Submitted by:
Lum Ramabaja

Submitted at:
Institute for Machine Learning, JKU Linz;
University of South Bohemia, Ceske Budejovice

Supervisor:
Univ. Prof. Dr. Sepp Hochreiter

Co-Supervisor:
Dipl.-Ing. Thomas Unterthiner, MSc

November 2018

Interconnected Neural Networks for Multi Task Learning

Bachelor Thesis
To obtain the academic degree of
Bachelor of Science
In the Bachelor's Program
Bioinformatics

Johannes Kepler
University Linz
Altenbergerstraße 69
4040 Linz, Austria

References

1. Caruana, R. Multitask Learning. in *Learning to Learn* (eds. Thrun, S. & Pratt, L.) 95–133 (Springer US, 1998).
2. Mayr, A., Klambauer, G., Unterthiner, T. & Hochreiter, S. DeepTox: Toxicity Prediction using Deep Learning. *Front. Environ. Sci. Eng. China* **3,** 80 (2016).
3. Girshick, R. Fast R-CNN. *arXiv [cs.CV]* (2015).
4. Collobert, R. & Weston, J. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. in *Proceedings of the 25th International Conference on Machine Learning* 160–167 (ACM, 2008).
5. Deng, L., Hinton, G. & Kingsbury, B. New types of deep neural network learning for speech recognition and related applications: an overview. in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* 8599–8603 (2013).
6. Ruder, S. An Overview of Multi-Task Learning in Deep Neural Networks. *arXiv [cs.LG]* (2017).
7. Anandkumar, A. & Ge, R. Efficient approaches for escaping higher order saddle points in non-convex optimization. *arXiv [cs.LG]* (2016).
8. Hochreiter, S. Untersuchungen zu dynamischen neuronalen Netzen.
9. Mitchell, T. M. The Need for Biases in Learning Generalizations. *citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5466* (1980).
10. Unterthiner, T., Mayr, A., Klambauer, G. & Hochreiter, S. Toxicity Prediction using Deep Learning. *arXiv [stat.ML]* (2015).

Contents

Abstract

Multitask learning is a subfield of machine learning in which many tasks are learned in parallel. The tasks usually share the same parameters for predicting. The information shared that way, can help each task improve its performance, while requiring less task specific data and less time to train. Multitask learning has shown promising results, from drug discovery, computer vision, to natural language processing.

In this thesis, I am going to introduce the interconnected neural network architecture for multitask learning. This architecture differs from the way a neural network with hard parameter sharing works, in that each task has its own subnetwork, which connects to the subnetwork of other tasks via connector weights. During forward propagation, information flows through the whole interconnected network, like in every feedforward neural network. During backward propagation, each error can flow only through its subnetwork and connector weights. That way, information cannot flow through other subnetworks. The interconnected model is then compared to a multitask model with hard parameter sharing, and to single task learning models. At the end, I will discuss about an interesting finding, that might make some understandings about task relatedness questionable.

1. Introduction

1.1 Overview

Multitask learning (MTL) is a transfer learning technique used to improve the generalization of a model. This is achieved by introducing the inductive bias of other tasks during the learning process. In MTL, the tasks share the same parameters and optimize more than one loss function. The shared parameters of the model can help each task improve its performance, while requiring less task specific data and less time to train [1]. Even though MTL is a technique that works for various kinds of models, in this thesis I am only going to focus on the applications and problems of MTL in deep neural networks.

MTL has shown promising results in many areas, from drug discovery [2], computer vision [3], to natural language processing [4] and speech recognition [5]. Regardless of its success however, most problems in machine learning are still formulated as single task problems (either in the form of binary labels, multi labels, or as single task regressions). One could argue that this will gradually change, as advances in (for example) autonomous vehicles, will help curate more multitask data sets. There are however limitations current MTL techniques face, that will make a switch from single task learning approaches, to a multitask learning approach relatively difficult.

Even though MTL has shown promising results, it suffers from a similar problem as the "pre-neural network" solutions in computer vision. Older computer vision techniques required manual curation of features, which was time consuming, and unique for every case. Similar in MTL, it is required to have predefined knowledge of the tasks, i.e. you have to figure out which tasks work well together, which is unique for every multitask problem.

