Alma Mater Studiorum
Università di Bologna

Department of Computer Science and Engineering
M.S. in **Computer Engineering**
Master Thesis in **Artificial Intelligence**

Domain Adaptation through Deep Neural Networks for Health Informatics

Candidate:
Lorenzo Donati

Supervisor:
Chiar.mo Prof.
Federico Chesani

Assistant supervisor:
Ing. Luca Cattelani

Session II

Academic Year 2016-2017

Contents

Introduction

The *PreventIT* project is an EU Horizon 2020 project (PHC-21-2015) aimed at preventing early functional decline at younger old age and promote active and healthy ageing by using mobile health technology to develop and deliver a risk score for functional decline and a personalized intervention with exercise integrated in daily life [1]. The analysis of causal links between risk factors and functional decline has been made possible by the cooperation of several research institutes' studies. However, since each research institute collects and delivers different kinds of data in different formats, so far the analysis has been assisted by expert geriatricians whose role is to detect the best candidates among the hundreds of datasets' fields and offer a semantic interpretation of the values. This manual data harmonization approach is very common in both scientific and industrial environments.

In this thesis project an alternative method for parsing heterogeneous data is proposed. Since all the datasets represent semantically related data, being all made from longitudinal studies on aging-related metrics, it is theoretically possible to train an artificial neural network to perform an automatic domain adaptation. To achieve this goal, a Stacked Denoising Autoencoder has been implemented and trained to extract a domain-invariant [2] representation of the data. Then, from the high-level representation provided by the Stacked Denoising Autoencoder, a second neural network classifier (either a Multilevel Perceptron or a Long Short-Term Memory network) has been trained to validate the model and ultimately to predict the probability of functional decline of the patient. This innovative approach to the domain adaptation process can provide an easy and fast solution to many research fields that now rely on human interaction to analyze the semantic data model and perform cross-dataset analysis.

Functional decline classifiers show a great improvement in their performance when trained on the domain-invariant features extracted by the Stacked Denoising Autoencoder. Furthermore, this project applies multiple neural network classifiers on top of the Stacked Denoising Autoencoder representation, achieving excellent results for the prediction of functional decline in a real case study that involves two different datasets.

The first chapter will deliver a brief introduction to artificial neural networks and a detailed explanation of the mechanisms behind all the network models used in this project.

The second chapter will discuss the implementation of the the project and the methods used for the validation of the model.

Finally, the third chapter will show the experimental results achieved by the trained model.

Chapter 1

Neural Networks

An artificial neural network is a system of hardware and/or software built after the operations of the biological neural networks that form human brains. Such systems learn (i.e progressively improve performance) to perform tasks by analyzing examples, generally without task-specific programming.

Neural networks are a form of connectionism [3] made up of a number of simple, highly interconnected processing elements (also called neurons or nodes), which process information in parallel by changing their state dynamically in response to external inputs in order to provide new outputs. Each connection between neurons, called *synapsis* in biology, can transmit a signal in one direction. The receiving (or post-synaptic) neuron can process the signal and then propagate its results downstream to the neurons connected to it.

Although computers are currently able to carry out some tasks in a more efficient way than the human brain, computers are not yet capable of matching the brain's cognitive capacity, its flexibility, robustness and most importantly its energy efficiency [4]. Artificial neural networks (from now on just Neural Networks, ANN or NN) usually involve a large number of processors operating in parallel and arranged in tiers or layers. The first layer receives the raw input information and then the signal travels towards the last layer that delivers the output back. Different layers may perform different kinds of transformations on their inputs, collecting information through the learning phase and processing it to acquire a specific knowledge on the problem.

Neural networks have been used on a variety of tasks, including natural language processing, computer vision, medical diagnosis, sentiment analysis, playing boardgames and in many other domains.

From the system engineering point of view, a neural network is considered as a "black-box" as it imitates a behaviour rather than a structure and can reproduce any function. Studying the inner structure of the neural network does not typically provide to the user any useful information about the original system being imitated. The physical organization of the original simulated system is never taken into consideration, but only its outcome. An advantage of the neural network is that it behaves as a non-linear black box, modeling and describing virtually any linear or non-linear dynamic. However the disadvantage consists in the knowledge being represented in a sub-symbolic way, thus offering a limited interpretation to the end user. As far as conventional statistics are concerned, the neural network may be considered as a non-identifiable model [5] in the sense that various networks with different topologies and parameters may be defined to produce the same results.

Although ANN being a not so recent field of research, having the earliest papers on the subject published in the late 1940s, many of the topics and applications of neural networks have only now acquired maturity and consolidation thanks to the great improvement in computational power of the last decade. In fact, the spread of multi-core processing units, parallel programming and virtualization are three of the fundamental technologies which have led to modern implementations of neural networks.

Differently from conventional computing which is typically sequential, i.e. having a set of operations that have to be completed in a given order, neural networks are inherently parallels in their behaviour because a huge amount of neurons are activated at the same time and process the information independently one from another. They have proven to be very competitive in the resolution of real-world problems compared to more traditional data-analysis methods, usually based on explicit statistical modeling [6].

1.1 Supervised and unsupervised learning

Typically, a neural network is initially trained by feeding large amounts of data to it. Training consists in providing the inputs and informing the network with what the expected output should be, so that it can compare the generated output with the true one and reduce the error in the process. In machine learning, the aforementioned process is called supervised training and the provided outputs are usually called labels. For example, a machine could be provided with a series of human face pictures and a corresponding series of labels identifying the face as smiling or not smiling, so that it can be trained to classify the faces with the right class. Therefore, a classifier can be seen as an implementation of the mapping function

$$Y = f(x) \tag{1.1}$$

where x is the input picture and Y is the output label. The goal of the supervised training is to approximate the mapping function so well that when you have new input data, you can predict the output variables for that data without resorting to additional training.

There may also be some basic rules about objects' relationships in the space being modeled. For example, the above facial recognition system might be instructed and modeled with additional information like "the eyes are under the eyebrows" or "the nose is in the middle of the face and above the mouth". Preloading rules in the network can make training faster and make the model more powerful sooner. But it also builds in assumptions about the nature of the problem space, which may prove to be either irrelevant and unhelpful or incorrect and counterproductive, making the decision about what rules to build in very important [7].

On the opposite side, unsupervised learning is when you only feed the input data to the machine and no corresponding output labels. The goal for unsupervised learning is therefore to model the underlying structure or distribution in the data in order to learn more about the data. Unlike the supervised learning above, in this process there is no correct answers and there is no teacher (hence the name unsupervised learning). Algorithms are left to their own devises to discover and present the interesting structure in the data. Unsupervised training algorithms are even more sensitive than the supervised ones to any *a priori* knowledge offered to the system, because it could impair the learning process by inserting an erroneous and difficult to detect human bias into the system.

1.1.1 Competitive learning and reinforcement learning

Competitive learning (CL) and Reinforcement learning (RL) are two very popular training techniques that may offer a deeper understanding on the variety of supervised and unsupervised algorithms used in machine learning and artificial intelligence. These Techniques are interesting from an design point of view and therefore they are briefly described here, however they are not implemented in the final project because of their inherent limitations.

Competitive learning, a variant of Hebbian learning [8], is a form of unsupervised learning in which the nodes compete for the right to respond to a subset of the input data. During the training, each node of the network differentiates from the others by increasing its specialization. Competitive learning is well suited to find clusters within data and, for this reason, vector quantization and self-organizing maps (Kohonen maps [9]) are often based on this algorithms. The principle on which competitive learning is based is that, although lacking redundancy, a network in which each neuron recognize a different kind of pattern (or feature) is smaller and more efficient than a network in which an arbitrary number of neurons are responsible for the same feature. To achieve CL, the learning algorithm needs a mechanism that lets the neurons compete for the right to respond to a given subset of inputs, such that only one neuron for each layer is active at a time. The principle that lets only one neuron win the competition is called “winner-take-all” [10].

This mechanism is usually implemented by means of a similarity measure between the neuron and the input to be recognized. In this case, only the “*most similar*” neuron to the input gets the right to specialize for it (e.g. the neuron that has the least Euclidean distance between the

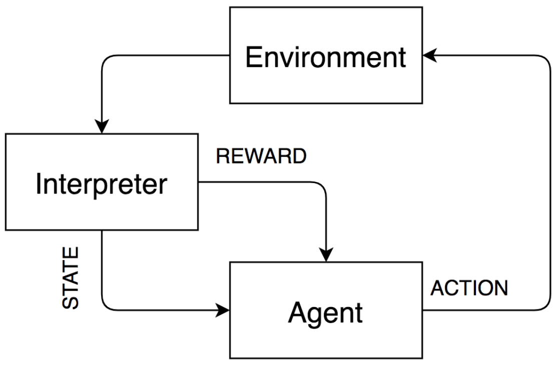


Figure 1.1: Reinforcement learning feedback loop

input vector x and its weight vector w, see Perceptron on page 8 for the definition of the node's weights).

One of the main problems of Competitive learning is obviously the limited redundancy of the derived system, which in turn can impair the robustness of the network to new stimuli not previously taken into consideration during training. Although many workarounds have been suggested in literature, they are outside the scope of this thesis and will not be discussed here.

Reinforcement learning, a technique inspired by behaviourist psychology, is a form of supervised learning in which the labels are not given to the system, but are discovered by it during the training process by acting on a certain environment [11]. In the operations research and control literature, the field where reinforcement learning methods are studied the most, this is called approximate dynamic programming [12].

Reinforcement learning is based upon the idea of having the system learn the correct group and order of operations to be executed in a physical or virtual environment in order to achieve a set goal. For example, a neural network could be built to play a game of Tetris and trained with Reinforcement learning in order to try postponing the *game over* screen (and as a consequence to play longer games).

Reinforcement learning differs from standard supervised learning because the correct input/output pairs are never presented, nor sub-optimal actions are explicitly corrected. Instead the focus is on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge) [13].

This form of "*learning-by-doing*" is more grounded in cognitive psychology than other supervised learning techniques, but requires a great amount of context to be added to the system in order to evaluate the actions' outcome on the environment and to provide the network (also called agent) with a proportionate reward (see figure 1.1).

1.2 Perceptron

The most basic form of neural network is made of only one neuron an is called Perceptron. Introduced for the first time in 1958 by Frank Rosenblatt [14], the Perceptron can hardly be called a neural *network* since it lacks any form of connectionism. However, it is an important case of study because all the other neural networks can just be seen as complex ensembles of connected Perceptrons. As previously stated, artificial neural networks are modeled after their biological counterparts, but with some important differences. Just like a biological neuron has dendrites to receive signals, a cell body to process them, and an axon to send signals out to other neurons, the artificial neuron has a number of input channels, a processing stage, and one output that can fan out to multiple other artificial neurons, as shown in figure 1.2.

However, the artificial neuron's output is modulated in amplitude while the biological one is modulated in frequency. Biological neurons propagate the information by releasing a certain fixed amount of chemical messengers called neurotransmitters. The more intense the signal, the more frequent the release of neurotransmitters is, but not their quantity. Although there are some neural networks modeled precisely on this kind of transmission, called Spiking Neural Networks [15], the great part of the neural networks currently in use have neurons whose output signal is a numeric value that increases or decreases with the intensity of the neuron's

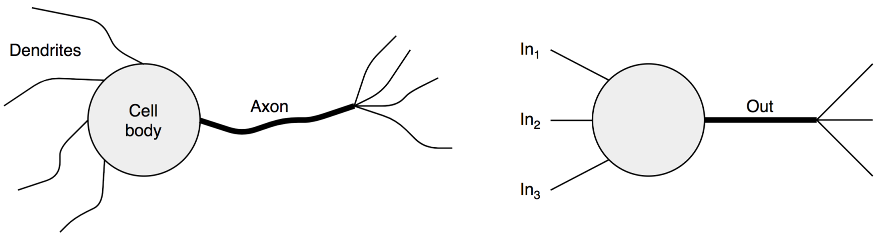


Figure 1.2: Artificial and biological neurons comparison

activation.

More specifically, a Perceptron can be seen as a linear classifier defined by a vector of weights and a bias, that takes an input vector and gives back a single numeric value as output. As shown in equation 1.2, the output is just the sum of the bias (b) and the dot product between the input vector (x) and the weights vector (w).

$$Y = (\sum_{i=1}^{n} w_i x_i) + b \tag{1.2}$$

Mathematically speaking, the Perceptron represents an hyperplane that divides the input space into two sections: one where the output is greater than zero and one where it is lesser than zero. From a spatial point of view, the weights alter the orientation of the hyperplane while the bias alters the position (though not the orientation) of that decision boundary, making the hyperplane independent from the origin point of the domain. During the training phase, the weights and bias are initialized at first with random values and then the entire input dataset is passed through the Perceptron one record at a time, confronting its output with the expected one. At each step (t), if the label differs from the Perceptron output, its weights are updated according to the equation 1.3 where c^t is the expected label and y^t is the provided one.

$$w_i^{t+1} = w_i^t + x_i(c^t - y^t) \tag{1.3}$$

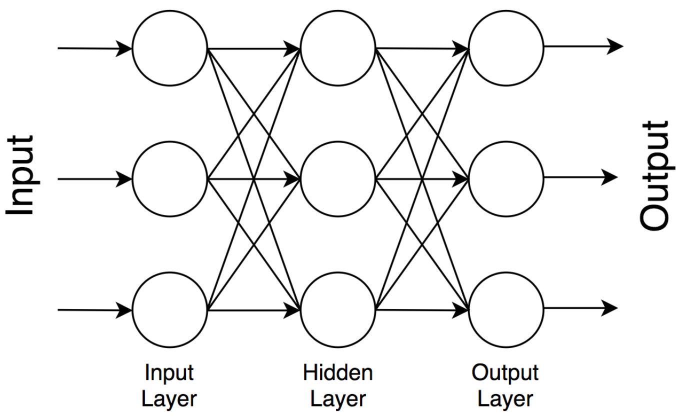


Figure 1.3: Multilayer perceptron

The bias is also updated in a similar manner, taking into consideration that $x_b = 1$ for each record because the bias is not multiplied by any element of the input vector. The training usually ends when all the input items can be successfully classified. Obviously, the Perceptron offers a very limited form of binary classification and the training process converges only when the two classes are linearly separable in the input space, because otherwise the weights will swing back and forth and the Perceptron will never be able to classify all the items at once.

1.3 Multilayer perceptrons

The next logical step from the simple Perceptron to achieve non-linear classification is to connect multiple Perceptrons in layers as seen in figure 1.3, this is called a Multilayer Perceptron (MLP).

Even if the stacking of Perceptrons alone is already able to separate more complex regions of the input space, it's not enough to approximate a non-linear function. Linear algebra shows that any number of layers built this way can be reduced to a two-layer input-output model. For this reason a non-linear activation function is introduced to compute the actual output of each node. The final output of a neuron is therefore

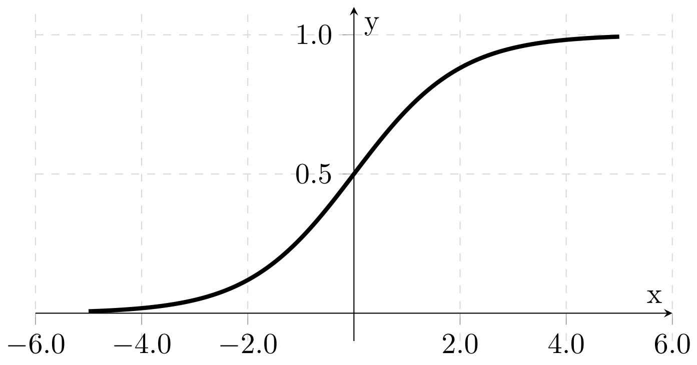


Figure 1.4: Logistic function

described in the equation 1.4 where f is the neuron's activation function.

$$Y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{1.4}$$

Each neuron of a given layer takes as input the the post-activation values of the previous layer, then multiplies that vector for its own weights vector, adds the bias and finally computes the activation function on the weighted sum. The resulting output is ultimately forwarded to the next layer until it reaches the final output layer whose vector of values is used for the classification, typically by having each output node representing a different class.

1.3.1 Activation functions

The two most common activation functions are both sigmoids, and are respectively the **hyperbolic tangent** ($f(x) = tanh(x)$), that ranges from -1 to 1, and the **logistic function** ($f(x) = (1 + e^{-x})^{-1}$) which is similar in shape but ranges from 0 to 1, as shown in figure 1.4.

Sigmoid functions have several useful properties because they are real-valued, bounded to a limited range (e.g. [-1,1]), monotonic, and differentiable having at each point a non-negative first derivative, which is bell shaped. A sigmoid function is also constrained by a pair of horizontal asymptotes as $x \to \pm\infty$, being defined for each $x \in \mathbb{R}$. Other activation functions proposed in literature and relevant to this thesis project are the ReLU, the Softplus and the Softmax.

