on DEAP Abbreviations: *V*, *A*: tasks, *LWm,n* loss weights for valence and arousal

While all values are rather small for the DEAP model, the absolute magnitude of all values in the deeper layers is very similar and non-zero.

4.5 Comparative Evaluation

All in all, Multi-Task Learning yielded more favourable results on DEAP than on OPPORTUNITY. A comparative evaluation of the models is given in the following Table 4.1 and Table 4.2. Besides reporting the classification scores for the models, the training time and parameter count is also given there. The model scores were averaged over the last 10 epochs for both datasets.

In summary, the results on OPPORTUNITY were mostly disappointing. While the single-task baseline was improved upon by one approach, the margin for this single model is rather small. Both SPS and CSN performed much worse than STL. On the other hand, the results for DEAP were not only much more positive, but also more conclusive early-on than those on OPPORTUNITY. Even before an extensive evaluation was carried out, the benefit of two of these three approaches became apparent. As the goal of this thesis was to demonstrate whether these approaches can potentially improve results over STL, a smaller number of experiments sufficed for this purpose on DEAP. These experiments demonstrate that the approaches work on some datasets, while they do not work on others without major efforts.

The training time varied greatly between the different models. STL trained the fastest for both datasets, because the number of parameters is larger for all of the MTL models. HPS incurred the smallest penalty, because more layers are shared.

Approach	Variant	F_1-macro $\pm$ CI	Training time (Overhead)
STL		0.711 ± 0.004	2.081 h
HPS	6,0 LW1,1	0.713 ± 0.004	2.166 h (4.06 %)
	6,0 LW2,1	0.712 ± 0.004	2.274 h (9.25 %)
	6,0 LW2,1 WOSD	0.701 ± 0.002	2.129 h (2.29 %)
	6,0,2,4 LW2,1,1,1	$\mathbf{0.719 \pm 0.003}$	2.299 h **(**10.47 %**)**
	all tasks	0.711 ± 0.004	2.530 h (21.56 %)
SPS	TW0.001 LW1,1	0.657 ± 0.004	4.019 h (93.11 %)
	TW0.005 LW2,1	*0.664 ± 0.005*	4.008 h (92.58 %)
	TW0.01 LW2,1	0.655 ± 0.007	4.021 h (93.19 %)
CSN	LW2,1 1000x50	0.641 ± 0.004	4.513 h (116.82 %)
	LW2,1 1000x500	*0.650 ± 0.002*	4.506 h (116.49 %)

Table 4.1: Evaluation of approaches on OPPORTUNITY. The best-performing variant in each group is set in italics, while the best performer overall is set in bold. Scores were calculated from epochs 41 to 50. Abbreviations: $6, 0$: tasks; *LW2,1*: loss weights for tasks; *WOSD*: without shared dense layers; *TWq*: weight of tensor trace norm; $r \times q$: number of dense units in each head; *CI*: 95 % confidence interval

Approach	Variant	V *Acc.* $\pm$ CI	A *Acc.* $\pm$ CI	Training time (Overhead)
STL		0.806 ± 0.007	0.811 ± 0.004	5.167 h (average V & A)
HPS	**VA LW1,1**	$\mathbf{0.86 \pm 0.0031}$	$\mathbf{0.863 \pm 0.003}$	5.622 h (8.80 %)
	VA LW1,1 Drop	0.725 ± 0.006	0.729 ± 0.005	5.625 h (8.86 %)
SPS	*TW0.0001*	*0.772 ± 0.006*	*0.751 ± 0.006*	22.386 h (333.25 %) (100 epochs)
	TW0.001	0.591 ± 0	0.564 ± 0	12.039 h (132.99 %) (78 epochs)
CSN	*LW1,1*	*0.842 ± 0.002*	*0.837 ± 0.002*	5.625 h (8.86 %)

Table 4.2: Evaluation of approaches on DEAP. The averages and confidence intervals ($\alpha = 0.05$) were calculated from epochs 191 to 200 (STL, HPS, CSN) and epochs 91 to 100 resp. 69 to 78 (SPS). Abbreviations: *A*: Arousal, *V*: Valence, *Acc*: sparse categorical cross-accuracy, *CI*: 95 % confidence interval.

While the number of parameters of the SPS and CSN models were very similar, the penalty on SPS was much larger for DEAP in comparison to CSN. However, on OPPORTUNITY, both SPS and CSN incurred a dramatic overhead that (more than) doubled the training time. On DEAP, the overhead on CSN was identical to that of the HPS model with dropout. Training times on DEAP were much longer in general due to the smaller batch size and large number of experiments.

Chapter 5

Discussion & Outlook

In our studies, Multi-Task Learning improved the results on both datasets overall, albeit inconsistently when it comes to the specific approaches. While it could be demonstrated that Hard Parameter Sharing (HPS) can be beneficial on both datasets, this could not be shown for the other two approaches. Of these, Soft Parameter Sharing (SPS) using the Tensor Trace Norm based on the approach in [YH16] did actually hurt performance for both datasets and incurred a significant overhead. The Cross-Stitch Networks (CSNs) proposed in [Mis+16] did improve results on DEAP with only a marginal overhead, but hurt performance on OPPORTUNITY with a dramatic overhead in training time.

In this chapter, the factors that may have influenced these results are discussed in detail. Furthermore, some points where further investigations are needed will be outlined.

5.1 General Considerations

Because of time constraints, only a small subset of possible architectures and meta-parameters could be tried out. This is of course the case for almost all publications in the field of Machine Learning, but is especially significant in this thesis. All three methods introduce additional hyperparameters to the underlying models, which interact with the pre-existing considerations needed both for STL and MTL, such as the batch size. An architecture that performs well on STL may not be the best-suited for SPS or CSN. The size of the convolutional layers, for example, was kept constant during all experiments. Furthermore, the results of HPS on OPPORTUNITY show that „wrong“ hyperparameter choices can hurt performance significantly, while some variations improve the results, if only by a small margin.

Another major factor in the results may be the way in which the datasets were prepared from the raw data made available by the respective researchers. While DEAP was already very clean, OPPORTUNITY required a much larger number of preprocessing steps. Because of the time needed for evaluating multiple preparation strategies on all approaches, the strategy was developed only using Single-Task Learning models. However, this strategy, which mainly consists of the two parameters *window length* and *step size* may not be the best for all of the approaches. The development of suitable preprocessing strategies is by itself a very active field of research [Can+15]

and an important topic in most, if not all, papers on (classification) algorithms [Li+18; Li+19; SG17; Tri+17].

In particular, the preprocessing done on the OPPORTUNITY dataset can be called into question because of the large number of unlabelled stretches present in the original data (where the subject did not perform any of the pre-decided activities). It would be very interesting to perform similar studies on a subset of the data, where unlabelled stretches are removed. This may not yield a large enough number of samples for proper training, however, leading to possibly very challenging experiments. In comparison to OPPORTUNITY, the DEAP dataset was already much cleaner, requiring less preprocessing. A point of contention may however be that each segmented data frame from each experiment was assigned the same label, since no other labels are available. This may be unrealistic because the reported emotion will not have been felt during the entire duration of the song. This means that some data frames may be much more valuable than others, which would make it very challenging for the networks to tell different combination of arousal and valence levels apart. These problems do of course not only affect the performance of the MTL, but rather any classification algorithm that is applied to the data.

5.2 Hard Parameter Sharing

Using Hard Parameter Sharing, the classification results on both datasets could be improved. However, the margin by which this improvement was achieved was rather small on OPPORTUNITY, where the result differed only after the third decimal place. Still, with a confidence of 95 %, the confidence intervals for the STL and HPS performance did not overlap, which can be interpreted as a significant improvement from HPS. On DEAP, this improvement was more substantial, with the better-performing network improving the accuracy by 5 % for both labels.

