ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

Study of Deep Learning algorithms: incremental solutions

Mémoire présenté en vue de l’obtention du diplôme
d'Ingénieur Civil en informatique, à finalité spécialisée

Antonio García Díaz

Directeur
Professeur Hugues Bersini

Service
IRIDIA

Année académique
2017-2018

Résumé

Nom de l'étudiant : Antonio García Díaz.

Titre complet du Master : Master en Ingénieur Civil en informatique, à finalité spécialisée.

Année académique : 2017-2018

Titre complet du Mémoire : Study of Deep Learning algorithms: incremental solutions.

Mots-clés : *Deep Learning*, Réseaux Neuronaux, topologie, structuration automatique, incrémental, TensorFlow.

Le *Deep Learning* est un champ particulièrement prometteur du *Machine Learning*, lui-même un des champs de recherche dans l'Intelligence Artificielle. Le *Machine Learning* a pour but de développer des algorithmes capables d'apprendre une tâche automatiquement et de devenir meilleurs à cette tâche. Ils accomplissent ceci par le moyen de données externes liées à cette tâche. Le *Deep Learning* est lié au développement d'un type particulier d'algorithmes de *Machine Learning*, appelés Réseaux Neuronaux Artificiels.

Ces Réseaux Neuronaux sont constitués de petits processeurs simples et interconnectés appelés « neurones », qui sont organisés ensemble en des couches. Un des problèmes clé lors de l'implémentation de Réseaux Neuronaux est de déterminer leur topologie, c'est-à-dire combien de couches et combien de neurones par couche devrait avoir le réseau. En 1994, un algorithme appelé EMANN a été developpé par le service IRIDIA de l'Université Libre de Bruxelles qui permettait de structurer automatiquement des topologies de Réseaux Neuronaux grâce à une solution incrémentale. À mesure que le réseau est entraîné, des nouvelles couches et neurones sont ajoutés ou enlevés, à des moments différents et en suivant certains critères, jusqu'à sélectionner une structure de réseau optimale.

Dans ce mémoire, des nouvelles implémentations d'EMANN appelées « algorithmes EMANN-like » ont été créés en utilisant la librairie TensorFlow. Ceux-ci ont été appliqués à une simple tâche de classification, le set de données de référence CIFAR-10, pour les tester et les comparer en termes de leur précision. L'effet de certaines valeurs de paramètres sur leur précision a aussi été évalué à travers de jusqu'à six groupes de tests et expériences. Il a été conclu principalement que EMANN a sa meilleure performance lorsque les fonctions d'activation utilisées pour ses neurones se comportent de façon similaire à une sigmoïde, et qu'il est probable que des paramètres corrélés avec une augmentation du nombre de neurones aient un effet très positif sur la précision du Réseau Neuronal construit par EMANN.

Abstract

Keywords: Deep Learning, Neural Networks, topology, self-structuring, incremental, TensorFlow.

Deep Learning is a promising subfield of Machine Learning, itself a subfield of Artificial Intelligence. Machine Learning focuses on the development of algorithms which can automatically learn a task and improve on it, through the usage of external data that is relevant to the task. Deep Learning is related to the development of one kind of Machine Learning algorithms, called artificial Neural Networks.

Neural Networks consist of various simple interconnected processors called neurons, organized together in layers. One of the key problems when implementing Neural Networks is to determine their topology: how many layers, and how many neurons per layer, the network should have. In 1994, an algorithm called EMANN was developed in the IRIDIA laboratories at the *Université Libre de Bruxelles* that allowed for self-structuring Neural Network topologies thanks to an incremental solution. As the network is trained, new layers and neurons are added and/or pruned at different moments, following certain criteria, until an optimal network structure is selected.

In this thesis, new implementations of EMANN called EMANN-like algorithms were created using the TensorFlow library. They were applied to a simple classification task, the CIFAR-10 benchmark dataset, to test them and compare them in terms of accuracy. The effect of certain parameter values on their accuracy was also assessed through up to six groups of tests and experiments. It was mainly concluded that EMANN performs much better when the activation functions used for its neurons behave in a similar way to a sigmoid, and that parameters correlated with an increase in the number of neurons are likely to have a very positive effect on the accuracy of the Neural Network constructed by EMANN.

Acknowledgements

First, I would like to express my sincere gratitude towards Prof. Hugues Bersini for directing this master thesis, for supervising my routine work, and for supporting me throughout all these years of my University career since the second Bachelor year (where he taught us, among others, the basics of Object Oriented programming). I am also especially grateful towards him for having co-developed the EMANN algorithm, which is the foundational pillar of this master thesis.

Next, I would like to express my deep gratitude towards Prof. Olivier Debeir from the LISA department, and his assistant Adrien Foucart PhD, for their expert advice and useful suggestions, as well as their willingness to help me debug my code during its early stages of development, which have been crucial for the production of this master thesis.

I would also like to thank Cédric Simar, for giving me insight on how to improve my test designs, and my implementation of EMANN.

Finally, I would like to give my heartfelt thanks to my parents, for their strong support along my University years, in particular during the preparation of this thesis. As well as with delicious and nourishing Spanish meals. (¡Gracias Mamá!)

Abbreviations and Acronyms

AI	Artificial Intelligence
(A)NN	(Artificial) Neural Network(s)
ANOVA	Analysis of variance
AT	Ascension threshold
CS	Connection strength
DL	Deep Learning
EMANN	Evolving Modular Architecture for Neural Networks
GPU	Graphics Processing Unit(s)
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Square Error
PP	Patience parameter
ReLU	Rectified Linear Unit
ST	Settling threshold
UT	Uselessness threshold

Table of contents

Introduction

Deep Learning is a promising subfield of Machine Learning, which is itself a subfield of Artificial Intelligence (AI). It can be seen as the "cutting-edge of the cutting-edge" of AI, focusing on a narrow subset of tools from Machine Learning to solve a very wide range of applications (1).

Machine Learning (ML) is the field of AI that focuses on the development of programs which can automatically learn a task and improve on it without being explicitly programmed, but through the usage of external data that is relevant to the task (2). Machine Learning programs achieve this by looking for patterns in the data that is fed to them, and by rectifying their decisions based on the success they have when performing the task. The primary aim of Machine Learning is to allow a program to learn and adjust itself with as little human intervention or assistance as possible.

Within the larger context of Machine Learning, Deep Learning (DL) is related to the development of algorithms called artificial Neural Networks (NN) (3). These algorithms take inspiration from the structure and function of the brain: they consist of various simple interconnected processors called neurons, organized together in layers (4). There is no unique definition for the term "Deep Learning", rather various specific and nuanced perspectives have been established by the leaders and experts in the field, and from these one can make sense of what Deep Learning is about. The general idea of Deep Learning could be simply defined as "solving machine learning problems by means of multi-layered artificial Neural Networks" (5).

One of the key problems when implementing Neural Networks with multiple layers is to determine the network's topology: how many layers, and how many neurons per layer, the network should have. Indeed, in most cases today, the inner structure of a neural network is crafted manually by a software engineer or a programmer, based on experimentation and testing of different topologies, and the results that they provide. A technique that would automatize the process of designing parts of a Deep Learning algorithm, adapting them towards a predefined classification goal, would be more than welcome.

Over the past few years, companies such as Google, Netflix and Amazon have popularized Deep Learning as a revolutionary technology (1). But during the mid-90's, several novel Machine Learning algorithms involving Neural Networks have been developed that could already be qualified as Deep Learning algorithms. One of these algorithms, developed at the IRIDIA Laboratories of the *Université Libre de Bruxelles*, is EMANN (Evolving Modular Architecture for Neural networks). It is notably for building the Neural Network's topology automatically thanks to an incremental approach (6). During the Neural Networks training phase, new layers and neurons may be added and/or pruned at different moments, following certain criteria, until an optimal network structure is selected.

The aim of this master thesis is to study the specifics of the original EMANN algorithm under the light of the currently relevant Deep Learning trends and techniques. Using Google's open-source TensorFlow library (7), new and updated versions of EMANN (called EMMAN-like algorithms) are implemented. A simple image classification task, the CIFAR-10 dataset (8), is used as the benchmark for testing the performance of these algorithms. Different algorithms are compared in terms of accuracy, and the effects of certain parameter values on their accuracy is also assessed. The Neural Network topologies produced by these algorithms are used to understand the influence of the number of layers and neurons on the accuracy of fully connected Neural Networks, when applied to a specific learning context such as CIFAR-10.

Background

The Age of Artificial Intelligence

In a broad sense, Artificial Intelligence (AI) can be defined as "any technology (be it a piece of software, an algorithm, a set of processes, a robot, etc.) that is able to function appropriately with foresight of its environment" (9). Over time, different authors and researchers have developed different perspectives on the specific goals and motivations of AI as a field of study. For instance, for some authors the aim of AI is to create computing systems (mainly software) that are capable of exhibiting intelligent behaviours, whilst for other authors the main concern of AI is the study and design of intelligent agents (software and hardware) that perceive their environment and take actions to maximize their chances of success (Fig.1) (10).

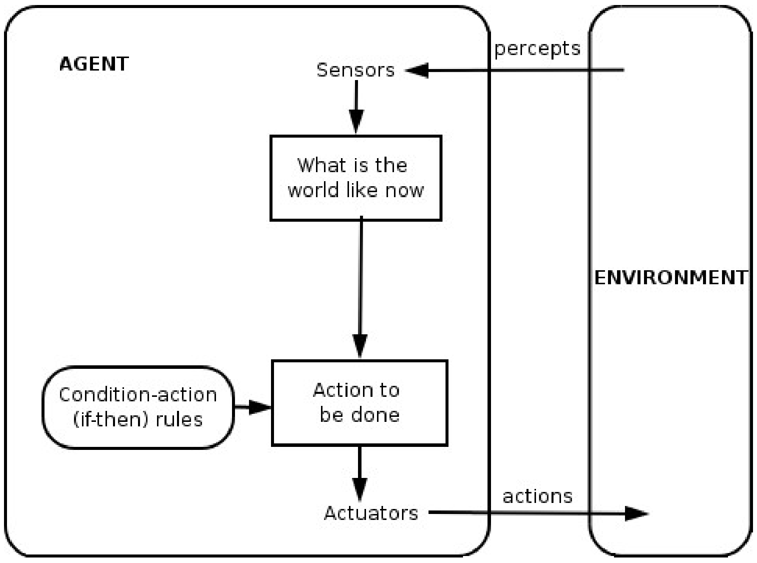

Fig. 1. Diagram of an intelligent agent interacting with its environment, one of the possible definitions of Artificial Intelligence (taken from (10)*).*

In recent years, the field of AI has advanced at a very rapid rate: much progress has been made in the techniques related to AI, and these techniques have found many applications in different domains. The beginning of this current "age of AI" has been mainly fuelled by three factors: the availability of strong computational power (in the form of powerful GPU processors), the development of more sophisticated algorithms (represented by the rise of Deep Learning techniques), and the availability of vast volumes of collected data (11).

It is estimated that, by 2030, AI will contribute 12.8 trillion euros to the global economy. This will represent an increase of 14% on today's world gross domestic product (GDP) (12) and as a consequence could boost productivity by up to 40% by 2035 (13). According to (14), many governments around the globe are deploying extensive AI strategic plans with comprehensive policy programmes, research activities, and extensive financial support measures for private investment. Asian governments (China, Japan, Singapore or South Korea) are currently taking the lead in AI. At the same time, Canada and the United States are also developing their own AI strategies. In Europe, only the United Kingdom and Finland have already adopted an AI strategy, but France will surely join in soon.

AI technologies, especially those based on Machine Learning, are increasingly being adopted both in the private and public sectors, for purposes such as pattern recognition in images and object detection, game playing, decision-making, medical and financial applications, and many more. In the future, AI will be present in the likes of self-driving cars, automated package delivery by drones, precision health analytics, and automated factory production processes (14) (15). AI is becoming a general-purpose technology, aiming to profoundly change all aspects of modern life, and many researchers predict that it will deeply transform society in years to come (16).

Machine Learning and Deep Learning

Nevertheless, even with the current boom of AI technologies and applications, confusions and misunderstandings about this field are still widespread among the general public. For instance, nowadays there is a frequent confusion on the difference between the concepts of Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL). While in common speech people have started to use these terms as synonyms, in reality Machine Learning is a sub-category of AI, and Deep Learning a specific subject of Machine Learning (Fig.2) (10).

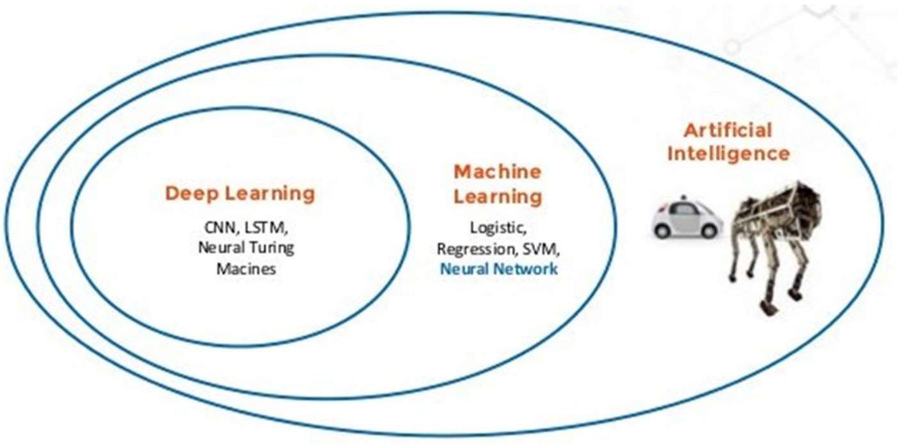
Deep Learning
CNN, LSTM, Neural Turing Macines
Machine Learning
Logistic, Regression, SVM, Neural Network
Artificial Intelligence

Fig. 2. Relationship between Artificial Intelligence, Machine Learning and Deep Learning (taken from (10)*).*

Specifically, **Machine Learning** is a branch of AI that is typically used when there is a large amount of data. It comes into play when, within a large dataset, some patterns exist and are available for use, but it is extremely difficult or infeasible to define them in terms of mathematical rules (17). Instead, Machine Learning uses computer software that can "learn" how to manage these problems, without the need to explicitly "program" them with a mathematical solution. ML techniques usually consist on taking some sample data (called a **training set**), train a multi-purpose model using that data (as well as an optional **verification set**), and use the trained model to make predictions on new data (called the **testing set**). Basically, ML algorithms make the computer create a program to produce a specific output from a known input, in such a way that this program can later provide an intelligent output from a different but similar input (10).

Even as a subfield of AI, Machine learning is a relatively large topic. ML algorithms are grouped in dozens of families, such as Support Vector Machines (SVM) (18), Random Forests (RF) (19) and Artificial Neural Networks (ANN) (20), are available for use both in research and industry. Each of these kinds of algorithms has its own characteristics and learns patterns in unique ways (Fig.3).