ReLU, or rectified linear unit, is an activation function defined as $f(x) = max(0, x)$. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering. This activation function was first introduced to a dynamical network by Hahnloser et al. in a 2000 paper in Nature with strong biological motivations and mathematical justifications [16]. The ReLU function is scale-invariant ($amax(0, x) = max(0, ax)$), computationally efficient (only comparison, addition and multiplication), and offers a sparse activation that helps avoiding over-sized neural networks (eg. in a randomly initialized network, only about 50% of hidden units are activated, having a non-zero output) [17].

However, ReLU also has some serious limitations, as it's not differentiable at zero (it is differentiable anywhere else, including points arbitrarily close to zero but not equal to it) and suffers from the "*Dying ReLU*" problem: ReLU neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. In this state, no gradient flows backward through the neuron (see Backpropagation at page 16), and so the neuron becomes stuck in a perpetually inactive state and "dies". In some cases, large numbers of neurons in a network can become stuck in dead states, effectively decreasing the model capacity. This issue typically arises when the learning rate (see equation 1.10) is set too high. The Dying ReLU problem may be mitigated by using a Leaky ReLUs instead, defined as $f(x) = max(ax, x)$ with $0 < a < 1$.

Softplus is a smoother version of the ReLU defined as $f(x) = ln(1 + e^x)$. The Softplus function maintains all the positive characteristics of the ReLU and it is also differentiable at zero. However, it is less efficient to be computed. Its difference in value from the ReLU is more evident around zero and get thinner as it approaches $\pm\infty$, as shown in figure 1.5. The Softplus derivative is exactly the logistic function previously discussed ($f(x) = (1 + e^{-x})^{-1}$).

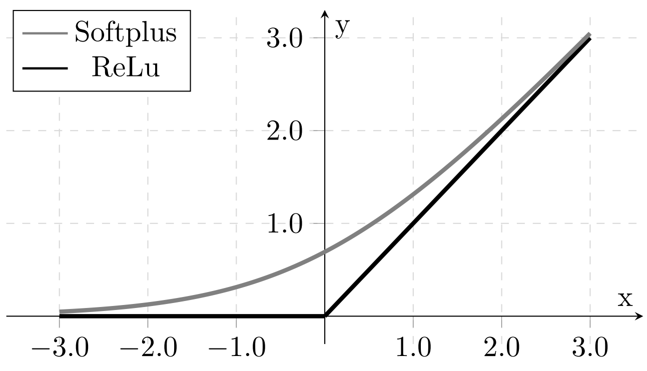


Figure 1.5: ReLU and Softplus comparison

Softmax, on the other end, is substantially different from all the previously discussed activation functions as it considers every neuron's output in a given layer. The post-activation output of each neuron therefore depends on both the value of its weighted sum and the value of every other neuron in the same layer. The Softmax function, or normalized exponential function, is a generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values into a $n-$dimensional vector $\sigma(z)$ of real values in the range [0, 1] that add up to 1, as described in the equation 1.5.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}} \qquad for \quad j = 1, \dots, n \tag{1.5}$$

Softmax function is exceptionally good when used in the output layer of a neural network classifier for a multiclass classification problem, i.e. a problem in which the system must discriminate among several classes, but the label of each input item is defined as a single class only. The reason being that the label can be projected as a one-hot vector (i.e. a vector where only one element is 1 and the rest are zeros) with each position in the vector representing a different class, while the neural network can be defined with a number of output nodes equals to the number of classes. This way, when the Softmax function is applied to the output layer, the resulting vector is shaped as a probability distribution of the input item over every class, having the sum of all the elements equal to 1 (because it must belong to one of the classes) and each element representing the probability of the respective class for that item.

1.3.2 Loss functions

Differently from the simple Perceptron, a MLP can have an arbitrary number of nodes in the output layer. For this reason, a loss function is needed to compare the expected label with the system output and evaluate the relative distance. Having a robust loss function is very important for a neural network, since its weights and biases are updated depending on the amplitude of the error. The two relevant loss functions for this thesis project are the RMSE and the Cross-entropy.

RMSE, or root-mean-square error, is a frequently used measure of the difference between values. The RMSE represents the standard deviation of the differences between predicted values and observed values. These individual differences are typically called residuals when the calculations are performed over the data that was used for the training and are called prediction errors when computed on the test set. As seen in the equation 1.6, the RMSE aggregates the magnitudes of the prediction errors into a single measure of predictive performance. The resulting value can be used as a measure of accuracy to compare prediction errors of different models on a given dataset, but not between datasets, as it is scale-dependent.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y^t - c^t)^2}{n}} \tag{1.6}$$

Although RMSE is one of the most commonly reported measures of disagreement, it should not be misinterpreted as average error, which RMSE is not. RMSE is the square root of the average of squared errors, thus RMSE confounds information concerning average error with information concerning variation in the errors. The effect of each error on RMSE is proportional to the size of the squared error, therefore larger errors have a disproportionately large effect on RMSE. Consequently, RMSE is sensitive to outliers.

Cross-entropy measures the average number of bits needed to identify an event drawn from the dataset, if a coding scheme is used which is optimized for an arbitrary probability distribution q, rather than the "true" distribution p of the dataset. In information theory, the Kraft –McMillan theorem [18] establishes that any directly decodable coding scheme, used on a signal to identify one value x_i out of a set of possibilities X, can be seen as representing an implicit probability distribution $q(x_i) = 2^{-l_i}$ over X, where l_i is the length of the code for x_i in bits. Therefore, cross entropy can be interpreted as the expected message length per datum when a wrong distribution q is assumed while the data actually follows a distribution p.

Given the two probability distributions p and q, the Cross-entropy between them is defined as:

$$CE(p,q) = H(p) + D_{KL}(p||q) \tag{1.7}$$

Where *H(p)* is the entropy of the true distribution p and $D_{KL}(p||q)$ is the Kullback–Leibler divergence [19] of q from p, also known as the relative entropy of p with respect to q. As discussed in the previous chapter, it is possible to transform both the expected label of a classification problem and the output of a neural network classifier to represent a discrete probability distribution over the set of classes. In this case, it is possible to write the Cross-entropy equation 1.7 in a discrete form:

$$CE(p,q) = -\sum_{k=1}^{C} p_k(x) \log q_k(x) \tag{1.8}$$

Where C is the number of different classes in the dataset, $q_k(x)$ is the probability of the item x belonging to the class k (i.e. the output of the k-

th output neuron), and $p_k(x)$ is the true probability of that item belonging to the class k as stated in the label (either 0 or 1).

Contrary to the RMSE, which is a metric measure and takes into consideration the difference between all the classes represented by the label and the output vectors, the Cross-entropy is not metric, because it does not satisfy the symmetry condition (i.e. $CE(a, b) \neq CE(b, a)$), and it is much more focused on the true class (i.e. the only one where $p_k(x)$ is not equal to zero), giving less emphasis to the wrong ones. To show why this may be important, consider the following working example: a neural network with three output nodes is trained to discriminate among three different classes (A, B and C). The item x belongs to class A and therefore its one-hot label is [1, 0, 0]. Supposing the network's output after Softmax is [0.6, 0.2, 0.2], by applying the Cross-entropy and RMSE equations you get a loss value of ∼0.222 and ∼0.283 respectively. However, if the output were [0.65, 0.35, 0] instead, the Cross-entropy would be lower as expected (∼0.187) since the item x is being classified as belonging to class A with a lower uncertainty, but the RMSE value would be higher than the previous one (∼0.286) because the elements of the vector corresponding to the wrong classes are more unbalanced.

1.3.3 Backpropagation and Gradient Descent

Training a multi-layered neural network is much more difficult than a simple Perceptron. As stated at page 10, during the training the information flows from the input nodes to the output nodes one layer at a time. Each neuron of a given layer gets the output of the previous layer's nodes as input and gives its computed output to the next layer. The signal is therefore fed forward to each subsequent layer, hence the name Feedforward Neural Network (FNN) generically used to refer to all the neural network's models where the information travels in one direction only without cycles or loops from the input to the output.

Once the output is computed for a certain input element, and the loss function is evaluated as well, the weights of the network's nodes are adjusted proportionally to the loss value and their contribution to it. This contribution is calculated backwards from the final layer to the first one and this process takes the name of **backpropagation**.

Measuring the contribution on the final loss of each weight of a deep neural network (i.e. a neural network with several layers between the input and output ones) is not a trivial task. The first algorithm that had successfully addressed the deep learning problem is called **Gradient Descent**.

The Gradient Descent (also called steepest descent [20]) is a first-order iterative optimization algorithm for finding the minimum of a function that tries to update the weights in several steps to gradually find the best ones. In the case of neural networks, the Gradient Descent algorithm updates at each step the weights proportionally to the negative of the gradient (or of the approximate gradient) of the loss function at the current point. The reason why it is important that the previously discussed activation functions have to be differentiable is that if a multi-variable function LF is defined and differentiable in the neighborhood of a certain point α, with LF being the loss function and its variables being all the weights of the network, then LF decreases fastest if one goes from α in the direction of the negative gradient of LF at α $(-\nabla LF(\alpha))$. The gradient is simply the vector of the first-order partial derivatives of LF with respect to each weight $w_j \in w$:

$$\nabla LF = [\frac{\partial LF}{\partial w_0}, \frac{\partial LF}{\partial w_1}, \dots, \frac{\partial LF}{\partial w_m}] \tag{1.9}$$

Therefore, progressing step by step, the loss value decreases if the vector of weights is updated using the equation:

$$w^{t+1} = w^t - \eta \nabla LF(w^t) \tag{1.10}$$

Where η is an arbitrary small correction parameter called **learning rate** that prevents the network from overshooting the function minimum.

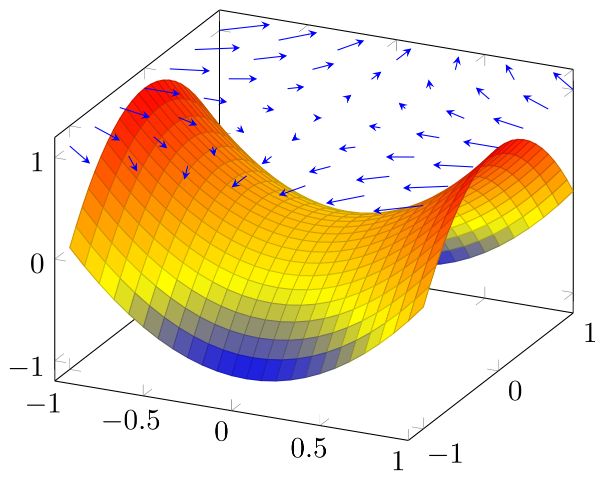


Figure 1.6: Gradient Descent representation: the length and direction of the arrows in the top plane represent the inverse gradient at the corresponding point of the loss function

If the loss function decreases at each step, the Gradient Descent algorithm is guaranteed to converge to a minimum. However, as shown in figure 1.6, only if the loss function is convex the found minimum is guaranteed to be a global optimum, otherwise it is only a local minimum. It is also important to note that the rate of convergence slows down reaching an asymptote as the function approaches the minimum, because the gradient shrinks smaller at each step. This gradual reduction of the learning rate is often called "annealing".

Stochastic gradient descent

Normally, the Gradient Descent algorithm is not computed with respect to the loss value of each single item in the dataset, but on the average of the losses on all the items together. Each time an optimization algorithm analyzes all the items inside the dataset, it is called an epoch. Averaging the loss value grants the algorithm correctness, but the resulting value is much smaller in magnitude and therefore the rate of convergence is very slow. Especially when used on a very large dataset, a learning methods that involves checking every item in it to compute a

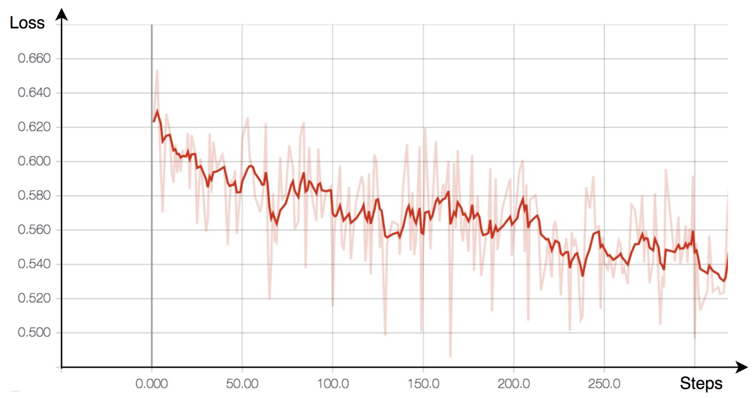


Figure 1.7: Stochastic Gradient Descent loss value

very small optimization step is too slow to be effective.

A useful alternative to the standard Gradient Descent is the **Stochastic Gradient Descent** (SGD), which trades off universal correctness for a much higher speed and a better scalability. SGD is a stochastic and iterative approximation of the Gradient Descent. The statistically-valid middle ground between computing the true gradient on the whole dataset and computing it over a single item, which would be too sensitive to the outliers, is to randomly select few items to form a batch at each step of the training and compute the gradient only on those items in order to update the network's weights. The selection can be either done at each step by picking a random subset of the items that were not previously considered during the training, requiring the appropriate data structures, or by just shuffling the dataset at the start of the epoch and then taking at each step a sequential batch of items from it.

As shown in figure 1.7, the loss optimization of SGD is not as smooth as it would be for standard Gradient Descent, having spikes when the selected batch diverges from the global trend. However, on a larger scale it still converges to a local minimum. The convergence of Stochastic Gradient Descent has been analyzed using the theories of convex minimization and of stochastic approximation. Briefly, when the learning rates η decrease with an appropriate rate, Stochastic Gradient Descent converges almost surely to a global minimum if the objective function is convex or pseudo-convex and converges almost surely to a local minimum otherwise. This is in fact a consequence of the Robbins-Siegmund theorem [21].

Adam optimizer

Gradient Descent is not the only optimization algorithm used to train neural networks. A more complex variation is the Adam optimizer, or Adaptive Moment Estimation [22]. Differently from Gradient Descent, which takes in consideration only the first-order partial derivatives, Adam computes at each step the averages of both the gradients and the second moments of the gradients. The update equation of the weights (w) at each step of the Adam algorithm is:

$$\begin{aligned} m^{t+1} &= \beta_1 m^t + (1-\beta_1)\nabla LF(w^t) \\ v^{t+1} &= \beta_2 v^t + (1-\beta_2)(\nabla LF(w^t))^2 \\ w^{t+1} &= w^t - \eta\, \frac{m^{t+1}}{1-\beta_1} \left(\sqrt{\frac{v^{t+1}}{1-\beta_2}} + \epsilon \right)^{-1} \end{aligned} \tag{1.11}$$

Where ϵ is a small number used to prevent division by zero, and β_1 and β_2 are the forgetting factors for gradients and second moments of gradients, respectively. Adam is particularly appropriate for non-stationary objectives and problems with very noisy or sparse gradients, since it also computes the second moments, and empirical results demonstrate that it is very fast, is not computationally expensive, and compares favorably to other stochastic optimization methods.

Like Gradient Descent and Adam, there are many more state-of-the-art optimization algorithms currently in use for deep learning, but they are not discussed here as they are not relevant to this thesis project.

Exploding and vanishing gradients

The biggest difficulty found in the training of deep artificial neural networks with gradient-based learning methods and backpropagation is that the gradient is likely to approach either zero or grows exponentially

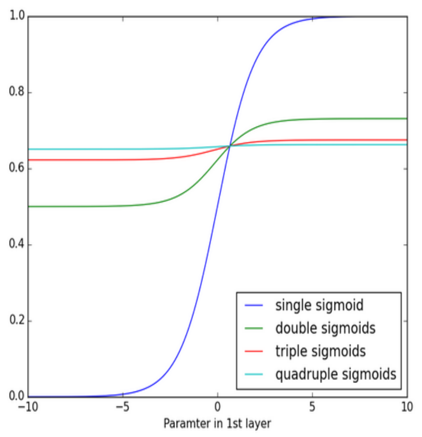


Figure 1.8: Vanishing gradient in composed logistic functions, courtesy of https://deeplearning4j.org [23]

as it get propagated back to the first layers. This dual problem is known as vanishing (or exploding) gradients. Traditional activation functions such as the hyperbolic tangent have outputs in the range $[-1, 1]$, and backpropagation computes gradients following the derivative chain rule $(f \circ g)' = (f' \circ g)g'$ (where $\circ$ is the function composition, i.e. $f \circ g(x) = f(g(x))$). As the activation functions get composed together by the chain rule, the variation of the resulting output grows smaller with each new layer, as shown in figure 1.8, and the slope becomes undetectable. For this reason, the partial derivative of the error with respect to a weight of the first layer of a deep neural network results in the multiplication of several small values, thus approaching zero. Since the gradient with respect to the weights of the first layers decreases exponentially with the depth of the network, their training becomes slower and slower as the network is being elongated to understand even more complex patterns in the input data.