Of the three MTL approaches tested, HPS is the simplest one in principle, so it is perhaps a bit surprising that the best results in our studies were all achieved by this technique. Of course, more complicated approaches do not always generate better results, as evidenced by the success of sparse (convolutional) neural networks in comparison to fully-connected architectures. The results on OPPORTUNITY have however demonstrated that this approach is only simple from an implementation perspective, but is rather sensitive to the choices of the hyperparameters and underlying architecture. Because the architecture for sharing is completely static, and the network has no way of „choosing“ what to share, this can make it very hard to find a dataset-specific set of tasks and a corresponding architecture, forcing experimenters to rely on trial and error. In particular, this makes it hard to transform an existing STL network into such a HPS architecture.

Because of this, the possible benefits of HPS are still unclear from our studies. Because we were able to demonstrate improvements when compared to STL, it is however very reasonable to assume that a different choice of hyperparameters might have resulted in much better results. To this end, a number of different approaches have been introduced

in the literature, which aim to reduce the dependency of HPS on these hyperparameters. For example, the loss weights in the overall loss function have been shown to crucially affect results when ill-chosen, prompting the development of uncertainty-based HPS models [KGC17]. Moreover, all models were trained using the same learning rate in the optimiser for each task-specific head, which may not be appropriate for all domains. This could be addressed with different learning rates and/or task-wise early stopping [Zha+14]. The plethora of further resources available for improving Multi-Task Learning demonstrate that it is a very active field of research with potential for big improvements, which will likely also apply to time-series data.

5.3 Soft Parameter Sharing

The basic idea of SPS is rather simple: multiple networks of identical (or at least very similar) architectures are trained in parallel for each task without any direct connections. The parameters of the networks are then forced to be broadly similar. This similarity has the effect that some parameters at each depth are shared (i.e. identical), while others are task-specific. This is in contrast to HPS, where parameters are either fully shared or fully task-specific. To achieve this similarity, a mostly arbitrarily chosen norm is imposed on the collection of layers at each depth. This function is fundamental to the performance of the method.

For our studies on Soft Parameter Sharing, we chose the approach presented in [YH16], where the authors demonstrated a benefit on using Tensor Trace Norm-regularised MTL in the domain of Natural Language Processing. However, the approach could not be shown to improve results over STL on the two datasets investigated. The method proposed in that paper was chosen due to multiple reasons, one being the publication within the workshop track of a top-tier conference (the International Conference on Learning Representations 2017). More importantly, the results the authors obtained showed an impressive improvement over STL in the chosen domain. Because of the public source code, the required adaptations were considered to be doable with little effort. Because of those factors, the bad performance on the two datasets was especially disappointing.

Besides choosing sub-optimal hyperparameters, one possible reason that the model performed this bad in our studies could be that semantic errors were introduced in this adaptation of the TensorFlow-specific source code to the Keras API. Because this code relies on low-level tensor operations and implements an essentially non-differentiable function [YH16], it is very difficult to debug. For instance, it is not clear how TensorFlow calculates the sub-gradient of the function, which is essential for the gradient descent of the loss.

Indeed, by removing the TTN term from the overall loss function, a network with an identical architecture to the one used for the studies on DEAP was able to outperform the other two SPS models, reaching an average accuracy of 0.816 ± 0.005 for Valence and 0.808 ± 0.01 for Arousal after 100 epochs. While this does not meaningfully outperform STL, this result is consistent with the observation that smaller values for the weight

of the TTN improve results on DEAP by reducing the influence of the TTN. It thus has to be assumed that either the TTN or SPS in general is unsuitable for the datasets evaluated due to some unknown reason or that the implementation for this study is faulty.

Regarding the first point, we were unable to find any reference where a similar approach was applied to time-series data. Most SPS approaches either use entirely synthetic datasets or low-dimensional real-life datasets [AEP07; BYL14]. However, because the basic idea of SPS (where layers are shared partially as needed) follows rather intuitively from the basic idea of HPS (where some layers are shared entirely), it is unreasonable to assume that SPS can not work on time-series data at all. The lack of papers in this field merely illustrates that further research into MTL, both applied to these datasets and in general, is urgently needed.

From this reasoning, the fault will probably lie in the use or the implementation of the TTN. Unfortunately, no other regulariser could be examined due to time constraints. These alternatives include classical difference metrics, such as the L_1 [OT06] and L_2 [Duo+15] norms, or a combination of these two [OT06]. Regardless of where the fault lies in our approach, the overhead incurred by the complex calculation of the sub-gradient of the loss can be directly attributed to our choice of regulariser. Choosing a different regulariser like L_2, which has a much simpler sub-gradient [Van19], would likely have a significantly smaller overhead.

5.4 Cross-Stitch Networks

Fundamentally, the ideas underlying Cross-Stitch Networks are very intuitive. While parameters, and thus also features, are either fully shared or fully task-specific with HPS, and partially shared using SPS, CSNs initially share nothing, and allow the optimisation process to assign weights to each input to the CS unit. This makes it possible for the network to decide on how much information from other tasks is required for each ultimate output. Initially, no information is shared because the parameter of each cross-stitch unit is initialised using the identity matrix. This approach was introduced in 2016 [Mis+16] and influenced a range of later works [Kok16; Rud17].

While CSNs did improve the classification results of arousal and valence over the single-task baseline on DEAP, they neither surpassed HPS, nor did they perform as well on OPPORTUNITY—indeed reaching a much lower F_1 score than STL. Furthermore, the time penalty on OPPORTUNITY was substantial, requiring more than double the time for training 50 epochs than the baseline. This penalty was not observed on DEAP, however, where the architecture required as much overhead as HPS. This demonstrates that the performance of CSNs is very much dependent on the dataset used, as well as the underlying architecture. Again, it is very possible that a different basis and choice of hyperparameters is able to perform much better.

We have not found a satisfactory explanation for the difference in training times. The networks for OPPORTUNITY had a much larger number of parameters, with the baseline having more than 6 million parameters, while the cross-stitch networks had only about 4.5 million. For DEAP, the baseline only featured about 87000 parameters, while the CSN actually featured double that amount, at approximately 174000 parameters. Based on this, the performance hit on DEAP should be larger. While we were only able to demonstrate improved performance for one of the two datasets, we believe that improvements on OPPORTUNITY should be possible to achieve using a different architecture. However, based on the performance penalty observed for this dataset, and the superior performance of HPS on both datasets, the use of Hard Parameter Sharing may prove the better choice for most applications. Regardless, one major contribution of this thesis is the public source code, where CSNs are implemented. This is of great value to the wider community because no Keras code for Cross-Stitch Networks was publicly available so far. The provided source code can thus be used for further experiments on this approach in other domains.

5.5 Outlook

Further studies on Multi-Task Learning for Time Series processing and classification are very much needed. The studies in this thesis have shown that some approaches work better on one dataset than another. While HPS has performed best on both datasets, studies should be carried out on more datasets to confirm this. For the datasets and approaches considered here, many permutations of the hyperparameters and (meta-) architectures should be investigated. Furthermore, the findings should be confirmed using experiments on further datasets.

Furthermore, the number and choice of tasks is a suitable target for improvement. While studies on DEAP have been restricted to the two-task problem of joint Arousal-Valence classification, more (arbitrarily chosen) permutations have been evaluated on OPPORTUNITY. Still, a large number of choices that have not been investigated so far exist on that dataset, of which some may improve results even more. For DEAP, the same problem exists, as the dataset provides at least one more axis (Dominance) on which the data could be classified (inferring liking from the data seems very challenging, but could well be possible).

The tasks used in this study were also very homogeneous, i.e. all tasks were classification tasks for the same data. Especially HPS allows for much more heterogeneous tasks, such as a combination of classification and regression tasks. This is also possible, yet more challenging, using the other approaches.

