LU TP 18-38
June 2018

TRAINABLE ACTIVATION FUNCTIONS FOR ARTIFICIAL NEURAL NETWORKS

Jim Öhman

Department of Astronomy and Theoretical Physics, Lund University

Bachelor thesis supervised by Mattias Ohlsson

Abstract

Artificial Neural Networks (ANNs) are widely used information processing algorithms based roughly on biological neural networks. These networks can be trained to find complex patterns in datasets and to produce certain output signals given a set of input signals. A key element of ANNs are their so-called activation functions, which control the signal strengths between the artificial neurons in a network, and which are normally chosen from a standard set of functions. This thesis investigates the performance of small networks with a new activation function, named the Diversifier, that differs from the common ones in that its shape is trainable, while the others are generally not. Additionally, a new method is introduced that helps to avoid the well known issue of overtraining. In the end it was shown that networks with the Diversifier performed slightly better compared to networks using two of the most common activation functions, the rectifier and the hyperbolic tangent, trained on two different datasets.

There have been articles covering explorations of different kinds of trainable activation functions, including, a trainable rectifier in a convolutional neural network (CNN) [5]. They also reported an improvement in performance. However, none of the ones read introduced something similar to the Diversifier.

Populärvetenskaplig sammanfattning

I dagens samhälle har maskininlärning en betydande roll, speciellt den delen som kallas för artificiella neuronnät (ANN). Dessa är användbara algoritmer som vagt härmar hur våra biologiska nätverk fungerar. De har visat sig vara skickliga i att lära sig klassificera, och generellt lära sig kopplingar mellan indata och utdata. ANN används flitigt för informationshantering av jättar som Facebook, Youtube och Google. Några användningsområden är bildigenkänning, bildförbättring, röstigenkänning, och datasortering, m.m.

Nyligen släppte ett Google ägt företag (Deep Mind) en ANN algoritm som kan lära sig olika brädspel genom att enbart spela med sig själv, utifrån att bara blivit given reglerna. Denna algoritm testades på de klassiska brädspelen: go, shogi, och schack, och efter bara en kort tids träning lyckades man uppnå, och passera, den mänskliga skicklighetsnivån. Denna imponerande bedrift visar att dessa nätverk klarar av att lösa komplexa och abstrakta problem.

En aspekt som nyligen gjort ANN mer användbara, och populära, är inte bara ökandet av datorkraft, men även hur nätverken tränas. Denna kandidatuppsats undersöker ifall det finns några positiva aspekter med att introducera ytterligare en träningsbar del av nätverken. Denna del är vad som kallas för aktiveringsfunktioner, som är en viktig komponent av artificiella neuroner eftersom de ser till att bestämma styrkan på de utgående signalerna beroende på styrkan på de ingående signalerna. Hur nätverkets aktiveringsfunktioner behandlar de ingående signalerna är något som man normalt inte har förändrat under träningen.

Contents

1 Introduction to Artificial Neural Networks **4**
1.1 The Simple Perceptron 4
1.2 The Multilayer Perceptron 5
1.3 Problem Types 6
1.3.1 Classification 6
1.3.2 Regression 8
1.4 Activation Functions 8
1.5 Training 10
1.5.1 Gradient Descent 11
1.5.2 Input Normalization 13
1.5.3 Weight Initialization 13
1.5.4 Optimizations 13
1.5.5 Regularizations 15

2 Trainable Activation Functions **16**

3 Method **19**
3.1 Datasets 19
3.1.1 Synthetic (Regression) 19
3.1.2 Bone Metastases (Classification) 20
3.2 Validation Methods 20
3.2.1 K-Fold Cross Validation 20
3.2.2 Performance Measures 21

4 Results **21**
4.1 Synthetic (Regression) 21
4.1.1 Performance Comparison 22
4.1.2 The Diversified Nodes 23
4.2 Bone Metastases (Classification) 24
4.2.1 Performance Comparison 25
4.2.2 The Diversified Nodes 26
4.2.3 Training Depth (Without Regularization) 28
4.3 The Restrictive Fixed Shape Method 28

5 Conclusions **30**

1 Introduction to Artificial Neural Networks

In this section, an introduction to, and information about, the construction and training of a feed-forward artificial neural network is presented.

1.1 The Simple Perceptron

The perceptron [1] can be thought of as a building block in artificial neural networks. It can be seen as one of the artificial neurons, as it is constructed to resemble the biological neuron.

An illustration of the perceptron can be seen in Fig. 1.

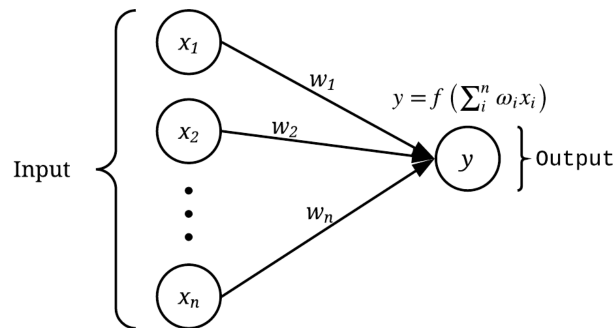


Figure 1: A sketch of the simple perceptron. Each circle represents a node in the network. In this figure there are n input nodes and 1 output node.

As a summary of Fig. 1, the perceptron takes inputs and computes outputs through a fairly straight forward mathematical procedure. An input vector $\vec{x} = (x_1, \ldots, x_n)$ consisting of the input values $\{x_i\}_i^n$ is multiplied with a vector of weights $\vec{\omega} = (\omega_1, \ldots, \omega_n)$, which are network parameters. The multiplication results in a weighted sum of the inputs: $\vec{\omega} \cdot \vec{x} = \sum_i^n \omega_i x_i$, that is then used as the argument to a usually non-linear function f called an activation function, see Sect. 1.4 for examples. The value $y = f(\sum_i^n \omega_i x_i)$ is then considered to be the output of the neuron.

The weights are the trainable parameters of the perceptron, usually trained such that a specific set of inputs $\{\vec{x}(j)\}_j$ produces a desired set of outputs $\{\vec{d}(j)\}_j$. The set of these input and output pairs: $\{\vec{x}(j), \vec{d}(j)\}_j$, is called a dataset. An introduction on how to train these networks is located under Sect. 1.5.

1.2 The Multilayer Perceptron

The multilayer perceptron (MLP) [2] is constructed from a set of layered simple perceptrons. The outputs from one layer of perceptrons are taken as the inputs to the next. An illustration of an MLP can be seen in Fig. 2.

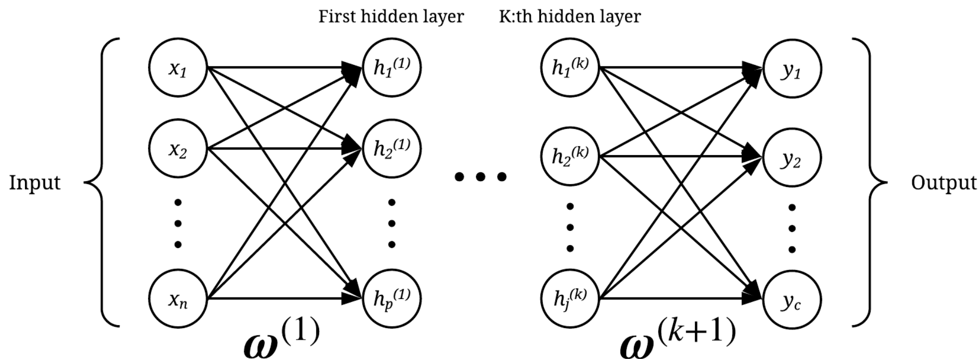


Figure 2: A sketch of the multilayer perceptron.

As an example, the output from perceptron a, in layer k, is given by:

$$h_a^{(k)} = f^{(k)} \left(\sum_i^m \omega_{ai}^{(k)} h_i^{(k-1)} \right)$$

where $f^{(k)}$ is the activation function for layer k, m is the number of nodes in layer $(k-1)$, and ω_{ai} is the a:th row and i:th column value of the weight matrix $\boldsymbol{\omega}^{(k)}$ connecting layer (k) and layer $(k-1)$.

There are three types of layers in an MLP: an input layer, a hidden layer, and an output layer. In this construction there can only be one input layer, and one output layer, that has a fixed number of nodes that depends on the problem that the MLP is applied on. However, the number of hidden layers and the number of hidden nodes are arbitrary. Those numbers are non-trainable parameters of the network, also called hyperparameters, and are to be chosen before any training begins. As a side note, a network with a large number of hidden layers has been given its own name as a deep neural network.

The simple perceptron and the multilayer perceptron that are shown in Fig. 1 and Fig. 2 are examples of feed-forward networks, which means that the information is propagated forward through the layers and none is propagated back. A network that can also propagate information back through the layers is called a recurrent network, which is used for time-series predictions, where the network benefits from remembering outputs at previous times. However, only feed-forward networks are considered from this point on.

1.3 Problem Types

In this subsection, two types of problems that the ANNs can be used for are introduced.

1.3.1 Classification

A classification problem is simply a task where a specific input needs to be labelled. For example, consider the inputs to consist of medical measurements taken of a patient. Through the values of that combination of measurements, doctors would be able to determine if the patient is free from a specific condition or not. This is an example of a binary classification problem.