1.4 Autoencoders

Among the many feedforward neural networks topologies, the most commonly used for unsupervised data analysis is the Autoencoder (AE), or

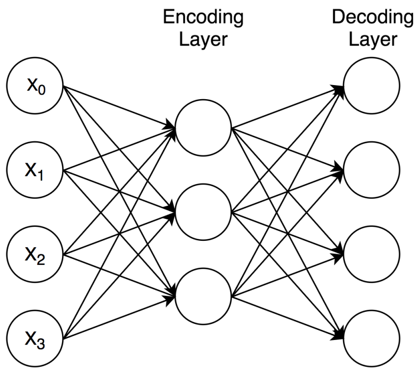


Figure 1.9: Basic Autoencoder

Diabolo Network. The aim of an Autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction. From a mathematical point of view, the Autoencoder tries to approximate an encoding function $h(x)$ and the respective decoding function $g(x)$.

Architecturally, the simplest form of an Autoencoder is very similar to a Multilayer Perceptron, but having the same number of nodes in both the input layer and the output layer while the hidden layer is dimensionally smaller. The hidden layer is called encoding layer and its output is the reduced representation $z = h(x)$ (also called code or image) of the input item. The output layer, or decoding layer, reconstruct the original item from the information stored in its image by applying the decoding function $x' = g(z)$ and then gives the result back to the optimization algorithm in order to measure the error and train the network. Being both functions represented by neural network layers, their final equation are:

$$\begin{aligned} z &= f(wx + b) \\ x' &= f'(w^T z + b') = f'(w^T f(wx + b) + b') \end{aligned} \tag{1.12}$$

Where f and f' are the activation functions of the two layers, b and b' are the two bias vectors, and w and w^T are respectively the weights matrix of the encoding layer and its transpose used by the decoding layer.

An Autoencoder is ultimately trying to learn an approximation to the identity function, so as to output a x' that is as similar as possible to the original x. The identity function seems a particularly trivial function to learn, but by placing constraints on the network, such as by limiting the dimension of the coded image, it is possible to discover interesting structures about the data. At the end of the training phase, the Autoencoder is split in two separate networks: the encoding layers works essentially as a feature extractor that offers a sub-symbolic representation of the input item, while the decoding layer can be used as a generative network by feeding it with random noise [24].

1.4.1 Stacked Autoencoders

The deep neural network version of the Autoencoder is called Stacked Autoencoder (SAE). In a Stacked Autoencoder, both the encoding function and the decoding function are implemented by a deep feedforward neural network with several layers. Each decoding layer's weights matrix is always the transpose of the weights matrix of the encoding layer located at the same distance from the image (e.g. when having k encoding layers and k decoding layers, the weights matrix of the decoding layer at depth i would be $w_i^{dec} = w_{k+1-i}^T$).

Stacked Autoencoders are usually trained in a two-step process. Initially, the network is pre-trained with a greedy layer-wise approach by training each layer in turn. In this phase, in order to train the $(n+1)-$layer, the items (x) of the dataset are encoded through all previous layers and the resulting images are used, as shown in the following equation.

$$\begin{aligned} y &= h_n(h_{n-1}(\dots h_2(h_1(x))\dots)) \\ y' &= g_{n+1}(h_{n+1}(y)) \end{aligned} \tag{1.13}$$

The reconstructed image (y') is then confronted with the image computed by the previous layers (y) to estimate the reconstruction error and update the weights of the $(n+1)-$layer accordingly. The pre-training helps the

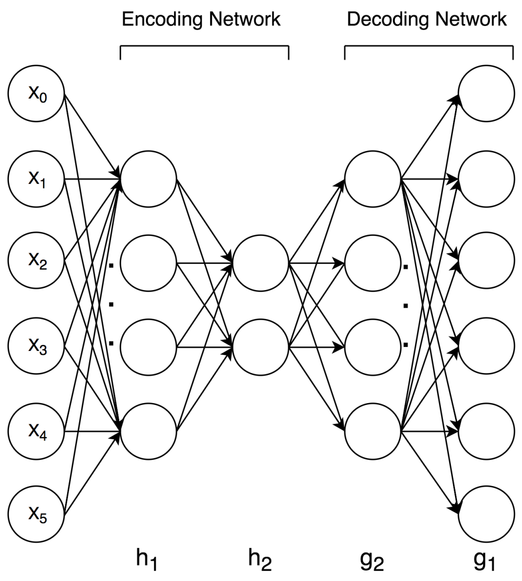


Figure 1.10: Stacked Autoencoder

network to avoid the exploding and vanishing gradients problem (see page 20) as demonstrated by J. Schmidhuber [25].

The second training phase, usually called fine-tuning, treats all layers of the Stacked Autoencoder as a single deep neural network model and tries to optimize its global behaviour. In this phase, the original items are encoded using all the encoding layers, then decoded using all the decoding layers and finally the reconstructed data is confronted with the original one. During the fine-tuning, the optimization algorithm updates all the network's weights at once, trying to find a minimum of the reconstruction error which in turn grants a better representation of the data through their coded image. As per standard Autoencoders, after the training of a Stacked Autoencoder the network is split in two and only the encoding layers are used to address dimensionality reduction problems.

1.4.2 Denoising Autoencoders

If the decoding layer of an Autoencoder has the same dimension of the input, it is theoretically possible to exactly map each value of the input vector on a different encoding node (i.e. the weights vector for each node is made of all zeros but a 1 corresponding to the mapped input). If the goal of the Autoencoder is to discover and extract interesting structures in the data instead of representing it in a lower-dimensional space, it is possible to use another technique which does not require to reduce the number of nodes to avoid trivial mapping. If the input is corrupted by a random noise before entering the encoding layer, the resulting Autoencoder can be trained to refine the data in order to retrieve the original item and for this reason is called a **Denoising Autoencoders** (DAE).

During the training of a Denoising Autoencoder, the loss value is computed with respect to the difference between the original uncorrupted data and the reconstructed output from the corrupted item, as shown in the following equation.

$$\begin{aligned} \tilde{x} &= noise(x) \\ \tilde{x}' &= g(h(\tilde{x})) \\ loss &= LF(x, \tilde{x}') \end{aligned} \tag{1.14}$$

Since the network output is never directly compared to its input, but only to the original uncorrupted data, the encoding layer can work as a robust feature extractor for an arbitrary large number of features once the training has ended. For the extraction of even more complex structures, it is possible to use a Stacked Denoising Autoencoder (SDAE) by adding the noise both during the layer-wise greedy pre-traing and later during the final fine-tuning.

1.5 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are network models where the connections between nodes may form directed cycles. Differently from the feedforward neural networks discussed so far, the layers of a recurrent neural network can propagate the information both forward to the next layer and backward to a previous one. Usually, the recurrent neural networks are still modeled in a hierarchical way, but with the output layer

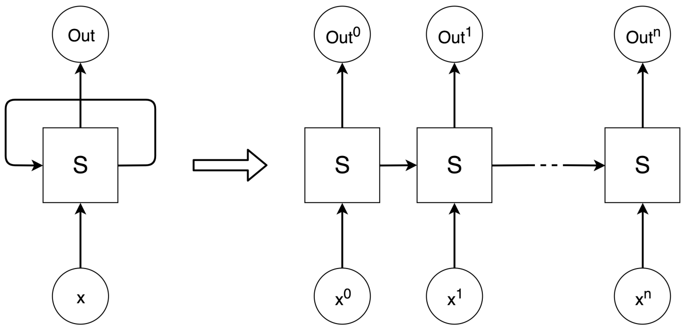


Figure 1.11: Recurrent Neural Network unfolding

sending the signal back to the input layer forming a cycle. The feedback introduced by the cycle allows the network to exhibit dynamic temporal behavior.

Since they can take time and sequencing into consideration during the training, the recurrent neural networks are often used to address time-dependent problems like speech recognition, stock analysis and even genome sequencing.

The simplest recurrent neural network is made by only one layer that takes in input the concatenation of the original item and the network's own output at the previous step. In practice, for a sequence of n items, the previous model is equivalent to n single-layer forward neural networks with the same weights and biases, each one sequentially trained with the output of the previous network and a new item of the sequence. This representation of a recurrent neural network as a sequence of feedforward neural networks, as shown in figure 1.11 is called *unfolding.* The feedback signal internally passed from one step to the next one is usually called *cell's state.* Very deep learning sequencing tasks that would require up to 1000 unfolded layers are nearly impossible for a standard feedforward neural network, but are solvable with a recurrent neural network [26].

Training a recurrent neural network is similar to train a traditional neural network. Backpropagation with Gradient Descent algorithm is still used, but it is adjusted with some slight differences. Since all the parameters (weights and biases) are shared by every time step in the network, the

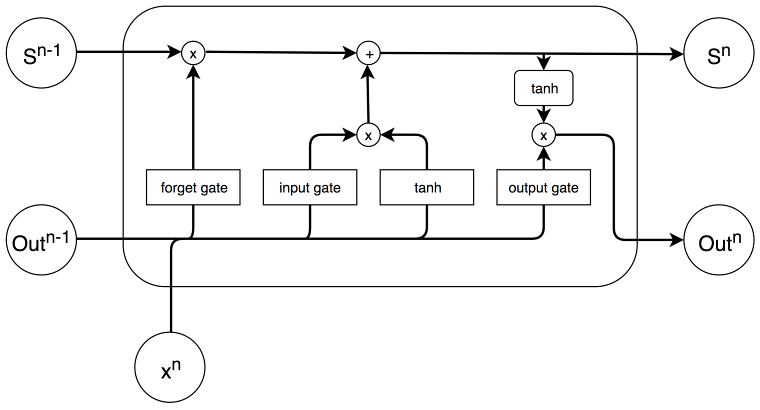


Figure 1.12: Long Short-Term Memory cell

gradient at each output depends not only on the calculations of the current time step, but also on the previous time steps (e.g. in order to calculate the gradient at $n = 5$, the algorithm needs to back-propagate four steps and sum up all the gradients together). This process is called **Backpropagation Through Time** (BPTT). Obviously, Backpropagation Through Time suffers even more from exploding and vanishing gradients (see page 20) than the standard Backpropagation because the total depth of the network is equal to the number of its layers multiplied by the length of the input sequence.

1.5.1 Long short-term memory

In the 1997, a variation of recurrent neural networks called Long Short-Term Memory (LSTM) was proposed by Sepp Hochreiter and Juergen Schmidhuber [27] as a solution to the vanishing gradient problem. LSTM helps to preserve the error that can be then back-propagated through time and layers. By maintaining a more constant error, it allows recurrent networks to continue to learn over several time steps, thereby also opening a channel to link causes and effects very distant in time.

Long Short-Term Memory models contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, like data in a computer's memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Unlike the digital storage on computers, however, these gates have a continuous value in the range [0, 1], because they are implemented as feed-forward neural networks with one output node activated by an **logistic function**. This continuous representation has the advantage over the digital representation {0, 1} of being differentiable, and therefore suitable for backpropagation [23].

A Long Short-Term Memory cell, as shown in figure 1.12, is composed of three main gates: an input gate (IG), an output gate (OG) and a forget gate (FG). Their equations for the $n-$th element of the sequence are:

$$\begin{aligned}
FG^n =& f_{LF}(w_{FG}\, x^n + u_{FG}\, Out^{n-1} + b_{FG})\\
IG^n =& f_{LF}(w_{IG}\, x^n + u_{IG}\, Out^{n-1} + b_{IG})\\
OG^n =& f_{LF}(w_{OG}\, x^n + u_{OG}\, Out^{n-1} + b_{OG})\\
S^n =& FG^n \bullet S^{n-1} + IG^n \bullet f(w_S\, x^n + u_S\, Out^{n-1} + b_S)\\
Out^n =& OG^n \bullet f(S^n)
\end{aligned} \tag{1.15}$$

Where x is the input item, Out is the output vector of the cell, S is the cell's state, the operator $\bullet$ denotes the element-wise Hadamard product, f_{LF} is the logistic activation function, and the initial values for S^0 and Out^0 are both equal to zero. The matrices of weights w, u, and the bias vectors b are the elements of the cell that will be updated by the optimizer during the training process. The core mechanism behind the success of Long Short-Term Memory networks is the plus operator in the cell's state update equation. By having an additive and not multiplicative operation, the error can be propagated back from the output regardless of the the time component, remaining in the block's memory without the risk of fading until the network learns how to cut off its value. Thanks to this internal memorization, the Long Short-Term Memory networks are inherently suited for tasks where outcomes are sparse and delayed in time (e.g. when an event at $n = 5$ has consequences only after many steps at $n = 100$), because it can be trained to understand time lags of arbitrary size.

Chapter 2

The domain adaptation project

With the rise of Big Data and the improvement in collective efforts to share information for research purposes, many new technologies are being developed to approach these massive amounts of data. This project addresses three of the most important problems found in Big Data analysis: *Volume*, *Variety* and *Velocity*.

As previously stated in the introduction (page 1), this project arises in the broad context of the European Horizon 2020 project *PreventIT* [28][29][30] on the analysis and prevention of functional decline in elderly patients. During the first iterations of the *PreventIT* project, three main issues in predicting functional decline have been identified:

1. The definition of functional decline is still a topic of debate in the geriatricians community. There are several parameters currently used as metrics for the elderly patient's *frailty* and/or his or her loss of autonomy [31], but no common agreement has been reached on which is the best one.

2. The nature of functional decline is inherently random when caused by unpredictable accidents (e.g. the decline in an elderly patient's autonomy as consequence of a disease, accidents, or surgery).

3. The data provided by the research institutes are the result of lon-

gitudinal studies made in different countries with different metrics and for this reason they are difficult to manage because of their size and heterogeneity.

While the first problem is a semantic one that does not closely involve health informatics research and the second problem could be feasible to approach just with statistical techniques, the third one is not easy to automate because it involves the feedback of an expert geriatrician who can understand the semantic of the data and detect what kind of transformations are required in order to map the data of a specific dataset on a generic structure that can be then provided to a classification tool. This man-in-the-loop approach is both very slow and likely to ending in human errors, especially as the datasets' documentation could be scarce or ambiguous. The problem of adapting an analysis model across different data distributions is known as domain adaptation [32].

Following the research on domain adaptation made by Xavier Glorot, Antoine Bordes, and Yoshua Bengio [33], the problem was approached in this project by implementing a Stacked Denoising Autoencoder trained to discover intermediate representations of the original data, that are independent from which dataset is being chosen as input. This method is made possible by the underlying semantic similarity among the attributes of the different dataset, since they all come from longitudinal studies specifically aimed at measuring aging-related values. In fact, it is possible to interpret variations in these values as effects of the true explanatory factors of the observed data which in turn also determines a variation in the functional autonomy of the patient. The functional decline is therefore treated as a consequence of one or more hidden variables. The Stacked Denoising Autoencoder tries then to extract these hidden variables as if they were features of the original data.

The evaluation of a deep neural network approach to domain adaptation is scientifically relevant for the functional decline detection problem because, although not tested yet in this environment, it has already led very interesting results, beating the previous state-of-the-art techniques in several cases [33][34][35][36].

Furthermore, the presence of fields in the datasets from which it is possible to extract a classification label (i.e. the functional decline) makes it possible to validate the trained model by applying a classifier on both ends of the Stacked Denoising Autoencoder and measuring the difference in classification errors between the pre-SDAE classifier results and the post-SDAE one. These metrics are essential to quantitatively measure the amount of knowledge transferred from one dataset to another when the Stacked Denoising Autoencoder is trained on a first dataset and then used to encode a second one during the validation phase.

As mentioned above, the deep learning approach to domain adaptation is also a relevant contribution to the field of Big Data, since it helps solving some of its most ubiquitous problems. First of all, it directly addresses the *Variety* problem by offering a high-level uniformed view of sparse and heterogeneous data. Second, it is very scalable and does not suffer from excessive *Volumes* of data, because the stochastic optimization approximates the optimal model without having to rely on a complete analysis of all the datasets at once, and the Stacked Denoising Autoencoder makes it possible to transfer knowledge without any need to do a thorough training on the target dataset (see section 2.2 for the in-depth description of the training process). Finally, it also deals with *Velocity* since the stochastic optimization is particularly well-suited for on-line learning where the data evolve through time (e.g. when a new annual report is being added to an already previously analyzed dataset), because the new introduced records can be interpreted just as a new batch to forward the training of the deep neural network.

2.1 Data extraction and transformation

As a proof of concept, this thesis project focuses on the transfer of knowledge between two datasets: The English Longitudinal Study of Ageing (ELSA) [37] and the The Irish Longitudinal study on Ageing (TILDA) [38][39]. Both datasets are divided in *waves* representing the annual report of each patient, but as expected the data is being presented to the user in an heterogeneous way.