Furthermore, conventional MTL approaches treat all tasks equally, without any discrimination between them. For multitask problems with a diverse set of tasks, using this kind of approach, i.e. forcing information sharing between the whole set of tasks, might result in ignoring the relatedness of subsets of tasks. On the other hand, many other techniques that don't follow this approach, assume some predefined knowledge of the tasks [6].

To address these challenges, I propose a new neural network structure for multitask learning problems, the interconnected neural network (ICNN). The assumption behind the ICNN idea was to come up with a design that does not require any prior knowledge about the set of tasks to be trained jointly. The idea was to create a network that regulates itself during training, so that information continuous flowing between related tasks, but not between unrelated tasks. The goal was to create an architecture that requires less manual task preparation for multitask problems.
I will discuss the issues of current MTL approaches in section 3, and introduce the concept of an interconnected neural network in section 4. In section 1.2, I will explain the basic idea of a feedforward neural network. In section six we will look at the performance of an interconnected neural network, and of other models. In section 7, I will discuss about potential pitfalls for interconnected neural networks, as well as discuss about further ideas for the architecture. At section 6.3, I will discuss about an interesting finding that might make some of our understandings about task relatedness questionable.

1.2 Feedforward neural networks

Artificial neural networks are a family of machine learning models, that can learn things without explicitly programming any rules. Such algorithms learn by inductive learning, i.e. they learn from past observations. The most quintessential neural network model, the feedforward neural network, can be thought of as a directed acyclic graph that maps inputs to an output. The nodes of the network are referred to as neurons, or units, whereas the edges as weights, or coefficients. Each neuron also has a bias, a term that specifies how easily the neuron fires. In a neural network, all neurons are organized in the form of stacked layers, where each unit of one layer is connected via weights to each unit of the next layer. Information flows from the input layer, to the intermediate layers (also known as hidden layers), up to the output layer; undergoing matrix transformations at each step (figure 1). To understand how information in a neural network is processed, let's first look at how a single neuron works:
In a nutshell, a neuron receives inputs $\{x_1, x_2, ..., x_n\}$, performs some computations, and produces a single output y (figure 1). To get to that y, each input x_i gets multiplied by a

corresponding weight w_i. This multiplication either increases, or decreases the input signal. Afterwards the weighted inputs are summed and a bias term b is added to it. The bias term determines how easily the neuron fires, it moves the firing threshold either up, or down. The total sum is then put through an activation function f (which is usually a nonlinear function, such as a sigmoid function, or a rectified linear function), which gives the final output y. The described procedure can be expressed as: $y = f(\sum_{i=0}^{n} w_i \cdot x_i + b)$, or in vector notation as $y = f(w \cdot x + b)$.

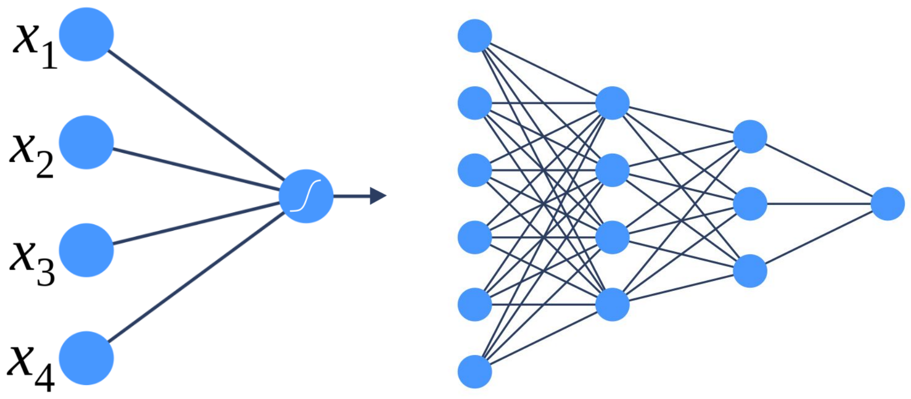

Figure 1. An illustration of a neuron on the left, and an illustration of a feedforward neural network on the right.