For **binary classification** the target outputs can be either 0 or 1, which corresponds to the two different classes. Thus, the output from the network must be between 0 and 1, which can simply be achieved by choosing an activation function with this range for the output layer, for example, the Sigmoid function, see Sect. 1.4. As the output is continuous, it can be interpreted as the probability of the input belonging to the class with representation 1. This means that when the network output for a specific input is, for example, bigger than or equal to 0.5, it is equivalent to that input belonging to the class assigned to 1, and vice versa if it is below 0.5.

For **multi-class classification** the target outputs can be set as vectors, where, for example, one class can have the vector representation $(1, 0, 0)$, another $(0, 1, 0)$, and the third $(0, 0, 1)$. Hence, three output nodes are required in this example. Also here, the output vector can be interpreted as probabilities, which requires the activation function for the output layer to be between 0 and 1, as well as satisfying the condition $\sum_i^n y_i = 1$. This can be fulfilled by applying the softmax function:

$$y_i = \frac{e^{t_i}}{\sum_j^n e^{t_j}}$$

where t_j is the weighted sum of the inputs for the j:th output node. Thus, the network output $\vec{y} = (0.3, 0.6, 0.1)$ gives a 30% probability for the input to belong to the class with representation $(1, 0, 0)$, 60% for $(0, 1, 0)$, and 10% for $(0, 0, 1)$. The input is then chosen to belong to the class with the highest probability.

A simple perceptron handles only binary classifications in which the classes are linearly separable. Being linearly separable means that one must be able to find a hyperplane separating the inputs into the two classes in the input space. As the network output $y = f(\vec{\omega} \cdot \vec{x})$ is interpreted as a probability, and is cut-off between the classes at output 0.5, a condition is set on the weights to be able to separate the classes:

$$f(\vec{\omega} \cdot \vec{x}) = 0.5.$$

For the Sigmoid function, this is satisfied if the argument is 0, which means that

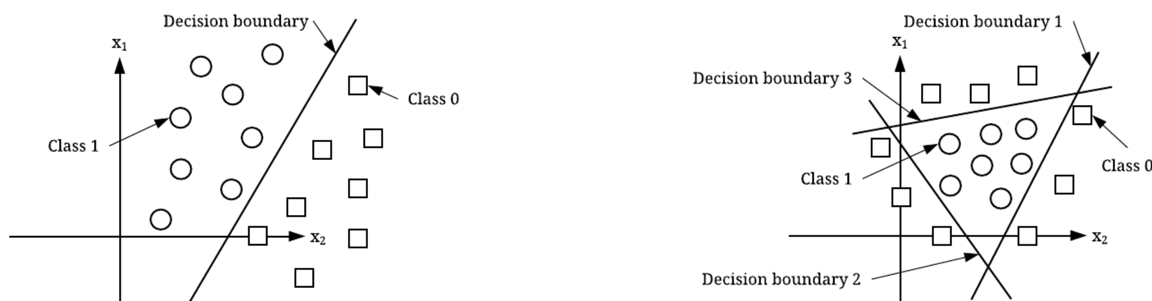
$$\vec{\omega} \cdot \vec{x} = 0.$$

This happens to be the equation for a hyperplane in input space. Thus, if the classes are linearly separable, the task of the simple perceptron is to find a normal to one of the hyperplanes separating the classes. Such a hyperplane is also called a decision boundary. However, if the classes are not separated by a hyperplane through the origin, this cannot be satisfied. To solve this, a so called bias node is added to the input layer, which is always given the value 1. This results in an additional weight, called a bias weight, that can be trained to shift the position of the hyperplane. Normally, a bias node is added to MLPs as well, to every layer except for the output layer, for this very reason.

An MLP is able to handle both binary and multi-class classification, and does not require a linear separation between the classes, given that at least one of the activation functions in the hidden layers are non-linear. It is easily shown that an MLP with only linear activation functions in the hidden layers is equivalent to a network with only an input and an output layer. The extra layers would just rescale the weights. Thus, it is the added complexity of the hidden layers that allows for the network to handle more complicated problems. However, a specific task will have an optimal number of hidden layers and nodes, therefore a larger network is not necessarily favourable.

The nodes in the first hidden layer have a theoretical interpretation, which is that they serve as creating decision boundaries in the input space, precisely as mentioned above for the simple perceptron. If the classes are not linearly separable, they can instead be separated through a set of hyperplanes determined by the weight vectors of the first layer nodes. Thus, knowing some of the structure between the classes before hand, allows one to choose a reasonable number of nodes in the first hidden layer. The nodes in other hidden layers does not have such a simple interpretation in the input space.

An illustration of a two dimensional dataset that is linearly separable into two classes, and one that is not, can be seen in Fig. 3.



(a) Linearly separable. Only one decision boundary is necessary, therefore no hidden nodes are required.

(b) Not linearly separable. A network would need at least three hidden nodes to separate this dataset.

Figure 3: Shown in the subfigures are examples of two dimensional datasets that are linearly separable, and not linearly separable, illustrated together with decision boundaries.

1.3.2 Regression

A regression problem requires one to find relationships between dependant variables and one or more independent variables. For example, given a dataset of points: $\{(x(j), y(j), z(j)\}_j^N$, find a function $z = f(x, y)$ between the independent variables x, y and the dependant variable z, given some conditions. The task of the network is to find a trend function that closely matches the outputs given the inputs.

As the network output has to be able to take on most values, both positive and negative, a suitable activation function for the output layer is simply a linear function: $f(x) = x$.

An illustration of two dimensional linear regression can be seen in Fig. 4.

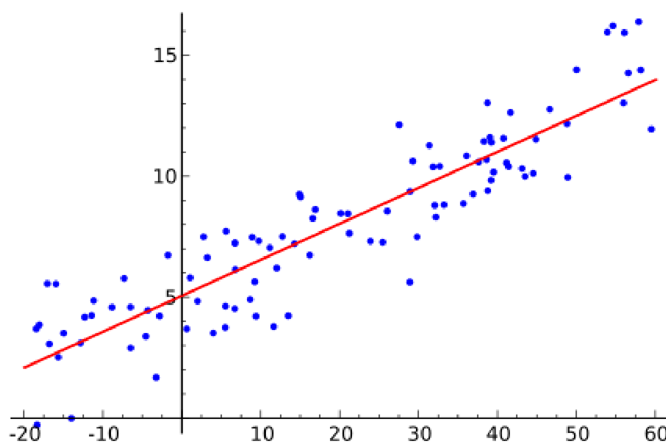
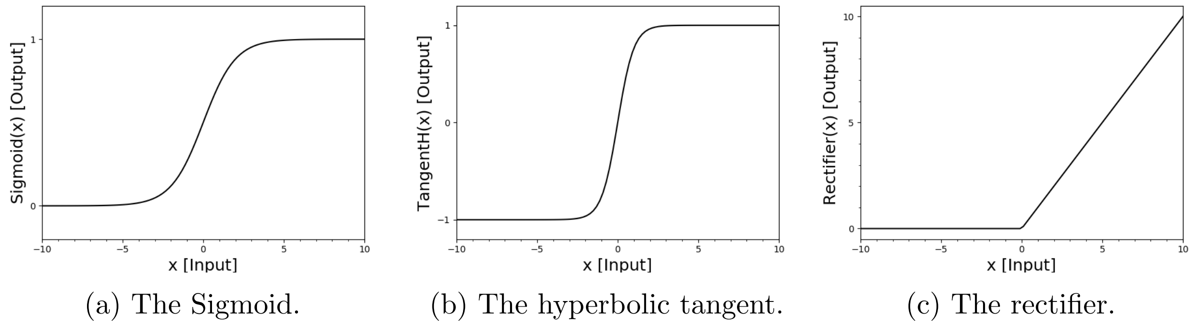


Figure 4: A sketch of linear regression. The dots are data points, and the line corresponds to the trend function, which is the target function of the network. [3]

1.4 Activation Functions

Activation functions determine the activity of nodes in the network, resembling the action potential for a biological neuron. They also serve to non-linearize the network and increase its complexity, which, as mentioned before, increases its problem solving abilities.

Three of the most used non-linear activation functions are shown in Fig. 5 and given by



(a) The Sigmoid. (b) The hyperbolic tangent. (c) The rectifier.

Figure 5: Three standard activation functions.

The sigmoid function (Fig. 5a),

$$\mathrm{Sigmoid}(x) = \frac{1}{1+e^{-x}}.$$

The hyperbolic tangent function (Fig. 5b),

$$\mathrm{Tanh}(x) = \frac{e^{2x}-1}{e^{2x}+1}.$$

The rectifier function (Fig. 5c),

$$\mathrm{Rectifier}(x) = \begin{cases} x, & \text{if } x > 0, \\ 0, & \text{else.} \end{cases}$$

The activation functions for an MLP are normally chosen layer wise, such that the nodes in one layer employs the same activation function. Additionally, they are chosen before the training starts.

Which activation function to use, and when, is not fully understood. However, some of them have advantages over others, depending on, for example, the size of the network.

The rectifier function is computationally easy, which makes it preferable in the hidden layers for deep neural networks.

However, the optimal choice, in terms of performance, of activation functions for a given problem, has to be determined by trial and error.

1.5 Training

In this subsection the training of a feed-forward ANN is discussed, introducing a mathematical method that can be used to update the weights, including: how to initialize the weights, how to optimize the learning procedure, and how to manage overtraining.

First of all, what is required for the network to learn a given task is a way for it to evaluate its performance on the task, and a way to change its internal structure in accordance to the evaluation, such that it improves. With a fixed number of nodes and layers, it is normally the weights that represent the internal structure.

Providing the network with a dataset, gives it a way of evaluating its performance by determining a difference between the network outputs and the targets, given the inputs. This is usually done through what is called an error function or cost function. Two such functions are introduced below.