Every MTL method incurred an overhead in training time. This comes atop of the time required for the implementation of the MTL method. By using public code for the implementation, the overhead from the latter is rather small, however, with most of the code being needed for data preparation, model selection and implementation, along with boilerplate code. Using multiple tasks with the Keras framework with

the presented methods is rather easy, so that it is feasible to try out different MTL approaches on top of an STL approach.

Furthermore, many other approaches are presented in the literature, such as Sluice Networks [Rud+17] or fully-adaptive methods [Lu+16]. Many of these methods are very exciting and intuitive and thus worthy of consideration.

5.6 Summary

All in all, we were able to show that Multi-Task Learning can help the classification of time-series data in the medical domain. Based on our experiments, the simplest method, Hard Parameter Sharing achieves the best results, but requires carefully chosen hyperparameters in addition to the usual decisions. No benefit from using Soft Parameter Sharing, where layers are regularised using the Tensor Trace Norm, could be demonstrated. Moreover, this method requires a lot of time for training due to the complexity of the gradient. Cross-Stitch Networks achieved a performance between ordinary Single-Task Learning and Hard Parameter Sharing on one dataset, while producing worse results than Single-Task Learning on the other.

The margins achieved in our experiments were mostly rather small, however. The overhead in training time for the models outperforming STL was always in the range of 4 % to 11 %, which is likely within an acceptable range for most applications.

Our experiments were limited to two datasets from different domains, which limits the extent to which our results are generally applicable. To aid further experiments, we provide all source code and pre-trained networks on GitHub. Based on our experiments, the use of Hard Parameter Sharing can be recommended for a wide range of applications, while the other two methods require further examination.

Bibliography

[AEP07] A. Argyriou, T. Evgeniou, and M. Pontil. „Multi-task feature learning“. In: *Advances in neural information processing systems*. 2007, pp. 41–48 (page 52).

[AlR+16] R. Al-Rfou et al. „Theano: A Python framework for fast computation of mathematical expressions“. In: *CoRR* (2016). arXiv: 1605.02688 (page 34).

[App19] Apple Inc. *Detecting Falls Using a Mobile Device*. United States Patent Application 62565988. Apr. 2019 (page 1).

[Ben09] Y. Bengio. „Learning Deep Architectures for AI“. In: *Foundations and Trends® in Machine Learning* 2.1 (2009), pp. 1–127. DOI: 10.1561/2200000006. URL: http://dx.doi.org/10.1561/2200000006 (pages 5, 7).

[BL94] M. M. Bradley and P. J. Lang. „Measuring emotion: The self-assessment manikin and the semantic differential“. In: *Journal of Behavior Therapy and Experimental Psychiatry* 25.1 (1994), pp. 49–59. DOI: 10.1016/0005-7916(94)90063-9 (page 23).

[Bon18] G. Bonaccorso. *Mastering Machine Learning Algorithms*. Packt, 2018 (pages 6, 8, 16, 17).

[BYL14] M. T. Bahadori, Q. (Yu, and Y. Liu. „Fast Multivariate Spatio-temporal Analysis via Low Rank Tensor Learning“. In: *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 3491–3499. URL: http://papers.nips.cc/paper/5429-fast-multivariate-spatio-temporal-analysis-via-low-rank-tensor-learning.pdf (page 52).

[Can+15] H. Candra, M. Yuwono, R. Chai, et al. „Investigation of window size in classification of EEG-emotion signal with wavelet entropy and support vector machine“. In: *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. Aug. 2015, pp. 7250–7253. DOI: 10.1109/EMBC.2015.7320065 (pages 28, 49).

[Car97] R. Caruana. „Multitask learning“. PhD thesis. Carnegie Mellon University, Pittsburgh: School of Computer Science, 1997 (pages 2, 9–11, 17).

[Che+15] T. Chen et al. „MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems“. In: *CoRR* (2015). arXiv: 1512.01274 (page 34).

[Cho+14] K. Cho, B. van Merrienboer, Ç. Gülçehre, et al. „Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation“. In: *CoRR* (2014). arXiv: 1406.1078 (page 9).

[Cho+15] F. Chollet et al. *Keras*. 2015. URL: https://keras.io (page 34).

[Cle79] W. S. Cleveland. „Robust Locally Weighted Regression and Smoothing Scatterplots“. In: *Journal of the American Statistical Association* 74.368 (1979), pp. 829–836. DOI: 10.1080/01621459.1979.10481038 (page 38).

[Den+09] J. Deng, W. Dong, R. Socher, et al. „ImageNet: A Large-Scale Hierarchical Image Database“. In: *CVPR09*. 2009 (page 25).

[Duo+15] L. Duong, T. Cohn, S. Bird, and P. Cook. „Low Resource Dependency Parsing: Cross-lingual Parameter Sharing in a Neural Network Parser“. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, pp. 845–850. DOI: 10.3115/v1/P15-2139 (pages 11, 52).

[Ekm+87] P. Ekman, W. V. Friesen, M. O'sullivan, et al. „Universals and cultural differences in the judgments of facial expressions of emotion.“ In: *Journal of personality and social psychology* 53.4 (1987), p. 712 (page 22).

[GAZ19] A. Ghorbani, A. Abid, and J. Zou. „Interpretation of neural networks is fragile“. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 3681–3688. DOI: 10.1609/aaai.v33i01.33013681 (pages 7, 44).

[Gru+18] A. Gruenewald, D. Kroenert, J. Poehler, et al. „Biomedical Data Acquisition and Processing to Recognize Emotions for Affective Learning“. In: *2018 IEEE 18th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE. 2018, pp. 126–132 (page 1).

[Hah+00] R. H. R. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung. „Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit“. In: *Nature* 405.6789 (2000), pp. 947–951. DOI: 10.1038/35016072 (page 7).

[Hal18] J. Hale. *Deep Learning Framework Power Scores 2018*. Sept. 2018. URL: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a (page 34).

[Haw04] D. M. Hawkins. „The problem of overfitting“. In: *Journal of chemical information and computer sciences* 44.1 (2004), pp. 1–12 (page 2).

[HDR19] S. Hayou, A. Doucet, and J. Rousseau. „On the Impact of the Activation Function on Deep Neural Networks Training“. In: *CoRR* (Feb. 2019). arXiv: 1902.06853 (page 7).

[HS97] S. Hochreiter and J. Schmidhuber. „Long Short-Term Memory“. In: *Neural Computation* 9.11/1997 (1997) (page 8).

[JSP19] T. Joshi, S. Sivaprasad, and N. Pedanekar. „Partners in Crime: Utilizing Arousal-Valence Relationship for Continuous Prediction of Valence in Movies“. In: *Proceedings of the 2nd Workshop on Affective Content Analysis*. Ed. by N. Chhaya, K. Jaidka, A. Sinha, and L. Ungar. 2019. URL: http://ceur-ws.org/Vol-2328/1_paper_9.pdf (page 24).

[Kar16] A. Karpathy. *CS231n Convolutional Neural Networks for Visual Recognition*. Course Material, Stanford University. 2016. URL: http://cs231n.github.io/ (page 8).

[KGC17] A. Kendall, Y. Gal, and R. Cipolla. „Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics“. In: *CoRR* (2017). arXiv: 1705.07115 (page 51).

[Koe+11] S. Koelstra, C. Muhl, M. Soleymani, et al. „DEAP: A database for emotion analysis; using physiological signals“. In: *IEEE transactions on affective computing* 3.1 (2011), pp. 18–31 (page 22).

[Kok16] I. Kokkinos. „UberNet: Training a 'Universal' Convolutional Neural Network for Low-, Mid-, and High-Level Vision using Diverse Datasets and Limited Memory“. In: *CoRR* (2016). arXiv: 1609.02132 (page 52).