- The data is very unbalanced, with ELSA having over 100 000 records of 570 fields spread over 6 waves, while TILDA has only 2 waves for a total of about 15 000 records each one having 1680 fields (almost triple the amount of ELSA's fields).

- While ELSA has both numerical (e.g. the age of the patient) and categorical (e.g the patient's blood type) data represented as numeric values, assigning in the second case to each category its unique natural number identifier, TILDA has a more chaotic approach with both numerical and categorical data expressed sometimes with numeric values, other times with strings describing the patient's answer and occasionally as mixtures of both numeric and descriptive values depending on the answer (e.g the measure of the patient's eye sight could be represented in the domain {"poor", 2, 3, 4, "excellent"}).

- Since both the datasets collect data over several years of study, the amount of information could vary from one year to another. For example, some patients could leave the study or new ones could enter after some waves, questions could be added or removed from the questionnaire, or the same data could be expressed in different ways depending on the wave (e.g. the unique identifier could vary or the descriptive string could be written differently).

- Finally, since the datasets deal with important and sensitive data, like health metrics or the economic and social status of the patients, the privacy must be guaranteed in the best way possible. This means that if a certain field's value is being attributed to only one or too few patients, the data must be aggregated with other neighbours to avoid people identification (e.g. if there is only one patient who is 94 years old, two being 91 years old and five being 87 years old, they should all be grouped together in a cluster of age "*greater than 85*"). This data anonymization was made by the research institutes and is not part of the project.

Because of all the aforementioned datasets' characteristics, a data preprocessing (see Data representation at page 36) must be performed before the neural network starts the training process.

2.1.1 Definition of functional decline

The first problem addressed in the project was to identify a suitable measure of the patient's functional decline. Fortunately, the two datasets share most of the health-related metrics. Although expressed with different formats, there is an almost direct correspondence between the two studies as far as questions regarding the patients' health are concerned. Thanks to the combined efforts of the two research institutes to provide an international standard representation of health-care data [40], the values of the fields corresponding to this kind of metrics can be easily mapped bijectively without requiring a domain adaptation.

Among the health-related answers, twenty-two of these involve Activities of Daily Living (ADL) and Instrumental Activities of Daily Living (IADL). These two metrics are the most commonly used when referring to functional decline [41].

The **Activities of Daily Living** are basic self-care tasks. When a patient's is not able to carry on one or more of these activities, a certain amount of external assistance is required, thus limiting one's autonomy [42]. The ADL generally address the following areas of tasks:

1. Mobility, also referred as ambulation or transferring.
2. Dressing.
3. Personal hygiene and grooming.
4. Toilet hygiene.
5. Bathing and showering.
6. Self-feeding (e.g. being able to eat from a plate).

The **Instrumental Activities of Daily Living**, instead, are not necessary for fundamental functioning, but they let a patient live independently in his community [43][44]. They are more complex than the ADL and they generally cover the following areas:

1. Preparing meals.
2. Shopping for groceries and necessities.
3. Moving within the community.
4. Managing money.
5. Cleaning and maintaining the house.
6. Taking prescribed medications.
7. Using the telephone or other form of communication.

Although the debate is still open on the definition of functional decline and on which metrics are more representative, in this project the total number of ADL and IADL that the patient is not able to carry on is used as a proxy to describe his or her functional status. Therefore, a functional decline is defined as the increase of the total number of functional difficulties over ADL and IADL.

2.1.2 Fields selection

As mentioned before, a small part of the hundreds of fields composing the two datasets can be automatically mapped together in a coherent representation. However, this was not possible with all the other fields. Instead of immediately investing time and efforts in making all the data fields suited to be parsed by the neural networks, the project started with only the shared ones and a small number of selected fields from each one of the two dataset. During the time in which the project was developed, the number of selected fields grew to encapsulate ever more information. In the end, about 14% of ELSA's fields and an equal number of TILDA's fields were selected to train the networks.

The chosen algorithm to select the best candidate fields was Minimum Redundancy Maximum Relevance (mRMR) [45]. The goal of mRMR is to identify the subset of features which have the strongest correlation to a certain classification variable while at the same time maintain the minimum mutual redundancy within the set. To compute the mRMR algorithm the functional status was used as the classification variable and the fields of both datasets were divided between categorical and numerical ones. Both numerical and categorical fields were then ranked with respect to the classification variable and ordered from the most relevant to the least. Each ordered list was then parsed from top to bottom selecting the subset with the least inner redundancy.

Having the functional status expressed as a number, in the selection of the numerical fields the **correlation** was used to both rank them and measure their redundancy. Therefore, the most relevant numerical field was the one having the highest correlation coefficient with the functional status, while the least redundant field was the one having the lowest sum of correlation coefficients computed over all the already taken fields.

Having two series A and B of d measurements each, the correlation coefficient is expressed by the equation:

$$Correlation(A, B) = \frac{\sum_{i=1}^{d}(a_i - \overline{a})(b_i - \overline{b})}{\sqrt{\sum_{i=1}^{d}(a_i - \overline{a})^2 \sum_{i=1}^{d}(b_i - \overline{b})^2}} \tag{2.1}$$

Where a_i and b_i are the values of the $i-$th record respectively for the field A and B, while $\overline{a}$ and $\overline{b}$ are the means computed over every value assumed by A and B.

The same technique was used for the categorical fields, but this time the **mutual information** was used instead of the correlation coefficient. The mutual information is a measure of the mutual dependence between two variables. More specifically, it quantifies the amount of knowledge obtained regarding the value of one of the two variables if the value of the other one is known. The mutual information between two variables A and B is expressed by the equation:

$$MI(A, B) = \sum_{v_A \in A} \sum_{v_B \in B} p(v_A, v_B)\, log\left(\frac{p(v_A, v_B)}{p(v_A)\, p(v_B)}\right) \tag{2.2}$$

Where v_A and v_B are all the possible values assumed by the fields A and

B, $p(v_A)$ and $p(v_B)$ are the respective probabilities that A would take the value v_A and B would take the value v_B, and $p(v_A, v_B)$ is the joint probability of A taking the value v_A and B taking the value v_B at the same time.

2.1.3 Data representation

After the selection of the best fields, the next design choice was to define a feasible representation of the numerical and categorical data. As discussed at page 31, there were many problems with the original datasets including, but not limited to, missing values for a given record, a huge amount of invalid codes caused by the data being not-applicable for a given record (e.g. the age of the patient's oldest child when the subject has no children), and even whole records missing because the patient either skipped a wave, dropped out of the study or was not yet inserted. These problems led to have very sparse datasets with many values all representing unavailable data, but each one for a different reason.

Since the neural networks ultimately work on numeric input nodes, in order to discriminate between a valid and an invalid numerical data, each numerical field was split in a pair of two nodes: the first one equal to the normalized value of that field (or zero when invalid) and the second node being either zero if the field was unavailable or 1 if it was valid. The categorical data were instead represented by one-hot vectors of nodes with length equal to the number of valid categories assumed by the field. Finally, all the nodes extracted by both the datasets were concatenated in a single input vector. Hence, with the exception of the few shared health-related fields, for each record of a given dataset all the fields linked to the other dataset were taken as missing.

With this representation, both categorical and numerical invalid values are expressed by vectors of zeros, and each node is always in the range [0, 1], leading to an homogeneous representation that is well suited for the training of neural networks.

2.1.4 Training, test, and validation sets

A common problem in machine learning is to understand when the results of a model are actually predictive of the application of that model to a real scenario. When the evaluation of a model is made using the same data also used for its training, the results are biased. In fact, in this case the best model is not the one that learns the best generalization of the problem, but the model that learns exactly how to map the input data on the expected output. Applying the model to a real scenario, which was not expected during the training, leads to poor performances because the model is not robust nor generalized enough. This problem is commonly referred as model *overfitting*.

To avoid overfitting and ensure that the results are predictive of the real application of the model, the dataset is usually split in two and only one of the two sets of observations is used during the training of the model, while the other one is held back and used only later during its evaluation. Furthermore, since complex models could require the tuning of hyper-parameters (e.g. the number of hidden nodes of a neural network) that are external to the training process, the dataset should be split again in order to have three distinct sets of data placed in a sort of hierarchical relation:

- **Training set:** a set of examples used to train the model and find the best inner parameters (e.g. the weights of the neural network).
- **Validation set:** a set of examples held back during the training and used to test the trained model in order to find the best set of hyper-parameters.
- **Test set:** a set of examples held back from every decision process and used only to simulate a real scenario in order to test the results of the best overall model.

As stated by Jason Brownlee [46] the division of the dataset in these three components ensures an unbiased evaluation of the final model that was previously fit on the training dataset.

The method described so far to split the dataset is called **holdout method** and the three subsets are typically created by randomly selecting examples from the original dataset and putting them into one of the subsets. The three subsets are also usually different in size, with the training set being the biggest one and the test set the smallest one. However, when working with small original datasets, this method may excessively reduce the number of examples that are available during the training phase and/or lead to an evaluation that is not representative of a real case scenario because the examples selected for the test set are too few to address every kind of data presented in the original dataset in a balanced way. To avoid this problem and use all the available data from the original dataset a possible solution is to use a form of cross-validation.

In a typical **k-fold cross-validation**, the original dataset is randomly partitioned into k equal sized subsets. Then, one of the subset is held out and used as a validation set while the other $k-1$ sets of examples are used for the training. This process is repeated k times, each time using a different subset as the validation set, and finally the overall performance of the model is measured by the average evaluation of each one of the k experiments on their respective validation sets. When also taking the test set into account, the overall process consists in having two nested k-fold cross-validations with the first one extracting a test set each time to evaluate the model and the second inner one extracting a validation set each time to evaluate the hyperparameters.

In scenarios with small dataset sizes, Max Kuhn and Kjell Johnson recommend using a 10-fold cross-validation in general, because of the desirable low bias and variance properties of the performance estimate [47].

Due to time constraints, in the project it was ultimately used the holdout method to separate training, validation and test set, with a respective size of 70%, 20% and 10% of the time series of the original datasets.

2.2 Experimental environment

As previously discussed in this chapter, the core of this thesis project is constituted by a Stacked Denoising Autoencoder that performs a domain adaptation between ELSA and TILDA datasets.

For what concerns the training of the Stacked Denoising Autoencoder, the first step was to split both the ELSA and the TILDA dataset in three subsets, using the holdout method and preserving the integrity of the time series, in order to have a permanent training set for the optimization of the weights and biases of the neural network, a validation set to compare models with different hyperparameters and choose the best ones, and finally a test set held out from the training process to have an unbiased result after the model was well defined. As described by Xavier Glorot, Antoine Bordes, and Yoshua Bengio in their paper [33], one of the best ways to approach a domain adaptation problem with a Stacked Denoising Autoencoder is to perform an unsupervised layer-wise pre-training on each layer of the network by feeding it with data taken from every dataset available in the experiment and after that perform a deep fine-tuning of the SDAE with data taken from only one of the datasets and use the resulting network to encode a different dataset in order to measure the amount of knowledge transfered between the two.

In order to compare Stacked Denoising Autoencoders with different hyperparameters, however, it was not possible to solely rely on the evaluation of the loss function, since even the loss function itself was treated as one of the measured hyperparameters. For this reason, a classifier was introduced in the experiment with the purpose of detecting if a functional decline would occur given a record of the datasets. The classifier was then used both on the original data and on the data encoded by the Stacked Denoising Autoencoder. The results were finally confronted using state-of-the-art metrics (see Validation metrics at page 51) and the hyperparameters which gave the best improvement between pre-SDAE classification and post-SDAE classification were ultimately chosen.

The whole training and validation process of the Stacked Denoising Autoencoder, given two datasets D_a and D_b and a $Classifier$, is described with pseudo-code in the algorithm 1. After the variables' initialization, each unique set of hyperparameters (HP) is tested by having the SDAE performing a layer-wise pre-training on the data from both the datasets followed by a deep fine-tuning on the source dataset only. The trained

```
begin findSDAEBestHyperparameters
    bestIncrement ← -∞;
    bestHP ← ∅;
    foreach set of hyperparameters HP do
        SDAE(HP).pretrain($D_a$.trainSet + $D_b$.trainSet);
        SDAE(HP).finetune($D_a$.trainSet);
        encodedTrain ← SDAE(HP).encode($D_b$.trainSet);
        encodedValidation ← SDAE(HP).encode($D_b$.validationSet);

        Classifier.train($D_b$.trainSet);
        preSDAEMetrics ← Classifier.test($D_b$.validationSet);
        Classifier.train(encodedTrain);
        postSDAEMetrics ← Classifier.test(encodedValidation);

        increment ← postSDAEMetrics - preSDAEMetrics;
        if increment > bestIncrement then
            bestIncrement ← increment;
            bestHP ← HP;
        end
    end
    return bestHP
end
```

Algorithm 1: SDAE hyperparameters validation

SDAE is then used to encode the training and validation sets of the target dataset and two different $Classifiers$ are trained respectively on the original raw training set and the encoded one. Finally, the two $Classifiers$ are tested on the raw and encoded validation sets and the extracted metrics are used to validate the SDAE hyperparameters and find the best ones.

It is important to note in this context that the deep fine-tuning on the source dataset is a slower and more computationally complex process respect to the previous greedy pre-training on the data taken from all the datasets.

2.2.1 Cost-sensitive learning and oversampling

From a first analysis of the domain it was clear that the two classes were heavily unbalanced, with about half of the records being invalid, a third having no functional decline and only a sixth of the total records reporting a decrease in the functional status. When it comes to dealing with unbalanced data the main problem is that, since the loss function is averaged on all the data, the training process of the neural network could easily tend to a local minimum where the most frequent class is well represented but the least common ones are totally ignored. This problem is particularly evident when dealing with unbalanced ordered classes in Health Informatics, because the least frequent ones are usually related to more severe health conditions [48]. Thus, a miss-classification of unusual classes holds a much worse outcome for the patient than a miss-classification of common ones (e.g. if a patient is diagnosed with a cancer when he or she is healthy, the process usually leads to further medical examinations which ultimately attest the first miss-classification and the real health condition of the patient. However, when a real cancer is not classified correctly the process normally stops until more evidence emerges, but at that time it could be too late to properly intervene).

To avoid this form of miss-representation of unbalanced data, the best solution in machine learning is to use a **cost-sensitive learning** [49][50][51]. In cost-sensitive learning, the total error averaged by the loss function is not only proportional, for each record, to the distance between the network output and the expected one (see Loss functions at page 14), but also to a certain miss-classification cost that depends on which are the expected class and the predicted one. The simplest implementation of cost-sensitive learning for multi-class classification is to use a weighted average of the measured distance as loss function, using a value inversely proportional to the probability (frequency) of the true class as weight for each record.

Cost-sensitive learning works well for classifiers, but neural networks that rely on unsupervised learning like the Stacked Denoising Autoencoder are unable to use it because they lack the information regarding the class of the record. For this reason a different approach is required that does not use the class information during the training but uses it *a-priori* to alter the class distribution of the dataset instead. The two main techniques used for unsupervised learning are called undersampling and oversampling.

When performing **undersampling**, the records belonging to each class are randomly selected until the selected subset has the same size of the least frequent class. Only the selected records are then used in the training process, while the other ones are discarded and ignored. The resulting training dataset is therefore balanced because every class is represented in the same amount, however, using this technique on datasets that are too small or where the least frequent class has too few records obviously leads to a further reduction in the size of available training data, with a consequent decline in the network performance.

When undersampling is not feasible, the opposite **oversampling** technique is usually preferred. In this technique the records belonging to each class are randomly selected with replacement from the original dataset until the selected subset has the same size of the most frequent class. The replacement lets the records that belong to an uncommon class to be selected multiple times in order to achieve a better representation of the domain. However, even this techniques has its problems because the items of an uncommon class are too similar to each other (being copies of the same few records) and the resulting network may not be very robust when addressing new records of the same class.

Luckily, the noise introduced by the Stacked Denoising Autoencoder perfectly counterbalances the cons of performing an oversampling, as stated at page 44, and for this reason the oversampling was used in the project to train the SDAE while the aforementioned simple cost-sensitive learning was used during the training of the classifier.

2.2.2 Weights and biases initialization

One of the most important factors in the training of a deep neural network is the initialization of the weigths and the biases, because the gradient descent techniques are very sensitive to the initial conditions, as shown in the figure 1.6 section at page 18. A common mistake is to initialize the weights and biases to zero in order to first analyze the simplest solutions where several nodes are inactive and then add complexity during the training through backpropagation. The problem with this approach is that every node of a given layer are equally updated by the backpropagation algorithm because they all share the same initial output and contribution to the final loss value. Hence, each layer can be collapsed into a single logic neuron and the expressive power of the network is heavily impaired.