The output of the neuron describes how the neuron “reacts” to a specific input. Our goal then is to tweak the weights / bias, so that the neuron reacts for specific inputs in a desired way.
In the case of the neural network, the first layer, the input layer, represents the actual training data in the form of a matrix (or vector, in the case of online learning). The inputs are processed in every neuron of the first hidden layer, and passed on as new inputs to the next layer. This process can be written as : $a_t = f(W_t^T \cdot a_{t-1} + b_t)$ where a_t is the output of the current layer, and a_{t-1} is the output of the previous layer. This is then repeated until the output layer is reached. We call this process *forward propagation.* The output layer represents the predicted targets for the given inputs. As one can imagine, the predicted targets will be completely wrong at the beginning of the training procedure. We then adjust the parameters by using an algorithm called *backpropagation.* After we repeat the forward pass and backward pass for many iterations (epochs), the network eventually learns to correctly map $A_i \rightarrow B_i$ (where A_i represents an input,

and B_i its target), even for unseen inputs. The ability of a model to correctly map unseen observations is also known as *generalization*.

1.3 Backpropagation and the gradient descent algorithm

Every feedforward neural network will initially have horribly wrong predictions. This makes sense, as nothing was "learned" yet. In machine learning, what is meant by "learning", is the minimization (or sometimes maximization) of a loss function. A loss function is simply a convex function, that allows to quantify the error of a model. In a nutshell, all what learning is, is the ability to be less wrong over time. This is achieved by optimizing the model's parameters (weights and biases) while training (while trying to minimize a loss function).
Estimating the right parameters however is not that straight forward. Most real-world problems are in the form of multivariate data. Finding the optimal parameters analytically for such problems, would be computationally enormously expensive. That is why neural networks and many other models, use the gradient descent algorithm for optimizing its parameters. The idea behind gradient descent is to iteratively update the given parameters in a way that minimizes the loss function. To better understand how gradient descent finds a minimum, imagine a ball rolling down a hill. In this analogy, the loss function represents the hill, whereas the ball represents the current values for the parameters θ . We want the ball to roll down the hill, through the steepest direction, until it lands at a valley (at a local / global minimum).

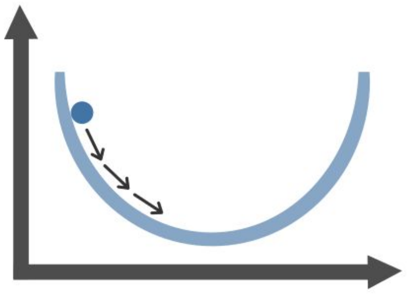

Figure 2. An oversimplified illustration of the gradient descent algorithm.

To do this, an update is repeatedly performed to the parameters:

$\theta_i := \theta_i - \alpha\frac{\partial C(\theta)}{\partial \theta_i}$

Where α is the *learning rate* and $\frac{\partial C(\theta)}{\partial \theta_i}$ is the partial derivative of the loss function, with respect to θ_i (where in the case of neural networks, θ_i is either the weight, or bias term). Following the analogy, the partial derivative tells us the steepness of the hill at a specific point (aka the gradient), whereas the learning rate represents the "step size" for the ball. When multiplied together, it tells us if the ball should "climb up" the hill, or go further down. Eventually after some iterations, we hope to land on a minimum.

One might think that such an algorithm would immediately get stuck in a local minima, and not work well. Unintuitively, that is not exactly the case when dealing with high dimensional functions. In fact, it is an NP-hard problem to even find a local minima [7]. Logically, the more dimensions we have, the lower the chances get that a critical point actually represents a local minima. Chances are much higher to find yourself in a saddle point, as shown in figure 3.

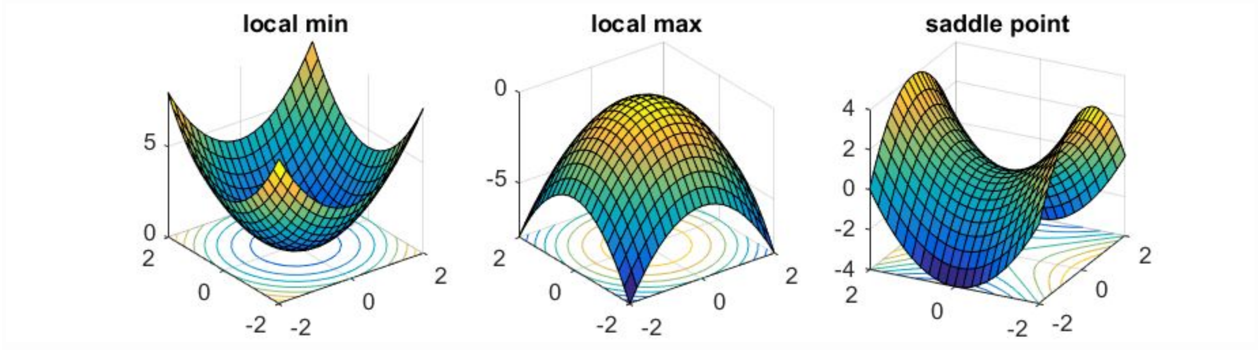

Figure 3. A depiction of a local minima, local maxima, and a saddle point. Image taken from "Rong Ge - Escaping from Saddle Points"