[KSK18] Y.-H. Kwon, S.-B. Shin, and S.-D. Kim. „Electroencephalography based fusion two-dimensional (2D)-convolution neural networks (CNN) model for emotion recognition system“. In: *Sensors* 18.5 (2018), p. 1383 (page 33).

[LeC+98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. DOI: 10.1109/5.726791 (pages 7, 25).

[Li+18] F. Li, K. Shirahama, M. Nisar, L. Köping, and M. Grzegorzek. „Comparison of feature learning methods for human activity recognition using wearable sensors“. In: *Sensors* 18.2 (2018), p. 679 (pages 1, 6, 25, 26, 30, 31, 50, 65).

[Li+19] Y. Li, W. Zheng, L. Wang, Y. Zong, and Z. Cui. „From Regional to Global Brain: A Novel Hierarchical Spatial-Temporal Neural Network Model for EEG Emotion Recognition“. In: *IEEE Transactions on Affective Computing* (2019). DOI: 10.1109/TAFFC.2019.2922912 (pages 1, 50).

[LRP17] D. Lopez-Martinez, O. Rudovic, and R. Picard. „Physiological and behavioral profiling for nociceptive pain estimation using personalized multitask learning“. In: *CoRR* (2017). arXiv: 1711.04036 (page 1).

[LS14] Y. Liu and O. Sourina. „Real-Time Subject-Dependent EEG-Based Emotion Recognition Algorithm“. In: *Transactions on Computational Science XXIII: Special Issue on Cyberworlds*. Ed. by M. L. Gavrilova, C. J. K. Tan, X. Mao, and L. Hong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 199–223. DOI: 10.1007/978-3-662-43790-2_11 (pages 28, 29).

[Lu+16] Y. Lu, A. Kumar, S. Zhai, T. Javidi, and R. S. Feris. „Fully-adaptive Feature Sharing in Multi-Task Networks with Applications in Person Attribute Classification“. In: *CoRR* (2016). arXiv: 1611.05377 (pages 11, 54).

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (page 34).

[Mis+16] I. Misra, A. Shrivastava, A. Gupta, and M. Hebert. „Cross-stitch Networks for Multi-task Learning“. In: *CoRR* (2016). arXiv: 1604.03539 (pages 3, 12, 19, 49, 52).

[Mon+18] J. Monteiro, R. Granada, J. P. Aires, and R. C. Barros. „Evaluating the Feasibility of Deep Learning for Action Recognition in Small Datasets“. In: *2018 International Joint Conference on Neural Networks (IJCNN)*. July 2018, pp. 1–8. DOI: 10.1109/IJCNN.2018.8489297 (page 2).

[MR74] A. Mehrabian and J. A. Russell. *An approach to environmental psychology*. The MIT Press, 1974 (page 23).

[Núñ+18] J. C. Núñez, R. Cabido, J. J. Pantrigo, A. S. Montemayor, and J. F. Vélez. „Convolutional Neural Networks and Long Short-Term Memory for skeleton-based human activity and hand gesture recognition“. In: *Pattern Recognition* 76 (2018), pp. 80–94. DOI: 10.1016/j.patcog.2017.10.033 (page 1).

[OT06] G. Obozinski and B. Taskar. „Multi-task feature selection“. In: *In the workshop of structural Knowledge Transfer for Machine Learning in the 23rd International Conference on Machine Learning (ICML 2006)*. 2006 (page 52).

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. „Learning representations by back-propagating errors“. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0 (page 6).

[Rog+10a] D. Roggen, A. Calatroni, M. Rossi, et al. „Collecting complex activity datasets in highly rich networked sensor environments“. In: *2010 Seventh International Conference on Networked Sensing Systems (INSS)*. June 2010, pp. 233–240. DOI: 10.1109/INSS.2010.5573462 (pages 20, 21).

[Rog+10b] D. Roggen, A. Calatroni, M. Rossi, et al. „Walk-through the OPPORTUNITY dataset for activity recognition in sensor rich environments“. In: (Jan. 2010) (page 20).

[Rog+12] D. Roggen, A. Calatroni, L.-V. Nguyen-Dinh, et al. *OPPORTUNITY Activity Recognition Data Set*. 2012. URL: https://archive.ics.uci.edu/ml/datasets/opportunity+activity+recognition (page 21).

[Rud+17] S. Ruder, J. Bingel, I. Augenstein, and A. Søgaard. „Latent Multi-task Architecture Learning“. In: *CoRR* (May 2017). arXiv: 1705.08142 (pages 12, 54).

[Rud17] S. Ruder. „An Overview of Multi-Task Learning in Deep Neural Networks“. In: *CoRR* (2017). arXiv: 1706.05098 (pages 2, 9–12, 17, 52).

[Rus03] J. A. Russell. „Core affect and the psychological construction of emotion“. In: *Psychological review* 110.1 (2003), p. 145 (pages 22–24).

[Rus80] J. A. Russell. „A circumplex model of affect“. In: *Journal of Personality and Social Psychology* 39 (1980), pp. 1161–1178 (page 23).

[SA16] F. Seide and A. Agarwal. „CNTK: Microsoft's Open-Source Deep-Learning Toolkit“. In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 2135–2135. DOI: 10.1145/2939672.2945397 (page 34).

[Sch+84] K. R. Scherer et al. „On the nature and function of emotion: A component process approach“. In: *Approaches to emotion*. 1984 (page 23).

[SG17] K. Shirahama and M. Grzegorzek. „On the Generality of Codebook Approach for Sensor-Based Human Activity Recognition“. In: *Electronics* 6.2 (June 2017), p. 44. DOI: 10.3390/electronics6020044 (pages 30, 50).

[SK18] M. N. Shiota and J. W. Kalat. *Emotion*. New York: Oxford University Press, 2018 (pages 22, 23, 27).

[Sri+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. „Dropout: a simple way to prevent neural networks from overfitting“. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (page 2).

[Tri+17] S. Tripathi, S. Acharya, R. Sharma, S. Mittal, and S. Bhattacharya. „Using Deep and Convolutional Neural Networks for Accurate Emotion Classification on DEAP Dataset.“ In: *Innovative Applications of Artificial Intelligence*. 2017. URL: https://aaai.org/ocs/index.php/IAAI/IAAI17/paper/view/15007/13731 (pages 33, 50).

[Tuc66] L. R. Tucker. „Some mathematical notes on three-mode factor analysis“. In: *Psychometrika* 31.3 (Sept. 1966), pp. 279–311. DOI: 10.1007/BF02289464 (page 18).

[Van19] L. Vandenberghe. *Subgradients*. Lecture for course 236C (Optimization methods for large-scale systems) at the University of Los Angeles. 2019. URL: http://www.seas.ucla.edu/~vandenbe/236C/lectures/subgradients.pdf (pages 19, 52).

[Var08] K. Varda. *Protocol Buffers: Google's Data Interchange Format*. Tech. rep. Google, June 2008. URL: http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html (page 61).

[Vau96] M. L. Vaughn. „Interpretation and knowledge discovery from the multilayer perceptron network: Opening the black box“. In: *Neural Computing & Applications* 4.2 (June 1996), pp. 72–82. DOI: 10.1007/BF01413743 (page 7).

[Vij+17] V. Vijayakumar, M. Case, S. Shirinpour, and B. He. „Quantifying and Characterizing Tonic Thermal Pain Across Subjects From EEG Data Using Random Forest Models“. In: *IEEE Transactions on Biomedical Engineering* 64.12 (Dec. 2017), pp. 2988–2996. DOI: 10.1109/TBME.2017.2756870 (page 1).

[VR17] V. Vasudevan and M. Ramakrishna. „A Hierarchical Singular Value Decomposition Algorithm for Low Rank Matrices“. In: *CoRR* (2017). arXiv: 1710.02812 (pages 19, 43).