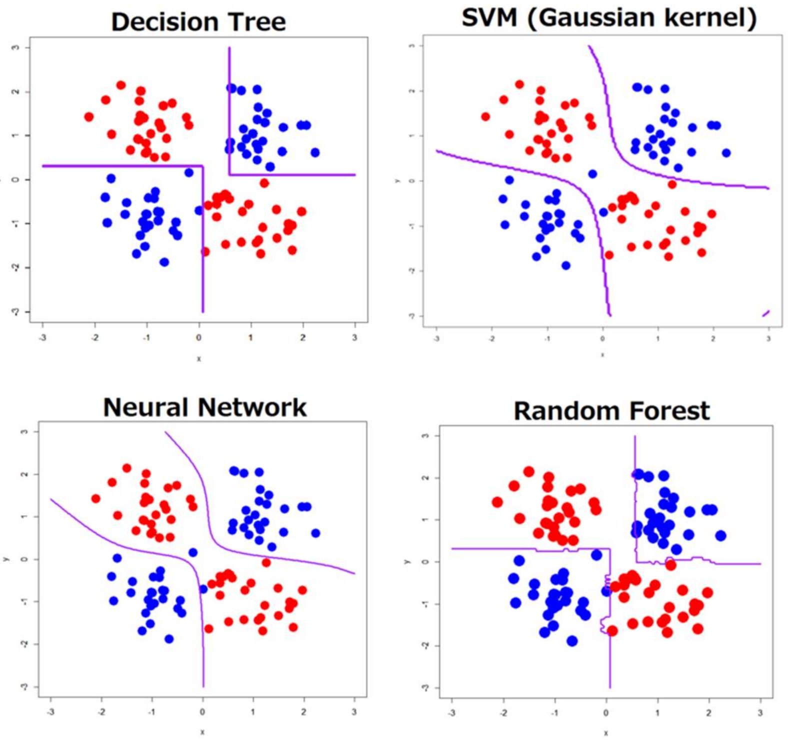

Fig. 3. This supervised learning example shows different Machine Learning algorithms learning output patterns in different ways (taken from (10)*).*

Across all these diverse methods, two main ML philosophies can be distinguished: supervised learning and unsupervised learning. **Supervised learning** is the ML task of learning a function, which maps an input to an output, based on example input-output pairs (21). The most important task types where a supervised learning approach can be applied are classification and regression (11). In classification, the output that the algorithm predicts is a "class" (a discrete label) assigned to the input. In regression, instead of discrete classes, the algorithm tries to predict a continuous function as basis for its output values. Image classification and market predictions are the most common examples of, respectively, a classification problem and a regression problem.

In contrast, **unsupervised learning** is the ML task of learning a function from the data itself, without any information on the expected output, and thus without evaluating the accuracy (the quality) of the predicted output. One of the most notable task where unsupervised learning is used is clustering, the identification of groups of similar examples within the data (11). Unsupervised learning is also used in tasks such as visualisation of high-dimension data (22), and automatic generation of new data (23).

Apart from supervised and unsupervised learning, other learning methods exist, such as semi-supervised learning and reinforcement learning. **Semi-supervised learning** typically performs training using a small amount of labelled data (expected outputs are known) together with a large amount of unlabelled data (expected outputs are not known). This technique is usually applied for prediction tasks, and it has attracted strong interest in the field of cognitive psychology as a computational model for human learning (24). **Reinforcement learning** is a task where an agent learns behaviour through trial-and-error interactions with a dynamic environment (25).

Deep Learning is a subset of Machine Learning related to the development of algorithms called artificial Neural Networks (ANN), but nowadays often simply called Neural Networks (NN). (3) NN algorithms have recently helped make Deep Learning technology widely known by the public, mostly because of Google DeepMind's AlphaGo software (26). In 2016, AlphaGo was able to win against Leo Sedol, the world champion of Go, one of the board games with the highest number of potential move combinations. Deep Learning techniques are currently used to make computers "learn" useful representation of features directly from images, texts and sounds (10).

Artificial Neural Networks

Artificial Neural Networks were developed to mimic the neural functions of the human brain (27). Some of the first forays into the field of algorithms inspired by neural behaviour date back to the mid-20th Century. These include the development of the threshold logic unit by Warren McCulloch and Walter Pitts in 1943, and that of the classic perceptron by Frank Rosenblatt in 1957 (20). Despite this, it has only been in the recent years that NN have experienced a renaissance, fostered by the development of DL techniques and by increasing computational power. The terrain for the current renaissance of NN was set in early 2000s, when it was discovered that NN could be trained efficiently using Graphics Processing Units (GPU). GPU are more efficient for the task than traditional CPU and provide a relatively cheap alternative to specialized hardware (28).

NN are formed by a set of interconnected processing units, called "**artificial neurons**", most commonly organised in layers. NN were originally designed to model the way a biological brain—containing over 10^{11} neurons—processes information, operates, learns and performs several tasks (15). The biological neurons in a human brain, for instance, can be divided into three parts: a number of dendrites which bring electrochemical information into the neuron, a cell body, and an axon which transmits information outside the neuron. An artificial neuron, although essentially a simple mathematical function $f : \bar{x} \to \bar{y}$, is constituted in a similar way to a biological neuron. It has a number of inputs $\bar{x} = (x_0, \dots, x_n)$ which correspond to the neuron dendrites, as well as a weight vector $\overline{w} = (w_0, \dots, w_n)$ where each parameter w_i affects its corresponding input x_i (like the synaptic strength of a dendrite affects the input passing through that dendrite). The processing unit combines the inputs with their corresponding weights via an inner product (a weighted sum) usually adding a bias weight (or threshold weight) b to the sum as follows:

$$x' = \bar{x} * \overline{w} + b = \sum_{i=0}^{n} (x_i * w_i) + b$$

The value x' is then fed to an **activation function**, corresponding to the cell body, that yields the output $y = f(x')$ of the neuron: the value which, in the biological neuron, would be propagated through an axon terminal. The outputs of neurons can then be used as inputs for other neurons, or as global outputs for the system.

In a NN, those layers that are not the output or input layer are called **hidden layers**. NN algorithms are defined by their number of hidden layers, and how these layers are connected together. A NN is called a **recurrent NN** if some of its neurons are linked within loops (the outputs of a neuron are used as inputs for a neuron in a previous layer). If no loops are present, it is called a **feed-forward NN**. A NN with more than two hidden layers can be considered a **"deep" NN**, and the deeper the NN is, the more complex patterns it will be able to recognise.

DL algorithms are characterised in that they build upon the concept of NN and expand it into new kinds of architectures. Some of the most popular sets of such algorithms are: Convolutional Neural Networks – CNN (29), Deep Belief Networks – DBN (30), Deep Auto-Encoders (31), Recurrent Neural Networks – RNN (32), and Generative Adversarial Networks – GAN (33).

Choosing an Activation Function

As said before, activation functions (the "cell bodies" of NN) are used to determine the output of artificial neurons. The resulting values are usually found between a range, such as $[0, 1]$ or $[-1, 1]$, that depends on the function (34). They can be interpreted as providing the answer to a "yes or no" question (for instance: "does this example belong to class A?" or "is this feature present?"). The extremes of the interval represent a pure "yes" or "no" answer, and the values in between represent the probability of a "yes" answer.

Activation functions can be divided into two types: linear activation functions and non-linear activation functions. In the case of **linear activation functions**, as the name implies, the function describes a line, and the range is $]-\infty, \infty[$. The most typical linear function is the identity function $f(x) = x$, which is to say that there is no activation function, and the result of the weighted sum is used directly as the output (Fig.4). This kind of activation function is simple, but it cannot properly represent a "yes or no" question, as the output is not confined within an interval.

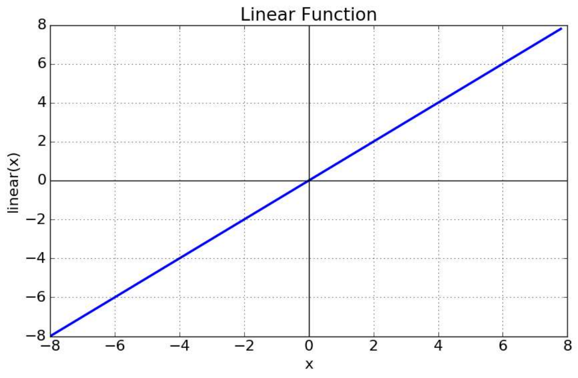

Fig. 4. The identity function returns the same value that it is given (taken from (34)*).*

In contrast, nonlinearity is the lack of linearity, the characteristic of a function whose output does not describe a line. **Nonlinear activation functions** are the most widespread activation functions, as their output can be confined to a range, and thus better represent the answer range of a "yes or no" question. Nonlinear activation functions are mainly divided on the basis of their range or the shape of their curves.

For instance, the logistic activation function, more commonly known as the **sigmoid function**, defines a curve with an S-like shape (hence the name "sigmoid"). The function is defined by the following equation:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid is especially useful in that its range is $[0, 1]$, meaning that its output can be directly interpreted as a probability between 0 and 1 (how likely a "yes" answer is). Still, the sigmoid is not without its problems, as its characteristic S-shape may cause a neuron to get stuck during training. Its weights may evolve in such a way that $\sigma(x)$ will always return the same value for all possible inputs, either 0 or 1.

The **Rectified Linear Unit (ReLU)** is currently the most widespread activation function, used in almost every modern DL application. ReLU is based on a linear (identity) activation function that is modified to become half-rectified: in the negatives, its output is always 0. Its range is $[0, \infty[$, and its equation is:

$$R(x) = max(0, x)$$

The use of ReLU as an activation function marked a breakthrough in the history of NN, enabling the fully supervised training of state-of-the-art deep NN (35). Compared to the sigmoid (Fig.5), ReLU allows for faster and more effective training of NN, especially those with a large number of layers.

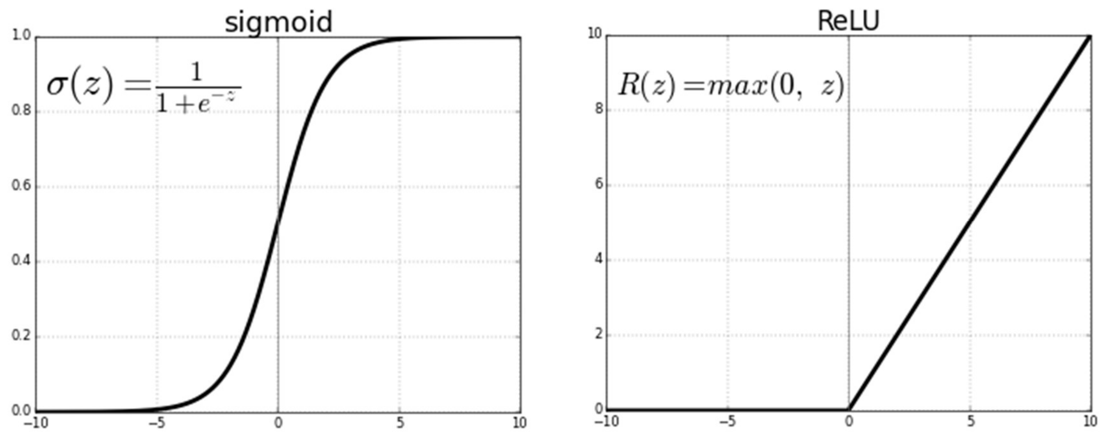

Fig. 5. Comparison of the sigmoid and ReLU activation functions (taken from (34)*).*

While numerous activation functions have been proposed to replace ReLU, none have managed to gain the widespread adoption that ReLU enjoys (36). Many practitioners have favoured the simplicity and reliability of ReLU because the performance improvements of other activation functions tend to be inconsistent across different models and datasets.

However, ReLU has one main problem: since all the negative values are transformed into 0, the model may not be able to properly fit parts of the data with negative inputs. This problem known as the "dying ReLU problem", can be solved by slightly modifying ReLU into the function known as **leaky ReLU**. The range of ReLU is increased (it is now $[-\infty, \infty[$) by replacing the rectification with a "leaky" linear function: in the negatives, $R(x) = a * x$, where a is a constant lower than 1 (Fig.6).

$$R(x) = max(a * x, x)$$

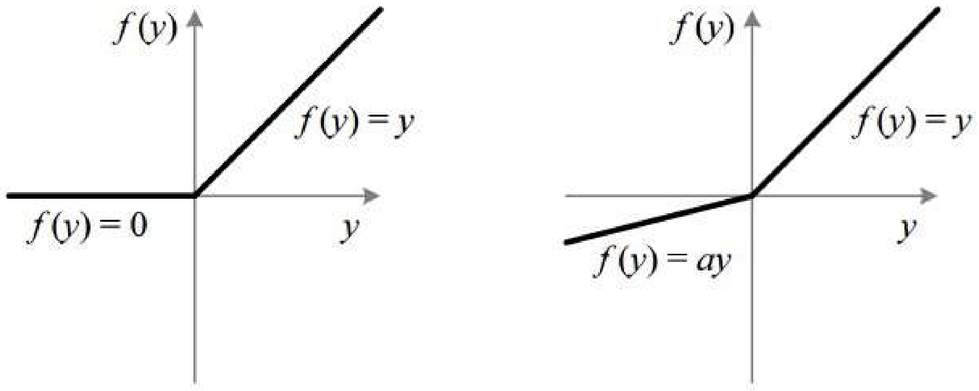

Fig. 6. Comparison of the ReLU and leaky ReLU activation functions (taken from (34)*).*

A second problem with ReLU, which it shares with linear functions, is that its wide range doesn't allow to properly translated its outputs into a range of probabilities (between 0 and 1). This is why whenever ReLU is used as an activation function, the final layer of neurons must always use a different activation function, with a $[0, 1]$ range. The sigmoid function is a good candidate for this task, but there are better alternatives.

One such alternative is the **softmax activation function**, commonly used for multi-class classification problems at the output layer (11). In these problems, the output layer has as many neurons as there are classes, and each of these neurons represents the "yes or no" question of whether or not the input corresponds to one of the classes. The answers to these "yes or no" questions are nonetheless not independent, since each input instance must be classified into one and only one class at the output.

Softmax, unlike the sigmoid function, takes care of this aspect by ensuring that the sum of all outputs across the layer is always equal to 1 (37). The function takes a vector of K arbitrarily large values (the outputs of all weighted sums in the layer) as an input, instead of a single value. Its output is always a vector of values between 0 and 1 that together sum to 1, and these values can be interpreted as interdependent probabilities for each class. Softmax achieves this through the following function:

$$SM(\overline{x_i}) = \frac{e^{x_i}}{\sum_{j=0}^{n} e^{x_j}} \quad \forall i = 0, \dots, n$$

The Multi-Layer Perceptron: Building a Simple Neural Network

The simplest structure for a feed-forward NN is known as the **multi-layer perceptron** (MLP). In it, the output of each neuron in each layer is connected to all of the neurons in the next layer (15), but not to any other neuron in a previous or further layer, or in the same layer. (Fig.7)

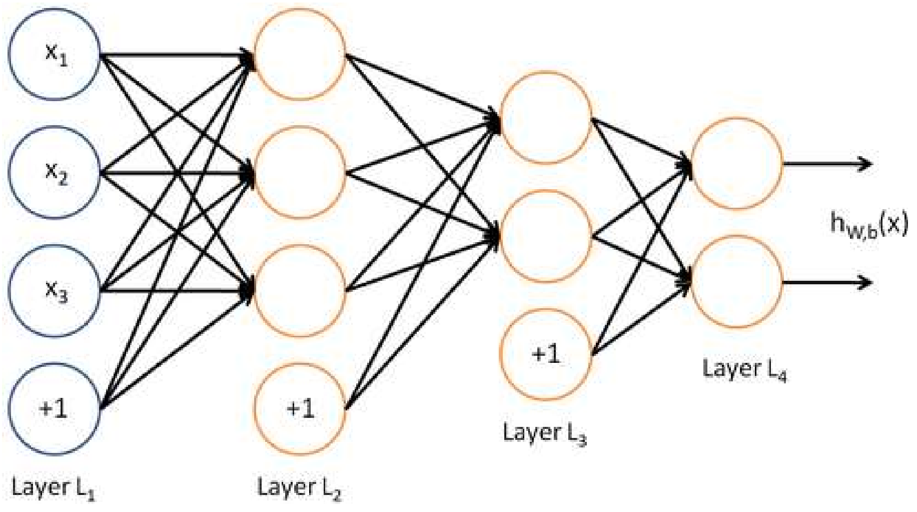

Fig. 7. Diagram of a multi-layer perceptron, a feed-forward NN where every neuron's output is connected to all the neurons in the next layer, and not to any other neuron (taken from (10)*).*

The output of the neurons in one layer thereby becomes the input of each neuron in the next layer. For the last layer, the output of the neurons is the global output of the NN. Hence it is often called the "output layer", in contraposition to the "hidden layers", the intermediate layers between the output and the input. An "input layer" exists but it does not contain neurons: it only distributes the input values to the first hidden layer of neurons.