Regardless of saddle points, gradient descent is a surprisingly effective algorithm for parameter optimization. The generalization of gradient descent algorithm to multi-layered feedforward networks, is known as the backpropagation algorithm, and it is the backbone of most of today's deep neural networks.

The main idea of backpropagation, is to calculate the error of the model at the output, and by using the chain rule, iteratively compute the gradients for each layer. To better understand this, let's take as an example a neural network with only one hidden layer, and compute the gradients for each step:

We will use for this example the cross entropy loss function:

$$L =- \sum_{i=0}^{n}(y_i \cdot log(a_i) + (1 - y_i) \cdot log(1 - a_i))$$

And the sigmoid function as activation function for the units in the neural network:

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

Where i tells us in which layer we are, and z represents the weighted sum

$$z_i \ = \ W_i^T \cdot a_{i-1} + b_i \, .$$

The error δ_2 of the output layer a_2 can be computed as:

$$\delta_2 \ = \ \frac{\partial L}{\partial z_2} \ = \ \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \ = \ \frac{a_2 - y}{a_2 \cdot (1-a_2)} \cdot a_2 \cdot (1 - a_2) \ = \ a_2 - y$$

Because $\frac{\partial L}{\partial a}$ and $\frac{\partial a}{\partial z}$ for this case equals to:

$$\frac{\partial L}{\partial a} =- \frac{y}{a} + \frac{1-y}{1-a} = \frac{a-y}{a \cdot (1-a)}$$

and

$$\frac{\partial a}{\partial z} = e^{-z} \cdot (1 + e^{-z})^{-2} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - (\frac{1}{1+e^{-z}})^2 = a - a^2 = a \cdot (1 - a).$$

Having this information, we can now compute $\frac{\partial L}{\partial w_2}$ and $\frac{\partial L}{\partial b_2}$:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2} = \frac{a_2 - y}{a_2 \cdot (1-a_2)} \cdot a_2 \cdot (1 - a_2) \cdot a_1 = (a_2 - y) \cdot a_1 = \delta_2 \cdot a_1 \, ,$$

because $\frac{\partial z_2}{\partial w_2} = a_1$, and : $\frac{\partial L}{\partial b_2} = \delta_2$,

and $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z} = \delta_2$

The calculated gradients can now be used to update the parameters of the last layer:

$$w_2 := w_2 - \alpha \frac{\partial L}{\partial w_2} \text{ and } b_2 := b_2 - \alpha \frac{\partial L}{\partial b_2} \, .$$

To compute the gradients of the layer below, the same logic can be applied:

$$\delta_1 = \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} = \delta_2 \cdot w_2 \cdot a_1 \cdot (1 - a_1)$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} = \delta_1 \cdot a_0$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} = \delta_1$$

And the update:

$$w_1 := w_1 - \alpha \frac{\partial L}{\partial w_1} \text{ and } b_1 := b_1 - \alpha \frac{\partial L}{\partial b_1} \, .$$

The more layers there are, the longer the “chain” for error calculation gets. One can see an obvious problem that arises when a network has many layers: parameters at the bottom of the network will update slower than the ones on the top. The continuous multiplication of gradients will cause the lower layer parameters to have smaller and smaller gradients, and thus very small parameter updates. This problem is also known as the vanishing gradient problem [8]. The opposite of the vanishing gradient, is the so called exploding gradient problem. It occurs when the gradients are too big, resulting in ever larger gradients down the network. Even though these two problems seem to ruin the whole concept of deep neural networks, they can be overcome with various regularization techniques.

Now that we understand how a neural network learns, we can go to the next step - see how neural networks learn with N loss functions.

2 Multi task learning

Until now we showed how neural networks can learn by optimizing a loss function. This type of learning is also called single task learning. It is also possible to train neural networks (or other kinds of models) by optimizing more than one loss function. Training models with multiple loss functions in parallel, is also known as Multitask learning (MTL).