Generally speaking, if the weights start too small in a deep neural network, the error signal used by the backpropagation decreases too much and become unusable to train the first layers of the network. Conversely, if they start too large, it is important to note that the output of a node depends on a weighted sum of every node's output from the previous layer, hence the feedforward signal may grow too much and saturate the output of the nodes in the last layers of the network.

Usually, a random initialization from a normal distribution with mean equal to zero is a good solution that lets the neurons of each layers specialize for different inputs while also searching at first the solutions around the simple zeros that could lead to a reduction in the number of nodes required to solve the problem. However, more complex initialization algorithms have been proposed in literature to quicken the training and avoid saturation and/or vanishing gradients. One of these techniques, developed by Xavier Glorot and Yoshua Bengio [52], initializes the biases and weights of each neuron with a random value taken from a uniform distribution in range:

$$x \sim U\left[-\sqrt{\frac{6}{in+out}}, \sqrt{\frac{6}{in+out}}\right] \tag{2.3}$$

Where x is the variable to be initialized, *in* is equal to the number of weights (or input nodes) of the current neuron and *out* is equal to the number of nodes in the current layer. This initialization is especially good when the neurons are activated by a logistic function, but it brings detrimental effects on the networks with very large hidden layers because the resulting weights are too small, thus the resulting training process is slowed too much.

In the project, the initialization algorithm was treated as an hyper-parameter, hence both the Gaussian initializer and the Xavier initializer were used during the validation phase.

2.2.3 Early stopping

One of the main source of overfitting when training a neural network is to overstate the number of epochs required to learn a good data representation. Thanks to the correctness of gradient descent algorithms with a proper learning rate, even when a stochastic approach is used, the error on the training set tends to get smaller at each epoch. This means that the network learns to better recognize the inputs that are available in the training sets, but it may also cause the network to over-specialize and lose its robustness as the number of epochs increases. In fact, when comparing the error on the train set to the one on the validation set, as shown in figure 2.1, it is evident that from an initial generalization of the model that brings both errors down, at a certain point in time the network starts to overfit and the validation error starts to rise again.

A form of regularization that avoids the negative effects of an excessive number of epochs is to add a noise to the network's inputs during the

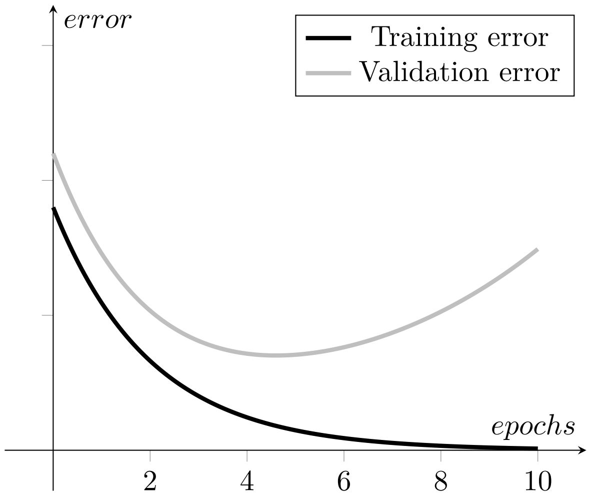


Figure 2.1: Early stop using training and validation errors

training phase, as performed by the Stacked Denoising Autoencoder. The random noise alters the inputs preventing the network from training on multiple repetitions of the same examples. However, even when using noise a certain rate of overfitting is ultimately expected in the long run because the noise itself can only corrupt the data up to a certain amount before destroying any information included in examples themselves.

Since the number of epochs used during the training process could be seen just as another hyperparameter, it is possible to use the data contained in the validation set in order to define its best value and avoid overfitting. The form of regularization that takes advantage of the validation data is called early stop. When performing an early stopping, at the end of each epoch the learning algorithm is allowed to "*peek*" into the validation set in order to compute the validation error. If the validation error starts to increase, the learning process is interrupted and the last best model is returned by the training algorithm. This technique quickens the training phase because it can interrupt it mid-process, however the results may be sub-optimal if the error on the validation set swings back and forth too much. Several solutions to the oscillating validation error have been proposed in literature [53].

In this project, an early stop regularization with a *look-ahead* parameter was used to avoid ovefitting in the training phase of the neural networks. The training process was programmed to check the validation error at the end of each epoch, but only to stop after a number of epochs from the last error reduction equal to the given look-ahead parameter. This way, a single-point oscillation of the validation error would be ignored by the early stopping, but a constant increase over several epochs would trigger the end of the training phase instead.

2.2.4 Chosen classifiers

In their work on domain adaptation, Xavier Glorot, Antoine Bordes, and Yoshua Bengio [33] used a Support Vector Machine to assert the amount of knowledge transferred by the Stacked Denoising Autoencoder. To support their hypothesis and expand the analysis on the domain adaptation field, in this project the Support Vector Machine was used side by side with three other classifiers during both the validation of the Stacked Denoising Autoencoder's hyperparameters and the final test of the network. The three additional classifiers chosen for the project were: a Multilayer Perceptron with a single hidden layer, a deep Multilayer Perceptron with several hidden layers, and a Long Short-Term Memory network. The implementation of the classifiers was possible thanks to Daniele Fongo's contribution to the project [54].

The application of neural networks to address both the domain adaptation and the classification process added a layer of complexity to the aforementioned validation algorithm (see page 40), because even the neural network classifiers required a training phase for their weights and biases followed by a validation phase for their hyperparameters.

In the project it was used a greedy approach for the training of the neural network classifier. At first, a validation phase was performed on each classifier to discover the best hyperparameters. The process (given two datasets D_a and D_b) is described with pseudo-code in the algorithm 2: after the variables' initialization, each unique set of hyperparameters (HP) for each different classifier (C) is tested by training the respective network on the data from both the datasets and then extracting the rel-

```
begin findClassifiersBestHyperparameters
    foreach classifier C do
        bestMetrics[C] ← 0;
        bestHP[C] ← ∅;
        foreach set of hyperparameters HP do
            C(HP).train($D_a$.trainSet + $D_b$.trainSet);
            metrics[C] ← Classifier.test($D_a$.validationSet +
             $D_b$.validationSet);
            if metrics[C] > bestMetrics[C] then
                bestMetrics[C] ← metrics[C];
                bestHP[C] ← HP;
            end
        end
    end
    return bestHP
end
```

Algorithm 2: Classifiers hyperparameters validation

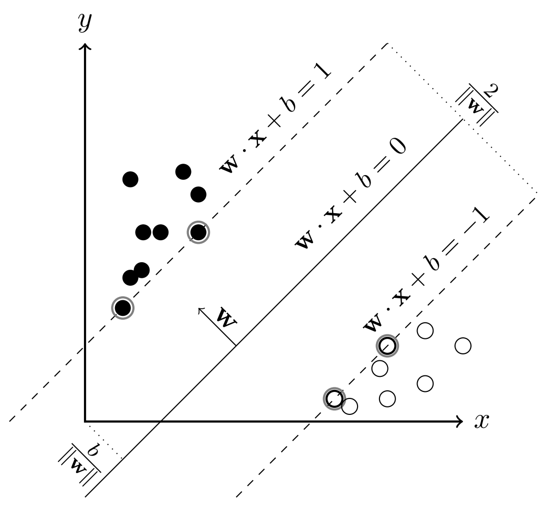
ative metrics, eventually updating the previous best found metrics and hyperparameters.

It is important to note that while the validation of the classifiers' hyperparameters was made on the joined datasets, either the joined training sets during the training or the joined validation sets during the extraction of the metrics, the final classifier used to attest the knowledge transfer of the Stacked Denoising Autoencoder was trained either on just the TILDA dataset and then tested on the ELSA or vice versa (see algorithm 1). In practice, it could be possible that the classifier's hyperparameters which were validated on both the datasets together would be suboptimal when applied to only one of them, however this implementation choice was ultimately made because of time constraints and limitations in the datasets' sizes.

SVM

The Support Vector Machine is a non-probabilistic binary linear classifier that tries to find the maximum-margin hyperplane which divides two different classes. If the points representing the data in the high-dimensional space of their parameters are linearly separable, it is possible to define an hyperplane that separates the points of the two classes by having one group on one side and one on the other. The margins are therefore defined as the maximum distance between the found hyperplane and a second hyperplane, parallel to the previous one, that still has all the points belonging to the respective class only in the region of the space that does not contain the original hyperplane, as shown in figure 2.2.

In order to extend the Support Vector Machine model to cases with many outliers, a hinge loss function is introduced. The loss function is equal to zero for each point that lies in the correct side of the margin's hyperplane and is proportional to the distance from it otherwise. The model is therefore trained to minimize its loss function. Differently from the Perceptron, the Support Vector Machine does not stop after finding an hyperplane which discriminate between the two classes, but tries to find the best and more stable one among the many possible hyperplanes.



y
x
$\mathbf{w}\cdot\mathbf{x}+b=1$
$\mathbf{w}\cdot\mathbf{x}+b=0$
$\mathbf{w}\cdot\mathbf{x}+b=-1$
$\frac{2}{\|\mathbf{w}\|}$
$\frac{b}{\|\mathbf{w}\|}$
$\mathbf{w}$

Figure 2.2: Support Vector Machine margins

The limitation of only addressing linearly separable data was overcome by Vladimir Vapnik et al. [55] by introducing a kernel function that maps each point of the original space into a new higher-dimensional feature space where the two classes are linearly separable. The kernel function allows the Support Vector Machine to approach much more complex classification problems. However, in order to compute the kernel function, the machine needs to analyze all the original data space at once. The complexity of finding the right kernel function is more than quadratic with respect to the number of records, making it unfeasible to compute when there are too many items in the datasets (i.e. more than 10K records [56]).

For this project, it was implemented a linear SVM classifier trained with Stochastic Gradient Descent and without any kernel transformation due to the size of the datasets. Thanks to the straightforward nature of the Support Vector Machine, its results were used in the project as a baseline to compare every other classification method both prior to and after the application of the Stacked Denoising Autoencoder.

MLP

As stated before, two different Multilayer Perceptrons were implemented in the project. The first one only had a single hidden layer while the other had multiple hidden layers between the input and the output layers. The reason behind this choice was to test both a deep neural network, that could possibly solve the functional decline prediction problem by itself, and the most simple feedforward classifier that could be interpreted as a neural network version of the baseline.

Only the deep Multilayer Perceptron had a *depth* and a *shape* hyperparameter, but otherwise they shared the same validation environment, with the tested hyperparameters being the number of *nodes per layer*, the *activation function* of the hidden nodes, the *initiaizer* and *optimizer* used during the training, the *learning rate* and *learning rate decay* used by the optimizer, and finally the *batch size*. Eventually, a parametric *noise* was also added to the Multilayer Perceptrons to avoid overfitting during the training process.

LSTM

The last classifier implemented in the project was a Long Short-Term Memory network. Since the original dataset consists of several time series, one for each patient, it was natural to choose a recursive neural networks as a classifier and the LSTMs held the best general results in literature among all the RNNs. Since the cause of a functional decline could be located many years before the actual decline occurrence, it could be impossible for a normal feedforward neural network to predict the event from a single record of the time series. In this case, the Long Short-Term Memory classifier should bring much more accurate results, therefore by comparing them with the metrics obtained from the deep feedforward neural network it should be possible to understand if the functional decline can be estimated by only checking the last record, or if the whole time series should be analyzed instead.

Since the time series from the two datasets had different lengths, the Long Short-Term Memory was implemented with a dynamic unfolding by providing the model with both the batched input and the relative list of lengths. Furthermore, many of the series started or ended with a succession of invalid records due to the patient leaving the study or entering after a while. For this reason, all the series were shifted and trimmed to cut the invalid ends on both sides.

The memory of the Long Short-Term Memory cell is represented by a state of parametric size, therefore the number of *internal nodes* was one of the tested hyperparameters, alongside with the *initializer*, the *optimizer*, the *learning rate* and *learning rate decay*, the *batch size*, and finally the input *noise*.

2.3 Validation metrics

All the neural network classifiers described so far were programmed to output a probability of functional decline. This behaviour was achieved by having two neurons in the output layer which were activated by a Softmax function. The two output neurons corresponded respectively to the probability of having a functional decline and the probability of not having it. The Softmax ensured that the two outputs were positive in value and that their sum was equal to one, therefore granting the correctness of the equality:

$$p(functional\ decline) = 1 - p(not\ functional\ decline) \tag{2.4}$$

When dealing with probabilistic classifiers (i.e. classifiers that output a probabilistic distribution over a set of classes) there are two main elements that can be evaluated: the uncertainty of the classification and the correctness of the classification. Both the previous characteristics are useful to attest the quality of the classifier and ultimately to confront different classification models in order to find the best one during the validation phase.

To evaluate the correctness of the classification, a useful tool is the Confusion Matrix. Since every classifier implemented in the project was

		Predicted Class	
		P	**N**
Real Class	**P**	True Positives (TP)	False Negatives (FN)
Real Class	**N**	False Positives (FP)	True Negatives (TN)

Table 2.1: Binary Confusion Matrix

a binary classifier, it is possible to define the predicted class by imposing a threshold that discriminates between the two classes. The threshold can be applied to either one of the two output neurons, as consequence of the equation 2.4. The typical threshold, also used in this project, is set to 50% for the binary classifiers, meaning that whichever neuron has the greatest value is the one used for the prediction. However, it is possible to adjust the threshold in order to further tune the classification (e.g. to only classify the record as anticipating a functional decline if the functional decline neuron has a value equal or greater than 0.7).

Once every record of the set is classified as either preceding a functional decline or not, it is possible to define a binary Confusion Matrix as shown in table 2.1, where the records are assigned to one of the matrix cells as following:

- **True Positives (TP)**: contains the number of records that precede a functional decline and are correctly classified.
- **False Positives (FP)**: contains the number of records that precede a functional decline but are incorrectly classified.
- **True Negatives (TN)**: contains the number of records that do not precede a functional decline and are correctly classified.
- **False Negatives (FN)**: contains the number of records that do not precede a functional decline but are incorrectly classified.

The Confusion Matrix is very useful when addressing the correctness of a classifier, but does not provide any information regarding the uncertainty of the classification (i.e. there is no difference between a record classified as preceding a functional decline because the respective output node has a value of 0.51 and another record where the same output is 0.99). To have a full picture of the classifiers' performance, in this project four different metrics were used during the validation phase, two addressing the correctness and two addressing the uncertainty.

```
begin validationProcess
    foreach Metric M do
        CHP = findClassifiersBestHyperparameters(M);

        foreach permutation P of $\{D_a, D_b\}$ do
            foreach Classifier C do
                SDAEHP[P, M, C] =
                  findSDAEBestHyperparameters(P, M, C, CHP[C]);
            end
        end
    end
    return SDAEHP
end
```

Algorithm 3: Validation process

Having multiple metrics to compare the classifiers, instead of only one, added yet another layer of complexity to the validation process, as described with pseudo-code in the algorithm 3. Therefore, in the final validation process, the first step was to find the best set of hyperparameters for each classifier, given a metric M, by using the procedure described in the algorithm 2 and providing it with the chosen metric M. The first step was independent from the direction of the knowledge transfer that had to be measured, since the classifiers were trained on the joined training sets and then validated on the joined validation sets. However, in the second step both the possible directions of the transfer were analyzed by permuting the order of the two datasets D_a and D_b (corresponding to ELSA and

TILDA). Then, for each classifier, the best set of SDAE hyperparameters was found by running the procedure described in the algorithm 1 and providing it with the direction of the analysis (i.e. the permutation P), the chosen metric M, and the chosen classifier C with the respective best hyperparameters $CHP[C]$ that were computed in the first step.

Therefore, the resulting validation process returns a three-dimensional matrix that for each direction, each metric, and each classifier, contains the best set of SDAE hyperparameters among all the combinations of possible hyperparameters for that specific neural network.

In the following sections, all the metrics implemented in the project are discussed in more details.

2.3.1 Accuracy

The accuracy is the simplest validation metric used in the project. Its goal is to establish the proportion between the amount of records which are correctly classified and the total amount of records used during the test. The value of the accuracy is computed using the following equation:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.5}$$

Where TP, FP, TN and FN are the four values of the Confusion Matrix shown in figure 2.1. The accuracy can be interpreted as the probability of the model correctly classifying any given record.

In addition to not delivering any information regarding the uncertainty of the classifier, the accuracy also suffers from the accuracy paradox when dealing with unbalanced datasets. In fact, if most of the records belong to the same class, the accuracy value may be higher when all the records are classified as belonging to the most common class. For example, if the values of TP, FP, TN and FN are respectively equal to [800, 200, 80, 20], the accuracy is $(800 + 80)/1100 = 80\%$, but if they were [1000, 0, 0, 100] instead, the accuracy would be $(1000 + 0)/1100 = 91\%$. In the first case, the classifier is able to discriminate between the two classes with a partial accuracy equal to 80% for either of the two and the total accuracy reflects this performance. However, in the second case the classifier has learned nothing about the the input space and always delivers the same class prediction, while also having a better overall accuracy thanks to the unbalanced class distribution in the dataset.