Considering a network with K output nodes, the **mean squared error** (MSE) can be constructed as:

$$E = \frac{1}{2N} \sum_{n=1}^{N} \sum_{k=1}^{K} \left(y_k(n) - d_k(n)\right)^2 .$$

Here, $d_k(n)$ is the target for output node k, $y_k(n)$ is the output from node k, and as before, N represents the total number of samples in the dataset. The division by two is a convention used to simplify the derivative of the function. This elementary error function is quite useful, since it can be applied as a performance evaluation for any problem where a dataset has been provided, for both classification and regression.

A less intuitive looking error function is the **binary cross-entropy error** (BCEE), which takes the form:

$$E = -\frac{1}{N} \sum_{n=1}^{N} \left(d(n) \log(y(n)) + (1 - d(n)) \log(1 - y(n))\right) .$$

This error is used for binary classification problems, and is derived by the probability interpretation of the outputs and the maximization of the conditional probability for a class given an input. For a binary classification, the conditional probability for observing either class can be written together as $p(d|\vec{x}) = y^d(1-y)^{1-d}$, where $d \in \{0, 1\}$ is the target and y the network output. Instead of maximizing $p(d|\vec{x})$ it is more natural to minimize the negative logarithm of it. Summing up the contributions from all the inputs in a dataset and taking the mean produces the binary cross-entropy error as it is shown above. It provides a measure of how far the network is from predicting the right class given an input, and would penalize a wrong prediction more than the MSE would. Thus, it is usually favourable to use BCEE over MSE as a performance evaluation for binary classification problems.

There are other ways of measuring the performance of a network, but given the mathematical method presented below, the MSE and BCEE are useful as they are easily differentiable and not very computationally heavy.

Providing the network with a dataset that it is supposed to learn is called supervised learning, given that what is a correct output for a specific input has in advance been determined by a "supervisor". Along side this, there is unsupervised learning, where the network is provided only a set of inputs. The task is then to find similarities or relationships between the inputs.

By using supervised learning, one hopes that the network will pick up on the general structure of the dataset, and would thus be able to fairly estimate the correct outputs from other inputs that was not part of the training dataset.

The network error for the training dataset is naturally called training error, while the error on data that was not used in the training is called validation error. It is usually the validation error that one wants to minimize as this gives a measure on how well the network would generally perform on the task.

It is easy to achieve perfection on the training dataset, given a large enough network, but this does often lead to a poor validation performance. This problem arises because the network starts to memorize the dataset instead of learning the general structure. The phenomenon is called overtraining and is discussed further in Sect. 1.5.5 with methods on how to control it.

How to proceed and update the weights in order to minimize the error function is discussed below.

1.5.1 Gradient Descent

A mathematical method that is often used in the training of a network is called gradient descent, where the weights of the network are updated by making use of the gradient of the chosen error function.

A networks error function spans a manifold within a $(T+1)$ dimensional space, where T stands for the total number of weights in the network. All possible values of the error function construct a landscape within that space, a hilly landscape with a lot of local minima and maxima. The shape of this error landscape depends on the activation functions in the network, and on the dataset.

The state of a network can be thought of as a point $(E, \vec{\omega}_{tot})$ represented in the error landscape, positioned at the current network performance E with weights given by the total weight vector: $\vec{\omega}_{tot} = (\omega_1, \ldots, \omega_T)$ containing all the weights of the network. As the network learns, it moves within this landscape, changing its weights with the objective to go as low as possible, minimizing the error.

Provided a starting performance, or equally a position in the error landscape, usually given by a semi-random initialization of the weights, one can find the steepest direction to move in order to reduce the error the most, by calculating the gradient of the error function with respect to the weights.

The gradient of the error function $E(w_1, \ldots, w_N)$ is given by:

$$\nabla E(w_1, \ldots, w_T) = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_T} \right).$$

This is a vector in the T dimensional subspace of the error space, and it can be shown that, locally, the negative of ∇E will point in the direction of steepest descent for the network. Adding $-\nabla E$ to the total weight vector $\vec{\omega}_{tot}$ causes the network to take one step in the steepest direction, but the step size has to be small as the gradient is likely to change within the landscape. A small positive constant η is multiplied to the gradient before the addition. If the step size is large the network might move too far and the error could instead increase. Naturally η is called the learning rate.

Hence, updating the weights according to:

$$\vec{\omega}(t+1)_{tot} = \vec{\omega}(t)_{tot} - \eta \cdot \nabla E(\vec{\omega}(t)_{tot}),$$

advances the network step by step in the directions that will minimize the error. In the above, t stands for the iteration.

The weights will continue to change, and the network move, until it converges to a minimum for the error function, where the gradient is zero. It is not guaranteed that the updating algorithm will converge to a minimum that is a global one, because there are usually more local ones. However, converging to a global minimum might not be desirable because of the previously mentioned phenomenon of overtraining. Thus, finding the best local minimum that would make both the training and validation error as small as possible is not an easy task, because the convergence depends on the learning rate and as well as on the starting position of the network. Though, one method that seems to help avoid some undesirable local minima introduces noise to the updates. This is done by not using the full amount of samples from the dataset when the weight updates are calculated. If the dataset contains N samples, a random selection of those will be collected as a mini-batch to use for each update. This causes the network move semi-stochastically in the error landscape, hence its name: stochastic gradient descent. A small terminology comment; an epoch is one training round that has iterated trough all mini-batches of the dataset once.

An additional issue are areas in which the error landscape is relatively flat, because convergence of the updating algorithm might take too long. Thus, weight initialization becomes important. However, it might be unavoidable to travel through flat areas as the network learns, but there are some methods that can be deployed to gradient descent that usually speeds up the learning process.

1.5.2 Input Normalization

Input normalization is a method that puts each input variable on equal terms, meaning that it transforms each input variable from the dataset to belong to distributions with zero mean and unit variance. This can improve the learning procedure by letting the input variables contribute similarly to the weight updates, thus making use of the data more effectively. It also has the effect of transforming all the inputs to be smaller, which can be helpful for computational purposes.

To do this transformation one has to calculate the mean and the variance of the input variables separately. For example, take the input variable x_1 from $\vec{x} = (x_1, \ldots, x_n)$, which has, given a specific set of inputs $\{\vec{x}(j)\}_j^N$, the mean:

$$\langle x_1 \rangle = \frac{1}{N} \sum_{j=1}^{N} x_1(j),$$

and the variance:

$$\sigma_1^2 = \frac{1}{N} \sum_{j=1}^{N} \left(x_1(j) - \langle x_1 \rangle \right)^2 .$$

It is transformed according to:

$$x_1 \rightarrow \frac{x_1 - \langle x_1 \rangle}{\sigma(x_1)}.$$

1.5.3 Weight Initialization

This is the process of choosing the initial values of the network weights.

This is important because the initial set of weights determine where in the error landscape the network will start during gradient descent training. If it is far from the ideal final position, it could get stuck in a suboptimal local minimum, or perhaps the landscape is quite flat and the updating algorithm takes too long.

If the dataset is normalized one can initialize the weights to make the inputs to the first hidden layer have zero mean and a fixed small variance. This is done by randomly selecting the initial value of each weight from the interval: $\left[-1/\sqrt{d}, 1/\sqrt{d}\right]$, where d is the number of inputs to the hidden node that the weight forwards information to. This method can speed up the learning process for certain activation functions with large derivatives close to the origin.

1.5.4 Optimizations

Optimization techniques aims to speed up the learning process and help the network avoid unwanted local minima during training. There are many such techniques, but the one used in this thesis is called **Adam** [4].

Standard gradient descent has a fixed learning rate for all the weights, such that the movement of the network within the error landscape stays rigid, having the same step size, which is usually not favourable.

The Adam optimization makes sure that each weight in the network has its own adaptive learning rate, introducing the effect of acceleration and momentum to the network as it moves around in the error landscape. With Adam the network will be able to more effectively avoid unwanted local minima, and move with speed in flatter areas.

The weights updates are done in a quite elaborate way:

$$\omega_i(t+1) = \omega_i(t) - \eta \frac{\hat{m}_i}{\sqrt{\hat{\nu}_i} + \epsilon},$$

where,

$$\begin{aligned} \hat{m}_i &= \frac{m_i(t+1)}{1-\beta_1^t}, \\ m_i(t+1) &= \beta_1 m_i(t) + (1-\beta_1)\frac{\partial E(t)}{\partial \omega_i}, \\ m_i(0) &= 0, \end{aligned}$$

and,

$$\begin{aligned} \hat{\nu}_i &= \frac{\nu_i(t+1)}{1-\beta_2^t}, \\ \nu_i(t+1) &= \beta_2 \nu_i(t) + (1-\beta_2)\left(\frac{\partial E(t)}{\partial \omega_i}\right)^2, \\ \nu_i(0) &= 0. \end{aligned}$$

Here, η is now the initial learning rate, and ϵ is a correction applied to avoid an initial bias towards small values, β_1 and β_2 are decay rate parameters. The authors of this method suggests that $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

To illustrate the effects of the method, assume that the network is initialized in a constant downslope of the error landscape. It will start moving downwards with each weight update, and in non-optimized gradient descent the movement of the network will be with the same speed no matter the iteration. However, by updating the weights with a portion of the previous gradients, as is done with m_i in Adam, the network will speed up as it moves downwards, effectively reaching the bottom faster than standard gradient descent. A downside with speeding up is that the network might surpass the minimum at the bottom, as it had gained too much speed. This is dealt with from the way m_i is constructed, as the speed has an upper limit that is dependant on the steepness of the slope. Though, if the slope is too steep, the same problem arises. This is why ν_i is introduced, its effect is partly to remove the dependence of the maximum speed on the slope, making it dependant only on the initial learning rate.