Because of their structure, MLP are defined as:

- Layered, because their neurons are grouped in layers;
- Feed-forward, because their connections are unidirectional from a previous layer to the next;
- Fully-connected, because every neuron is connected to all of the neurons in the next layer;

Training Through Backpropagation

Perceptrons are not only simple NN structures, they are also among some of the oldest. The original perceptron algorithm was invented by Rosenblatt in 1958 (38). It aimed to connect together and train several NN of the kind originally produced by McCulloch and Pitts in 1943 (39).

These were some of the first ANN algorithms to be produced, and their activation functions were also quite primitive (15). A mere binary threshold was used to produce the output of each neuron, returning 0 for negative values and 1 for positive values. This activation function could be easily used to train a perceptron without a hidden layer and solve linearly separable problems. Problems that are not linearly separable, such as the XOR gate, could only be modelized with the arrival of methods that could train weights beyond one layer. The oldest and most popular of such methods is the backpropagation algorithm (40).

The **backpropagation algorithm**, based on gradient descent optimization, is arguably the most common training algorithm for ANN (15). Its name is an abbreviation of "backward propagation of errors". This alludes to its way of functioning: weight updates that minimize a measure of error, called the **loss function**, are calculated by propagating this measure backwards through the network. From the output to the input layer, this backwards propagation is calculated by computing the partial derivative (gradient) of the loss function with respect to each weight of the NN. Since this gradient gives the direction according to which the loss function grows, the error is minimized by subtracting from the weights a portion of the gradient.

A simple yet complete example of the backpropagation algorithm can be described for the case of an MLP with one hidden layer (41). In this case, the forward propagation of the input through the neural network is described by the following equations.

$$\begin{aligned}
&z_1 = a_0 * w_1 + b_1 \\
&a_1 = f_1(z_1) \\
&z_2 = a_1 * w_2 + b_2 \\
&a_2 = f_2(z_2)
\end{aligned}$$

Where a_λ is the output of layer λ, $f_\lambda(x)$ is its activation function, and z_λ is the result of the weighted sum of that layer (calculated by means of the layer's weights w_λ and biases b_λ). The inputs are a_0, and the global outputs are a_2 (the outputs of the last layer). Note that before any forward propagation takes place, the weights themselves have to be initialized to some values, for instance small random values (27).

To begin the backpropagation process, the received output a_2 of the network is compared to the desired output y by means of the loss function. In this case, the function used as the loss function is the **Mean Square Error (MSE)**, also known as the L$_2$-norm. This function's gradient (here represented with n as the number of outputs) is very easy to calculate, as it is merely the difference between the received outputs and the desired outputs (42):

$$\begin{aligned}
&MSE = \sum\nolimits_{i=0}^{n} \frac{\left(a_{2,j} - y_j\right)^2}{n} \\
&\nabla MSE = a_2 - y
\end{aligned}$$

The value of the gradient of the loss function is the **error value** of the output layer. This value is then propagated back through the network to calculate the error values of the hidden layer neurons. The loss function gradients for the neurons in hidden layers can be solved using the chain rule of derivatives.

$$\begin{aligned}
&\nabla z_2 = \nabla MSE * f_2'(z_2) \\
&\boldsymbol{\nabla b_2 = \nabla z_2} \\
&\boldsymbol{\nabla w_2 = a_1^T * \nabla z_2} \\
&\nabla a_1 = \nabla z_2 * w_2^T \\
&\nabla z_1 = \nabla a_1 * f_1'(z_1) \\
&\boldsymbol{\nabla b_1 = \nabla z_1} \\
&\boldsymbol{\nabla w_1 = a_0^T * \nabla z_1}
\end{aligned}$$

After the backwards propagation, the weights and biases are updated by subtracting a proportion η of their previously calculated gradients from their current value. The proportion ratio η is called the **learning rate** (11), and can be fixed or dynamic (in this case it is fixed). Thus, for each layer λ, the values of the weights and biases are updated in the following manner:

$$\begin{aligned}
w_\lambda &\leftarrow w_\lambda - \eta * \nabla w_\lambda \\
b_\lambda &\leftarrow b_\lambda - \eta * \nabla b_\lambda
\end{aligned}$$

After the weights and biases have been updated, the algorithm continues by executing the two phases of forward and backward propagation, with different inputs every time. The NN is said to “learn” if its weights and biases are converging towards a set of values that minimizes the loss function.

Self-Structuring Neural Networks

An important aspect of the construction of a NN model is the **topology of the network** itself (its structure). It is indeed needed to create a network topology that is adapted to the specific problem that it must solve, and that is able to obtain the best performance for that problem. The search for an optimal topology is due not to the limitations of the learning algorithm that is used, but rather to inherent limits of the network structure and of some of its settings (43). Particular aspects of the topology problem include determining the suitable number of hidden layers, the suitable number of neurons per hidden layer, and the proper connection scheme for these neurons and layers. The optimal setting of parameters such as the learning rate, momentum value, and epoch size of the NN scheme may also be considered part of the topology problem (44).

One of the most obvious aspects of the topology problem is the choice of the size of the NN. A NN that is too small will not be able to properly “learn” the problem, while a NN that is too large may produce poor generalization performance (45). On one hand, a poorly structured NN (46) or a NN with few hidden neurons (47) may underfit the dataset, as it will lack the representational power to model the diversity of the dataset items, or the complexity inherent in them. On the other hand, a system that has been structured to suit every single item in the dataset may cause the system to be overfitted (48), owing to its excess information capability (49). Another aspect of the topology problem is to find an acceptable value for the error rate of the NN model, the measure of error after which the model has converged and is unable to be further improved (46). For instance, the model designer may set the error rate to a value that is unreachable, causing the model to stick in local minima (44), or to a value after which the model could actually be further improved.

The topology of NN is commonly decided by means of **trial and error**: the parameters are tuned in order to better accomplish a certain task. This technique may prove itself fruitful in domains where rich prior experience and knowledgeable experts are available (47). But since this kind of prior knowledge and expertise is hard to get in practice, the process often becomes a tedious and time-consuming search for the correct values (48). Despite the criticisms of trial and error, there are relatively very few scientific papers focused on improving a NN’s performance on basis of dynamically modifying its topology.

The existing public available researches, although few, can nevertheless offer interesting strategies for the modern NN researcher. Algorithms where the NN structure is **self-constructing**, such as those described in these papers, provides a timely solution that might displace some of the burden from system designers. It is for this reason that they have in recent times gained the attention of many researchers (47). For instance, in his 1994 paper on Dynamic Learning Algorithms (43), Sam Waugh reviews various strategies for gradually improving NN architectures by means of pruning: pruning on connections, pruning on weights, and construction or pruning of hidden nodes. More recently, in 2016 Fadi Thabtah, Rami M. Mohammad, and Lee McCluskey (47), presented an opposite solution where the NN model is restructured by incremental means: tuning some parameters, adding new neurons to the hidden layer or sometimes adding a new layer to the network.

It should nevertheless be reminded that automating the structuring process of NN topologies is not only about adding new neurons or hidden layers. Sometimes indeed additional hidden neurons do not essentially correlate with an improved accuracy (50), and the tuning of several parameters (such as the learning rate) must be made in conjunction with the addition of new neurons and layers. Sadly, many of these parameters, including the learning rate, are still set through a trial and error procedure (51), and statistical tests must be made to find the proper values that optimize the automatized structuring process.

Different Approaches to Self-Structuring

Three main approaches exist to the automation of an ANN's structuring process (51). These are: constructive algorithms, pruning algorithms, and constructive-pruning algorithms.

Constructive Algorithms (also known as **incremental algorithms**) start with a simple NN structure, such as one hidden layered NN with a single neuron in the hidden layer (49). Recursively new items and parameters (hidden layers, hidden neurons, connections, etc.) are added to the initial structure, until a satisfactory result is reached. After each addition, either the entire network topology or only the recently added parameter(s) is retrained and cannot be modified. Constructive approach algorithms tend to be relatively simple to tune because they usually depend on few initial parameters (i.e. the number of neurons in the input layer, the epoch size, the new parameters added to the network). These algorithms are also computationally efficient, since they search for small structures (47). It should nevertheless be considered that the constructive approach has some hurdles that should carefully be addressed (45). It is indeed the programmer who, when building the algorithm, should decide when to add a new parameter, when to stop the addition process, and when to stop training and building the network.

The first constructive algorithm is most likely Dynamic Node Creation (DNC) (52). This algorithm adds neurons in the hidden layer one by one whenever the MSE on the training dataset stops variating, until the network achieves a predefined desired error rate. The algorithm retrains the entire network, meaning that each addition is irreversible. Although DNC is simple, the structures that it produces tend to be complicated because it keeps adding hidden neurons until the desired error rate is reached (47).

Another example of a constructive algorithm is the Cascade-Correlation algorithm (CC-Alg.) (53). This algorithm generates NN with multiple hidden layers, each of which consists of only one hidden neuron where each neuron is linked to previously added neurons. Its main issue is that the generalization ability of the produced NN may decrease if the number of hidden layers increases (45). Several improvements on the CC-Alg. have been proposed to facilitate adding more than one neuron in each hidden layer, such as the one proposed by L. Ma and K. Khorasani (54).

Pruning Algorithms start with an oversized NN structure (such as a NN with multiple hidden layers and a large number of neurons in each layer) (47). Recursively, some items and parameters (hidden layers, hidden neurons, connections, etc.) are removed from the network. After each training process, the new pruned structure is retrained for a while so that the remaining parameters can compensate for the roles of the removed parameters. If the network performance improved, more parameters are removed and the resulting network is again retained. However, if the network performance does not improve, the deleted parameters are restored and others are removed instead. This process is repeated recursively until achieving the final network topology. Usually, only one item or parameter is removed in each pruning phase (52). This approach is not without disadvantages: it may be too time consuming, and the programmer needs to decide beforehand how big the initial NN structure should be for the specific problem in question.

Constructive-Pruning Algorithms work in two phases: a constructive phase and a pruning phase (47). During the constructive phase new hidden layers, neurons, and connections are added. This phase may result in a ridiculously complicated structure, so a pruning phase is employed to simplify the network structure while preserving the network performance. In some algorithms the pruning phase may run simultaneously with the constructive phase, and in others it is started as soon as the training phase is accomplished. Examples of constructive-pruning algorithms for NN can be found at (45), (49) and (55).

The EMANN algorithm

Overview of the EMANN algorithm

As described in the original 1994 paper by Tristan Salomé and Hugues Bersini (6), **EMANN** (Evolving Modular Architecture for Neural Networks) is a multi-purpose algorithm for building self-structuring Neural Network classifiers. This algorithm follows the constructive or incremental approach to building the NN's topology (although some pruning aspects are present too), and simultaneously builds and trains the network for a given problem.

The incremental approach is evident throughout the design of the EMANN algorithm, particularly in two of its main aspects. The first aspect is that the algorithm conceives the NN as a **cascade of modules** built on top of each other. Each of these modules is a fully-connected MLP with one hidden layer, and sigmoid as the activation function for all layers. These modules are incrementally added "on top of" each other by making sure that they are complementary to the previous ones. Their output layer always has the same shape (that of the NN's global output), while their input layer depends on previous modules. The second aspect is that each module, once it is added, **increases the number of neurons** inside its hidden layer through a complex series of steps, that depends on a series of parameters. These steps take place at the same time as the training epochs (which are performed through backpropagation or a similar algorithm), and their completion is marked by the unfolding of these epochs. They are described in detail in the next few sections.

The concept of Connection Strength (CS)

In EMANN, the weights and biases of a module's hidden layer play an important role in its incremental building. During the training of a module, each neuron is assigned a value called the **connection strength (CS)**. This local variable is crucial for the incremental evolution of a module's structure, and much of the module building algorithm depends on its fluctuations. It is calculated as the average of all the (absolute values of the) weights connected to the neuron. For a neuron j in layer λ, connected to n inputs through an equal number of weights, the CS is given by the following equation:

$$CS_j = \frac{\sum_{i=0}^{n} w_{\lambda,ij}}{n}$$

Since in EMANN the activation function for all layers is a sigmoid, the CS was identified as an important variable because of its relationship with the behavior of this activation function. The value of the CS correlates with how "sharp" a neuron's output slope is around the inflection point of the sigmoid, and can therefore become a particularly useful indicator of how much the neuron has trained. A neuron is considered "settled" if its CS is higher than a **settling threshold (ST)**, and "useless" if its CS is lower than a **uselessness threshold (UT)**.

The "Ascension" stage

After a new module has been created, the first stage of its construction is called the "Ascension" stage. At the beginning of this stage, the module starts with only one neuron in its hidden layer. Neurons are progressively added to the layer until the algorithm observes that a neuron has become "settled". The rate at which the neurons are added is given by a fixed **ascension threshold (AS)** parameter: one new neuron is added every AS training epochs. At the end of this stage, the module's hidden layer should have become filled with a large number of neurons.

It is worth noting that the AT has an alternative value for the first trained module that is different from the one used for all other modules. This peculiarity was added because it was found to help preventing the so-called "copy problem", where new modules would first tend to replicate the outputs of the module directly below them.

The "Improvement" stage

Once at least one neuron in the module's hidden layer has become "settled", the module switches to the "Improvement" stage. During this stage, the algorithm waits for a fixed number of training epochs, called the **patience parameter (PP)**, to move on to the next stage. If another neuron settles before PP training epochs have been performed, a new neuron is added and the number of waiting epochs is reset to zero.

Ideally, this stage is considered as the moment when the optimal number of hidden neurons has more of less been reached, and the module is trying to reach a solution. The new neurons are meant to give the module one last push to make the solution even better.

Pruning and recovery

Once PP epochs have elapsed since the last neuron settled, the module enters its "Pruning" stage. This is the only stage where neurons can be removed from the module, which afterwards will retain its acquired topology throughout the rest of the network's construction. The accuracy of the network (for the training set) is calculated and stored in a variable, then every neuron that is "useless" is pruned from the hidden layer.

After the hidden layer has been pruned, the network's accuracy is once again calculated, and compared with its value before pruning. If the accuracy has decreased since the layer was pruned, a fourth "Improvement" stage takes place where the module is simply trained until the pre-pruning accuracy has been recovered. Otherwise, the module is retained and the algorithm moves on to build the next module.