Although not being a very reliable metric for the aforementioned reasons, the accuracy is still a useful metric to be analyzed because of its computational simplicity, its understandability, and because the bias introduced by the class unbalance is equally present for every classifier, since they are all tested on exactly the same data subsets.

2.3.2 Cohen's Kappa

The Cohen's Kappa [57] is the second metric implemented in the project that addresses the classifiers' correctness. The Cohen's Kappa is more robust than the simple accuracy because it also takes into account the possibility that the agreement between the predicted class and the real label is just the result of chance. When dealing with only two classes, the value of the Kappa can be computed as:

$$\begin{aligned} Kappa &= \frac{Accuracy - p_c}{1 - p_c} \\ p_c &= \frac{TP + FP}{N}\frac{TP + FN}{N} + \frac{TN + FN}{N}\frac{TN + FP}{N} \end{aligned} \tag{2.6}$$

Where TP, FP, TN and FN are the four values of the Confusion Matrix shown in table 2.1 and N is the sum of all the previous cells together. The value of p_c can be interpreted as the classification accuracy achieved by a probabilistic classifier that randomly assigns the classes with a probability proportional to the marginal totals of the analyzed Confusion Matrix. In fact, the four terms of the second equation (p_c) correspond to the first row of the Confusion Matrix, the first column, the second row and the second column respectively.

The magnitude of the Cohen's Kappa can be therefore interpreted as the improvement of the classifier with respect to the aforementioned random classifier. The value of the Kappa is represented by a real number in the interval [-1, 1], with $Kappa = 1$ (100% improvement) when the evaluated classifier performs a perfect classification, $Kappa = 0$ when the classification is apparently random and only driven by chance, and $Kappa = -1$ when the evaluated classifier is in total disagreement with the real class distribution of the original dataset (i.e. it classifies every record as belonging to the wrong class).

Although being an improvement over the simple Accuracy because it also takes into consideration the accidental agreement between the predicted classes and the labels of the dataset, the Cohen's Kappa is more prone to misinterpretation because it confuses the disagreement due to class allocation with the disagreement caused by their quantity. For example, if the values of TP, FP, TN and FN are respectively equal to [0, 1, 10, 1], the classification is clearly better than if they were [1, 10, 1, 0] instead, however the Kappa in the first case is equal to -0.091 while in the second case would be equal to 0.016. The second matrix of the example is equal to the first one with just the two rows rearranged in the opposite order, but by doing so the number of record correctly classified dropped from 10 out of 12 to only 2 out of 12. However, it also solves the classification of 0 records out of 1 that the original matrix has in the first row, hence boosting the relative classification of the *True* records and achieving a higher Kappa score.

2.3.3 Brier score

The Brier score is the first metric implemented in the project which directly addresses the uncertainty of the classifier. When dealing with a categorical classifier that outputs a proper probability distribution among all the possible class categories, like the one granted by the Softmax activation function, it is possible to compute the Brier score as the mean squared difference between the datasets' labels and the model's outputs. For example, if the network's output for a given record is [0.8, 0.2] and its real label is [1, 0], the contribution to the Brier score corresponding to that record would be equal to $((1 - 0.8)^2 + (0 - 0.2)^2)/2 = 0.04$.

Because the problem treated in this thesis is formulated around the functional decline, the Brier score can be used to measure the uncertainty and the correctness of the classification at the same time. However if the original functional status extracted from the datasets were used without converting them to binary categorical variables, the Brier score would be inappropriate. In fact, the Brier score assumes that all the possible classes are equivalently distant one from another and is not suited to analyze probability distributions over ordinal classes.

The Brier score has a value in the range [0, 1] where 0 means that the classifier is absolutely certain of its predictions and also performs a perfect classification, while a value equal to 0.25 indicates a classifier that is very uncertain about its classification (i.e. its outputs are close to [0.5, 0.5], independently from the actual real label), and a score of 1 means that the classifier is absolutely certain about its predictions, but the final classification is always the opposite of the expected one.

Differently from the other metrics implemented in the project, the Brier score is a distance. Therefore, the lower the value is, the better the performance of the classifier.

2.3.4 AUC-ROC

The Area Under the Receiver Operating Characteristic Curve [58] is the last metric implemented in this project. As stated at page 51, in a binary classification problem it is possible to achieve different classification ratios between the two classes by simply changing the value of the discrimination threshold. By increasing the positive threshold (e.g. setting the network to only classify a record as positive if the corresponding neuron has a value equal or greater than 0.99) the number of False Positives should decrease while the number of False Negatives should increase.

As shown in figure 2.3, it is possible to plot the distribution of the positive and negative records with respect to the classifier's threshold. Even if the two distributions are partially overlapping (i.e. there is no threshold that perfectly discriminates between positives and negatives), it is still possible to determine the best threshold by sliding it within the [0,

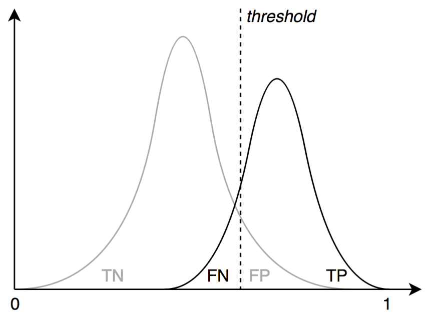


Figure 2.3: Positives and Negatives distribution

1] interval and measuring the number of True Positives and False Positives. The ROC curve, as shown in figure 2.4, is therefore created by plotting the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)** for each threshold value, where the TPR is equal to the number of True Positives divided by the total number of Positives and the FPR is equal to the number of False Positives divided by the total number of Negatives.

The dotted line in figure 2.4 represents the values provided by the random guess. Therefore, if the ROC curve lays under the dotted line (i.e. in bottom-left region) the classifier is worse than a random classifier, while if it lays over the dotted line it is considered a good classifier. The perfect classifier, only achievable if the Positives and Negatives distributions do not overlap, has a ROC curve that goes straight up to the top-left corner and then stays flat until it reaches the top-right corner of the figure.

Since the ROC space is bounded by a square of side one, the value of the area under the ROC curve is represented by a real number within the interval [0, 1]. The AUC-ROC value describes the probability that the positive output node of the classifier will rank a randomly chosen positive record higher than a randomly chosen negative one. The AUC-ROC can be seen as a measure of the quality of the discriminative power of the classifier and for this reason is appropriate even when the classifier's outcome does not get discretized. By projecting the AUC-ROC score into the [-1, 1] interval, the result can also be interpreted as a Kappa statistics which also takes the classifier's uncertainty into consideration.

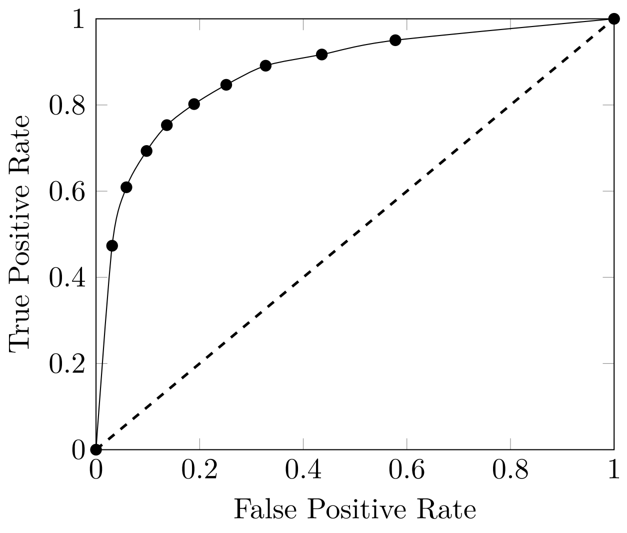


Figure 2.4: Receiver Operating Characteristic curve

2.4 Networks implementation

The project was fully implemented in Python because it offers several very useful scientific-oriented libraries. Among these libraries, the most relevant for the purposes of this thesis are Numpy, Scikit-learn and TensorFlow.

NumPy [59] is the fundamental package for scientific computing with Python. It contains among other things:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is licensed under the BSD license, enabling reuse with few restrictions.

The whole testing environment and data management of the project was programmed on top of the NumPy functionalities. More specifically, all the data extracted from both ELSA and TILDA datasets were converted to NumPy multidimensional array structures in order to feed them to the neural networks.

Scikit-learn [60] is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language. Emphasis is put on ease of use, performance, documentation, and API consistency. It has minimal dependencies and is distributed under the simplified BSD license, encouraging its use in both academic and commercial settings.

The Support Vector Machine used in the project was implemented on top of the Scikit-learn *SGDClassifier* class. Furthermore, some of the metrics (namely the AUC-ROC and Brier scores) were also implemented using Scikit-learn functionalities to obtain more accurate results.

2.4.1 TensorFlow and TensorBoard

TensorFlow [61] is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

Both the optimizers, the xavier initializer, the LSTM cell, and the cross-entropy function were implemented on top of the respective TensorFlow functionalities because they are faster and more numerically stable than the alternative raw formulations [62].

The TensorFlow library also includes a suite of visualization tools called TensorBoard. TensorBoard can be used to visualize TensorFlow graphs, to plot quantitative metrics about the execution of the graphs, and show additional data (e.g. images or signals) that pass through them.

The figures 2.5, 2.6, and 2.7 show respectively the TensorFlow graphs for the Stacked Denoising Autoencoder, the Multi-Layer Perceptron and the Long Short-Term Memory network extracted from TensorBoard. Furthermore, the figure 1.7 at page 19 was made with the scalar summary tool of TensorBoard during the training phase of one of the neural networks implemented in this project. Each arrow represents a certain number of tensors, while the rounded rectangles indicate a high-level node of the computational graph. More specifically *input* refers to the batched input vector, *seq-length* is the batched vector of the time series lengths used by the LSTM, *target* is the batched label vector, *cost_vector* represents the cost mask vector used for the cost-sensitive learning, *optimizer* is the homonym operation, *loss* is the node that computes the error, *metric* is the metrics extractor node, and finally *SDAE*, *Feedforward* and *LSTM* are the three kind of networks implemented in the project.

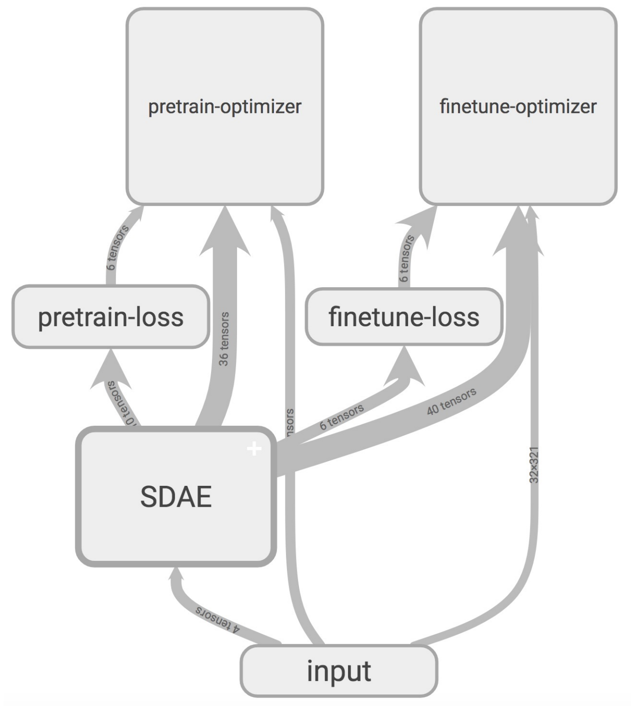


Figure 2.5: TensorFlow graph of the Stacked Denoising Autoencoder

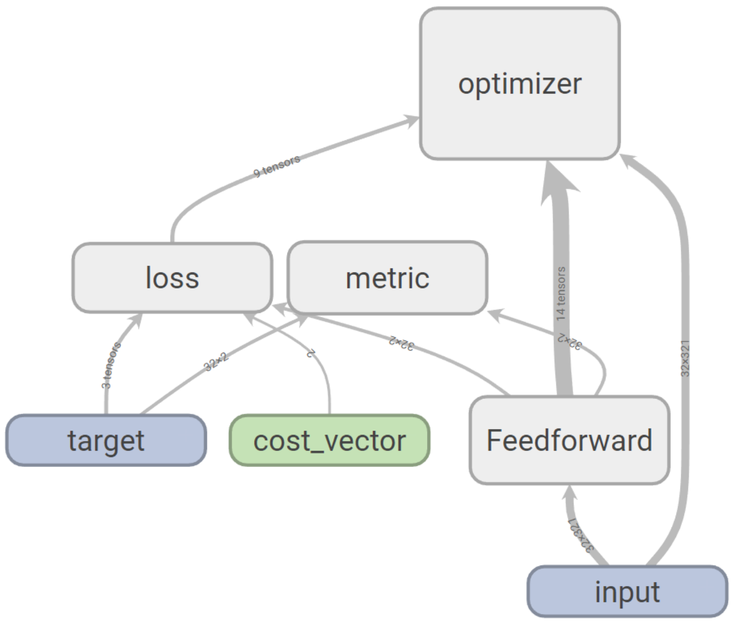


Figure 2.6: TensorFlow graph of the Multi-Layer Peceptron

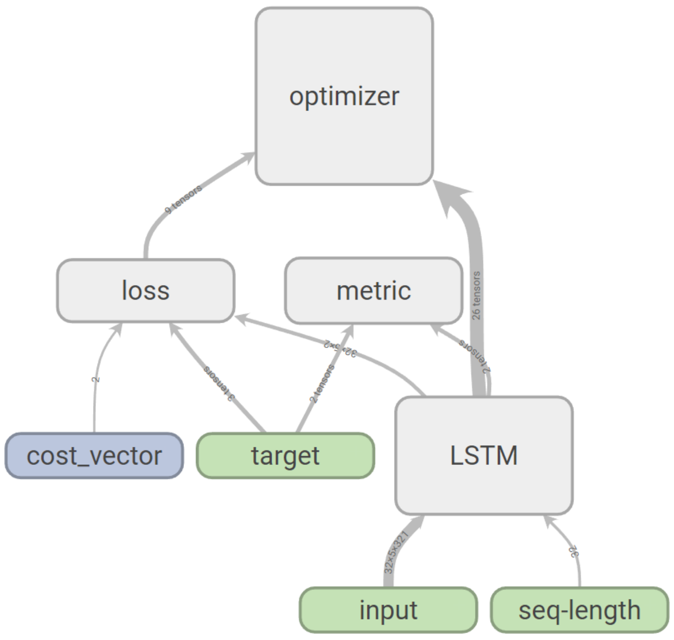


Figure 2.7: TensorFlow graph of the Long Short-Term Memory network

Chapter 3

Experimental results

In the final extensive test performed within the project, the input layer consisted of 321 nodes in range [0, 1], of which about 60 were shared between the ELSA and TILDA datasets while the other belonged to only one of them and thus had a value equal to zero for the record of the other dataset.

3.1 Classifiers optimization

The grid search analysis in the validation process of the classifiers (described in the algorithm 2) was made on the following sets of hyperparameters:

- **LSTM:** a state size in {50, 100, 200, 300} (its optimal value was **200** for all the metrics); a Gaussian or Xavier initializer (**Xavier** always outperformed the Gaussian); an optimization function using either Stochastic Gradient Descent or Stochastic Adam (**Adam** always gave the best performance); a learning rate in {0.1, 0.01, 0.001} (**0.001** was always optimal); a learning rate decay either fractional or exponential (**exponential** led to the best results); a batch size in {32, 64, 128, 256} records (**128** was the most agreed on and the one that was ultimately chosen, but 32 led to the best performance w.r.t. the AUC-ROC metric); and finally an optional Gaussian noise (the tests with inputs corrupted by the **noise** always improved the final results).

- **Deep MLP:** a shape either rectangular (same size across all the hidden layers) or triangular (size decreasing with the depth), thus a final sizes of the hidden layers in {[150, 150], [300, 300], [150, 150, 150], [300, 300, 300]} for the rectangular shaped networks and in {[100, 50], [200, 100], [200, 100, 50], [300, 200, 100]} for the triangular ones (**[300, 300, 300]**, i.e. a constant size of 300 hidden nodes with a depth equal to 3, gave the best results across all the metrics); an activation function for the hidden nodes either implemented as a Softplus, a logistic function, or an hyperbolic tangent (the **hyperbolic tangent** was always the best); a Gaussian or Xavier initializer (**Xavier** always outperformed the Gaussian); an optimization function using either Stochastic Gradient Descent or Stochastic Adam (**Adam** always gave the best performance); a learning rate in {0.1, 0.01, 0.001} (**0.001** was always optimal); a learning rate decay either fractional or exponential (**exponential** led to the best results); a batch size in {32, 64, 128, 256} records (**128** was always optimal); and finally an optional Gaussian noise (the **noise** always improved the final results).