If the network then were to encounter an uphill, it will not immediately change direction as it would in standard gradient descent, but will continue upwards, slowing down, as if it had momentum.

In other words, updating with a track of previous gradients, as in Adam, essentially introduces gravity to the error landscape, as well as an atmosphere that puts an upper limit on the speed. Thus, during training, the network can be thought of as a ball rolling in a hilly landscape, through an atmosphere, trying to minimize its gravitational potential.

1.5.5 Regularizations

As previously mentioned, it is relatively easy to train the network perfectly on a given dataset, just provide a large enough network. Though, this results in memorization, where the network memorizes the dataset instead of learning its general structure. This phenomena is called overfitting and is usually undesirable as the network will perform badly on data that was not used in the training. There are several methods that can be used to prevent this. Such a method is **weight decay**, and is introduced below.

Weight decay corresponds to adding a penalty term to the error function, which stops the network from growing its weights too large. There are two main types of penalty terms, called the **L1-norm** and the **L2-norm**, which has the following forms:

$$\text{L1:} \ \lambda \sum_i |w_i| \,,$$

$$\text{L2:} \ \lambda \sum_i w_i^2.$$

Here ($\lambda > 0$) is another hyperparameter, and the sums are taken over all of the weights in the network.

As an example, for the L2-norm, the MSE is transformed according to:

$$E \to \hat{E} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{k=1}^{K} (y_k(n) - d_k(n))^2 + \lambda \sum_i w_i^2,$$

and similarly for the L1-norm.

Both the L1 and L2 method changes the gradient and therefore how the weights are updated. As λ is taken to be strictly positive, and the error function is to be minimized, the weights are pushed towards small values. Having all of the weights equal to zero would most likely not minimize the unmodified error, which means that the network has to keep the weights that are important and let the rest go towards zero, effectively making the network learn more of the general structure.

One key difference between the two methods is that the derivative of the L1-norm does not depend on the size of the weights, thus pushing them all equally fast towards zero. For the L2-norm, the push does depends on how big the weights are, which means that those closer to zero become less penalized with each iteration.

Which one of these that works best depends on the problem, and is something that has to be empirically tested.

2 Trainable Activation Functions

As previously mentioned, activation functions are generally selected from a set of common ones, and are chosen to each layer before training starts. This might not be ideal as activation functions play a big part in shaping the error landscape. The optimal minimum for a specific landscape might not be the optimal minimum among all possible landscapes.

To avoid the task of having to choose between activation functions, one can introduce **Parametrized Activation Functions** (PAFs) with trainable parameters that allows the network to change the structure of the functions as it learns.

The parameters introduced below are trainable through gradient descent and optimizable by Adam, precisely in the same way as normal weights.

As a first simple approach one could take a linear combination of two common activation functions in the following manner:

$$\mathrm{PAF}^{(k)}(x, \alpha_k, \beta_k) = \alpha_k \cdot f_1(x) + \beta_k \cdot f_2(x), \tag{1}$$

where k stands for the k:th layer.

If α_k and β_k are allowed to be any value, then the PAF will ultimately have some issues with overfitting. To avoid this, α_k and β_k could be restricted to a finite interval $[a, b]$, such that with each update they will be multiplied by a factor returning them to a or b if they exceed their limits.

With that construction, a continuous transformation between $f_1(x)$ and $f_2(x)$ can be achieved by setting the restriction interval to [0,1] and $\beta_k = 1 - \alpha_k$. This could be of interest if one wants to see which activation function seems to be most natural for a specific dataset.

Restricting α_k and β_k in the way proposed above might not be optimal, as the derivatives do not saturate if the network tries to reach values outside of the interval, this is unnecessarily computationally demanding. Instead, one can use a new function $g(\alpha)$ with range [0,1] to encode the transformation between the two functions right from the start. This would stop the network from trying to achieve a specific combination of the functions that it is unable to. A natural choice for $g(\alpha)$ would be the Sigmoid function (Fig. 5a), as it is continuous, has a range between [0,1], and changes relatively quickly around the origin.

Thus, a PAF of this form would be:

$$\mathrm{PAF}^{(k)}(x, \alpha_k) = \Sigma(\alpha_k) \cdot f_1(x) + (1 - \Sigma(\alpha_k)) \cdot f_2(x),$$

where Σ is decided to stand for the Sigmoid function for brevity.

Additionally, one can parametrize the activation function to each node instead of to each layer, giving the nodes individual activation functions:

$$\mathrm{PAF}^{(ik)}(x, \alpha_{ik}) = \Sigma(\alpha_{ik}) \cdot f_1(x) + (1 - \Sigma(\alpha_{ik})) \cdot f_2(x).$$

Here i stands for the i:th node in the k:th layer.

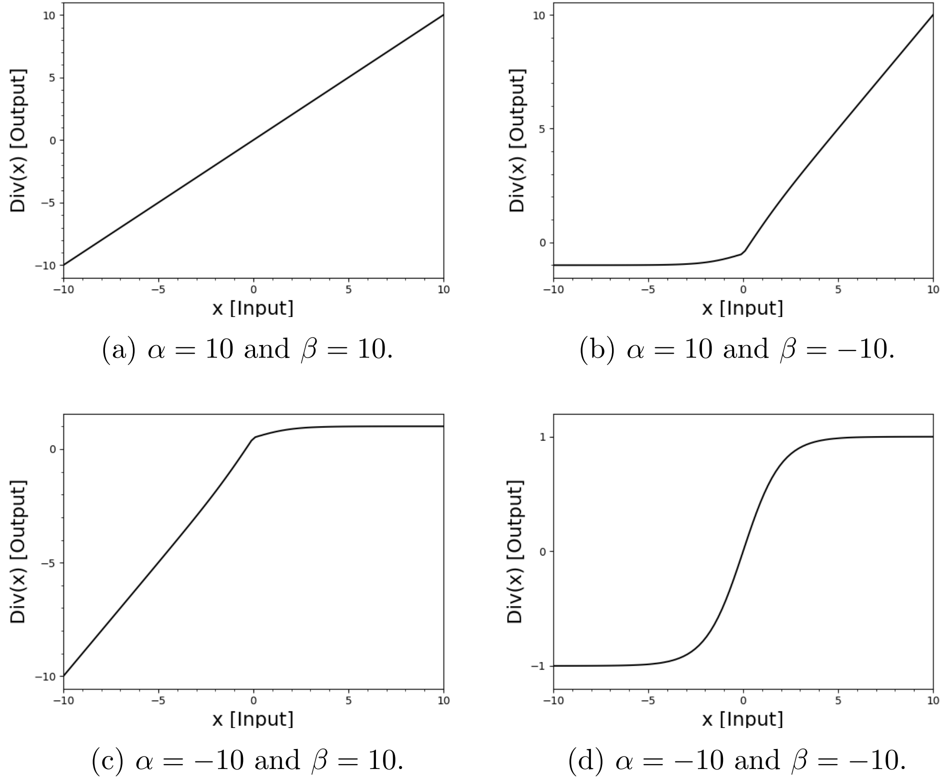
Controlling the PAF outputs for positive and negative inputs separately introduces another flexibility. To achieve this, a form similar to (1) has to be used, with two parameters, and activation functions $f_1(x)$ and $f_2(x)$ of which one is zero for positive x and the other is zero for negative x.

As the rectifier and the hyperbolic tangent are typically top contenders in performance, a PAF incorporating similar forms could be of interest.

This thesis introduces one such parametrized activation function, which combines all three of the above approaches. It has been given the name **the Diversifier**:

$$\mathrm{Div}^{(ik)}(x, \alpha_{ik}, \beta_{ik}) = \begin{cases} \Sigma(\alpha_{ik}) \cdot x + (1 - \Sigma(\alpha_{ik})) \cdot \Sigma(x), & \text{if } x \geq 0, \\ \Sigma(\beta_{ik}) \cdot x - (1 - \Sigma(\beta_{ik})) \cdot \Sigma(-x), & \text{else.} \end{cases}$$

An illustration of four different shapes that the function can transform itself to, is shown in Fig. 6.



(a) $\alpha = 10$ and $\beta = 10$.

(b) $\alpha = 10$ and $\beta = -10$.

(c) $\alpha = -10$ and $\beta = 10$.

(d) $\alpha = -10$ and $\beta = -10$.

Figure 6: Four different shapes of the Diversifier function.

As can be seen in Fig. 6, the Diversifier has continuous transformations to three known activation function shapes: the linear shape (Fig. 6a), as well as a shape (Fig. 6b) with similarities to the rectifier but with a non-zero tail, which it can also reflect (Fig. 6c), and lastly, a shape very close to the hyperbolic tangent function (Fig. 6d). The terminology for its shapes will be rectifier and hyperbolic tangent for simplicity.

With the Diversifier **a new regularization method** is available. By initializing the Diversifier with a shape close to the linear function one restricts the network, as it removes the effects of the hidden layers, at first making it a simple perceptron. This can be prolonged by letting the first 300 epochs or so be fixed, then suddenly switching on the learning rate for the activation function parameters (AFPs). This method has a simple interpretation for classification problems, during the first 300 epochs the weights are forced to act together to find one decision boundary that splits the two classes the best. When the learning rate is applied again, and the hidden nodes gain their effect, new decision boundaries seem to end up in favourable positions. The limit of 300 epochs for restriction is not special, the idea is that one would like the normal weights to converge before the learning rate is switched on for the AFPs, and how many epochs that takes depends on the dataset.