Building multiple modules

As stated before, EMANN incrementally adds modules "on top of" each other, connecting the inputs of each new module to the elements underneath it. Modules use as their input all of the outputs of previous modules, together with the global input. This implementation (Fig.8) is meant to ensure that modules are truly complementary to each other. As a consequence, even if some modules have processed their inputs incorrectly, the modules above them are not affected. New modules are added until the difference in MSE before and after the addition of the last module was less than a fixed **improvement threshold** parameter.

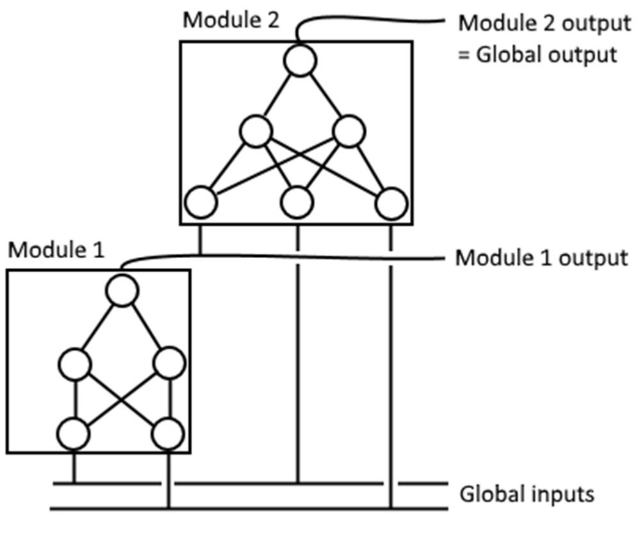

Fig. 8. Diagram of the incremental addition of new modules. Using as inputs all of the previous modules' outputs ensures that modules are truly complementary to each other (slightly modified from that on (6)*).*

Methodology

Why TensorFlow?

For carrying the experiments and tests required for this master thesis, EMANN was implemented using Python (version 2.7.12), and the TensorFlow library (version 1.4.0). **TensorFlow** (7) is an open source software library for high performance numerical computation, originally developed by researchers and engineers from the Google Brain team. TensorFlow provides a full set of matrices and vectors operations that are relevant to the implementation of ML and DL algorithms, which are designed to seamlessly exploit both the GPU and CPU resources of the computer.

The greatest advantage of TensorFlow, how low-level the package is, may also be the greatest inconvenient to working with it. While a low-level package allows to implement highly customized networks, including self-structuring ones, it also makes some of the simplest tasks more complicated than they are in most packages. In the end, the advantages outweigh the disadvantages, as nowadays not many ML packages for Python allow for the coding of self-structuring NN, and while TensorFlow was not meant for it, with some clever coding it is possible to implement such algorithms and many more.

The data set: CIFAR-10

Finding a proper benchmark task was required to test the TensorFlow implementation of EMANN. In the end, the CIFAR-10 classification task was used, since TensorFlow tutorials on the Web tended to make use of such a classification task. In particular, the choice of using CIFAR-10 as the benchmark for implementing EMANN was greatly influenced by one online tutorial (5), which shows how to build a simple MLP with one hidden layer in TensorFlow.

CIFAR-10 (8) is a dataset consisting of 60000 32x32 RGB images of animals and vehicles, to be classified into 10 mutually exclusive classes. These classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Implementing EMANN with TensorFlow

EMANN was implemented as three separate Python files. One first file, the executable `run_EMANN_algo.py`, contains the main EMANN code and the different stages for constructing a module: "Ascension", "Improvement", "Pruning", and "Recovery". A second file, `EMANN_original.py`, contains an object class representing a module, an initializer for such a class, and several relevant functions and methods for the module class. These include (among others) functions for adding and pruning neurons, calculating the CS of neurons, performing backpropagation-based training of the network, and measuring the accuracy of the module with respect to a given dataset. A third file, `data_helpers.py`, was directly taken from the tutorial for building a single-layered MLP, and used for preparing the different data batches from CIFAR-10's training dataset.

During the implementation of EMANN with TensorFlow, one of the main issues was to implement an MLP topology that could be automatically incremented and pruned without losing its weight values. Very early in development, it was decided to follow the tutorial's example of conceiving the mechanisms of the MLP as **matrix operations**, and to remove the bias terms to simplify the calculations.

Using this formalism, for instance, calculating the weighted sum for all the neurons in a layer would correspond to a matrix product of the previous layer's outputs (a vector matrix) with the weights for that layer (a 2D matrix). For a layer λ with m neurons, for which the previous layer $\lambda - 1$ had n neurons, the calculation of the weighted sum would amount to the following equation:

$$[a_{\lambda-1;1} \quad \cdots \quad a_{\lambda-1;n}] \times \begin{bmatrix} w_{\lambda;1,1} & \cdots & w_{\lambda;1,m} \\ \vdots & \ddots & \vdots \\ w_{\lambda;n,1} & \cdots & w_{\lambda;n,m} \end{bmatrix} = [z_{\lambda;1} \quad \cdots \quad z_{\lambda;m}]$$

The $z_{\lambda;i}$ terms for all $i = 1, \dots, m$ would then be passed through the λ layer's activation function to obtain its output. In TensorFlow, matrices and vectors such as these are known as tensors. The definition of tensors and their interrelationships as a flow of operations form the basis of TensorFlow's main formalism (hence the name).

Based on this matrix formalism, it became clear how the processes of adding and pruning neurons in a hidden layer could be implemented. Adding a new hidden neuron would amount to adding a new column to the weight matrix before the hidden layer, and a new row to the weight matrix after that layer. Pruning a neuron would amount to removing its corresponding column and row in these weight matrices. The flow of operations through the other tensors would then have to be updated to take into account these changes.

Although the TensorFlow library does not contain operations for directly adding or erasing lines in a tensor, it does contain a method for grouping together a list of tensors contained in a Python list into a single tensor of a higher dimension. It was thus decided to store the weight matrices not as 2D tensors, but as **Python lists containing 1D tensors (vectors)**. For the first weight matrix these tensors represent columns, and for the second one they represent rows. Python lists can be appended or pruned with ease, so a list of 1D tensors can form the perfect basis for a self-structuring topology.

The flow of operations in which these lists of vectors are involved is achieved through the TensorFlow operations "stack" and "unstack". For the forward propagation of outputs, each weighted sum is defined as the matrix product of the previous layer's outputs with the "stack" of the list of weight tensors, put together along the proper dimension (rows or columns). For the backpropagation, the new vector lists are recalculated as the "unstack" of their respective weight matrices.

Changes in EMANN

Because of the nature of the benchmark problem (CIFAR-10) and of the goals of this master thesis, some key modifications were made for the implementation of EMANN that was produced in Python and TensorFlow, with respect to the original implementation described in the 1994 paper (6). An extensive overview of these differences and their reasons to be are presented in this section.

Concerning the definition of "settled" and "useless" neurons, it was observed that in further layers the maximum and minimum values of CS tended to be different than in the first one. Consequently, it was decided that in the new implementation the ST and UT would have (like the AT) alternative values for building the first module, different from the ones used for all other modules.

In EMANN, there are two stages where the algorithm must periodically check for "settled" neurons: the "Ascension" stage and the "Improvement" stage. In the original implementation, the benchmark was simple enough to allow for the possibility of checking the state of neurons once at every epoch. The current implementation is far more complex, and it would take far too much time to perform this "standard check" on every epoch. A fixed parameter (**check threshold**) was used to determine how many epochs to wait between each "standard check" for "settled" neurons in the module's hidden layer.

After several tests, it was noticed that many of the structures built by EMANN showed a very low number of neurons, as well as very low accuracy levels when compared to an implementation of a fixed single-layered MLP topology. It was then decided that the original EMANN algorithm was too slow when increasing the number of neurons in modules. The proposed solution was to make a slight modification to the first steps of a module's construction. Each new module starts with a fixed **initial unit number**, which by default would be set to 1 neuron (as in the original algorithm). During the "Ascension" stage, every AS epochs, a fixed number of neurons called "**unit increase**" (less or equal to the **initial unit number**) is added to the module, also set to 1 by default.

Since one of the main goals of this master thesis was to use EMANN to automatically build fully-connected topologies similar to MLP, the way in which modules were stacked together was also modified in the new implementation. In the original formulation, modules used as their input all of the outputs of previous modules, together with the global input. This was changed in the present implementation: the input of each module is now the hidden layer of the previous module (Fig.9). This emulates, through the construction of each module, the building of one hidden layer in the MLP.

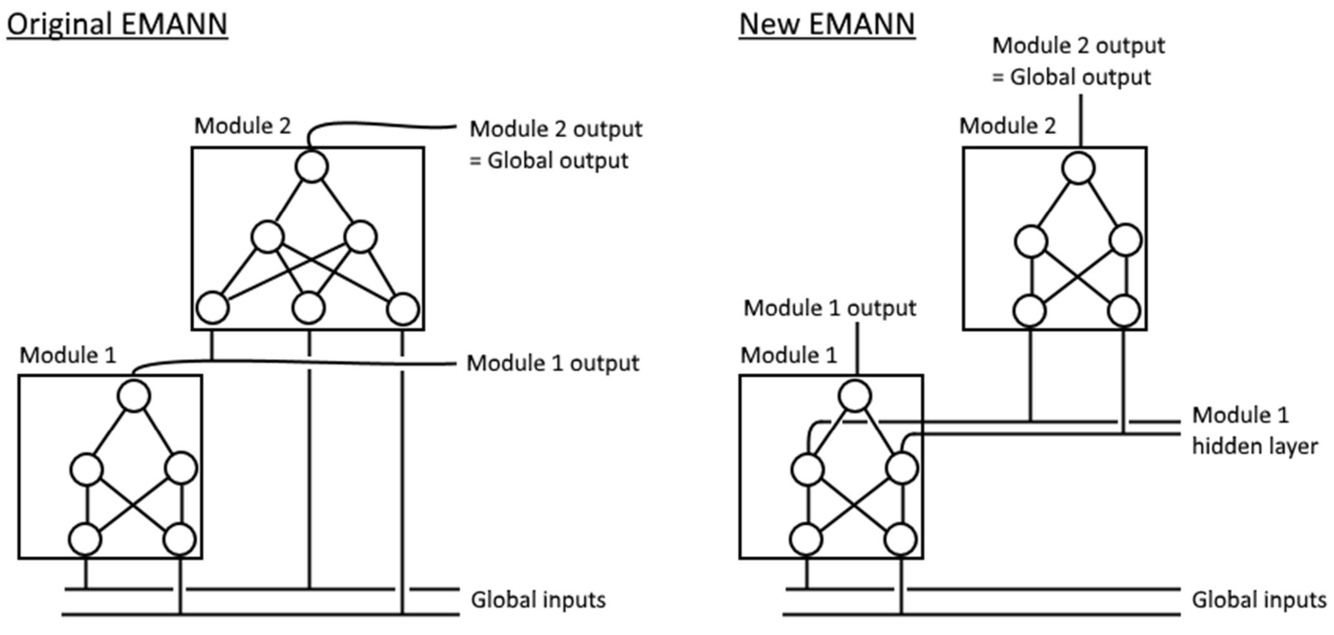

Fig. 9. Diagram of the incremental addition of new modules, in the original EMANN algorithm and in the new implementation of EMANN. The new implementation is meant to resemble the structure of an MLP.

Finally, in order to make the new implementation of EMANN more flexible for testing, it was decided to add more flexible stopping criterions for certain parts of the algorithm. Most notably, the **number of modules to build** was made into an independent parameter, rather than tied to the difference in MSE between two modules. This allowed to only build one or two modules every time EMANN was launched, without needing to manually shut down the algorithm. In later tests, since in some cases a module would not leave the ascension threshold, it was decided to set an **ascension limit** parameter. The purpose of this was to ensure that if a certain (large) number of neurons was reached, the algorithm would skip directly into the improvement stage regardless of whether any neurons had been counted as "settled" or not.

EMANN-like: new kinds of modules

One of the main objectives of this master thesis was to use EMANN as a basis for creating various new incremental algorithms that make use of currently relevant DL trends and techniques. These **EMMAN-like** algorithms share their module-building procedure with the implemented EMANN, but the activation functions in their modules are different.

As previously explained, the module object had been implemented in a separate Python file from the main body of the EMANN algorithm. EMANN-like algorithms were thus implemented as modifications of the module object file, which could be loaded and used by the main executable file without making any modifications to it (other than changing which file to import the module from).

The final implementation allowed for testing three kinds of EMMAN-like algorithms:

- `EMANN_original.py`: an algorithm that closely resembled the original EMANN, using the sigmoid function as the activation function for all of its layers.
- `EMANN_softmax.py`: an algorithm that uses softmax as the activation function for all of its layers.
- `EMANN_ReLU.py`: an algorithm that uses a leaky ReLU function as the activation function for the hidden layer (with a leak constant of 0.2), and softmax as the activation function for the output layer.

Test Design

The tests were run in an HP Pavilion 15 Notebook PC, running a Linux Ubuntu 16.04 Bash console within a Windows 10 Home (64 bit) OS, with the following specifications:

- 8.00 GB of RAM (7.78 GB available).
- CPU: Intel Core i7 4500U with four cores (1.80 GHz).
- GPU: NVIDIA GeForce GT 740M.

In order to assess the effect of several parameters in the performance of the various implemented EMANN-like algorithms, a number of tests were designed.

The **first test** consisted in a comparison, among the three EMANN-like implementations, of the accuracy with respect to different values of the settling threshold (ST) and ascension threshold (AT) parameters. The accuracy levels were measured for both the last training batch and the testing set at two points during the module building algorithm: at the end of the "Ascension" stage, and at the end of EMANN's full module construction cycle.

The **second test** consisted in yet another comparison among the three implementations. This time, the impact of different values for the ST and AT parameters was measured on the total number of neurons in the hidden layer. Since it was observed that the number of neurons changes very little after the "Ascension" stage, this measure was only made at one of the two previously mentioned points of the algorithm: the end of a full EMANN cycle.

The **third test** was designed to gain insight on a phenomenon that was observed in the two previous tests. In the sigmoid and softmax using implementations, the maximum CS of neurons always grew consistently, and easily crossed past the ST value. This was not the case in the leaky ReLU implementation, which was very often unable to reach the ST. It was found that in that implementation the maximum CS tended to increase rapidly, then stagnate for a while on a random value (depending on how rapidly it was able to increase), and finally begin to slowly but steadily decrease. The test consisted in running the ReLU implementation 10 times with different values of ST and AT. The number of times when the neurons were unable to settle was counted, and the influence of ST and AT on this phenomenon was assessed.

The **fourth test** consisted in a comparison, among the three EMANN-like implementations, of the accuracy level with respect to different values of the patience parameter. Since this parameter is relevant to the algorithm only from the "Improvement" stage onwards, after the "Ascension" stage is already finished, the accuracy levels were once again only measured at the end of the full EMANN cycle.

The **fifth test** consisted in assessing the influence on the accuracy, among the three EMANN-like implementations, of different values for the initial unit number and for the unit increase during the "Ascension" stage.

The **sixth test** consisted in assessing the level of accuracy acquired by a second module, receiving the outputs of the first module's hidden layer, for the best performing modules configuration tested in previous experiments.

For the first four tests, results were statistically analyzed through a one-way or two-ways Analysis of Variance (ANOVA) depending on the test. In the tests, three replicas were executed for each of the different combinations, and a 5% (0.05) probability confidence level was used for assessing the statistical significance of the results. For the last two tests, a simple assessment based on running several tests iterations was made. In all tests, discussion on the obtained accuracies was informed by a "**reference test**", performed on a simple MLP structure with one hidden layer, containing a fixed number of neurons on this layer. This structure was taken from the online tutorial (5) used as a basis to implement EMANN in TensorFlow.