[Wan+19] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu. „Deep learning for sensor-based activity recognition: A survey“. In: *Pattern Recognition Letters* 119 (2019). Deep Learning for Pattern Recognition, pp. 3–11. DOI: 10.1016/j.patrec.2018.02.010 (page 6).

[YH16] Y. Yang and T. M. Hospedales. „Trace Norm Regularised Deep Multi-Task Learning“. In: *CoRR* (2016). arXiv: 1606.04038 (pages 3, 11, 18, 19, 41–43, 49, 51, 63).

[Yin+17] Z. Yin, Y. Wang, L. Liu, W. Zhang, and J. Zhang. „Cross-subject EEG feature selection for emotion recognition using transfer recursive feature elimination“. In: *Frontiers in neurorobotics* 11 (2017), p. 19. DOI: 10.3389/fnbot.2017.00019 (pages 29, 33).

[Zha+14] Z. Zhang, P. Luo, C. C. Loy, and X. Tang. „Facial landmark detection by deep multi-task learning“. In: *European conference on computer vision*. Springer. 2014, pp. 94–108. DOI: 10.1007/978-3-319-10599-4_7 (pages 12, 51).

[ZZ18] L. Zhang and J. Zhang. „Synchronous Prediction of Arousal and Valence Using LSTM Network for Affective Video Content Analysis“. In: *CoRR* (2018). arXiv: 1806.00257 (page 24).

[ZZL17] W.-L. Zheng, J.-Y. Zhu, and B.-L. Lu. „Identifying stable patterns over time for emotion recognition from EEG“. In: *IEEE Transactions on Affective Computing* (2017) (page 33).

Appendix A

Implementation Details

A.1 General Implementation

The main culprit in regards to the very high memory usage while training on OPPORTUNITY using TF1 was the amount of memory consumed by keeping the entire dataset in system memory at all times. This is a problem in many networks which was historically often addressed by just adding more RAM. An alternative solution, however, has been introduced with functionality specific to the TensorFlow implementation of Keras. While this `dataset` functionality was already available experimentally in earlier versions of TensorFlow, the recently-released TensorFlow 2 specifically builds on this effort to deliver a convenient way of reducing memory consumption.

Every `tf.dataset` is build on Protocol Buffer (*Protobuf*) messages, a *language-neutral, platform-neutral mechanism for serialising structured data* [Var08]. These messages each describe sets of instances of `tensorflow.Tensor`, which serve as the input(-s) and ground truth label(-s) for the network. When the dataset is created, the data is serialised into these messages using a user-defined message specification and written to a file. This serialisation relies on very primitive structures, namely lists of integers, floating-point numbers and bytes. Because every variable can be described as a list of bytes, everything can be serialised with the `BytesList` primitive. While serialising NumPy arrays as lists of floats seems to be possible, the TensorFlow documentation recommends serialising these arrays to a byte string. Thus, for both datasets used, the input data, which is available in two-dimensional tensors for each sample, was serialised into `BytesList` instances.

To consume a dataset that was created in this way, a number of steps have to be carried out: First, a `tf.data.TFRecordDataset` needs to be created from a list of filenames. Then, a function is mapped over every message in this set of files that describes how to deserialise the data. Crucially, data structures that were stored as a `BytesList` have to be deserialised as well. Interestingly, the shape of the tensor is not stored along the data—because the shape of a tensor can be changed using reshaping operations—and has to be set to the desired target shape during deserialisation. The overall deserialisation function should return a tuple `(inputs, outputs)` that will be used during training. Because multi-input and -outputs networks are easy to implement using the Keras API, these values may also be tuples themselves. Also, not every value that was serialised has to be returned. In this way, the multi-task networks were only provided with the ground-truth values they needed.

After this step, the data can be shuffled, repeated n times (or *ad infinitum*) and batched. While these steps are similar to a NumPy-based pipeline, the crucial difference lies in the *laziness* inherent to the TensorFlow approach. This means that data is only read from disk when it is needed, so that only a minimal amount of memory is consumed, because the dataset describes a plan how to generate data from the files. Of course, asynchronous buffering is also possible and automatically provided so that the current batch along with the next few batches will always be in RAM. While this approach yields significant improvements with regards to memory size, it also comes with some caveats. In particular, lazily shuffling a large dataset is not trivial. TensorFlow solves this by providing a buffer of size b that is filled with the first b examples in the dataset. The file pointer remains at position $b + 1$ until another sample is requested. Whenever this occurs, TensorFlow randomly chooses one sample and

replaces the item with the next item in the dataset at the position where the file pointer is at this point. Hence, the performance of the network may be dependent on the size of this shuffle buffer, which becomes a meta-parameter for the method.

A.2 DEAP

In our experiments on DEAP, this size of the shuffle buffer proved to be an important factor with regards to network performance. Because it was selected too small, with a buffer size of $b = 1000$, every minibatch of 32 data frames consisted of data from at most two subjects when using one-second dataframes with an overlap of half a second. Using these parameters, every experiment was split into 125 data frames for a total of 5000 frames per subject. This may have resulted in a subject-independent classifier with very little accuracy during validation. While increasing the size of this buffer already improved results significantly, the approach we decided on required pre-shuffling the entire dataset during the preprocessing, when enough memory is available. From a code perspective, this was done by wrapping the class that writes the examples to disk in a container which cached the entire data in a `numpy` array, which was shuffled using `numpy` methods before writing out the samples. This ensured a high-quality shuffle, identical to traditional pipelines.

A.3 Cross-Stitch Networks

Because of the automatic calculation of the graph, the implementation of cross-stitch networks described in subsection 3.1.4 relies on what may be aptly described as a "hack". While theoretically, it should have been possible to return n output tensors for n tasks, doing this resulted in a `graph disconnect` exception because TensorFlow could not infer the computations required. By stacking the n output values, which are of shape $s = (s_1, \ldots, s_k)$ into a tensor of shape $(n, s_1, \ldots, s_k)$, which gets returned by the `CrossStitch` layer and is then immediately unstacked outside of that class into the original set of n shape-s-tensors, this error could be avoided. This unstacking automatically adds `TensorFlowOpLayers` to the network graph that encapsulate the respective operations. However, because this is essentially a no-operation, no problems should arise from this solution.

A.4 Soft Parameter Sharing

```
def loss_trace_norm(self, ttn_layers, ttn_weight=None):''
"""calculate the tensor trace norm for every collection in ttn_layers,
optionally weighted by ttn_weight"""
    def loss(y_true, y_pred):
    """loss function returned by loss_trace_norm incorporating the TTN"""
        def _ttn(layer_idx, ttn_layer):
        """calculate the TTN for the set of layers"""
          # get layer for every layer name in ttn_layer
          weights = [self.model.get_layer(
                     name=conv_name).weights[0] for conv_name in ttn_layer]
          # assume shapes are the same
          shapeX = weights[0].get_shape().as_list()
          dimX = len(shapeX)
          #stack on the last axis
          stack = tf.stack(weights, axis=dimX)
```

```
16            t = TensorTraceNorm(stack)
17            #returns a tensor with dim (n, 1)
18            tr = tf.reduce_sum(t)
19            return tr
20
21        # ttn loss gets added to categorical cross_entropy loss
22        cce = tf.keras.losses.categorical_crossentropy(y_true, y_pred)
23        # calculate the ttn loss for every set in the collection
24        ttn = [_ttn(layer_idx, ttn_layer)
25                    for layer_idx, ttn_layer
26                    in enumerate(ttn_layers)]
27        # reduce the collection by summing
28        sttn = tf.reduce_sum(ttn)
29        if ttn_weight is not None:
30          # optionally weight the tensor trace norm loss component (gamma)
31          sttn = tf.math.scalar_mul(ttn_weight,
32                                    sttn)
33        final_loss = cce + sttn
34        return final_loss
35
36      if ttn_layers is None:
37        # this returns a function, not a scalar!
38        return tf.keras.losses.categorical_crossentropy
39      else:
40        # the inner function above!
41        return loss
```

Listing A.1: Implementation of Tensor Trace Norm Loss, utilising functions from https://github.com/wOOL/TNRDMTL [YH16], as used in this thesis.