MTL is a transfer learning technique used to improve the generalization of a model by training tasks in parallel while sharing the models' parameters. The shared parameters of the model can help each task improve its performance, while requiring less task specific data and less time to train [1].

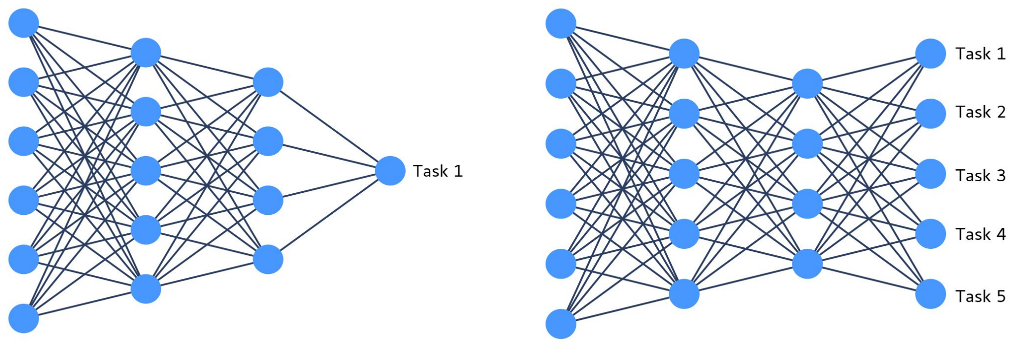

Figure 4. On the left we have a single task architecture for a neural network, and on the right we have a multitask architecture for a neural network.

Figure 4. Shows a structural comparison of a single task learning (STL) neural network, and a MTL neural network. In the case of the STL approach, backpropagation is performed the same way as explained in section 1.3 - The error of the task is calculated, and then propagated backwards to correct the weights and biases. In The case of the MTL approach however, backpropagation is done in parallel - the errors of each task are summed together, and then propagated backwards. We say that tasks are trained jointly. This approach can allow for faster training with fewer observations. It also can lead to better performances, than learning the task separately [1].

2.1 Intuition

At first it sounds surprising that such an approach works at all, let alone that it can increase the performance of individual tasks. One can imagine that the inductive bias introduced by other tasks, can help each task learn better. For a model to be able to make assumptions on unseen data, it is necessary for it to have biases for choosing one generalization, over another; in fact, a model without such biases cannot exist [9]. In the case of MTL, the training signals of each task serve as a bias to every other task. This causes the model to learn the parameters needed to explain more than one task. The model becomes biased, in the sense that it prefers internal representations that do well on many tasks. This can also be viewed as a regularization technique, where generalization is enforced by forcing the model to work well on many related tasks. In contrast, other regularization techniques prevent overfitting by punishing complexity in general. As a result, the multitask bias causes the model to generalize better, than a model with a single source of inductive bias.
To better understand how the inductive bias effects MTL, we must understand its underlying mechanisms, most of which were proposed by [1].

2.1.1 Statistical data amplification

The extra information introduced by the training signals of other tasks in MTL, can be used to ignore the noise pattern of individual tasks. That way a model can learn a more generalized representation, by ignoring task specific noise. We can say that MTL increases the sample size used for training the model. Learning only a single task can result in overfitting, while learning two tasks in parallel can result in better generalization.

2.1.2 Attribute selection

If we have a single task A and a single hidden layer H, and a high dimensional input, it can be sometimes difficult to distinguish the relevant features for H. If we train however two tasks A and B in parallel, it will distinguish relevant features for H easier, because of data amplification.

2.1.3 Eavesdropping

Let's say we have features F, And two tasks A and B. And let's assume that A finds it difficult to learn F, while B finds it easy. This can occur for various reasons (other features could prevent A from learning F, etc.). If task A and B are trained together however, task A can learn F through task B.

2.1.4 Representation Bias

With MTL, a model is biased to form hidden representations that are prefered by other tasks. [1] shows this behaviour beautifully with an experiment: Let's assume there are two local minima A and B a for the loss function of task T. Learning another task T0 can also lead to two minima A and C. If there is no preference for the two tasks (neither T or T0 prefer to land on one of the minima), if T and T0 are trained together, they usually fall into A. This shows that in MTL tasks prefer hidden representations useful to both tasks. The opposite experiment is also true: If task T prefers A more than B, and task T0 has no preference between B and C, T0 will usually fall into C, and T will fall into A.