Results and Discussion

All tests, including the “reference test”, were performed with a training batch size of 400 examples per epoch. The “reference test” was performed with a learning rate of 10^{-3}, but since this setting proved to be too coarse for EMANN implementations to converge, all the other tests were performed with a learning rate of 10^{-4}.

Reference Test: Prebuilt Single-Layer MLP

The single-layer MLP was prebuilt as described in the aforementioned online tutorial (5). Its number of hidden neurons was set to 120, and the number of epochs for the training was set to 2000. Results were as follows:

Training	Testing
0,5550	0,4522
0,6075	0,4691
0,5850	0,4691
0,5850	0,4577
0,6325	0,4726
0,5825	0,4686
0,5500	0,4597
0,6075	0,4681
0,6425	0,4702
0,6125	0,4561

Average	0,5960	0,4643
STD	0,0303	0,0072

From these results it can be deduced that, with little deviation, the accuracy of a single-layer MLP architecture trained on the CIFAR-10 dataset tends to be around 60% for the last training set batches, and around 46% for the testing set. These results are relevant when discussing the accuracy obtained by EMANN implementations.

First Test: Effect of "Ascension" Stage Parameters on Accuracy

For these tests, only one EMANN module was built, with UT = 0.013 as its uselessness threshold and PP = 4000 epochs as its patience parameter. The tested settling threshold values were ST = 0.014, 0.016, and 0.019. The tested ascension threshold values were AS = 200 epochs, 500 epochs, and 800 epochs.

The sigmoid using module implementation produced the following results:

Ascension Threshold	Settling Threshold	Ascension Stage Only				Full EMANN Cycle			
		Accuracy Tr.		Accuracy Test.		Accuracy Tr.		Accuracy Test.	
		AVERAGE	STD	AVERAGE	STD	AVERAGE	STD	AVERAGE	STD
200	0,014	0,1300	0,0180	0,1356	0,0313	0,1767	0,0406	0,1882	0,0275
	0,016	0,1617	0,0311	0,1598	0,0693	0,2367	0,0256	0,2290	0,0617
	0,019	**0,2992**	0,0884	**0,2864**	0,0218	**0,3650**	0,0552	**0,3628**	0,0402
500	0,014	0,1092	0,0232	0,1105	0,0224	0,1233	0,0375	0,1367	0,0061
	0,016	0,1283	0,0194	0,1475	0,0238	0,1692	0,0337	0,1675	0,0239
	0,019	0,1333	0,0118	0,1375	0,0345	0,2017	0,0257	0,2210	0,0126
800	0,014	0,1350	0,0075	0,1309	0,0319	0,1575	0,0208	0,1605	0,0028
	0,016	0,1492	0,0407	0,1500	0,0184	0,1583	0,0195	0,1646	0,0019
	0,019	0,1108	0,0191	0,1103	0,0493	0,1808	0,0196	0,1619	0,0382

	Ascension Stage Only		Full EMANN Cycle	
	Accuracy Tr.	Accuracy Test.	Accuracy Tr.	Accuracy Test.
Probability Ascension Threshold	**0,0149**	**0,0008**	**0,0000**	**0,0000**
Probability Settling Threshold	**0,0009**	**0,0120**	**0,0001**	**0,0000**
Probability Interaction	**0,0016**	**0,0011**	**0,0145**	**0,0017**

The probabilities resulting from the ANOVA indicate that the variation in results is statistically significant (marked in bold) for the variation of both parameters in all cases, as well as for the interaction between these parameters. It appears that the accuracies obtained in this test are quite poor when compared to those from the "reference test", although there seems to be no sign of overfitting as the testing set accuracy is always close to the one for the training batches. Of notable interest is that a significantly higher accuracy than for any other combination (36.2%) is found for the combination ST = 0.019 and AT = 200 (highlighted in bold).

The softmax using module implementation produced the following results:

Ascension Threshold	Settling Threshold	Ascension Stage Only				Full EMANN Cycle			
		Accuracy Tr.		Accuracy Test.		Accuracy Tr.		Accuracy Test.	
		AVERAGE	STD	AVERAGE	STD	AVERAGE	STD	AVERAGE	STD
200	0,014	0,1742	0,0643	0,1667	0,0800	0,2933	0,0621	0,2656	0,0288
	0,016	0,1850	0,0066	0,1959	0,0385	0,2600	0,0157	0,2448	0,0366
	0,019	**0,2967**	0,0063	**0,2861**	0,0066	**0,3100**	0,0044	**0,3043**	0,0110
500	0,014	0,1392	0,0300	0,1339	0,0423	0,2067	0,0091	0,1959	0,0408
	0,016	0,2125	0,1128	0,1946	0,0634	0,2400	0,1054	0,2319	0,0698
	0,019	0,2492	0,0184	0,2437	0,0161	0,2667	0,0068	0,2650	0,0079
800	0,014	0,1300	0,0218	0,1339	0,0639	0,2033	0,0243	0,2222	0,0469
	0,016	0,1667	0,0416	0,1688	0,0932	0,2400	0,0432	0,2254	0,0592
	0,019	0,2275	0,0238	0,2276	0,0080	0,2633	0,0201	0,2600	0,0350

	Ascension Stage Only		Full EMANN Cycle	
	Accuracy Tr.	Accuracy Test.	Accuracy Tr.	Accuracy Test.
Probability Ascension Threshold	0,1818	0,1968	0,0992	0,1084
Probability Settling Threshold	**0,0005**	**0,0003**	0,2144	**0,0493**
Probability Interaction	0,6947	0,9210	0,8046	0,8168

In this case, the ANOVA indicates that the variation in results is only statistically significant in what concerns the variation of the settling threshold (in bold). Yet again, the accuracies tend to be poorer than in the "reference test", although in average better than for the sigmoid using module. Once again, the best accuracies (30.4%) are found for the combination ST = 0.019 and AT = 200 (highlighted in bold).

The ReLU using module implementation produced the following results:

Ascension Threshold	Settling Threshold	Ascension Stage Only				Full EMANN Cycle			
		Accuracy Tr.		Accuracy Test.		Accuracy Tr.		Accuracy Test.	
		AVERAGE	STD	AVERAGE	STD	AVERAGE	STD	AVERAGE	STD
200	0,014	0,1600	0,0205	0,1347	0,0055	0,1492	0,0138	0,1285	0,0058
	0,016	0,1675	0,0238	0,1572	0,0186	0,1542	0,0225	0,1438	0,0023
	0,019	0,0967	0,0388	0,1030	0,0057	0,1425	0,0325	0,1424	0,0095
500	0,014	0,1025	0,0487	0,1023	0,0214	0,1217	0,0341	0,1224	0,0294
	0,016	0,1108	0,0139	0,1174	0,0106	0,1450	0,0238	0,1401	0,0159
	0,019	0,1250	0,0274	0,1403	0,0006	0,1283	0,0153	0,1097	0,0167
800	0,014	0,1208	0,0347	0,1266	0,0264	0,1508	0,0170	0,1271	0,0151
	0,016	0,0983	0,0444	0,1103	0,0353	0,1325	0,0229	0,1259	0,0140
	0,019	0,1197	0,0371	0,1284	0,0284	0,1625	0,0087	0,1454	0,0140

	Ascension Stage Only		Full EMANN Cycle	
	Accuracy Tr.	Accuracy Test.	Accuracy Tr.	Accuracy Test.
Probability Ascension Threshold	0,1422	0,4440	0,2429	0,1751
Probability Settling Threshold	0,8756	0,7591	0,9508	0,3610
Probability Interaction	**0,0333**	**0,0138**	0,4387	0,1195

The results of the ANOVA only indicate statistical significance for the interaction between two parameters at the end of the ascension stage. This means that there are some combinations which may show statistically significant differences, but it cannot truly be concluded which combination is best without further tests and a more detailed analysis. The accuracy results are nevertheless very poor for this module implementation.

Second Test: Effect of "Ascension" Stage Parameters on Neuron Count

The settings for these tests were the same as for the first tests. Only one module was built each time, with an uselessness threshold of UT = 0.013 and a patience parameter of PP = 4000 epochs. The tested settling threshold values were once again ST = 0.014, 0.016, and 0.019, and the tested ascension threshold values were also once again AS = 200 epochs, 500 epochs, and 800 epochs.

This group of tests produced the following results:

		ReLU		Sigmoid		Softmax	
Ascension Threshold	Settling Threshold	AVERAGE	STD	AVERAGE	STD	AVERAGE	STD
200	0,014	3,00	1,00	3,33	2,08	11,00	1,73
	0,016	4,67	1,53	5,00	2,00	11,00	7,21
	0,019	4,33	0,58	**16,00**	6,00	**26,00**	5,29
500	0,014	4,33	1,15	1,00	0,00	3,67	2,08
	0,016	4,33	0,58	2,33	1,15	5,67	5,51
	0,019	3,33	0,58	3,67	0,58	13,67	3,06
800	0,014	3,67	2,08	1,00	0,00	5,00	2,65
	0,016	4,33	0,58	1,00	0,00	6,67	4,16
	0,019	3,67	2,52	1,67	1,15	10,00	1,00

	ReLU	Sigmoid	Softmax
Probability Ascension Threshold	0,9802	**0,0000**	**0,0003**
Probability Settling Threshold	0,4398	**0,0002**	**0,0001**
Probability Interaction	0,6742	**0,0012**	0,1746

The calculated probabilities from the ANOVA indicate that only the result variations for the sigmoid using module and the softmax using module are statistically significant, and they are so for the variations of both parameters (in bold). In both module implementations the greatest number of neurons seems, like the greatest level of accuracy, correlated with the combination ST = 0.019 and AT = 200 (highlighted in bold).

A logical explanation of why a greater number of neurons would be correlated with greater accuracy would be that, since with more neurons the network is able capture more nuances and patterns within the data, then its accuracy when classifying samples also increases.

Third Test: Analysis of Settling Failure in ReLU Module

The aim of this third test was to assess the frequency of a problem where, on many occasions, the neurons in the ReLU module would fail to settle during the “Ascension” stage. Once again, only one module was built each time with a uselessness threshold of UT = 0.013 and a patience parameter of PP = 4000 epochs. The tested ST and AT values were the same as in previous tests.

This group of tests produced the following results:

	AT = 200	AT = 500	AT = 800
ST = 0.014	10	20	40
ST = 0.016	60	40	30
ST = 0.019	70	70	60

The probabilities from the ANOVA test are 6.62% for the settling threshold and 95.60% for the ascension threshold. This indicates that, although the variation in results is not statistically significant, it comes close to be significant for the variation in the settling threshold. A larger number of failed launches of the ReLU module appears indeed to be correlated with larger values of the settling threshold.

A logical explanation for this correlation could be found in the sudden increase in CS exhibited by these modules at the beginning of training. Indeed, as explained before, the maximum CS of ReLU using modules tends to increase rapidly, then stagnate for a while before finally decreasing steadily. Depending of the time before the CS begins to stagnate, this stagnation may happen at different CS values, and higher values may be harder to reach before stagnation than lower ones.

Fourth Test: Effect of Patience Parameter on Accuracy

As in the previous tests, only one EMANN module was built with UT = 0.013 as its uselessness threshold. On the basis of results obtained in previous tests, it was decided to set the ascension threshold to AS = 200 epochs. The settling threshold was set to ST = 0.019 for the sigmoid and softmax using modules, as this value tended to give a higher accuracy. For the ReLU using module, the settling threshold was set to ST = 0.014, a value that had been linked to a lower probability of a failure to settle. The tested patience parameter values were PP = 2000, 4000, and 8000.

This group of tests produced the following results:

		ReLU		Sigmoid		Softmax	
Patience parameter	Settling Threshold	AVERAGE	STD	AVERAGE	STD	AVERAGE	STD
2000	Accuracy Training	0,1658	0,0161	0,3542	0,1185	0,2517	0,1028
	Accuracy Testing	0,1504	0,0177	0,3400	0,0732	0,2448	0,0850
5000	Accuracy Training	0,1450	0,0375	0,3667	0,0429	0,2692	0,0063
	Accuracy Testing	0,1334	0,0178	0,3564	0,0339	0,2642	0,0159
8000	Accuracy Training	0,1467	0,0014	0,3933	0,0756	0,2883	0,0290
	Accuracy Testing	0,1378	0,0109	0,3509	0,0720	0,2937	0,0044

	Accuracy Training	Accuracy Testing
Probability Patience Parameter	0,789180749	0,776952754
Probability Module Implementation	**2,98173E-06**	**1,01356E-07**
Probability Interaction	0,924554771	0,818298923

The probabilities from the ANOVA indicate statistically significant differences in the results from different module implementations (in bold). Indeed, it appears that after a long “Improvement” stage, the sigmoid and softmax EMANN-like implementations have higher accuracies than the ReLU one, which retains a very poor accuracy. The best accuracies are found in the sigmoid using implementation (sometimes reaching 39% for the last training batches, and 35% for the testing set).

It could also appear that a greater patience parameter is linked to slightly greater accuracy levels, at least for the sigmoid and softmax using implementations. The ANOVA test nevertheless indicate that these correlations are most probably not statistically significant. In case they were, this could be related to the fact that a longer waiting time allows for more neurons to settle, and thus for the indirect addition of new neurons. Indeed, in previous tests parameters that favorized the addition of new neurons were also correlated with favorizing higher accuracy values.

Fifth Test: Effects of Ascension with Multiple Unit Increase

Several experiments were launched on the sigmoid using module and on the ReLU using module implementations. The same values were always used for the ascension threshold (AT = 200), uselessness threshold (UT = 0.013), and patience parameter (PP = 4000). For the settling threshold, experiments launched with the sigmoid using module implementation always used ST = 0.019, while those launched with the ReLU using implementation sometimes used ST = 0.014 and sometimes ST = 0.019. The “initial unit number” and “unit increase” parameters were set to values other than 1, and the evolution of the accuracy for both the training batches and testing set was observed.

It appears that, when the “initial unit number” and “unit increase” parameters are set to higher values (such as 5 or 10), the accuracy of the sigmoid using implementation increases very quickly, but it becomes very difficult for units to settle. When the number of neurons is made to quickly reach 100 units (or to start close to 100), the levels of accuracy may even reach values of 50% for the recent training batches, and 40% for the testing set.

Similarly, in the ReLU using implementation, the accuracy also rises quickly for higher values of “initial unit number” and “unit increase”. However, the accuracy levels never seem to go much further than 27% for the training batches, and 21% for the testing set. In addition, the trouble with settling is even greater for this implementation, as not even a value of 0.014 for the ST seems to be enough for the module to leave the “Ascension” stage.

In both cases, the more quickly neurons are added, the more time the algorithm takes to execute. This may be due to the large number of neurons, and the fact that they are stored as a Python list of TensorFlow tensors that needs to be constantly put together.

Sixth Test: Effects of Module Topology on a Second Module

On various occasions throughout the other tests, the experiment was made to allow EMANN to build a second module on top of the first one. On most cases, the second module could only produce random results (around 10% accuracy, for a classification task with 10 classes). However, in cases where the first module contained a high number of neurons in its hidden layer, and its accuracy had reached a high enough level, the second module could reach levels of accuracy above 20%.

Further tests need to be made in this direction, and new parameters and features need to be added to the EMANN implementation to allow a second module to reach an equal or higher accuracy than the first one.