Appendix B

Source code

The code for this thesis is available on GitHub at https://github.com/LtSurgekopf/TimeSeriesMTL and can be used unter the terms of the MIT licence.
For both datasets (OPPORTUNITY and DEAP), the entry point into the application is the same, either `hps.py` or `sps.py`. To run it, a number of command line parameters (implemented using `argparse` in the modules `utils/app_{hps,sps}.py`) have to be supplied, while some have sensible defaults. Crucially, the dataset path is also a parameter, so that different datasets can be used, possibly even one not included in this study.
A very important parameter is that of the model to be tested, which corresponds to a function in `models.base_models_mtl`. While currently the parameters of these functions are not mapped to command line parameters, these could be easily added if needed.
Because experiments on DEAP were carried out after those on OPPORTUNITY, it behaves a bit differently, as changes in this branch were not backported to OPPORTUNITY. The code specific for this dataset assumes that a dataset description file is present next to the `tfrecord` files of the dataset. This file describes parameters such as the input shape, the number of classes, the label names, along with the validation split and the number of samples available for training and testing. Also, the files for the selected validation split are listed, so that a datasets can be created from each list of files. This meta-data file significantly reduces the possibility of calculation errors when manually defining the number of training and testing steps, because they can be automatically calculated from the number of examples in each case along with the batch size. It is automatically created when creating the DEAP dataset with the provided scripts.
The source code distribution contains the following files:

README.md Help for this package. Refer to this for more guidance on usage.

conda-config.yml Can be used with Anaconda 3 to install an environment with all required packages. Change the username in the last line to yours before invoking `conda env create -f conda-config.yml`

hps.py Main entry point for HPS. This can also be used for STL.

sps.py Main entry point for SPS and CSNs

shared.py Functions and definitions required for both HPS and SPS

evaluation.py Evaluation callback and file transformer for OPPORTUNITY. Also provides a `scipy`-based implementation of the F_β score. The callback is provided during training and writes JSON files. The script can also be invoked directly to process these files.

r-eval.rmd RMarkdown notebook for generating graphics from the results in CSV format

models/base_models_mtl.py Required model definitions for HPS and SPS

models/generate_mtl_head.py Function for generating MTL heads atop a provided model

utils/app_{sps,hps}.py Argparse interface for HPS and SPS. Change command line parameters here.

utils/{deap, opportunity}.py Dataset-specific functions

utils/f_scores.py Implementation of the F_β function. This function can be provided to `model.fit`, but yields significantly worse results than those obtained by the industry-standard `scipy` implementation.

utils/misc.py An interface for logging that just redirects all standard output to a file

utils/utilitary_mtl.py Alternative implementations of the F_β score

dataio/deap/make_deap_dataset.py Create a `tf.dataset` from the DEAP files[1]

dataio/deap/deap_reader.py Adapter for the DEAP dataset in `tfrecord` format which lazily yields tuples for the models required.

dataio/opportunity/clean_opp_data.py First step for processing the OPPORTUNITY dataset files[2]. This removes `NaN` values from the end of each run entirely and replaces them with the previous non-`NaN` value when encountered during the experiment. Also removes channels. Implementation via[3] [Li+18]

dataio/opportunity/compute_time_windows_opp.py Extract sliding windows from the files generated by the previous step. Also performs clipping of extremely large values. Implementation via [Li+18]

dataio/opportunity/create_tf_dataset.py Create files in the `tf.dataset` format from the files obtained by the previous step.

dataio/opportunity_adapter.py Adapter for reading the `tfrecord` files for OPPORTUNITY within `tf.keras`

[1] obtained via https://www.eecs.qmul.ac.uk/mmv/datasets/deap/, a licence is required
[2] from https://archive.ics.uci.edu/ml/datasets/opportunity+activity+recognition
[3] using code from https://www.info.kindai.ac.jp/~shirahama/sensors/

Appendix C

Architectures

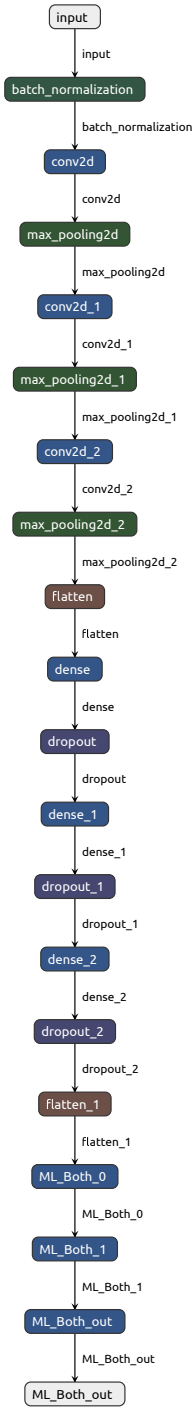
C.1 Single-Task Learning

Layer	Parameter	Value
input	shape	$(?, 64, 107, 1)$
conv_2d, conv_2d_{1,2}	kernel size filters activation	$\{(11, 1), (10, 1), (6, 1)\}$ {50, 40, 30} ReLU
max_pooling2d, max_pooling2d_{1,2}	pool size	$\{(2, 1), (3, 1), (1, 1)\}$
dense, dense_{1,2}	units activation	1000 ReLU
dropout, dropout_{1,2}	rate	0.4
ML_Both_{0,1}	units activation	500 ReLU
ML_Both_out	units activation	18 softmax

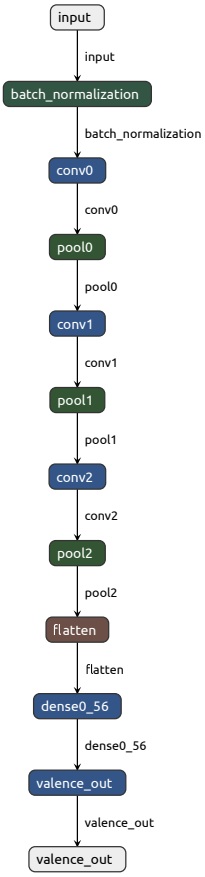
Table C.1: Architecture of the baseline STL network for OPPORTUNITY. The layer names refer to those in Figure C.1(a).

Layer	Parameter	Value
input	shape	$(?, 40, 128, 1)$
conv_{0, 1, 2}	kernel size filters activation	$(1, 8), (1, 6), (1, 4)$ 64, 32, 16 ReLU
pool_{0, 1, 2}	pool size	$(1, 4), (1, 3), (1, 2)$
{valence, arousal}_0	units activation	56 ReLU
{valence, arousal}_out	units activation	2 softmax

Table C.2: Architecture of the baseline STL network for DEAP. Label names refer to those in Figure C.1(b)



(a) OPPORTUNITY



(b) DEAP

Figure C.1: Baseline architectures for OPPORTUNITY and DEAP, generated using the application *Netron* from the architecture descriptions in JSON format generated by the respective scripts