Having a fixed non-linear shape in the first part of the training also seems to improve performance. As will be seen in the results section, the regression problem required the Diversifier to be initialized with the hyperbolic tangent shape for maximum performance, where this method was applied.

Initializing to a non-linear fixed shape for the first hundreds of epochs has been given the name the **fixed shape method**. While initializing to a linear fixed shape has been given its own name as the **restrictive fixed shape method**.

As mentioned before, the AFPs are treated the same as normal weights; they are trained through gradient descent with their own learning rate, and are optimizable by the Adam method.

A Diversifier introduces three new hyperparameters: the initialization of α and β, and the learning rate of the AFP.

3 Method

The MLP architecture was constructed from scratch in Python, as well as all the additional methods such as Gradient Descent, Adam, and K-Fold Cross Validation [7] (briefly covered in Sect. 3.2.1). The standard weight updates were set to follow the method of back-propagation [8], and were adjusted to also accept the activation function parameters. The adjustment was straight forward, as it is only the last part of the chain derivative of, for example, $\partial E/\partial \alpha$ that is different from $\partial E/\partial \omega$, assuming that α and ω belong to the same layer, which means that the back-propagated errors can be used for the activation function parameters as well.

For the datasets described in Sect. 3.1, a reasonable network size of 1 hidden layer with 10 hidden nodes was used for both, and hyperparameters such as learning rate and L2 weight decay parameter, were optimized when the hyperbolic tangent was used in the hidden layers, and also for the rectifier. The optimization was done through 10 times repeated 10-fold cross validation over a thousand epochs with a full batch. Both activation functions had exactly the same weight initializations and folds. The hyperparameters were optimized in the sense of maximizing validation performance.

Then, for a fair comparison with the Diversifier, the same network size was used, with hyperparameters equal to the optimal of either the ones for the hyperbolic tangent or the rectifier. This was then run in another 10 times repeated 10-fold cross validation on the datasets, with precisely the same weight initialization and folds as for the others.

The performances of the networks were captured by graphing the training and validation measures described in Sect. 3.2.2.

3.1 Datasets

Below is a description of the two datasets that were used in the comparison between the Diversifier, the hyperbolic tangent and the rectifier.

3.1.1 Synthetic (Regression)

This is a synthetic dataset for a regression problem, where the data is generated through the formula:

$$y = 2x_1 + x_2 x_3^2 + e^{x_4} + 5x_5 x_6 + 3\sin(2\pi x_6) + \alpha\epsilon.$$

The variable ϵ is normally distributed noise with zero mean and unit varance, and α is the parameter controlling its size. The variables $x_1, \ldots, x_4$ are taken from a normal distribution with zero mean and unit variance, and x_5, x_6 are taken uniformly from the interval $[0, 1]$.

There are a total of 6 independent variables $x_1, \ldots, x_6$ and one dependant variable y. Thus, 6 input nodes and one output node was required.

For this thesis 2000 samples were generated with $\alpha = 0.7$.

3.1.2 Bone Metastases (Classification)

This dataset [9] consists of a set of measurements and indicators taken from scans of 1013 patients undergoing bone scintigraphy. These patients have previously been diagnosed with prostate cancer, and have undergone scintigraphy to examine if any bone metastases have appeared.

This is a binary classification problem, as the targets are binary. The presence of metastases from the measurements has been determined by experts beforehand.

A total of 46 measurements were used, therefore 46 input nodes were required, as well as one output node for the target.

3.2 Validation Methods

To measure the validation performance of a network on a specific task, one can use part of the data from the provided dataset as a validation dataset, usually around a third of the size of the original. The network is then trained on two thirds and evaluated on the rest. This is an example of Hold-Out Validation. A downside with this approach is that not all of the data is being used to train the network, which is especially important for the generalization performance if the original dataset is small. An additional downside is that the validation dataset and the training dataset could be different enough to introduce a bias, and give a distorted estimation.

Optimally one would like to use all of the data in the training phase to avoid a large bias, but still be able to estimate the validation performance. This can partly be achieved by the method of K-Fold Cross Validation that is introduced below.

3.2.1 K-Fold Cross Validation

The method proceeds as follows: Firstly, the dataset is divided in to K equal parts. Secondly, one of these parts is chosen as the validation dataset, and the rest as the training dataset. Thirdly, the network is trained and evaluated on the chosen sets respectively, and the performance measures are registered. Fourthly, a new part for the validation dataset is chosen, and the process is repeated until all parts have been used for both validation and training.

The final estimate of the network performance is taken as the average of the K performances.

As not all of the data is used in each of the K trainings, the final estimated performance is biased. Using a sizeable K will decrease this bias, as more of the dataset is used in each training. Though, this will increase the variance between each folds performance, due to the increasing probability that the validation dataset contains a proportionately large amount of hard to classify samples. To deal with a larger variability one could repeat the K-Fold Cross Validation using different divisions, many times, and then take the final estimate as the average of the before final estimates, this allows for an unbiased reduction of the variance in the mean.

A value of K between 5 and 10 is considered appropriate for network testing, and for the repetition it is the same.

3.2.2 Performance Measures

The are many different ways of quantifying the performance of a network on a specific problem. The ones used in this thesis are introduced below.

For **regression problems** there is the before mentioned Mean Squared Error, as well as the Correlation Coefficient [10], which measure the linear relationship between the network outputs and targets. A correlation coefficient value of 1 would imply a perfect match, while a value of 0 would imply zero correlation.

For **binary classification problems** there is the before mentioned Binary Cross-Entropy Error, as well as standard Accuracy, which is simply the total amount of correct classifications in ratio with the total number of classifications.

4 Results

In the sections below are the results from the comparison between the Diversifier, the rectifier and the hyperbolic tangent, on the different datasets.

Together with the graphs of training and validation performances, an example of the final forms for the Diversified nodes are shown. Additionally, a comparison between the Diversifier with and without the new regularization method is presented.

For the graphs below; the sharp lines represents the mean values, and the transparent areas represents one standard deviation from the mean.

4.1 Synthetic (Regression)

For this dataset, a network size of 6 input nodes, 10 hidden nodes, and 1 output node was used together with a linear output activation function. The hidden layer had the same activation function for all nodes, either the rectifier, the hyperbolic tangent, or the Diversifier.

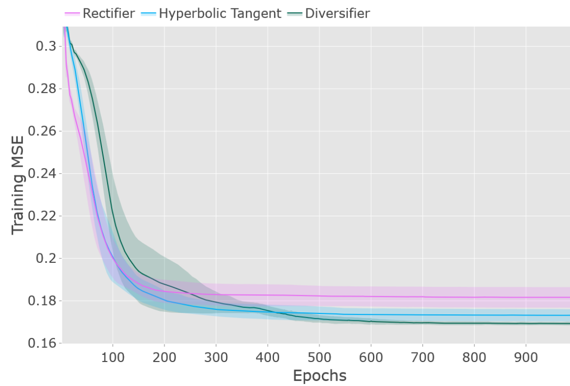
The input data and the output data were normalized according to Sect. 1.5.2, and the weights were initialized according to Sect. 1.5.3. The applied optimization method was Adam, and the applied regularization method was the L2-norm.

For the hyperbolic tangent, the optimal hyperparameters were:
Learning rate = 0.03, and L2-parameter = 0.002.

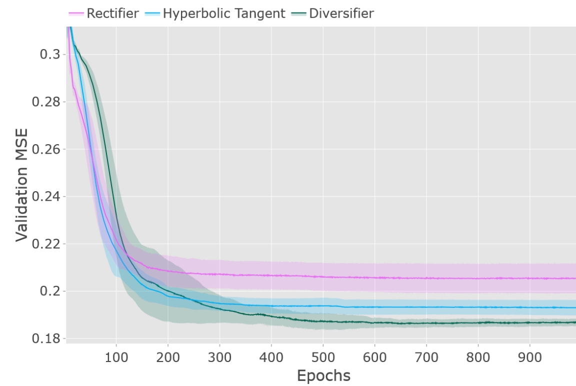
For the rectifier, the optimal hyperparameters were:
Learning rate = 0.05, and L2-parameter = 0.002.

The hyperparameters for the Diversifier was chosen to be the same as for the rectifier. Additionally, the fixed shape method was used; the Diversifier was initialized to the hyperbolic tangent shape, with $\alpha_{ik} = \beta_{ik} = -10$ for all ik, and was kept fixed for the first 300 epochs, then the learning rate was applied to the AFPs as well.

4.1.1 Performance Comparison

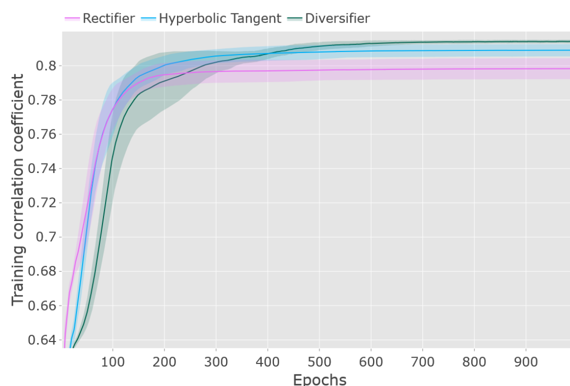


(a) Training Mean Squared Error.
Final Values:
Rectifier = 0.1816,
Hyperbolic Tangent = 0.1732,
Diversifier = 0.1693.