Conclusion

While it has been observed that the EMANN algorithm needs to be further tweaked to build modules with optimal performance when confronted to a relatively complex (for MLP) problem such as CIFAR-10, various pertinent conclusions can be taken from the experiments and tests performed within this thesis.

Firstly, EMANN seems to perform much better when the activation functions of its modules behave in a similar way to a sigmoid. In particular, the softmax activation function seems to provide very good results. This can be explained by the fact that CIFAR-10 is a classification task with mutually exclusive classes, precisely the kind of task that softmax was designed for. Although popular, the ReLU activation function seems in no way relevant for an algorithm centered such as EMANN, whose performance relies on increasing connection strength in neurons.

Secondly, in cases where the activation function behaves similarly to a sigmoid, better results are obtained with a high settling threshold and a low ascension threshold. This is correlated with a high number of neurons in a module's hidden layer, as these two parameters control the rate at which new neurons are added to it. With more neurons in a layer, more possibilities for memorizing patterns emerge, and thus the module can reach higher levels of accuracy.

Thirdly, and once again given that the activation function behaves similarly to a sigmoid, it appears for now that any parameter correlated to an increase in the number of neurons will have a very positive effect on the accuracy of the constructed network. This is true both for parameters already in EMANN and for newly introduced parameters ("initial unit number" and "unit increase").

Future Work

As a result of the experiments and tests presented and discussed above, several lines of future research can be suggested. Among these lines are the following:

- Further study the influence of these parameters on the accuracy of further modules.
- Explore EMANN-like module implementations with other activation functions. Preferably those with similar behavior to the sigmoid function (because of the use of CS), such as the tanh activation function.
- Fully exploring the effect of an "initial unit number" and a "unit increase" parameter that are larger than 1, as well as alternate ways to increase the number of neurons added during the "Ascension" stage.
- Performing tests using other benchmark databases for image classification, such as MNIST.
- Trying to implement an algorithm for self-structuring convolutional NN that uses an approach similar to the one EMANN uses. And/or implement EMANN for the fully-connected NN usually added at the output of a convolutional NN.

Bibliography

1. **Marr, Bernard.** What Is The Difference Between Deep Learning, Machine Learning and AI? [Online] Forbes, December 8, 2016. [Cited: May 21, 2018.] https://www.forbes.com/sites/bernardmarr/2016/12/08/what-is-the-difference-between-deep-learning-machine-learning-and-ai/.

2. **Anonymous.** What is Machine Learning? A definition. [Online] Expert System. [Cited: May 21, 2018.] http://www.expertsystem.com/machine-learning-definition/.

3. **Brownlee, Jason.** What is Deep Learning? [Online] Machine Learning Mastery, August 16, 2016. [Cited: May 21, 2018.] https://machinelearningmastery.com/what-is-deep-learning/.

4. **Schmidhuber, Jürgen.** Deep learning in neural networks: An overview. *Neural Networks.* January 2015, Vol. 61, pp. 85-117.

5. **Beyer, Wolfgang.** How to Build a Simple Image Recognition System with TensorFlow. [Online] December 5, 2016. [Cited: February 21, 2018.] https://www.wolfib.com/Image-Recognition-Intro-Part-1/.

6. **Salomé, Tristan and Bersini, Hugues.** An Algorithm for Self-Structuring Neural Net Classifiers. IRIDIA, Université Libre de Bruxelles. 1994.

7. **Anonymous.** About Tensorflow. [Online] TensorFlow. [Cited: November 2, 2017.] https://www.tensorflow.org/.

8. **Krizhevsky, Alex.** Learning multiple layers of features from tiny images. *Master Thesis. Department of Computer Science. University of Toronto.* 2009.

9. **Nilsson, Nils J.** The quest for artificial intelligence. Cambridge University Press, 2009.

10. **Araujo dos Santos, Leonardo.** Artificial Intelligence. [Online] 2018. https://legacy.gitbook.com/book/leonardoaraujosantos/artificial-inteligence/details.

11. **Bishop, Christopher M.** Pattern Recognition and Machine Learning. Springer, 2006.

12. **PWC.** Sizing the prize – What's the real value of AI for your business and how can you capitalise?. [Online] 2017. https://www.pwc.com/gx/en/issues/analytics/assets/pwc-ai-analysis-sizing-the-prize-report.pdf .

13. **Accenture.** Why Artificial Intelligence is the Future of Growth. [Online] 2016. https://www.accenture.com/t20170927T080049Z__w__/us-en/_acnmedia/PDF-33/Accenture-Why-AI-is-the-Future-of-Growth.PDFla=en.

14. **EPSC.** The Age of Artificial Intelligence. Towards a European Strategy for Human-Centric Machines 2018. Strategic notes. Issue 29. [Online] 2018. https://ec.europa.eu/epsc/sites/epsc/files/epsc_strategicnote_ai.pdf.

15. **Yannakakis, Georgios N. and Togelius, Julian.** Artificial intelligence and games. Springer, 2018.

16. **Ng, Andrew.** Why AI is the new electricity. [Online] 2016. https://singjupost.com/andrew-ng-artificial-intelligence-is-the-new-electricity-at-stanford-gsb-transcript/?print=pdf.

17. **Wan, Hao.** A new scheme for training ReLU based multi-layer feedforward neural networks. *Master Thesis. KTK Royal Institute of Technology. School of Computer Science and Communication.* 2017.

18. **Campbell, Collin and Ying, Yiming.** Learning with support vector machines. Morgan& Claypool Publishers., 2011, Vol. 5, pp. 1-95.

19. **Cha Zhang, Cha; Ma, Yunqian.** Ensemble Machine Learning: Methods and Applications. Springer Publishing Company, Incorporated., 2012.

20. **Rojas, Raul.** Neural Networks: A Systematic Introduction. *Springer.* New York, NY, USA : Springer, 1996.

21. **Russell, Stuart J.; Norvig, Peter.** Artificial Intelligence: A Modern Approach. Third Edition, Prentice Hall, 2010.

22. **Mwangi, Benson, Soares, Jair C. and Hasan, Khader M.** Visualization and unsupervised predictive clustering of high-dimensional multimodal neuroimaging data. *J. Neurosci. Methods*, 2014, Vol. 30, pp. 19-25.

23. **Bengio, Yoshua.** Deep Learning of Representations for Unsupervised and Transfer Learning. *JMLR: Workshop and Conference Proceedings. Workshop on Unsupervised and Transfer Learning*, 2012, pp. 17-37.

24. **Zhu, Xiaojin.** Semi-Supervised Learning. Computer Sciences, University of Wisconsin-Madison, 2008.

25. **Kaelbling, Leslie Pack; Littman, Michael L.; Moore, Andrew M.** Reinforcement learning: a survey. J*ournal of Artificial Intelligence Research*, 1996, pp. 237-285.

26. **Marblestone, Adam H., Wayne, Greg and Kording, Konrad.** Towards an integration of deep learning and neuroscience. [Online] 2016. https://www.biorxiv.org/content/biorxiv/early/2016/06/13/058545.full.pdf.

27. **Stenroos, Olavi.** Object detection from images using convolutional reural networks. *Master Thesis. Aalto University. School of Science. Master's programme in computer, communication and information sciences.* 2017.

28. **Steinkraus, Dave, Buck, Ian and Simard, Patrice Y.** Using GPUs for machine learning algorithms. *Document Analysis and Recognition. Proceedings. Eighth International Conference, IEEE,* 2005, pp. 1115-1120.

29. **LeCun, Yann.** LeNet-5, convolutional neural networks. [Online] 2013. http://yann.lecun.com/exdb/lenet/.

30. **Hintom, Goeffrey E.** Deep belief networks., 4(5): 5947. [Online] 2009. http://scholarpedia.org/article/Deep_belief_networks.

31. **Liou, Cheng-Yuan, et al.** Autoencoder for words. *Neurocomputing.* Elsevier, 2014, Vol. 139, 2, pp. 84-96.

32. **Schmidhuber, Jürgen.** Deep learning in neural networks: An overview. *Neural networks.* Elsevier, 1993, Vol. 61, pp. 85-117.

33. **Goodfellow, Ian J.; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua.** Generative Adversarial Nets. [Online] 2014. http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf.

34. **Sharma, Sagar.** Activation Functions: Neural Networks - Sigmoid, tanh, Softmax, ReLU, Leaky ReLU explained!!!. [Online] 2017. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

35. **Krizhevsky, Alex, Sutskever, Ilya and Hinton, Geoffrey E.** ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems.* 2012, pp. 1097-1105.

36. **Ramachandran, Prajit, Zoph, Barret and Le, Quoc V.** Searching for activation functions. [Online] 2017. https://arxiv.org/pdf/1710.05941.pdf.

37. **Polamuri, Saimadhu.** Difference between softmax function and sigmoid fonction. [Online] 2017. http://dataaspirant.com/2017/03/07/difference-between-softmax-function-and-sigmoid-function/.

38. **McCulloch, Warren S. and Pitts, Walter.** A Logical Calculus for Ideas Imminent in Nervous Activity. *Bulletin of Mathematical Biophysics.* 1943, Vol. 5, 4, pp. 115-133.

39. **Rosenblatt, Frank.** The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review.* 1958, Vol. 65, 6, pp. 386-408.

40. **Werbos, Paul John.** Beyond regression: new tools for prediction and analysis in the behavioural sciences. *PhD thesis, Harvard University.* 1974.

41. **Aloni, Dan.** Back propagation with TensorFlow (Updated for TensorFlow 1.0). *Dan Aloni's blog.* [Online] March 6, 2017. [Cited: May 30, 2018.] http://blog.aloni.org/posts/backprop-with-tensorflow/.

42. **Fortuner, Brendan.** Loss Functions. *ML Cheatsheet.* [Online] April 20, 2017. [Cited: May 30, 2018.] http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html.

43. **Waugh, Sam.** Dynamic Learning Algorithms. Department of Computer Science. University of Tasmania, 1994.

44. **Basheer, Imad; Hajmeer, M.** Artificial neural networks: fundamentals, computing, design, and application. *J. Microbiol. Methods.* Elsevier, 2000, Vol. 43, pp. 3-31.

45. **Kwok, Tin-Yau and Yeung, Dit-Yan.** Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE Transactions on Neural Networks.* 1997, Vol. 8, 3, pp. 630-645.

46. **Duffner, Stefan and Garcia, Christophe.** An online backpropagation algorithm with validation error-based adaptive learning rate. *Artificial Neural Networks – ICANN 2007, Porto, Portugal.* 2007.

47. **Thabtah, Fadi; Mohammad, Rami M.; McCluskey, Lee.** A Dynamic self-structuring neural network model to combat phishing. *2016 International Joint Conference on Neural Networks (IJCNN).* 2016, pp. 4221-4225.

48. **Mohammad, Rami M., Thabtah, Fadi and McCluskey, Lee.** Predicting phishing websites based on self-structuring neural network. *Neural Computing and Applications.* 2013, Vol. 25, 2, pp. 443-458.

49. **Islam, Monirul; Sattar, Abdus; Amin, Faijul; Yao, Xin; Murase, Kazuyuki.** A New Adaptive Merging and Growing Algorithm for Designing Artificial Neural Networks,. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics.* 2009, Vol. 39, 3, pp. 705-722.

50. **Mohammad, Rami M., Thabtah, Fadi and McCluskey, Lee.** Predicting Phishing Websites using Neural Network trained with Back-Propagation. *ICAI.* Las Vegas, 2013.

51. **Thabtah, Fadi, Mohammad, Rami M. and McCluskey, Lee.** An Improved Sel-Structuring Neural Network. *Trends and Applications in Knowledge Discovery and Data Mining: PAKDD 2016 Workshops, BDM, MLSDA, PACC, WDMBF. Revised Selected Papers.* Auckland, New Zealand, 2016, pp. 35-47.

52. **Ash, Timur.** Dynamic node creation in backpropagation networks. *Connection Science.* 1989, Vol. 1, 4, pp. 365-375.

53. **Fahlman, Scott. E. and Lebiere, Christian.** The Cascade-Correlation Learning Architecture. *Advances in neural information processing systems.* 1990, pp. 524-532.

54. **Ma, L. and Khorasani, K.** A new strategy for adaptively constructing multilayer feedforward neural networks. *Neurocomputing.* Elsevier, 2003, Vol. 31, pp. 361-385.

55. **Yang, Shih-Hung and Chen, Yon-Ping.** An evolutionary constructive and pruning algorithm for artificial neural networks and its prediction applications. 86 (2012) 140–149. *Neurocomputing.* Elsevier, 2012, Vol. 86, pp. 140-149.