- **Single layer MLP:** a hidden layer size in {100, 200, 300, 400, 500, 600, 700, 800} (**400** was always optimal); an activation function implemented as a Softplus, a logistic function, or an hyperbolic tangent (the **hyperbolic tangent** was always the best); a Gaussian or Xavier initializer (**Xavier** always outperformed the Gaussian); an optimization function using either Stochastic Gradient Descent or Stochastic Adam (**Adam** always gave the best performance); a learning rate in {0.1, 0.01, 0.001} (**0.01** was always optimal); a learning rate decay either fractional or exponential (**exponential** led to the best results); a batch size in {32, 64, 128, 256} records (**128** was always optimal); and finally an optional Gaussian noise (the **noise** always improved the final results).

The fractional learning rate decay mentioned before was implemented using the following equation:

$$\eta_t = \eta_0 \, \frac{1}{1+t} \tag{3.1}$$

Where η_0 is the original learning rate reported by the respective hyper-paramenter and η_t is the learning rate used during the $t-$th epoch of the training in order to ensure the correctness of the stochastic optimization. The exponential learning rate decay was instead computed as:

$$\eta_t = \eta_0 \, \alpha^t \tag{3.2}$$

Where the value of α was set to 0.99 by default.

The Gaussian noise used to corrupt the inputs has a mean equal to 0 and a standard deviation of 0.05. This choice was made to avoid destroying any useful information inside the original data ranged within [0, 1], while also grating some robustness to the network.

The final results of the classifiers' hyperparameters highlighted some interesting characteristics across all the networks and all the metrics. For example the hyperbolic tangent always outperformed the other activation functions, while the Xavier initializer and the Adam optimizer were always better than the respective counterparts. The learning rate equal to 0.001 (except for the single layer MLP) shows that in this experiment a slow but more precise training of the deep neural networks was preferable to a hasty but less careful one. The fractional learning rate decay never came up because its implementation led to a drop in the learning rate that was probably too sudden and therefore it gave too much importance to the first few epochs compared to the remaining ones. A batch size of 128 records was always preferred to smaller o bigger sizes probably due to the internal variance of the datasets, and finally the noise always helped improving the performance, probably because it reduced the risk of overfitting during the training process.

On the other end, The number of hidden nodes is an interesting case study because it varies depending on the type of neural network. The single layer MLP, although being tested with a big amount of possible sizes, had the best performance with only 400 hidden nodes, which is the smallest available number over the input size (321 nodes). Among all the shapes, depths, and sizes, the deep MLP selected a constant width of 300 hidden nodes along 3 hidden layers, which was the biggest network available during the validation process. The LSTM preferred an inner state of just 200 nodes instead, that was not the biggest size available and it was actually smaller than the number of nodes in the input layer. However, it is important to note that the LSTM cell is made of three different networks each of which has a hidden layer of the same size, leading to a total number of hidden nodes equal to 600 (that is still less than the deep MLP's 900 nodes anyway).

3.2 SDAE optimization

For what concerns the hyperparameters of the SDAE, the validation process was more complex, as described by the algorithm 3. The basic structure of the Stacked Denoising Autoencoder was the same described by Xavier Glorot, Antoine Bordes, and Yoshua Bengio in their research on the domain adaptation, with the a parametric masking noise on the input layer and a gaussian noise on the hidden ones. The masking noise was tested in 0.3, 0.6, 0.9 while the Gaussian noise was implemented with the same mean and variance of the one described for the previous neural networks.

The masking noise perfectly matches the datasets representation provided to the input layer. In fact, because the missing values of a certain record (i.e. the ones belonging to the other dataset or the invalid ones) are set to zero, the model can not discriminate between masked nodes (i.e. nodes whose value is set to zero by the masking noise) and invalid nodes without also learning the underlying characteristics of the two datasets.

However, differently from the model reported in the aforementioned paper, the activation function of the outermost decoding layer was implemented as a logistic function. Since the output of that layer was confronted with the uncorrupted input during the training process, the choice of using a logistic function was made to match the outputs range (i.e. [0, 1]) with the original data.

Except for the noise layout and the activation function of the outermost decoding layer, every other hyperparameter was validated through a grid search analysis. As for the deep MLP, multiple shapes, sizes and depths were explored for the SDAE. However, due to time constraints, both the encoding and the decoding networks were only tested with a depth equal to either 1 or 3, leading to a size of the encoding layers in {100, 300, 500} for the single layer SDAE, in {[300, 150, 50], [150, 75, 25]} for the triangular deep SDAE, and in {[300, 300, 300], [100, 100, 100]} for the rectangular deep SDAE. The decoding layers were obviously reversed in size in order to maintain the symmetry. In addition to the cross-entropy loss function proposed by Xavier Glorot, Antoine Bordes, and Yoshua Bengio, a RMSE loss function was also tested in the validation phase. Because the cross-entropy needs a probability distribution with values within [0, 1] to work properly, it was always tested with decoding layers activated by a logistic function. However, since the RMSE does not have this limitation it was tested with several decoding activation functions, namely the logistic function, the hyperbolic tangent, the Softplus function and the ReLU. The encoding activation functions were also tested using all the previous functions.

The other validated hyperparameters were the Gaussian or Xavier initializer, the Stochastic Gradient Descent or Stochastic Adam optimizer, the learning rate in {0.1, 0.01, 0.001}, the fractional or exponential learning rate decay, and the batch size in 32, 64, 128, 256.

Some of the hyperparameters outperformed every other hyperparameter in the respective category independently from the direction of the knowledge transfer, the classifier, or the metric used. These hyperparameters are the **hyperbolic tangent** activation function for both the encoding and decoding layers, the **exponential** learning rate decay, the **Xavier** initializer, the **Adam** optimizer, and the batch size equal to *128*

		Loss	LR	Noise	Size
EtoT	**Accuracy**	RMSE	0.01	0.9	300
	Kappa	CE	0.01	0.3	500
TtoE	**Accuracy**	RMSE	0.001	0.9	300
	Kappa	RMSE	0.01	0.9	300

Table 3.1: Best SDAE hyperparameters w.r.t. SVM classification

records. It is noteworthy that all the previous hyperparameters were also shared by all the neural network classifiers.

The rest of the hyperparameters depended on the specific direction, classifier and metric used in for validation. Therefore, the following sections will report every combination of the aforementioned variables with the respective set of hyperparameters that led to the best overall performance. To achieve a more straightforward description and avoid confusion, the direction of the knowledge transfer will be always referred as **EtoT** or **TtoE**, meaning that the SDAE is either fine-tuned on the ELSA dataset and then used to encode the TILDA dataset or fine-tuned on the TILDA dataset and then used to encode the ELSA dataset respectively.

3.2.1 SVM

For what concerns the Support Vector Machine, the best performance was achieved when the encoding SDAE had the hyperparameters reported in table 3.1. Except for the Cohen's Kappa in direction EtoT which was optimal when the SDAE was implemented using a cross-entropy loss function, a 30% masking noise and a single encoding/decoding layer of size 500, the other results were pretty consistent along all the experiments, with a high masking noise (90%), a RMSE loss function and a single hidden layer of 300 nodes. The best learning rate was always equal to 0.01 except when tested through the SVM accuracy in direction TtoE (0.001).

The depth of the encoding network and the high masking noise match the experimental results of Xavier Glorot, Antoine Bordes, and Yoshua Bengio. In their benchmark experiment, they tested the domain adap-

		Loss	LR	Noise	Size
EtoT	**Accuracy**	RMSE	0.01	0.3	300
	Kappa	RMSE	0.01	0.3	300
	Brier	RMSE	0.01	0.3	300
	AUCROC	CE	0.001	0.3	300, 300, 300
TtoE	**Accuracy**	CE	0.001	0.3	300, 300, 300
	Kappa	CE	0.001	0.3	300, 300, 300
	Brier	CE	0.001	0.3	300, 300, 300
	AUCROC	CE	0.001	0.3	300, 300, 300

Table 3.2: Best SDAE hyperparameters w.r.t. single layer MLP classification

tation provided by the Stacked Denoising Autoencoder with a Support Vector Machine and the reported best performance was given by a single layer SDAE with a 80% masking noise in the input layer.

3.2.2 Single layer MLP

When tested through the single layer MLP classifier, the resulting best hyperparameters were very polarized, as reported in table 3.2, with the 30% masking noise being the only common hyperparameter across all the metrics and directions.

The best Stacked Denoising Autoencoder according to all the metrics in direction TtoE end even the AUC-ROC in direction EtoT was a deep SDAE with three encoding/decoding layers of constant size equal to 300 nodes, a learning rate equal to 0.001, and a cross-entropy loss function. On the other side, all the metrics in direction EtoT (except the AUC-ROC) reported the best improvement when paired with a SDAE that had a single hidden layer of 300 nodes, a learning rate equal to 0.01, and a RMSE loss function.

		Loss	LR	Noise	Size
EtoT	**Accuracy**	RMSE	0.01	0.3	300
	Kappa	RMSE	0.01	0.3	300
	Brier	RMSE	0.01	0.3	100
	AUCROC	RMSE	0.01	0.6	300
TtoE	**Accuracy**	RMSE	0.001	0.3	100
	Kappa	RMSE	0.001	0.3	100
	Brier	RMSE	0.001	0.6	300, 300, 300
	AUCROC	CE	0.001	0.3	300, 300, 300

Table 3.3: Best SDAE hyperparameters w.r.t. deep MLP classification

3.2.3 Deep MLP

Compared to the single layer MLP, using the deep MLP to validate the SDAE hyperparameters led to more heterogeneous results, as reported in table 3.3. RMSE was reported as the best loss function by all the metrics in both directions, except for the AUC-ROC in direction TtoE which preferred a cross-entropy loss function. The best learning rate according to all the metrics in direction EtoT was equal to 0.01, while for the metrics in direction TtoE the 0.001 learning rate performed better. A low 30% masking noise was preferred by all the metrics except the AUC-ROC in direction EtoT and the Brier score in direction TtoE that preferred a 60% masking noise. Finally, the shape and size of the encoding and decoding networks were the hyperparameters with the highest disagreement. The Accuracy, Cohen's Kappa and AUC-ROC in direction EtoT were higher with a single layer SDAE of size 300; the Accuracy and Cohen's Kappa in direction TtoE and the Brier score in direction EtoT reported better results with a single layer SDAE of size 100; and finally the Brier score and AUC-ROC in direction TtoE preferred a deep Stacked Denoising Autoencoder with three layers of 300 nodes each.

		Loss	LR	Noise	Size
EtoT	**Accuracy**	RMSE	0.01	0.3	300
	Kappa	RMSE	0.01	0.3	300
	Brier	CE	0.01	0.3	500
	AUCROC	RMSE	0.01	0.3	500
TtoE	**Accuracy**	CE	0.001	0.6	300, 300, 300
	Kappa	CE	0.001	0.6	300, 300, 300
	Brier	CE	0.001	0.6	300, 300, 300
	AUCROC	CE	0.001	0.6	300, 300, 300

Table 3.4: Best SDAE hyperparameters w.r.t. LSTM classification

3.2.4 LSTM

The best hyperparameters for the SDAE were again somewhat clustered when measured through the classification of the LSTM network on the encoded data, as reported in table 3.4. All the metrics in direction TtoE preferred a deep SDAE with three layers of 300 nodes, a 60% masking noise, a learning rate equal to 0.001, and a cross-entropy loss function. The metrics in direction EtoT instead preferred a SDAE with a single encoding/decoding layer of size equal to either 300 nodes (Accuracy and Cohen's Kappa) or 500 nodes (Brier score and AUC-ROC), a low 30% masking noise, a learning rate equal to 0.01, and a RMSE loss function (except for the Brier score that was lower when the SDAE was trained with a cross-entropy loss function).

3.3 Classifiers and metrics comparison

After the validation phase, the best candidates were tested on the ELSA and TILDA test sets. The results are shown in the figures 3.1, 3.2, 3.3, and 3.4. Each chart reports a different metric and displays four bars for each classifier (dMLP means deep MLP and sMLP stands for single layer MLP). The pale red and pale blue bars show the value of the metric when the classifier was tested on the raw test set of the TILDA and

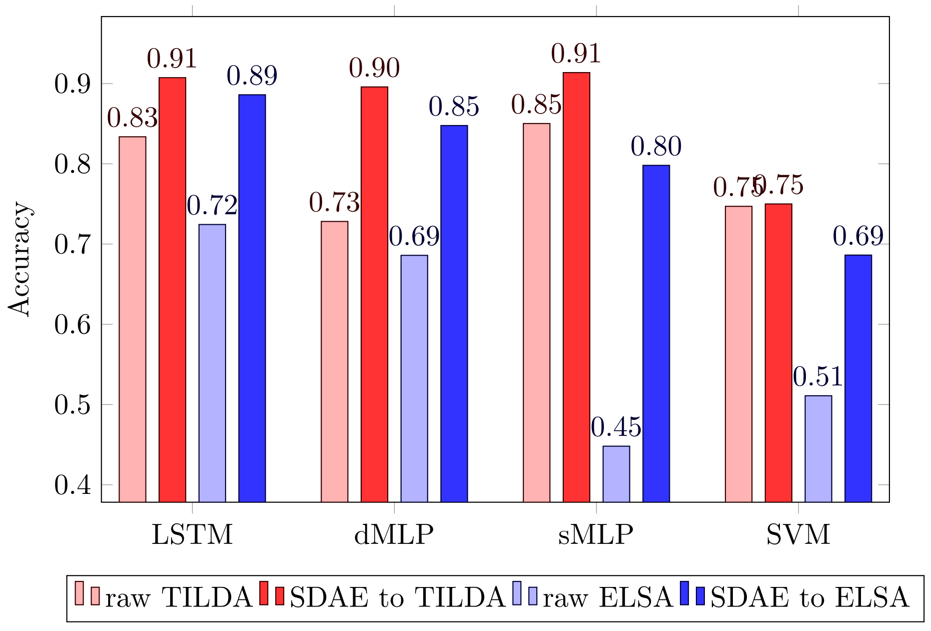


Figure 3.1: Accuracy of the four classifiers on ELSA and TILDA test sets, both before and after SDAE encoding

ELSA dataset respectively. The bright red bar reports the value of the metric when the classifier was tested on the TILDA test set encoded by a SDAE which was previously fine-tuned on the ELSA dataset and was built with the best hyperparameters extracted for that specific metric, classifier, and direction during the validation phase. Conversely, the bright blue bar shows the value of the metric when the classifier was tested on the encoded ELSA test set with the best SDAE fine-tuned on the TILDA dataset.