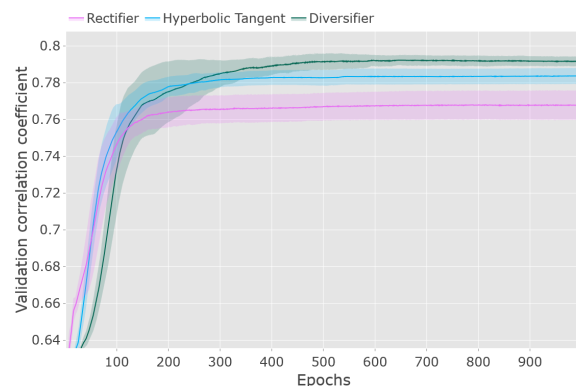


(b) Validation Mean Squared Error.
Final Values:
Rectifier = 0.2055,
Hyperbolic Tangent = 0.1930,
Diversifier = 0.1867.

Figure 7: Tracks of Training and Validation Mean Squared Error, for all three activation functions.



(a) Training Correlation Coefficient.
Final Values:
Rectifier = 0.7982,
Hyperbolic Tangent = 0.8090,
Diversifier = 0.8141.



(b) Validation Correlation Coefficient.
Final Values:
Rectifier = 0.7678,
Hyperbolic Tangent = 0.7837,
Diversifier = 0.7917.

Figure 8: Tracks of Training and Validation Correlation Coefficient values, for all three activation functions.

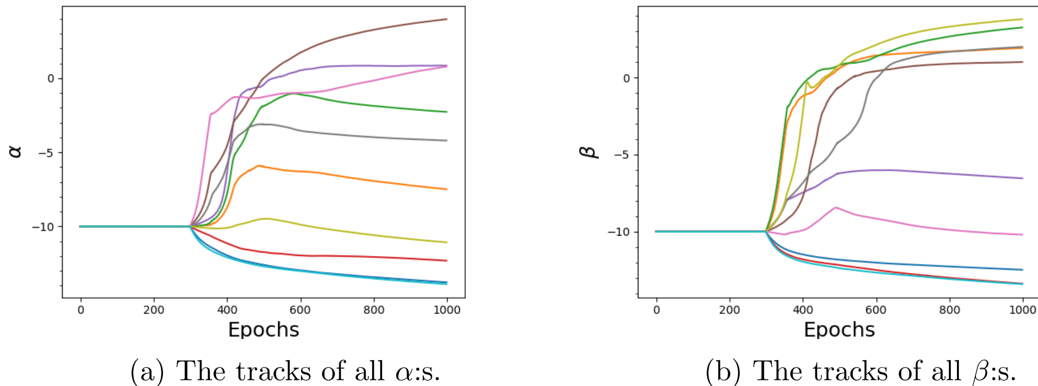
As can be seen in Fig. 7 and Fig. 8, the Diversifier reached a better performance on both training and validation. It was relatively small, but considering that the hyperparameters were not optimized for the Diversifier, it might be improvable.

The rectifier appeared to perform relatively badly on this dataset. This is also the case for the Diversifier if it is initialized with the rectifier or linear shape.

Initializing the Diversifier with the hyperbolic tangent shape managed to not only outperform the rectifier but also the actual hyperbolic tangent. Interestingly, the most common shape that the Diversifiers adopted for the hidden nodes was the rectifier shape, even though initializing them as rectifiers gave a bad performance. In the set of final shapes there were also some hyperbolic tangent and linear ones, as can be seen in Sect. 4.1.2. This might suggest that the Diversifier has a harder time reaching the hyperbolic tangent shape, which some of the nodes might be optimal with for this dataset.

4.1.2 The Diversified Nodes

Below in Fig. 9 is an example of the evolution for the Diversifier parameters α_{ik} and β_{ik}, taken from one of the folds in the previous cross-validation on the regression dataset.



(a) The tracks of all α:s.

(b) The tracks of all β:s.

Figure 9: Tracks of α_{ik} and β_{ik} for the Diversified nodes during training from one fold on the regression dataset. Each color represents an individual node.

As mentioned before, the fixed shape method was applied. Hence, no movement of any α_{ik} and β_{ik} is seen until 300 epochs is reached. After that point, it is clear that they are being trained. Although not all of the α_{ik} and β_{ik} had converged, the shapes did, values above 1 or below -1 does not change the shape significantly due to the sigmoid function that they always are confined within.

Presented in Fig. 10 are the final shapes that the parameters in Fig. 9 produced.

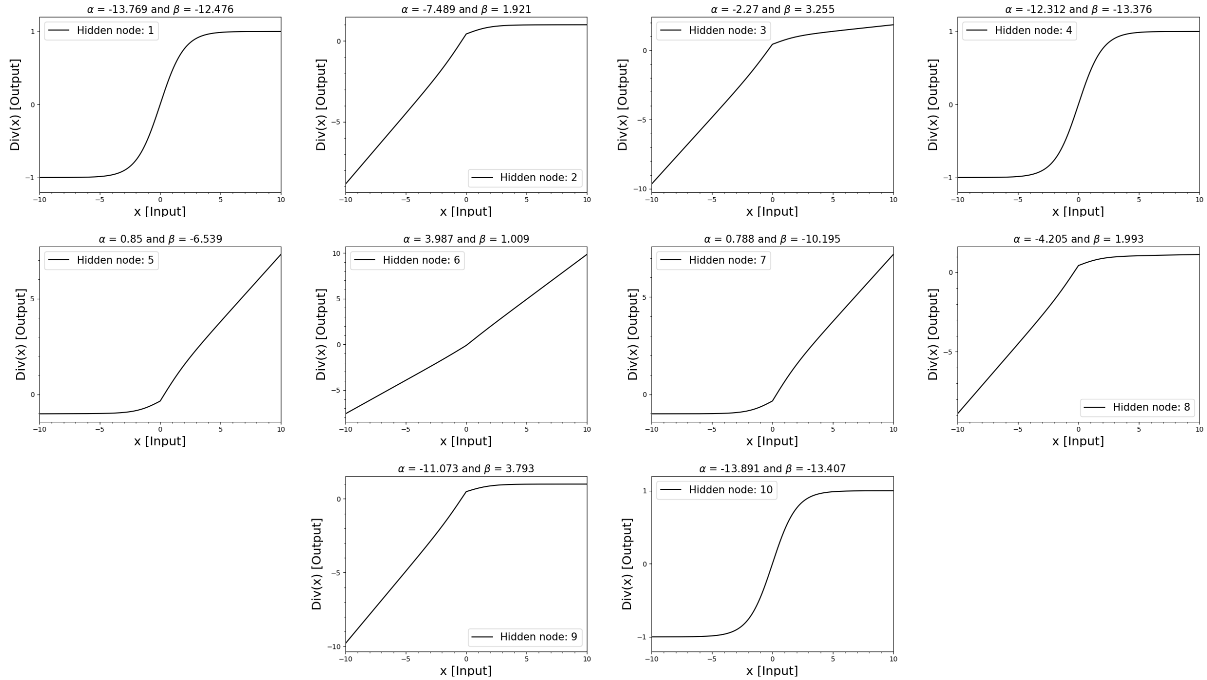


Figure 10: Final shapes of the Diversified nodes for the regression dataset.

4.2 Bone Metastases (Classification)

For this dataset, a network size of 46 input nodes, 10 hidden nodes, and 1 output node was used, together with the sigmoid as the output activation function. The hidden layer had the same activation function for all nodes, either the rectifier, the hyperbolic tangent, or the Diversifier.

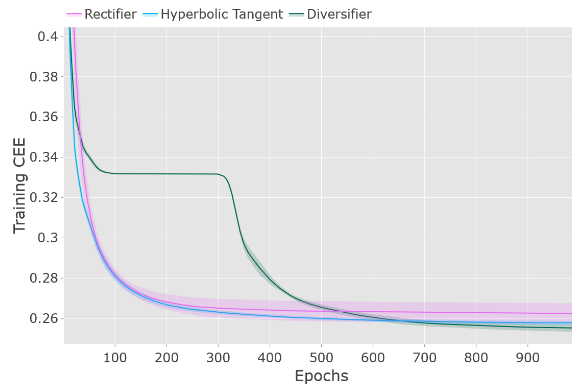
The input data was normalized according to Sect. 1.5.2, and the weights were initialized according to Sect. 1.5.3. The applied optimization method was Adam, and the applied regularization method was the L2-norm.

For the hyperbolic tangent, the optimal hyperparameters were:
Learning rate = 0.02, and L2-parameter = 0.01.

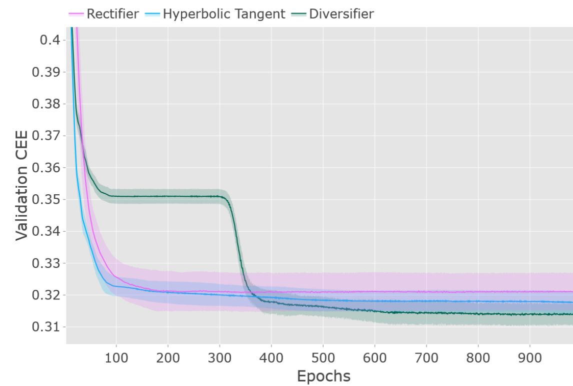
For the rectifier, the optimal hyperparameters were:
Learning rate = 0.01, and L2-parameter = 0.01.