Annex A: Source Code for EMANN Executable

```python
'''''
Trains and evaluates an EMANN classifier for CIFAR-10.

Modification of Wolfgang Beyer's code form his tutorial:
"Simple Image Classification Models for the CIFAR-10 dataset using TensorFlow".
'''

# Needed for compatibility between Python 2 and 3.
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Import relevant modules (numpy, TensorFlow, time modules, OS file paths).
import numpy as np
import tensorflow as tf
import time
from datetime import datetime
import os
# data_helpers.py contains functions for loading and preparing the dataset.
import data_helpers
# two_layer_fc.py contains the Module class, which defines a 2-layer NN.
from EMANN_softmax import Module, connection_strength, get_activation_funct

# Silence TensorFlow warning logs on build.
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

print('--------------------------------------------------------------')
print('----- CIFAR-10 TensorFlow EMANN classifier training demo -----')
print('--------------------------------------------------------------\n')


# -----------------------------------------------------------------------------
# ----------------------------CONSTANT DEFINITIONS-----------------------------
# -----------------------------------------------------------------------------


# Define as constants: number of color channels per image, number of classes.
IMAGE_PIXELS = 3072
CLASSES = 10

# Define some parameters for the model, as external TensorFlow flags.
flags = tf.flags
FLAGS = flags.FLAGS
# Parameters related to training epochs.
flags.DEFINE_integer('batch_size', 400, 'Batch size, divisor of dataset size.')
flags.DEFINE_float('reg_constant', 0.1, 'Regularization constant for weights.')
flags.DEFINE_float('learning_rate', 1e-4,
                   'Learning rate for the training epoch (backpropagation).')
# Parameters used in EMANN algorithms (with alt values).
flags.DEFINE_integer('uselessness_tresh', 0.15,
                     'Minimum connection strength for a unit to be useless.')
flags.DEFINE_integer('settling_tresh', 0.5,
                     'Maximum connection strength for a unit to be settled.')
flags.DEFINE_integer('ascension_tresh', 400,
                     'During ascension, number of epochs for adding a unit.')
flags.DEFINE_integer('uselessness_tresh_alt', 0.013,
                     'Different uselessness treshold for the first module.')
flags.DEFINE_integer('settling_tresh_alt', 0.019,
                     'Different settling treshold for the first module.')
flags.DEFINE_integer('ascension_tresh_alt', 200,
                     'Different ascension treshold for the first module.')
# Parameters used in EMANN algorithms (without alt values).
flags.DEFINE_integer('initial_unit_num', 1,
                     'Initial number of units in a module.')
flags.DEFINE_integer('unit_increase', 1,
                         'Number of units added each ascension_thresh.')
flags.DEFINE_integer('ascension_limit', 120,
                         'Maximum number of neurons before stopping ascension.')
flags.DEFINE_integer('patience_param', 8000,
                         'During improvement, number of epochs before stopping.')
flags.DEFINE_integer('improvement_tresh', 5e-3,
                         'Minimum MSE difference for adding a new module.')
# Parameters related to the algorithm's implementation.
flags.DEFINE_integer('check_tresh', 100,
                         'Number of epochs to skip before the network is checked.')
flags.DEFINE_string('train_dir', 'tf_logs',
                        'Directory for the program to place the training logs.')

FLAGS._parse_flags()
print('\n-------SELECTED VALUES FOR NN PARAMETERS-------')
for attr, value in sorted(FLAGS.__flags.items()):
    print('{} = {}'.format(attr, value))
print()


# -----------------------------------------------------------------------------
# ----------------------------FUNCTION DEFINITIONS-----------------------------
# -----------------------------------------------------------------------------


def get_next_feed_dict(batches):
    '''''
    Get a random input data batch for the next training epoch.

    Args:
        batches: A generator that yields randomly generated batches.

    Returns:
        feed_dict: A dictionary that defines the next data batch for training.
           '''
           batch = next(batches)
           images_batch, labels_batch = zip(*batch)
           feed_dict = {
               images_placeholder: images_batch,
               labels_placeholder: labels_batch
           }
           return feed_dict


       def standard_module_check(sess, new_module, accuracy, feed_dict,
                                     test_feed_dict, num_epochs, logs):
           '''''
           Standard check operation performed on the module every check_tresh epochs.
           Calculates the accuracy, the maximal and minimal CS, and adds the epoch
           data to the custom training log file.

           Args:
               sess: Ongoing TensorFlow session.
               accuracy: Operation for the module's accuracy calculation.
               feed_dict: Random input data batch for the current epoch.
               test_feed_dict: Input data batch from the testing set.
               num_epochs: Number of training epochs already elapsed.
               logs: A writer for custom training log files.

           Returns:
               train_accuracy: The current accuracy of the module on the training set.
           '''

           # Calculate the accuracy on the training set.
           train_accuracy = sess.run(accuracy, feed_dict=feed_dict)
           test_accuracy = sess.run(accuracy, feed_dict=test_feed_dict)
           # Print the calculated training set accuracy.
           print('Epoch {:d}: training set accuracy {:g}'.format(
               num_epochs, train_accuracy))
        logs.write('{:d}\t{:d}\t{:d}\t{:g}\t{:g}\t{:g}\t{:g}\n'.format(
            num_epochs, new_module.a_1_units, new_module.settled,
            train_accuracy, test_accuracy, new_module.minCS, new_module.maxCS))

        return train_accuracy


    def EMANN_ascension(sess, new_module, labels_placeholder, batches,
                        test_feed_dict, units_settled, num_epochs, logs):
        '''
        Run the ascension stage of EMANN on a module.

        Returns:
            None.
        '''
        # From the module, get definitions for the following operations:
        # loss function, training epoch, and accuracy calculation.
        loss = new_module.loss(labels_placeholder)
        train_epoch = new_module.training(labels_placeholder, FLAGS.learning_rate)
        accuracy = new_module.evaluation(labels_placeholder)

        # Get the proper tresholds for the module.
        settling_tresh = FLAGS.settling_tresh
        ascension_tresh = FLAGS.ascension_tresh
        if new_module.module_ID == 0:
            settling_tresh = FLAGS.settling_tresh_alt
            ascension_tresh = FLAGS.ascension_tresh_alt

        # Perform training epochs until one unit settles
        # or the hidden layer has reached ascension_limit units.
        # Add a new unit every ascension_tresh epochs.
        while units_settled == 0:  # and new_module.a_1_units < FLAGS.ascension_limit:
            # Get the (random) input data batch corresponding to epoch i.
            feed_dict = get_next_feed_dict(batches)
            # Perform a new training epoch.
            sess.run([train_epoch, loss], feed_dict=feed_dict)

            # Every ascension_tresh epochs, add unit_increase units to the hidden layer.
            if num_epochs % ascension_tresh == 0:
                # Add the new unit and update the graph.
                new_module.add_new_unit(sess, FLAGS.unit_increase,
                                        reg_constant=FLAGS.reg_constant)
                # Update the definitions of loss, training epoch, and accuracy.
                loss = new_module.loss(labels_placeholder)
                train_epoch = new_module.training(labels_placeholder,
                                                  FLAGS.learning_rate)
                accuracy = new_module.evaluation(labels_placeholder)

            # Every check_tresh epochs, check the state of the module.
            if num_epochs % FLAGS.check_tresh == 0:
                # Check if a unit has settled.
                units_settled = new_module.units_settled(settling_tresh)

                # Perform the standard module check.
                train_accuracy = standard_module_check(sess, new_module,
                                                       accuracy, feed_dict,
                                                       test_feed_dict,
                                                       num_epochs, logs)

            # Increase the epoch counter.
            num_epochs += 1

        return units_settled, num_epochs


    def EMANN_improvement(sess, new_module, labels_placeholder, batches,
                          test_feed_dict, units_settled, num_epochs, logs):
        '''
        Run the improvement stage of EMANN on a module.
```

```python

        Returns:
            None.
        '''
        # From the module, get definitions for the following operations:
        # loss function, training epoch, and accuracy calculation.
        loss = new_module.loss(labels_placeholder)
        train_epoch = new_module.training(labels_placeholder, FLAGS.learning_rate)
        accuracy = new_module.evaluation(labels_placeholder)

        # Get the proper settling treshold for the module.
        settling_tresh = FLAGS.settling_tresh
        if new_module.module_ID == 0:
            settling_tresh = FLAGS.settling_tresh_alt

        # Perform training epochs until no units have settled in awhile.
        # Add a new unit every time a unit settles.
        patience_countdown = FLAGS.patience_param
        while patience_countdown >= 0:
            # Get the (random) input data batch corresponding to epoch i.
            feed_dict = get_next_feed_dict(batches)
            # Perform a new training epoch.
            sess.run([train_epoch, loss], feed_dict=feed_dict)

            # Every check_tresh epochs, check the state of the module.
            if num_epochs % FLAGS.check_tresh == 0:
                # Check if new units have settled, if so add a new unit.
                new_units_settled = new_module.units_settled(settling_tresh)
                if new_units_settled > units_settled:
                    # Update the number of settled units.
                    units_settled = new_units_settled
                    # Reset the patience countdown.
                    patience_countdown = FLAGS.patience_param + 1

                    # Add the new unit and update the graph.
                    new_module.add_new_unit(sess, 1,
                                            reg_constant=FLAGS.reg_constant)
                    # Update the definitions of loss, training epoch, and accuracy.
                    loss = new_module.loss(labels_placeholder)
                    train_epoch = new_module.training(labels_placeholder,
                                                      FLAGS.learning_rate)
                    accuracy = new_module.evaluation(labels_placeholder)

                # Perform the standard module check.
                train_accuracy = standard_module_check(sess, new_module,
                                                       accuracy, feed_dict,
                                                       test_feed_dict,
                                                       num_epochs, logs)

            # Increase the epoch counter and decrease the patience countdown.
            num_epochs += 1
            patience_countdown -= 1

        return units_settled, num_epochs


    def EMANN_module_training(sess, module_ID, inputs_placeholder, input_num,
                              labels_placeholder, batches, test_feed_dict, logs):
        '''''
        Run the EMANN algorithm to train a module (network with one hidden layer).

        Args:
            sess: Ongoing TensorFlow session.
            module_ID: Identifier for the module to be built.
            inputs_placeholder: Represents the inputs for this module.
            input_num: Number of input channels for this module.
            labels_placeholder: Represents the expected outputs for this module.
            batches: A generator of random data batches used for training.
            test_feed_dict: A spare data batch used for testing.
```

```
                logs: A writer for custom training log files.

            Returns:
                new_module: A new module to add to the EMANN classifier.
            '''
            print('Setting up the new module...', end='')
            # Create the module, with initial_unit_num units in its hidden layer.
            new_module = Module(module_ID, inputs_placeholder, input_num,
                                FLAGS.initial_unit_num, CLASSES,
                                reg_constant=FLAGS.reg_constant)
            # From the module, get definitions for the following operations:
            # loss function, training epoch, and accuracy calculation.
            loss = new_module.loss(labels_placeholder)
            train_epoch = new_module.training(labels_placeholder, FLAGS.learning_rate)
            accuracy = new_module.evaluation(labels_placeholder)
            print(' DONE!')

            # Get the proper settling threshold (useful for the standard check).
            settling_tresh = FLAGS.settling_tresh
            if new_module.module_ID == 0:
                settling_tresh = FLAGS.settling_tresh_alt

            # Initialize global variables.
            sess.run(tf.global_variables_initializer())
            # Current number of settled units is 0.
            settled_units = 0
            # Current number of performed epochs is 0.
            num_epochs = 0

            # Get the (random) input data batch corresponding to epoch i.
            feed_dict = get_next_feed_dict(batches)
            # Perform the first training epoch and increase the epoch counter.
            sess.run([train_epoch, loss], feed_dict=feed_dict)
            units_settled = new_module.units_settled(settling_tresh)
            num_epochs += 1

            # ------------------------ ASCENSION STAGE ------------------------
            print('\n--------------1. ASCENSION STAGE---------------\n')
            logs.write('--------------1. ASCENSION STAGE---------------\n')
            logs.write('Epoch\tUnits\tSettl\tAccTr\tAccTes\tCS Min\tCS Max\n')
            units_settled, num_epochs = EMANN_ascension(sess, new_module,
                                                        labels_placeholder, batches,
                                                        test_feed_dict,
                                                        units_settled, num_epochs,
                                                        logs)

            # ----------------------- IMPROVEMENT STAGE -----------------------
            print('\n-------------2. IMPROVEMENT STAGE--------------\n')
            logs.write('-------------2. IMPROVEMENT STAGE--------------\n')
            logs.write('Epoch\tUnits\tSettl\tAccTr\tAccTes\tCS Min\tCS Max\n')
            units_settled, num_epochs = EMANN_improvement(sess, new_module,
                                                          labels_placeholder, batches,
                                                          test_feed_dict,
                                                          units_settled, num_epochs,
                                                          logs)

            # ------------------------- PRUNING STAGE -------------------------
            print('\n---------------3. PRUNING STAGE----------------\n')
            logs.write('---------------3. PRUNING STAGE----------------\n')
            logs.write('Epoch\tUnits\tSettl\tAccTr\tAccTes\tCS Min\tCS Max\n')
            # Save the current accuracy.
            pre_pruning_accuracy = sess.run(accuracy, feed_dict=feed_dict)
            test_accuracy = sess.run(accuracy, feed_dict=test_feed_dict)
            units_settled = new_module.units_settled(settling_tresh)
            print('Epoch {:d}: training set accuracy BEFORE PRUNING {:g}'.format(
                    num_epochs, pre_pruning_accuracy))
            logs.write('{:d}\t{:d}\t{:d}\t{:g}\t{:g}\t{:g}\t{:g}\n'.format(
                num_epochs, new_module.a_1_units, new_module.settled,
                pre_pruning_accuracy, test_accuracy, new_module.minCS, new_module.maxCS))
```

```

    # Prune all useless units.
    if new_module.module_ID == 0:
        new_module.units_prune(FLAGS.uselessness_tresh_alt)
    else:
        new_module.units_prune(FLAGS.uselessness_tresh)
    # Update the definitions of loss, training epoch, and accuracy.
    loss = new_module.loss(labels_placeholder)
    train_epoch = new_module.training(labels_placeholder, FLAGS.learning_rate)
    accuracy = new_module.evaluation(labels_placeholder)

    # Save the accuracy after pruning.
    train_accuracy = sess.run(accuracy, feed_dict=feed_dict)
    test_accuracy = sess.run(accuracy, feed_dict=test_feed_dict)
    units_settled = new_module.units_settled(settling_tresh)
    print('Epoch {:d}: training set accuracy AFTER PRUNING {:g}'.format(
            num_epochs, train_accuracy))
    logs.write('{:d}\t{:d}\t{:d}\t{:g}\t{:g}\t{:g}\t{:g}\n'.format(
        num_epochs, new_module.a_1_units, new_module.settled,
        train_accuracy, test_accuracy, new_module.minCS, new_module.maxCS))

    # ------------------------ RECOVERY STAGE -------------------------
    print('\n--------------4. RECOVERY STAGE----------------\n')
    logs.write('--------------4. RECOVERY STAGE----------------\n')
    logs.write('Epoch\tUnits\tSettl\tAccTr\tAccTes\tCS Min\tCS Max\n')
    # Perform training epochs until the accuracy is the same as before pruning.
    while train_accuracy < pre_pruning_accuracy:
        # Get the (random) input data batch corresponding to epoch i.
        batch = next(batches)
        images_batch, labels_batch = zip(*batch)
        feed_dict = get_next_feed_dict(batches)
        # Perform a new training epoch.
        sess.run([train_epoch, loss], feed_dict=feed_dict)

        # Every check_tresh epochs, check the state of the module.
        if num_epochs % FLAGS.check_tresh == 0:
            # Perform the standard module check.
            units_settled = new_module.units_settled(settling_tresh)
            train_accuracy = standard_module_check(sess, new_module,
                                                   accuracy, feed_dict,
                                                   test_feed_dict,
                                                   num_epochs, logs)

        # Increase the epoch counter.
        num_epochs += 1

    # Perform one last standard module check.
    units_settled = new_module.units_settled(settling_tresh)
    train_accuracy = standard_module_check(sess, new_module, accuracy,
                                           feed_dict, test_feed_dict,
                                           num_epochs, logs)

    test_accuracy = sess.run(accuracy, feed_dict=test_feed_dict)
    print('-------------------------------------------------------------')
    print('\aFINAL TESTING SET ACCURACY {:g}'.format(test_accuracy))
    print('-------------------------------------------------------------')

    return new_module


# -----------------------------------------------------------------------------
# -----------------------------MAIN PROGRAM CODE-----------------------------
# -----------------------------------------------------------------------------


# Start measuring the runtime.
beginTime = time.time()

print('Loading data sets...', end='')
```

```python

    # Load the CIFAR-10 data (see data_helpers.py).
    # 50000 training set images, 10000 testing set images.
    # Data sets are: images_train, labels_train, images_test, labels_test, classes.
    data_sets = data_helpers.load_data()

    print(' DONE!')

    print('Creating placeholders for datasets and modules...', end='')

    # Define input placeholders. An input contains:
    # 1) a batch of "N" images, where each is a batch of 3072 floats (1024 pixels),
    # 2) a batch of "N" labels, where each is an int (from 0 to 9).
    images_placeholder = tf.placeholder(tf.float32, shape=[None, IMAGE_PIXELS],
                                        name='images')
    labels_placeholder = tf.placeholder(tf.int64, shape=[None],
                                        name='image-labels')
    # Initialize the list of modules.
    modules_list = []
    # Initialize the list of inputs for modules.
    input_list = [images_placeholder]
    input_num = IMAGE_PIXELS

    print(' DONE!')

    print('\n---------EMANN TRAINING SESSION BEGINS---------\n')

    with tf.Session() as sess:
        print('Generating {:d} random batches...'.format(FLAGS.batch_size), end='')
        # Build a generator of random input data batches, of size batch_size.
        zipped_data = zip(data_sets['images_train'], data_sets['labels_train'])
        batches = data_helpers.generate_random_batches(
            list(zipped_data), FLAGS.batch_size)
        print(' DONE!')
        # Retrieve the test data batch to be used after training each module.
        test_feed_dict = {
                images_placeholder: data_sets['images_test'],
                labels_placeholder: data_sets['labels_test']
        }

        # Put custom training logs in a .txt file.
        date_and_time = datetime.now().strftime('%Y%m%d-%H%M%S')
        logfile = FLAGS.train_dir + '/' + get_activation_funct()
        logfile = logfile + '__' + date_and_time + '.txt'
        # Create a file writer for custom log files.
        log_writer = open(logfile, 'w')
        log_writer.write('Parameters (1st layer):\n')
        log_writer.write('ascension_tresh = {}\n'.format(
            FLAGS.ascension_tresh_alt))
        log_writer.write('settling_tresh = {}\n'.format(FLAGS.settling_tresh_alt))
        log_writer.write('uselessness_tresh = {}\n'.format(
            FLAGS.uselessness_tresh_alt))
        log_writer.write('\nParameters (other layers):\n')
        log_writer.write('ascension_tresh = {}\n'.format(FLAGS.ascension_tresh))
        log_writer.write('settling_tresh = {}\n'.format(FLAGS.settling_tresh))
        log_writer.write('uselessness_tresh = {}\n'.format(
            FLAGS.uselessness_tresh))
        log_writer.write('\nParameters (all layers):\n')
        log_writer.write('patience_param = {}\n'.format(
            FLAGS.patience_param))

        # Itteratively build a certain number of modules (layers).
        for i in range(1):
            # Build a new module and add it to the modules list.
            print('Building module number: ', len(modules_list))
            log_writer.write('\nMODULE NUMBER {}\n'.format(len(modules_list)))
            new_module = EMANN_module_training(sess, len(modules_list),
                                               input_list[-1], input_num,
                                               labels_placeholder,
                                                         batches, test_feed_dict,
                                                         log_writer)
                modules_list.append(new_module)
                input_list.append(new_module.a_1)
                input_num = new_module.a_1_units

            # After all EMANN modules has finished training, the training stops.
            print('\n---------END OF EMANN TRAINING SESSION---------\n')
            # Close the custom training log file.
            log_writer.close()

        endTime = time.time()
        print('Total time elapsed: {:5.2f}s'.format(endTime - beginTime))
```