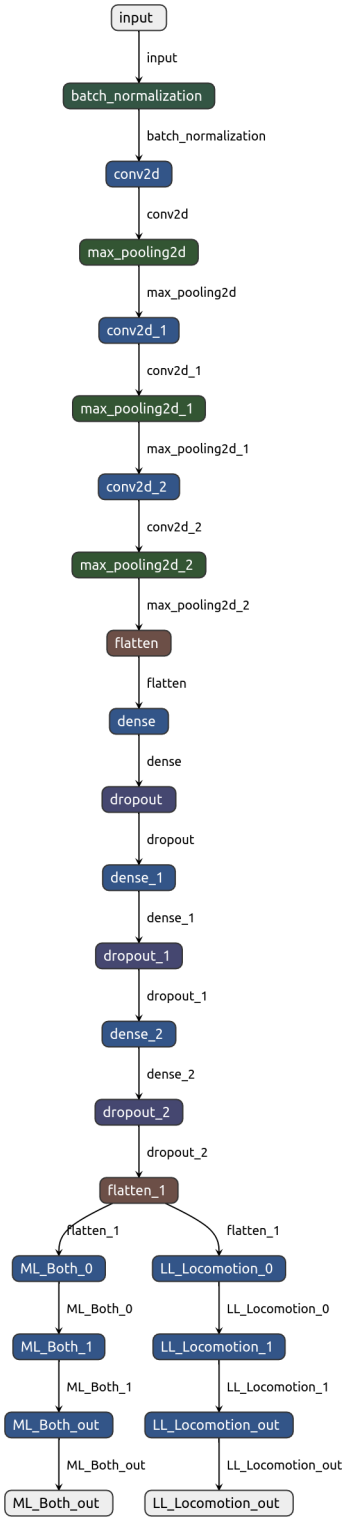
C.2 Hard Parameter Sharing

Layer	Parameter	Value
input	shape	$(?, 64, 107, 1)$
conv_2d, conv_2d_{1,2}	kernel size filters activation	$\{(11, 1), (10, 1), (6, 1)\}$ {50, 40, 30} ReLU
max_pooling2d, max_pooling2d_{1,2}	pool size	$\{(2, 1), (3, 1), (1, 1)\}$
dense, dense_{1,2}	units activation	1000 ReLU
dropout, dropout_{1,2}	rate	0.4
ML_Both_{0,1}, LL_Locomotion_{0,1}	units activation	500 ReLU
{ML_Both, LL_Locomotion}_out	units activation	$\{18, 6\}$ softmax

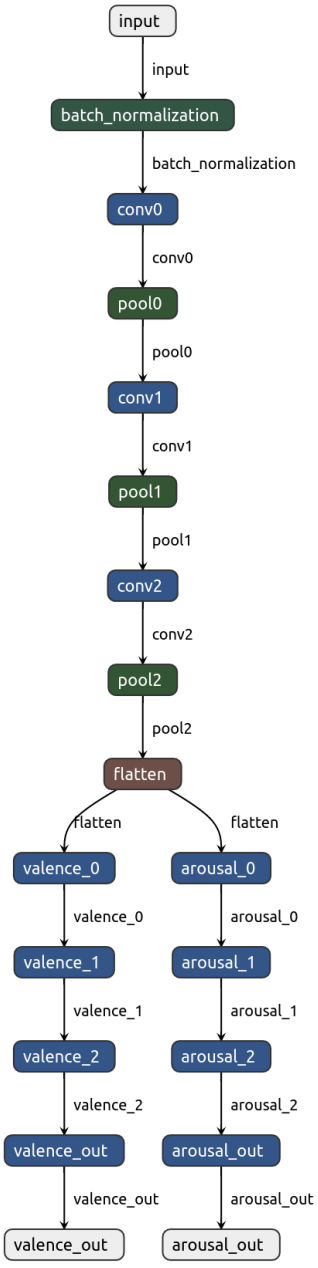
Table C.3: Architecture of the two-task network for OPPORTUNITY. The layer names refer to those in Figure C.2(a).

Layer	Parameter	Value
input	shape	$(?, 40, 128, 1)$
conv_{0, 1, 2}	kernel size filters activation	$(1, 8), (1, 6), (1, 4)$ 64, 32, 16 ReLU
pool_{0, 1, 2}	pool size	$(1, 4), (1, 3), (1, 2)$
[valence, arousal]_{0, 1, 2}	units activation	[48, 32, 16] ReLU
[valence, arousal]_out	units activation	2 softmax

Table C.4: Architecture of the best-performing two-task network for DEAP. Label names refer to those in Figure C.2(b)



(a) OPPORTUNITY



(b) DEAP

Figure C.2: Architectures of the HPS networks for OPPORTUNITY and DEAP. For OPPORTUNITY, networks with more than these tasks featured the same underlying architecture.

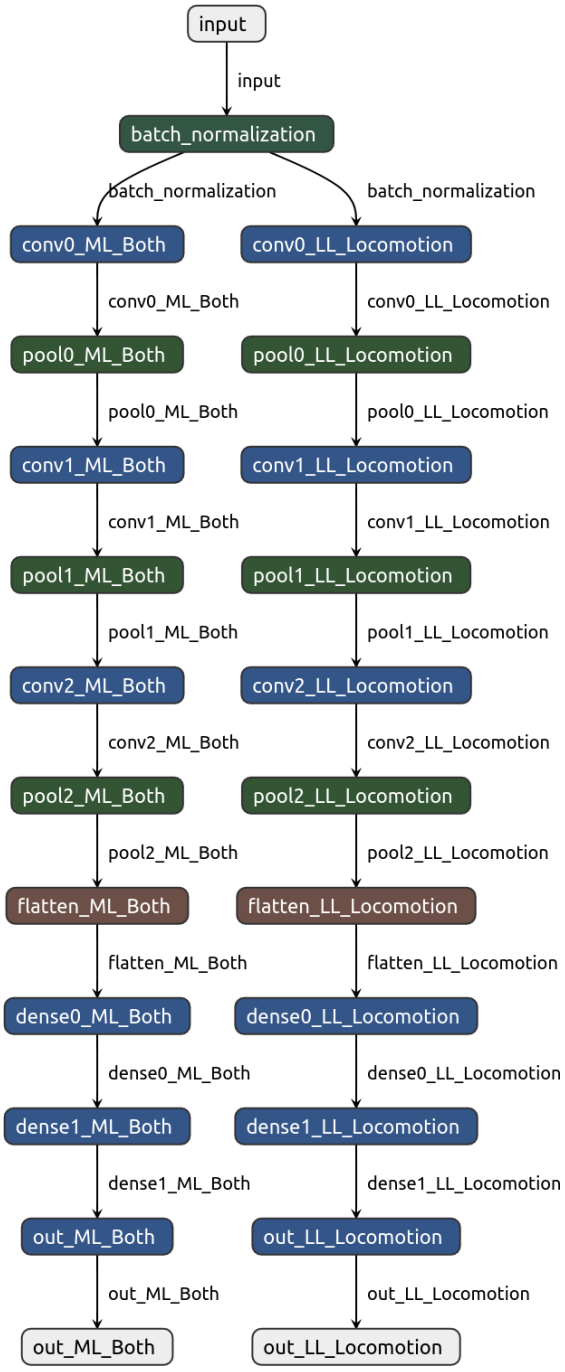
C.3 Soft Parameter Sharing

Layer	Parameter	Value
input	shape	$(?, 64, 107, 1)$
conv{0,1,2}_ML_Both, conv{0,1,2}_LL_Locomotion	kernel size filters activation	$\{(11, 1), (10, 1), (6, 1)\}$ {50, 40, 30} ReLU
pool{0,1,2}_ML_Both, pool{0,1,2}_LL_Locomotion	pool size	$\{(2, 1), (3, 1), (1, 1)\}$
dense{0,1}_ML_Both dense{0,1}_LL_Locomotion	units activation	{1000, 500} ReLU
out_{ML_Both, LL_Locomotion}	units activation	$\{18, 6\}$ softmax