The hyperparameters for the Diversifier was chosen to be the same as for the hyperbolic tangent. Additionally, the new regularization method was applied; the Diversifier was initialized with a shape close to linear, with $\alpha_{ik} = \beta_{ik} = 1$ for all ik, and kept fixed for the first 300 epochs, then the learning rate was applied to the AFPs as well.

4.2.1 Performance Comparison

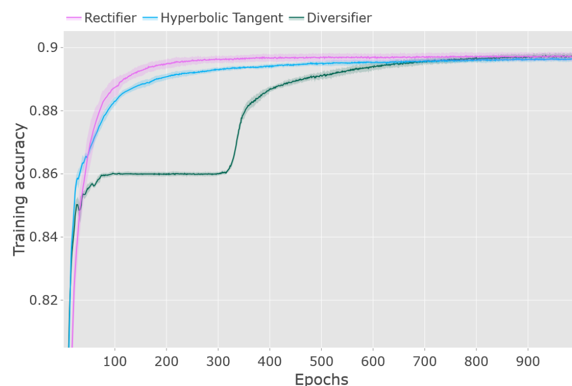


(a) Training Cross Entropy Error.
Final Values:
Rectifier = 0.2625
Hyperbolic Tangent = 0.2578
Diversifier = 0.2553

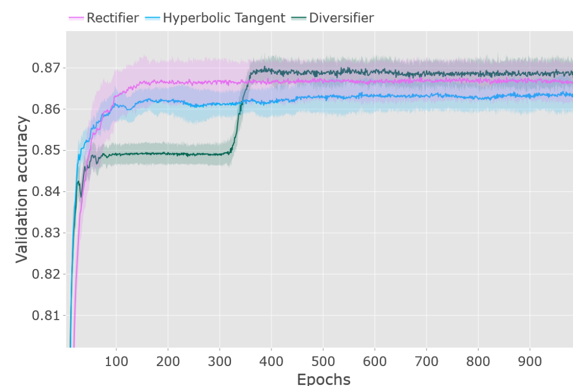


(b) Validation Cross Entropy Error.
Final Values:
Rectifier = 0.3211
Hyperbolic Tangent = 0.3178
Diversifier = 0.3140

Figure 11: Tracks of Training and Validation Cross Entropy Error, for all three activation functions.



(a) Training Accuracy.
Final Values:
Rectifier = 89.73%
Hyperbolic Tangent = 89.63%
Diversifier = 89.71%



(b) Validation Accuracy.
Final Values:
Rectifier = 86.63%
Hyperbolic Tangent = 86.36%
Diversifier = 86.83%

Figure 12: Tracks of Training and Validation Accuracy, for all three activation functions.

As can be seen by the tracks of the Diversifier in Fig. 11 and Fig. 12, the restrictive fixed shape method was applied. The results are similar to the regression problem. The Diver-

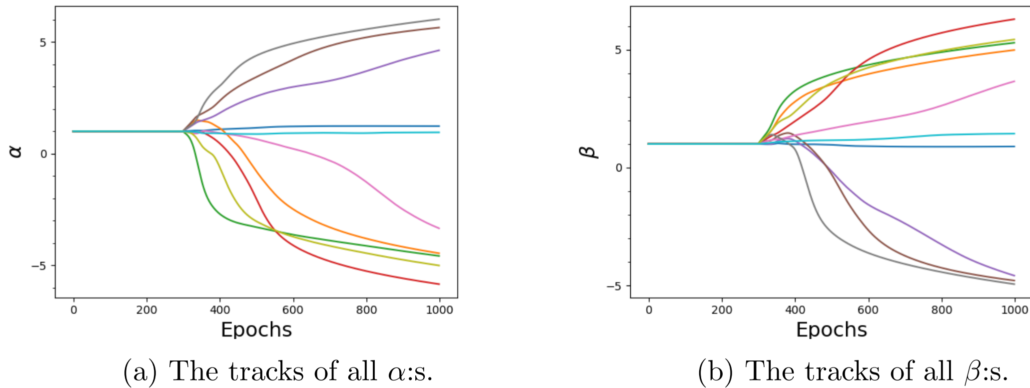
sifier had a slightly better performance on both training and validation, even though the rectifier had the top training accuracy by a small margin it seemed to have plateaued while the Diversifier had not.

Initializing the Diversifier with the hyperbolic tangent shape and using the fixed shape method did not yield a preferable result compared to the rectifier and the hyperbolic tangent. Interestingly, there was no great mixture of shapes for this dataset, which will be shown in Sect. 4.2.2. The dominant shape is still the rectifier and its reflected version, but this time there is no hyperbolic tangent shape. Also, the restrictive fixed shape method seemed to have pressured some shapes to be linear.

Initializing the Diversifier heavily to the Hyperbolic Tangent shape with $\alpha_{ik} = \beta_{ik} = -10$ for all ik, did not seem to affect the final shapes, they were all strongly driven to rectifiers. However, it did affect the final performance negatively. This is likely because of interplay between normal weights and the activation function parameters during training.

4.2.2 The Diversified Nodes

Below in Fig. 13 is an example of the evolution for the Diversifiers parameters α_{ik} and β_{ik} taken from one of the folds in the previous cross-validation on the classification dataset.



(a) The tracks of all α:s. (b) The tracks of all β:s.

Figure 13: Tracks of α_{ik} and β_{ik} for the Diversified nodes during training from one fold on the classification dataset. Each color represents an individual node.

As can be seen Fig. 13, the tracks of the α_{ik} and β_{ik} seem to be close to reflected images of each other. The final shapes of the parameters above are presented in Fig. 14.

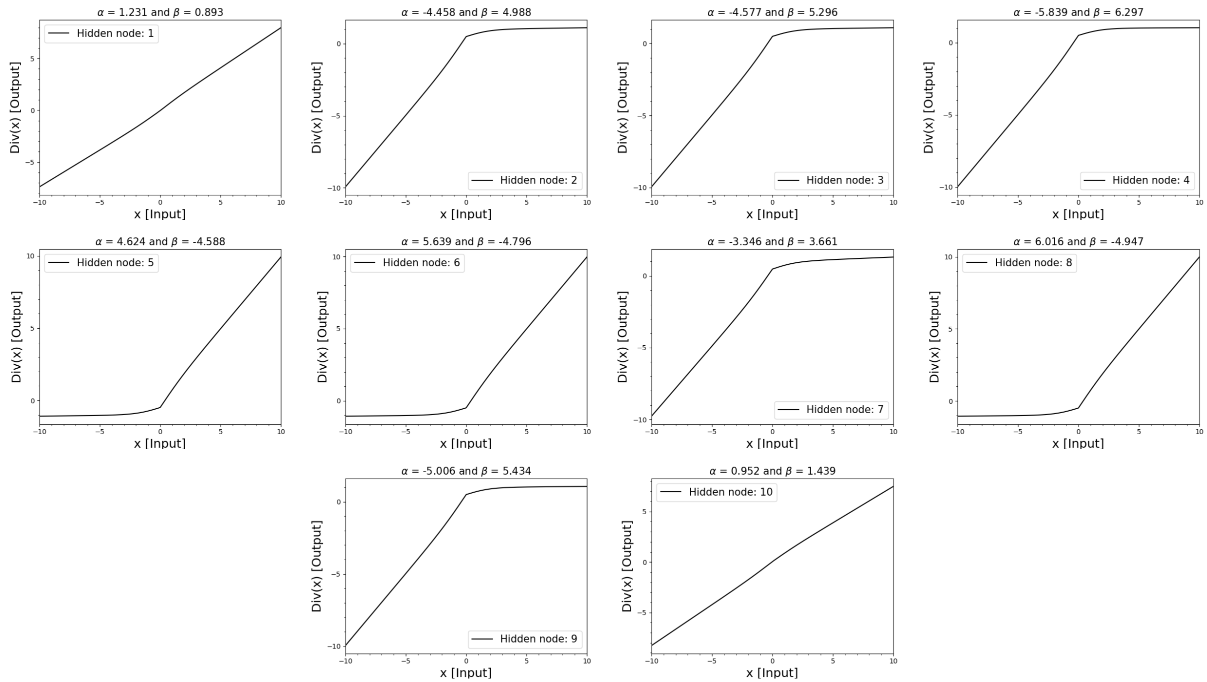
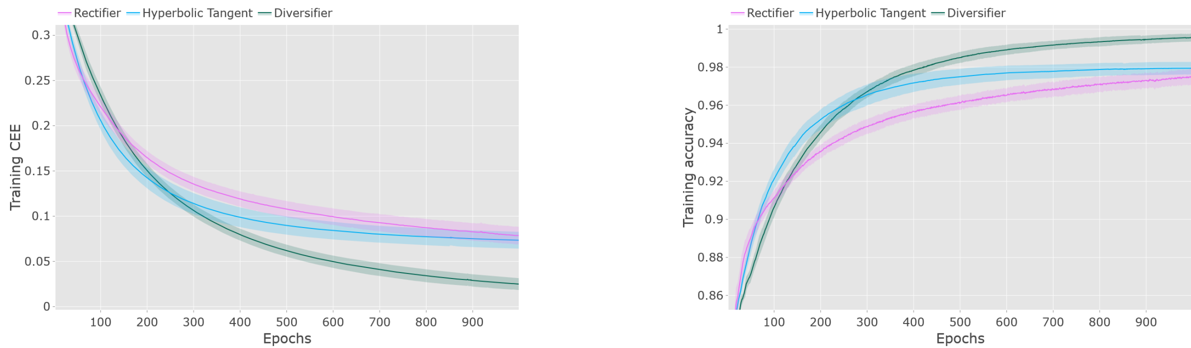


Figure 14: Final shapes of the Diversified nodes on the classification dataset.