Annex B: Source Code for Original (Sigmoid) Module

```python
'''''
Class representing an EMANN module, a 2-layer fully-connected neural network
with the possibility of adding and pruning units from its hidden layer.

The algorithm used on these modules is an accurate recreation of the original
EMANN algorithm, as described by its creators in the original paper:
SALOME Tristan, BERSINI Hugues,
An Algorithm for Self-Structuring Neural Net Classifiers,
IRIDIA - Universite Libre de Bruxelles, June 28 - July 2 1994.

Modification of Wolfgang Beyer's code form his tutorial:
"Simple Image Classification Models for the CIFAR-10 dataset using TensorFlow".
'''

# Needed for compatibility between Python 2 and 3.
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# Import relevant modules (numpy, TensorFlow).
import numpy as np
import tensorflow as tf


# ----------------------------------------------------------------------
# ---------------------------USEFUL FUNCTIONS---------------------------
# ----------------------------------------------------------------------


def get_activation_funct():
    '''''
    Returns the name of the activation function for the hidden layer,
    in this case sigmoid.
    '''
    return "sigmoid"


def connection_strength(weights):
    '''''
    The connection strength operation, to be applied on a tensor (vector)
    containing the weights of a unit.

    The CS of a unit is the average of (the absolute values of) its weights.
    '''
    CS = tf.reduce_mean(tf.abs(weights))
    return CS.eval()


def sigmoid_prime(x):
    '''''
    The derivative of the sigmoid operation, defined using TensorFlow's
    sigmoid operation.

    sigmoid_prime(x) = sigmoid(x) * (1 - sigmoid(x))
    '''
    return tf.multiply(tf.nn.sigmoid(x),
                       tf.subtract(tf.constant(1.0), tf.nn.sigmoid(x)))

# ----------------------------------------------------------------------
# --------------------------MODULE OBJECT CLASS-------------------------
# ----------------------------------------------------------------------


class Module:
    ''''''Defines a module, a 2-layer fully-connected neural network'''

    def __init__(self, module_ID, input_channel, input_units, hidden_units,
                 classes, reg_constant=0):
        '''
        Initialization method.
        Sets the initial weights and biases for each layer in the network.
        Defines the forward pass through the module.

        Args:
            module_ID: Identifier for the module.
            input_channel: Input data placeholder.
            input_units: Number of input entry units (i.e. color channels).
            hidden_units: Number of initial hidden units.
            classes: Number of image classes (number of possible labels).
            reg_constant: Regularization constant (default 0).

        Returns:
            None.
        '''
        # The module's ID is used when defining TensorFlow variables.
        self.module_ID = module_ID

        # Define the initial number of units for each layer.
        self.a_0_units = input_units
        self.a_1_units = hidden_units
        self.a_2_units = classes

        # Variables for the number of settled units, and the min and max CS.
        self.settled = 0
        self.minCS = 0
        self.maxCS = 0

        # ------------------LAYER #1------------------
        # Define the layer's weights as a list of vectors.
                # N.B.: The matrix used for backpropagation is found by stacking
                #       these vectors along axis=1 (transpose of just staking them).
                self.w_1 = []
                for unit in range(self.a_1_units):
                    self.w_1.append(tf.get_variable(
                        name='m_'+str(self.module_ID)+'__w_1_'+str(unit+1),
                        shape=[self.a_0_units],
                        # Weights are initialized to normally distributed variables,
                        # the standard deviation being 1/sqrt(a_0_units).
                        initializer=tf.truncated_normal_initializer(
                            stddev=1.0 / np.sqrt(float(self.a_0_units))),
                        # L2-regularization adds the sum of the squares of all the
                        # weights in the network to the loss function. The importance
                        # of this effect is controlled by reg_constant.
                        regularizer=tf.contrib.layers.l2_regularizer(reg_constant)
                    ))

                # ------------------LAYER #2------------------
                # Define the layer's weights as a list of vectors.
                # N.B.: The matrix used for backpropagation is found by stacking
                #       these vectors along axis=0 (transpose of just staking them).
                self.w_2 = []
                for unit in range(self.a_1_units):
                    self.w_2.append(tf.get_variable(
                        name='m_'+str(self.module_ID)+'__w_2_'+str(unit+1),
                        shape=[self.a_2_units],
                        initializer=tf.truncated_normal_initializer(
                            stddev=1.0 / np.sqrt(float(self.a_1_units))),
                        regularizer=tf.contrib.layers.l2_regularizer(reg_constant)
                    ))

                # -----------DEFINE THE FORWARD PASS----------
                # Keep a reference to the input channels.
                self.a_0 = input_channel
                # Define the hidden units' output (w.r.t. input units and weights).
```

```python
        # N.B.: The activation function is sigmoid (value between 0 and 1).
        self.z_1 = tf.matmul(self.a_0, tf.stack(self.w_1, axis=1))
        self.a_1 = tf.nn.sigmoid(self.z_1)
        # Define the output units' output (w.r.t. hidden units and weights).
        # N.B.: The activation function is also the sigmoid.
        self.z_2 = tf.matmul(self.a_1, tf.stack(self.w_2, axis=0))
        self.a_2 = tf.nn.sigmoid(self.z_2)

    def add_new_unit(self, sess, unit_count, reg_constant=0):
        '''
        Adds new units to the module's hidden layer (new weight vectors).
        Then redefines the forward pass to take the new unit into account.

        Args:
            sess: Ongoing TensorFlow session.
            unit_count: Number of new units to be added.
            reg_constant: Regularization constant (default 0).

        Returns:
            None.
        '''
        # For each unit to be added.
        for unit in range(unit_count):
            # Increase the number of units in the hidden layer.
            self.w_1.append(tf.get_variable(
                name='m_'+str(self.module_ID)+'__w_1_'+str(self.a_1_units+1),
                shape=[self.a_0_units],
                initializer=tf.truncated_normal_initializer(
                    stddev=1.0 / np.sqrt(float(self.a_0_units))),
                regularizer=tf.contrib.layers.l2_regularizer(reg_constant)
            ))
            self.w_2.append(tf.get_variable(
                name='m_'+str(self.module_ID)+'__w_2_'+str(self.a_1_units+1),
                shape=[self.a_2_units],
                initializer=tf.truncated_normal_initializer(
                    stddev=1.0 / np.sqrt(float(self.a_1_units))),
                regularizer=tf.contrib.layers.l2_regularizer(reg_constant)
            ))
            # Initializes these new weight vectors.
            sess.run(self.w_1[-1].initializer)
            sess.run(self.w_2[-1].initializer)
            # Update the unit count.
            self.a_1_units += 1

        # Redefine the forward pass accordingly.
        self.z_1 = tf.matmul(self.a_0, tf.stack(self.w_1, axis=1))
        self.a_1 = tf.nn.sigmoid(self.z_1)
        self.z_2 = tf.matmul(self.a_1, tf.stack(self.w_2, axis=0))
        self.a_2 = tf.nn.sigmoid(self.z_2)

        print("Added an unit! Number of units:", self.a_1_units)

    def units_settled(self, settling_tresh, extra_cond=False):
        '''
        Checks how many units in the module's hidden layer have settled
        (their connection strength is above a settling treshold).

        An additional condition for settling can be added, where the unit
        must have at least one of its weights above the settling treshold.
        This condition is used to avoid "copy" problems in succesive modules.

        Args:
            settling_tresh: The settling treshold for CS or weights.
            extra_cond: Adds an extra condition for weights (default False).

        Returns:
            settled: The number of units that have settled.
        '''
        self.maxCS = connection_strength(self.w_1[0])
```

```python
            self.minCS = connection_strength(self.w_1[0])
            self.settled = 0

            # Check for every unit if the CS is above the threshold.
            for i in range(self.a_1_units):
                CS = connection_strength(self.w_1[i])
                self.maxCS = max(CS, self.maxCS)
                self.minCS = min(CS, self.minCS)
                if(CS > settling_tresh):
                    if(extra_cond):
                        # Check for every weight if it is above the threshold.
                        current_weights = self.w_1[i].eval()
                        for j in range(len(current_weights)):
                            if(current_weights[j] > settling_tresh):
                                self.settled += 1
                    else:
                        self.settled += 1

            print("CS from", self.minCS, "to", self.maxCS,
                  "Settling tresh:", settling_tresh)
            print("Settled units:", self.settled, "\n")
            return self.settled

        def units_prune(self, uselessness_tresh):
            '''''
            Prunes all the useless units in the hidden layer (their connection
            strength is below an uselessness treshold).

            Args:
                uselessness_tresh: The uselessness treshold for CS.

            Returns:
                None.
            '''
            toKeep = []
            # Check for every unit if the CS is above the threshold.
            for i in range(self.a_1_units):
                CS = connection_strength(self.w_1[i])
                if(CS > uselessness_tresh):
                    toKeep.append(i)  # If so, keep the unit.

            print("Number of units before pruning:", self.a_1_units)
            print("Units to prune:", self.a_1_units - len(toKeep))

            # Update the number of units.
            self.a_1_units = len(toKeep)
            # Recreate the weight vector lists without the pruned units.
            self.w_1 = [self.w_1[i] for i in toKeep]
            self.w_2 = [self.w_2[i] for i in toKeep]

            # Redefine the forward pass after pruning.
            self.z_1 = tf.matmul(self.a_0, tf.stack(self.w_1, axis=1))
            self.a_1 = tf.nn.sigmoid(self.z_1)
            self.z_2 = tf.matmul(self.a_1, tf.stack(self.w_2, axis=0))
            self.a_2 = tf.nn.sigmoid(self.z_2)

            print("Pruned units! Number of units:", self.a_1_units)

        def training(self, y, eta):
            '''''
            Define what to do in each training epoch (backward propagation).

            Args:
                y: Labels tensor, of type int64 - [batch size].
                eta: Learning rate to use for backward propagation.

            Returns:
                train_epoch: Operation for a training epoch.
            '''
            # We use the sigmoid prime function to backpropagate the difference
            # between a_2 and y, and deduce how to alter all the variables.

            # ------------------LAYER #2------------------
            # Difference between probabilities and actual labels.
            # N.B.: One-hot encoding converts the labels into probabilities.
            diff = tf.subtract(self.a_2, tf.one_hot(y, self.a_2_units))
            # Difference between old and new values for weights.
            d_z_2 = tf.multiply(diff, sigmoid_prime(self.z_2))
            d_w_2 = tf.unstack(tf.matmul(tf.transpose(self.a_1), d_z_2), axis=0)

            # ------------------LAYER #1------------------
            # Difference between hidden layer values and its expected values.
            d_a_1 = tf.matmul(d_z_2, tf.transpose(tf.stack(self.w_2, axis=0)))
            # Difference between old and new values for weights.
            d_z_1 = tf.multiply(d_a_1, sigmoid_prime(self.z_1))
            d_w_1 = tf.unstack(tf.matmul(tf.transpose(self.a_0), d_z_1), axis=1)

            # Use the deduced differences to update the weights.
            # N.B.: The differences are in fact modulated by the learning rate.
            train_epoch = []
            for unit in range(self.a_1_units):
                train_epoch.append(tf.assign(self.w_1[unit], tf.subtract(
                    self.w_1[unit], tf.multiply(eta, d_w_1[unit]))))
                train_epoch.append(tf.assign(self.w_2[unit], tf.subtract(
                    self.w_2[unit], tf.multiply(eta, d_w_2[unit]))))

            return train_epoch

        def loss(self, y):
            '''''
            Define how the loss value is calculated from the logits (z_2)
            and the actual labels (y).
            N.B: Cross entropy compares the "score ranges" probability
                 distribution with the actual distribution
                 (correct class = 1, others = 0).

            Args:
                y: Labels tensor, of type int64 - [batch size].

            Returns:
                loss: Loss tensor, of type float32.
            '''
            with tf.name_scope('Loss'):
                # Cross entropy operation between logits and labels.
                # N.B.: Use z_2 because a_2 already is a softmax (of z_2).
                # N.B.2: One-hot encoding converts the labels into probabilities.
                cross_entropy = tf.reduce_mean(
                    tf.nn.softmax_cross_entropy_with_logits(
                        logits=self.z_2, labels=tf.one_hot(y, self.a_2_units),
                        name='cross_entropy'))

                # Operation for the final loss function.
                # N.B.: We add all the L2-regularization terms
                #       (squares of weights, multiplied by the reg_constant).
                loss = cross_entropy + tf.add_n(tf.get_collection(
                    tf.GraphKeys.REGULARIZATION_LOSSES))

            return loss

        def evaluation(self, y):
            '''''
            Define an operation for calculating the accuracy of predictions,
            how well the probabilities can predict the image's actual label.

            This operation is the proportion of correct predictions, the mean
            of the results of the binary test "does the component of a_2 with
            the greatest value correspond to the expected output y?".

```

```
            Args:
                y: Labels tensor, of type int64 - [batch size].

            Returns:
                accuracy: Percentage of images whose class was correctly predicted.
            '''
            # Define an operation for comparing the prediction with the true label.
            correct_prediction = tf.equal(tf.argmax(self.a_2, 1), y)

            # Define an operation for calculating the accuracy of predictions.
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

            return accuracy

        def quadr_error(self, y):
            '''''
            Define an operation for calculating the mean square error (MSE)
            of the predicted labels.

            Args:
                y: Labels tensor, of type int64 - [batch size].

            Returns:
                quadr_error: Mean square error of the predictions.
            '''
            quadr_error = tf.losses.mean_squared_error(
                labels=tf.one_hot(y, self.a_2_units), predictions=self.a_2)

            return quadr_error
```