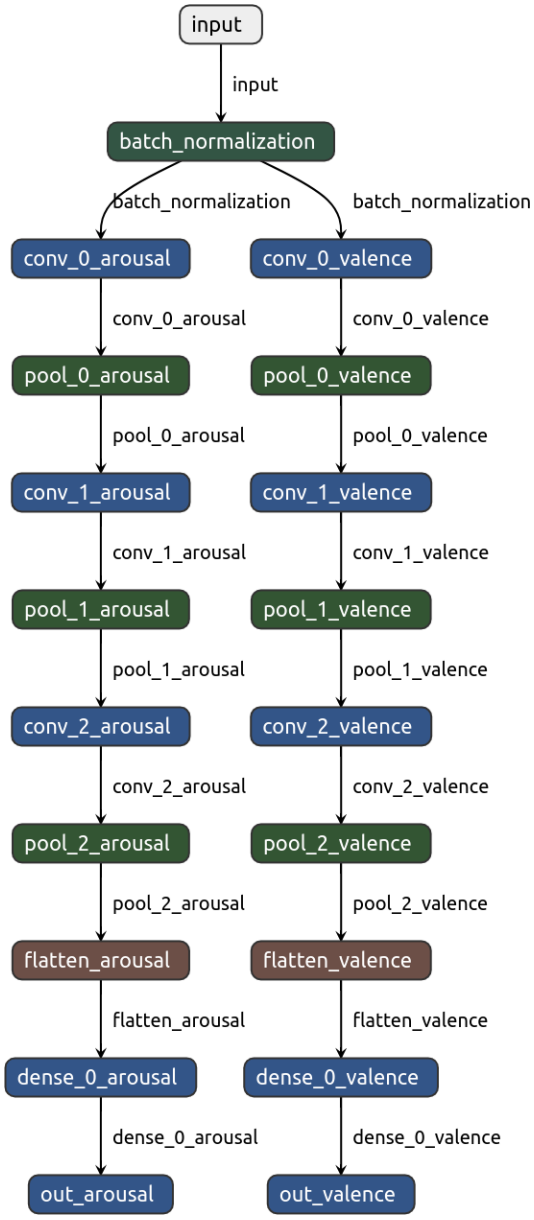
Table C.5: Architecture of the SPS network for OPPORTUNITY

Layer	Parameter	Value
input	shape	$(?, 40, 128, 1)$
conv_{0,1,2}_valence, conv_{0,1,2}_arousal	kernel size filters activation	$(1, 8)$, $(1, 6)$, $(1, 4)$ 64, 32, 16 ReLU
pool_{0,1,2}_valence, pool_{0,1,2}_arousal	pool size	$(1, 4)$, $(1, 3)$, $(1, 2)$
dense_0_{valence, arousal}	units activation	56 ReLU
[valence, arousal]_out	units activation	2 softmax

Table C.6: Architecture for the SPS for DEAP



(a) OPPORTUNITY



(b) DEAP

Figure C.3: Architectures for the SPS networks for OPPORTUNITY and DEAP

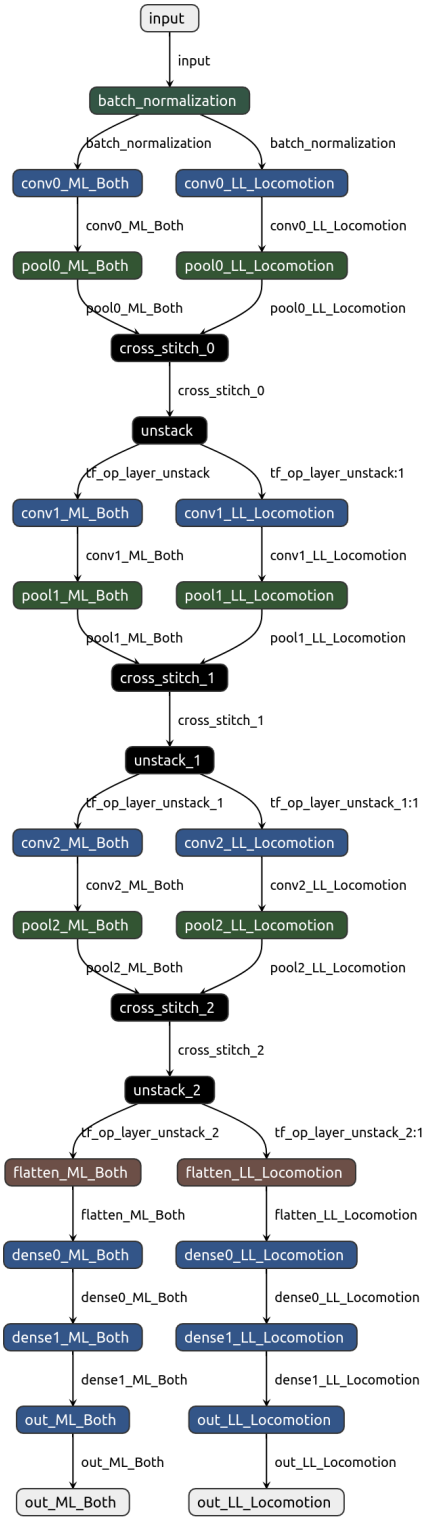
C.4 Cross-Stitch Networks

Layer	Parameter	Value
input	shape	$(?, 64, 107, 1)$
conv{0,1,2}_ML_Both, conv{0,1,2}_LL_Locomotion	kernel size filters activation	$\{(11, 1), (10, 1), (6, 1)\}$ {50, 40, 30} ReLU
pool{0,1,2}_ML_Both, pool{0,1,2}_LL_Locomotion	pool size	$\{(2, 1), (3, 1), (1, 1)\}$
dense{0,1}_ML_Both dense{0,1}_LL_Locomotion	units activation	{1000, 500} ReLU
out_{ML_Both, LL_Locomotion}	units activation	$\{18, 6\}$ softmax

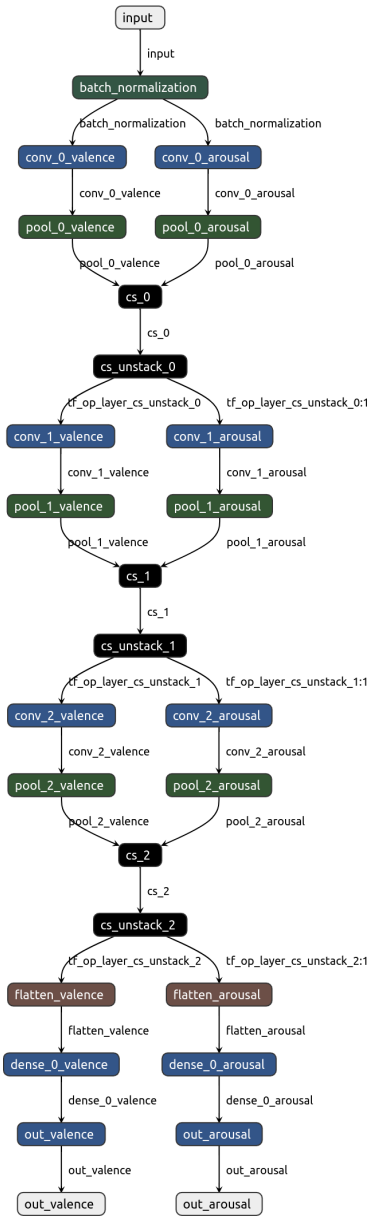
Table C.7: Architecture of the CSN for OPPORTUNITY

Layer	Parameter	Value
input	shape	$(?, 40, 128, 1)$
conv_{0,1,2}_valence, conv_{0,1,2}_arousal	kernel size filters activation	$(1, 8), (1, 6), (1, 4)$ 64, 32, 16 ReLU
pool_{0,1,2}_valence, pool_{0,1,2}_arousal	pool size	$(1, 4), (1, 3), (1, 2)$
dense_0_{valence, arousal}	units activation	56 ReLU
[valence, arousal]_out	units activation	2 softmax

Table C.8: Architecture for the CSN for DEAP



(a) OPPORTUNITY



(b) DEAP

Figure C.4: Architectures of the CSNs for OPPORTUNITY and DEAP.