3.4 Results interpretation

From the analysis of the hyperparameters it is possible to extract some noteworthy observations. For example, some of the hyperparameters seem to be linked to the direction of the transfer, with EtoT generally working better when performed by a single-layer SDAE that has a learning rate equal to 0.01, while TtoE preferring a deep rectangular SDAE with a learning rate of 0.001. This trend is probably caused by the underlying characteristics of the two datasets (e.g. ELSA's records are quantitatively

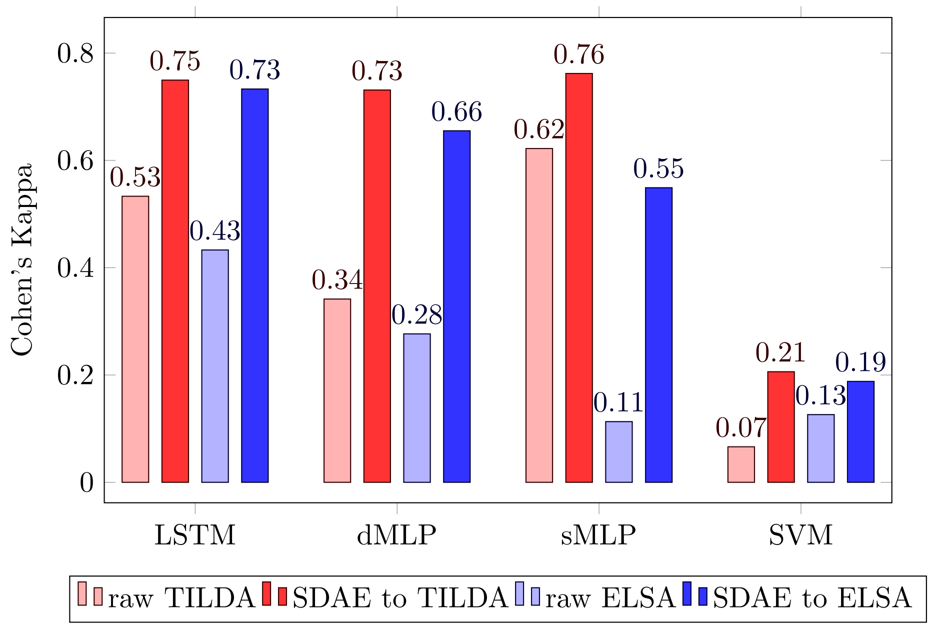


Figure 3.2: Cohen's Kappa of the four classifiers on ELSA and TILDA test sets, both before and after SDAE encoding

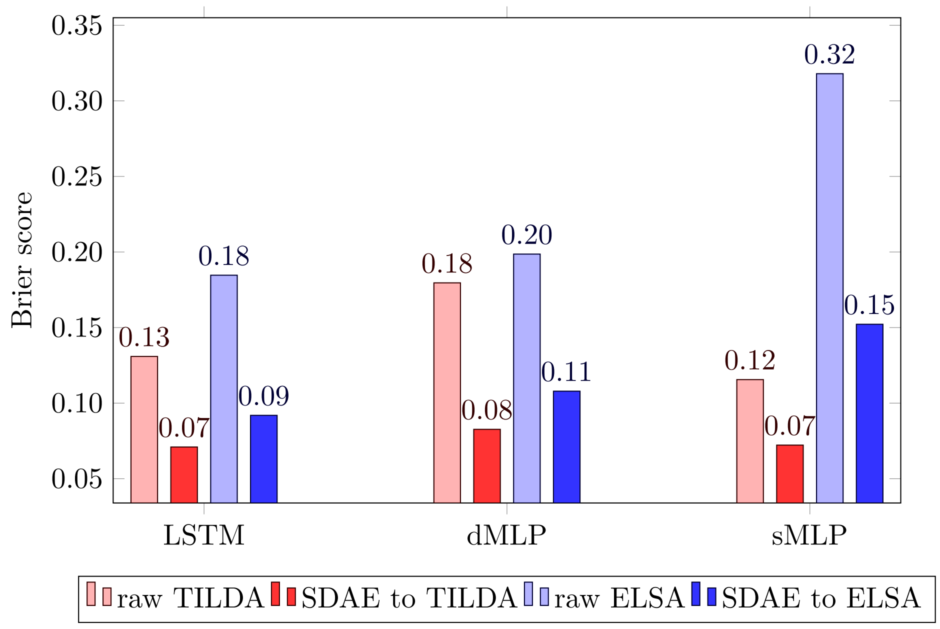


Figure 3.3: Brier score of the three probabilistic classifiers on ELSA and TILDA test sets, both before and after SDAE encoding

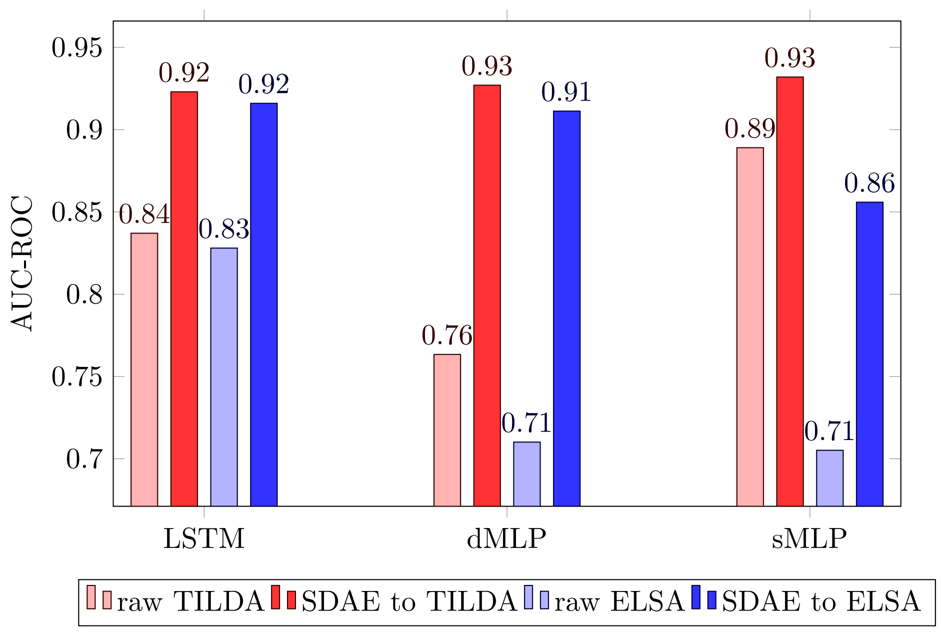


Figure 3.4: AUC-ROC of the three probabilistic classifiers on ELSA and TILDA test sets, both before and after SDAE encoding

more than TILDA's and often contain more invalid fields and invalid waves in the time series). The nature of ELSA's data could require a deeper and more accurate analysis from the Stacked Denoising Autoencoder in order to learn how to extract interesting features from a less complex environment (i.e. TILDA's data).

Furthermore, the high masking noise seems to bring better results only when paired with the Support Vector Machine, while all the neural network classifiers performed better when the encoding SDAE was trained with a low corrupting noise.

It is also worth mentioning that the RMSE and cross-entropy loss functions were almost equally represented in the results and that the best width for the encoding and decoding layers was almost always about the size of the input layer, meaning that the Stacked Denoising Autoencoder was probably able to extract a great amount of features from the original data (N.B. every record of a given dataset actually fills only about half of the input nodes with non-zero values, therefore the number of hidden features extracted by the SDAE exceeds the number of valid inputs).

The bar charts reported in the current chapter show how effective the domain adaptation performed by the SDAE is. In fact, all the metrics improved when the SDAE encoding was applied, independently from the chosen classifier or the direction of the transfer.

Regarding the classifiers comparison, a possible observation is that the neural networks always outperformed the linear Support Vector Machine by a large margin, while within the neural network classifiers the deep ones were the most consistent across all the experiments and also the classifiers that generally led to better results. The LSTM performed especially well compared to every other classifier when applied on the raw data of both the datasets, probably thanks to the temporal analysis performed by the network. However, it is important to note that regardless of the initial performance on the raw data, after the encoding performed by the SDAE all the neural network classifiers reached similar values for almost all the metrics, meaning that the domain adaptation process was probably more relevant to the final classification than the choice of the actual classifier.

Conclusions

In this project a Stacked Denoising Autoencoder was designed and implemented in TensorFlow alongside with four different classifiers, namely a Support Vector Machine, a Multilayer Perceptron, a Deep Multilayer Perceptron, and a Long Short-Term Memory classifier. Then, several tests were designed and performed on the aforementioned networks in order to measure their performance and usefulness in a real case study.

More specifically, this thesis provided experimental evidence to support the hypothesis of benefiting from the use of a Stacked Denoising Autoencoder to perform an unsupervised feature extraction for a domain adaptation problem. The features representation extracted by the Stacked Denoising Autoencoder greatly improved the performance of all the tested classifiers.

Furthermore, the project extended a previously documented technique, which involved a Support Vector Machine to classify the encoded data [33], with several other classifiers and demonstrated that, in this operational scenario, the deep neural network classifiers are particularly well-suited to predict the functional decline of an elderly patient from the features extracted by a Stacked Denoising Autoencoder. The neural network classifiers trained through supervised deep learning greatly outperformed the linear Support Vector Machine classification, achieving an average accuracy, Cohen's Kappa, Brier score, and AUC-ROC equal to 74.31%, 0.393, 0.173, and 0.785 respectively just on the raw data.

When trained and tested on the feature extracted by the Stacked Denoising Autoencoder all the metrics of the deep classifiers showed an improvement, with the average accuracy rising to 88.42% (+0.141), the average Kappa reaching 0.717 (+0.321, almost doubled), the average Brier score dropping to 0.088 (-0.085, almost halved), and the average AUC-ROC reaching 0.919 (+0.135).

In a future scenario, it is possible to extend this project to every field of the two original datasets that were used to measure the knowledge transfer achieved by the Stacked Denoising Autoencoder, and even to extend the analysis to other similar datasets.

Another possible improvement is to perform a 10-fold cross-validation instead of the simple holdout that was used in this project, in order to achieve more accurate results and train the networks with all the information contained in the datasets.

Finally, the validation phase of the Stacked Denoising Autoencoder could be extended to include even more hyper-parameters with the purpose of finding a new model that would perform even better than the one presented in this project.

List of Figures

List of Tables

Acknowledgements

The data relative to ELSA were made available through the United Kingdom Data Archive. ELSA was developed by a team of researchers based at the NatCen Social Research, University College London and the Institute for Fiscal Studies. The data were collected by NatCen Social Research. The funding is provided by the National Institute of Aging in the United States, and a consortium of United Kingdom government departments coordinated by the Office for National Statistics. The developers and funders of ELSA and the Archive do not bear any responsibility for the analyses or interpretations presented here.

TILDA is an interinstitutional initiative led by Trinity College Dublin. TILDA data have been co-funded by the Government of Ireland through the Office of the Minister for Health and Children, by Atlantic Philanthropies, and by Irish Life; have been collected under the Statistics Act, 1993, of the Central Statistics Office. The project has been designed and implemented by the TILDA study team, Department of Health and Children. Copyright and all other intellectual property rights relating to the data are vested in TILDA. Ethical approval for each wave of data collection is granted by the Trinity College Research Ethics Committee. TILDA data is accessible for free from the following sites: Irish Social Science Data Archive at University College Dublin http://www.ucd.ie/issda/data/tilda/; Interuniversity Consortium for Political and Social Research at the University of Michigan (`http://www.icpsr.umich.edu/icpsrweb/ICPSR/studies/34315`).

Bibliography

1. The preventit project, aimed at preventing early functional decline at younger old age. `https://ec.europa.eu/eip/ageing/commitments-tracker/c2/preventit-project-aimed-preventing-early-functional-decline-younger-old-ag` `en`.

2. Mingsheng Long, Jianmin Wang, and Michael I. Jordan. Unsupervised domain adaptation with residual transfer networks. *CoRR*, abs/1602.04433, 2016.

3. Jeffrey L. Elman. *Rethinking innateness.* Neural network modeling and connectionism. MIT, Cambridge, Mass., 1998. A Bradford book.

4. S. Song B. Zhang, L.P. Shi. Creating more intelligent robots through brain-inspired computing, 2016. Brain-inspired intelligent robotics: The intersection of robotics and neuroscience sciences.

5. A. Raue, C. Kreutz, T. Maiwald, J. Bachmann, M. Schilling, U. Klingmüller, and J. Timmer. Structural and practical identifiability analysis of partially observed dynamical models by exploiting the profile likelihood. *Bioinformatics*, 25(15):1923–1929, 2009.

6. Alberto Prieto, Beatriz Prieto, Eva Martinez Ortigosa, Eduardo Ros, Francisco Pelayo, Julio Ortega, and Ignacio Rojas. Neural networks: An overview of early research, current frameworks and new challenges. *Neurocomputing*, 214(Supplement C):242 – 268, 2016.

7. C. Lee Giles and Christian W. Omlin. Extraction, insertion and re-

finement of symbolic rules in dynamically driven recurrent neural networks. *Connection Science*, 5(3-4):307–337, 1993.

8. Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, June 1949.

9. Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, Jan 1982.

10. Stephen Grossberg. Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11(1):23–63, 1987.

11. Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

12. Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

13. Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *J. Artif. Int. Res.*, 4(1):237–285, May 1996.

14. F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

15. Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

16. Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947 EP –, Jun 2000.

17. Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 807–814, USA, 2010. Omnipress.

18. L. G. Kraft. A device for quantizing, grouping, and coding amplitude modulated pulses. M.Sc. Thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass., 1949.

19. S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951.

20. P. Debye. Näherungsformeln für die zylinderfunktionen für große werte des arguments und unbeschränkt veränderliche werte des index. *Mathematische Annalen*, 67(4):535–558, Dec 1909.

21. H. Robbins and D. Siegmund. *A Convergence Theorem for Non Negative Almost Supermartingales and Some Applications*, pages 111–135. Springer New York, New York, NY, 1985.

22. Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

23. Deep learning for java. open-source, distributed, deep learning library for the jvm, 2015.

24. Anders Boesen Lindbo Larsen and Søren Kaae Sønderby. Generating faces with torch, 2015.

25. J. Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, March 1992.

26. J. Schmidhuber. Netzwerkarchitekturen, Zielfunktionen und Kettenregel. *(Network Architectures, Objective Functions, and Chain Rule.)* Habilitationsschrift *(Habilitation Thesis)*, Institut für Informatik, Technische Universität München, 1993.

27. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

28. White paper of the project results in preventit (deliverable 8.2), 2017. Early risk detection and prevention in ageing people by self-administered ICT-supported assessment and a behavioural change intervention delivered by use of smartphones and smartwatches.

29. Clemens Becker Chris Todd Kristin Taraldsen Mirjam Pijnappels Kamiar Aminian Jorunn L. Helbostad, Beatrix Vereijken and Sabato Mellone. Mobile health applications to promote active and healthy ageing, 2016.

30. Lindy Clemson Elisabeth Boulton Helen Hawley-Hague Clemens Becker Michael Schwenk Michaela Weber, Nacera Belala. Feasibility and effectiveness of intervention programmes integrating functional exercise into daily life of older adults: A systematic review, 2017.

31. Peter Hartley, Jennifer Adamson, Carol Cunningham, Georgina Embleton, and Roman Romero-Ortuno. Clinical frailty and functional trajectories in hospitalized older adults: A retrospective observational study. *Geriatrics & Gerontology International*, 17(7):1063–1068, 2017. GGI-0104-2016.R2.

32. Hal Daumé III and Daniel Marcu. Domain adaptation for statistical classifiers. *CoRR*, abs/1109.6341, 2011.

33. Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 513–520, USA, 2011. Omnipress.

34. Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969 – 978, 2009. Special Section on Graphical Models and Information Retrieval.

35. Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

36. Marc' Aurelio Ranzato and Martin Szummer. Semi-supervised learning of compact document representations with deep networks. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 792–799, New York, NY, USA, 2008. ACM.

37. James Nazroo James Banks, G.D. Batty and Andrew Steptoe. *The dynamics of ageing: Evidence from the English Longitudinal Study of Ageing 2002-15 (Wave 7).* The Institute for Fiscal Studies, 2016.

38. Rose Anne Kenny. The irish longitudinal study on ageing (tilda), 2009-2011, Jul 2014.

39. Brendan J. Whelan and George M. Savva. Design and methodology of the irish longitudinal study on ageing. *Journal of the American Geriatrics Society*, 61:S265–S268, 2013.

40. A Nolan, C O'Regan, C Dooley, D Wallace, A Hever, H Cronin, E Hudson, and RA Kenny. The over 50s in a changing ireland: economic circumstances, health and well-being. *Dublin: The Irish Longitudinal Study on Ageing*, 2014.

41. Linda S. Noelker and Richard Browdie. Sidney katz, md: A new paradigm for chronic illness and long-term care. *The Gerontologist*, 54(1):13–20, 2014.

42. Brie Williams. *Consideration of Function & Functional Decline.* 2014.

43. Harrington M. Pass L. Bookman, A. and E. Reisner. *Family Caregiver Handbook.* MIT, 2007.

44. Cynthia Williams. *Healthy Aging & Assessing Older Adults.* 2011.

45. J. R. Berrendero and Alma-Delia Cuevas. The mrmr variable selection method: a comparative study for functional data. 2015.

46. Jason Brownlee. What is the difference between test and validation datasets?, 2017.

47. Max Kuhn and Kjell Johnson. *Applied Predictive Modeling.* Springer, 2013.

48. B. Misganaw and M. Vidyasagar. Exploiting ordinal class structure in multiclass classification: Application to ovarian cancer. *IEEE Life Sciences Letters*, 1(1):15–18, June 2015.

49. Gary Weiss, Kate McCarthy, and Bibi Zabar. Cost-sensitive learning vs. sampling: Which is best for handling unbalanced classes with unequal error costs?, 01 2007.

50. Bianca Zadrozny, John Langford, and Naoki Abe. Cost-sensitive learning by cost-proportionate example weighting. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 435–, Washington, DC, USA, 2003. IEEE Computer Society.

51. Zhi-Hua Zhou and Xu-Ying Liu. Training cost-sensitive neural networks with methods addressing the class imbalance problem. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):63–77, Jan 2006.

52. Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 249–256, May 2010.

53. Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 55–69, London, UK, UK, 1998. Springer-Verlag.

54. Fongo Daniele. Previsione del declino funzionale tramite lútilizzo di reti neurali ricorrenti, 2017.

55. Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152, New York, NY, USA, 1992. ACM.

56. Scikit-learn: Machine learning in Python - sklearn.svm.svc. `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`.

57. Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

58. J. A. Hanley and B. J. McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, April 1982.

59. S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

60. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

61. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

62. Tensorflow - api r1.4. `https://www.tensorflow.org/api_docs/python/`.