Fig. 14 shows that the rectifier shape and its inverted form are dominant for the Diversifier on this classification dataset. And as mentioned before, this would be the end result even if the Diversifier was initialized heavily to the hyperbolic tangent shape. Investigating this further by looking at shapes from other folds reveal that the majority of the Diversified nodes repeatedly end up with a specific shape, while some of them goes between either being the rectifier shape or its reflected version. Interestingly, there seemed to be a consistency in how many of the rectifier shapes are reflected, and a consistency in how many are linear.

4.2.3 Training Depth (Without Regularization)

Another comparison that can be made is how well a network with a fixed size can memorize a dataset, in this case the classification dataset. Memorizing can sometimes be useful, especially if the dataset does not contain a lot of noise. Fig. 15 shows the training results in one graph for each of the three activation functions using the same network size as before on the classification dataset. The learning rate was set on 0.03 for all of them. Additionally, the Diversifier was initialized close to the Hyperbolic Tangent shape with $\alpha_{ik} = \beta_{ik} = -5$ for all ik.



(a) Training Cross Entropy Error.
Final Values:
Rectifier = 0.0787
Hyperbolic Tangent = 0.0735
Diversifier = 0.0249

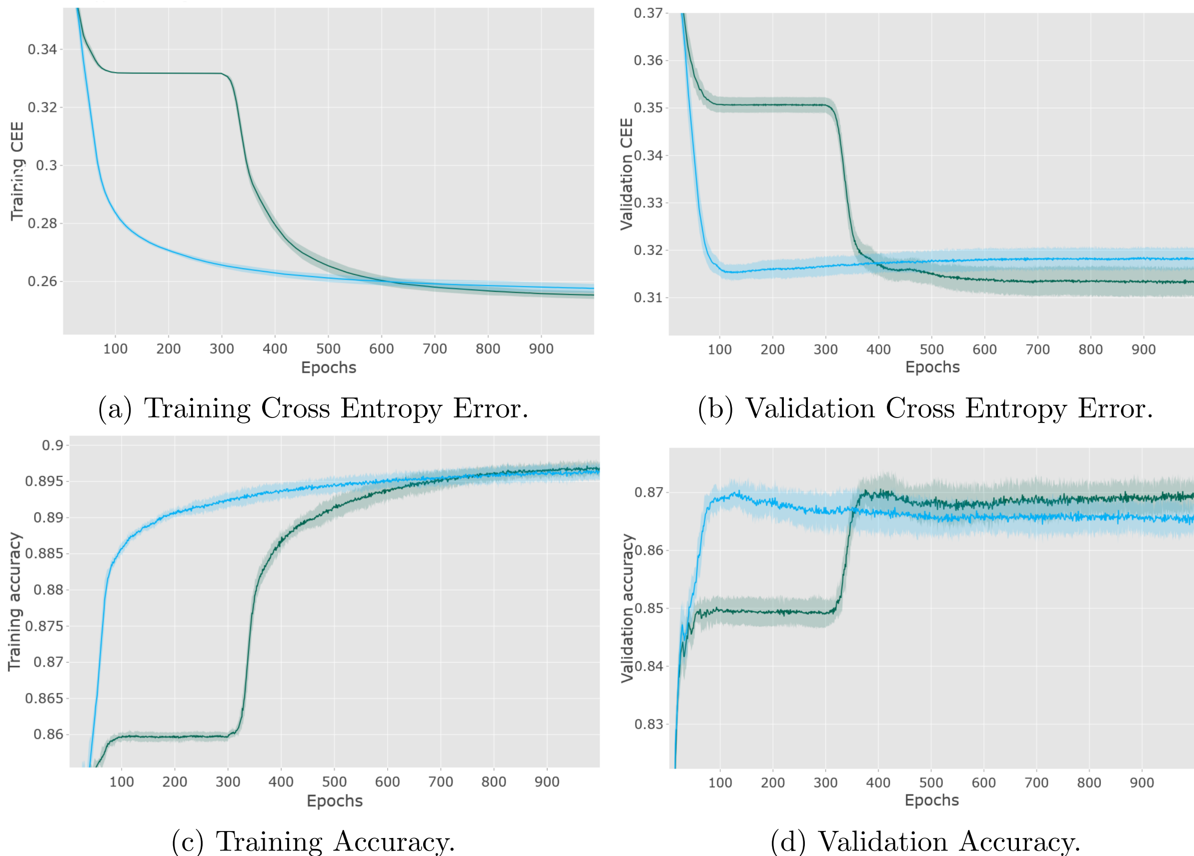
(b) Training Accuracy.
Final Values:
Rectifier = 97.50%
Hyperbolic Tangent = 97.94%
Diversifier = 99.56%

Figure 15: Track of Training Cross Entropy Error and Accuracy without any regularization, for all three activation functions.

In Fig. 15 it is evident that the Diversifier is able to reach a better result than the others. This sizeable difference was present only when the Diversifier was initialized to the hyperbolic tangent shape, otherwise it would perform similarly or just slightly better.

4.3 The Restrictive Fixed Shape Method

In this subsection, the benefit of the new regularization method is displayed. In Fig. 16 is a comparison between the Diversifier with, and without, the new regularization method, taken through a 10 times repeated 10-fold cross validation on the classification dataset. The network size is the same as before. The first 300 epochs were set to be fixed with $\alpha_{ik} = \beta_{ik} = 1$ for all ik for the restricted one. Both were given exactly the same set up, folds, and weight initializations.



(a) Training Cross Entropy Error.

(b) Validation Cross Entropy Error.

(c) Training Accuracy.

(d) Validation Accuracy.

Figure 16: Tracks of Training and Validation Cross Entropy Error and Accuracy for the Diversifier with L2 regularization only (blue track), and the Diversifier with L2 regularization and the restrictive fixed shape method (green track).

There is no doubt from Fig. 16 that the new regularization method had positive effects. Not only did it prevent some overtraining, as can be seen in Fig. 16b and Fig. 16d, but it also manages to reach a better training result, Fig. 16a and Fig. 16c.

Initializing the network with a linear activation function in the hidden layers can be applied without any parametrization of the activation functions, after say 300 epochs one could suddenly switch the activation function to a non-linear and train normally for the rest. This was tested for the hyperbolic tangent and the rectifier. For the hyperbolic tangent one could see a very tiny difference. For the rectifier there was a substantial negative effect. That this method works well for the Diversifier suggests that it is the continuous shape change that is responsible. Initializing the Diversifier heavily to the linear shape, with $\alpha_{ik} = \beta_{ik} = 10$ for all ik, removes the positive effects of the restriction, as it is too restrictive, most of the shapes gets stuck as a linear function.

As mentioned previously, initializing the Diversifier to the hyperbolic tangent shape and keeping it fixed for the first hundreds of epochs also comes with an improvement, though smaller. This suggests that pre-training the weights in this way with either the restrictive or non-restrictive method could generally improve the performance of the Diversifier.

5 Conclusions

A new parametrized activation function called the Diversifier was introduced, with continuous transformations between four shapes. Additionally, a new regularization method with two varieties connected to the Diversifier was presented. Furthermore, a performance comparison between the Diversifier, the rectifier and the hyperbolic tangent was made, through one regression dataset and one classification dataset. The Diversifier with its new regularization method proved to be favourable to the rectifier and the hyperbolic tangent in terms of performance. Without any regularization the difference in performance was substantial. With regularization the difference was smaller but consistent. Small improvements are regarded as significant on more standard datasets such as MNIST [11]. This sparks some interest in applying the Diversifier and its regularization method to deep convolutional neural networks. More work is needed to fully assess the usefulness of the Diversifier and properly understand the regularization method that was introduced.

Acknowledgements

I would like to say thank you to my supervisor Mattias Ohlsson for all the helpful advice, and also Ludvig Siwe for all the interesting discussions leading to further explorations.

References

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[2] Mattias Ohlsson, "Lecture notes on introduction to artificial neural networks and deep learning (fytn14/extq40)," 2018. Sect. 3.3, p. 24-26.

[3] Wikipedia, the free encyclopedia, "Linear regression," 2018. Avaiable at `https://en.wikipedia.org/wiki/File:Linear_regression.svg` (last visited May 9, 2018.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[6] L. Trottier, P. Gigu, B. Chaib-draa, *et al.*, "Parametric exponential linear unit for deep convolutional neural networks," in *Machine Learning and Applications (ICMLA), 2017 16th IEEE International Conference on*, pp. 207–214, IEEE, 2017.

[7] Mattias Ohlsson, "Lecture notes on introduction to artificial neural networks and deep learning (fytn14/extq40)," 2018. Sect. 3.8.2, p. 49.

[8] Mattias Ohlsson, "Lecture notes on introduction to artificial neural networks and deep learning (fytn14/extq40)," 2018. Sect. 3.3.3, p. 21-24.

[9] J. Kalderstam, M. Sadik, L. Edenbrandt, and M. Ohlsson, "Analysis of regional bone scan index measurements for the survival of patients with prostate cancer," *BMC medical imaging*, vol. 14, no. 1, p. 24, 2014.

[10] R. Taylor, "Interpretation of the correlation coefficient: a basic review," *Journal of diagnostic medical sonography*, vol. 6, no. 1, pp. 35–39, 1990.

[11] MNIST database, `http://yann.lecun.com/exdb/mnist/` (last visited May 14, 2018.).