2.2 The limitations of multi task learning

MTL has shown promising results in many areas, from drug discovery [2], computer vision [3], to natural language processing [4] and speech recognition [5]. There are however limitations to MTL that need to be addressed, for it to become a more widespread (relative to single task learning):

1) It is not that easy to use MTL “out of the box”. Training a model with unrelated tasks can lead to suboptimal performances, or even worse performances. One task can override the weights that were useful to the other task. One has to figure out which tasks learn well together, and which ones do not. To make things more complicated, even very unrelated tasks can sometimes help improve generalization of other tasks [1]. Because of this, relying solely on correlation matrices to figure out task relatedness, could lead to a suboptimal selection of tasks to be trained together. Even though a lot of research has been put into this problem [6], most techniques rely on predefined knowledge about the tasks.
2) Another problem of MTL is that of weight overriding. Two tasks might learn well together, but might reach optimal performance at different points during training. In such a scenario, early stopping can actually hurt performance, as one task might override the weights that were useful to the other task [1].

Having information sharing in models can be of great interest for many fields. If we look for example at computer vision, we can see that many objects share similarities with one-another. Sharing that kind of information could improve performance significantly. Conventional MTL approaches however treat all tasks equally, without any discrimination between them. For multitask problems with a diverse set of tasks, using this kind of approach, i.e. forcing information sharing between the whole set of tasks, might result in ignoring the relatedness of subsets of tasks. Many other techniques that don't follow this approach, assume some predefined knowledge of the tasks [6].

To address these challenges, I propose a new neural network structure for multitask learning problems, the interconnected neural network (ICNN). The idea for the ICNN was to have a model that does not require any predefined knowledge of task relatedness. The assumption made is that the ICNN architecture can regulate itself during training, so that information continuous flowing between related tasks, but not between unrelated tasks.

3 Interconnected neural networks

The interconnected neural network is a type of artificial neural network for multitask problems. In this network, a set of tasks can be trained jointly, with the hypothesis that there is no need for prior knowledge of the task relatedness. The architecture is designed in a way, so that subsets of related tasks can be isolated from unrelated subsets of tasks. The network then regulates itself during training, so that information continuous flowing between related tasks, but not between unrelated tasks. This is achieved by doing backward propagation through the network slightly differently than in conventional MTL approaches.

Each task in an ICNN has its own subnetwork. Each subnetwork is positioned in parallel to each-other. We can define a subnetwork as a neural network with two kinds of weights: The root weights (W) - the weights that connect only to layers of the same subnetwork; and the connector weights (CW) - the weights that connect one layer of the subnetwork, with another layer of another subnetwork. Similar to usual neural networks, each layer connects only to the layers above them, may that be layers of the same subnetwork, or those of other subnetworks. This means that each layer of each subnetwork is connected to all other subnetworks.

There is in fact no technical difference between connector weights, and the usual weights; the only real difference occurs during backpropagation. During forward propagation the same input data is fed to each subnetwork. Each subnetwork then computes N weighted sums, one with its root weights, and $N-1$ weighted sums with its connector weights - each connector weight matrix connects the layer of one subnetwork to the next layer of the other subnetworks. This way the information of other subnetworks can flow to each other subnetwork. This is done for each separate subnetwork. As a result, the output of each subnetwork layer will be:

$$a_{j,i} = f(\sum_{i=0}^{N} z_{j-1,i})$$

Where $a_{j,i}$ is the j-th layer of the i-th subnetwork, $f()$ is an activation function, N is the total number of tasks, and $z_{j-1,i}$ is the weighted sum of the previous layer, at the i-th subnetwork. There are probably better ways of summing the signal of each subnetwork, but we will continue using this approach in this paper.

The process is simpler than it might look at first glance, to recap: the information of each subnetwork “flows” through the other subnetworks during forward propagation, the information propagates as usual. Backpropagation for an ICNN on the other hand is done slightly differently, than in other MTL approaches:

The errors of each task are not summed together. In most MTL approaches, the errors of each task get summed, and propagated backwards. That is how the inductive bias of other tasks is introduced. In the interconnected neural network on the other hand, the errors are not summed. The error of each task is only used to update its root weights and connector weights. The error of one subnetwork, cannot propagate through other subnetworks. In other words, once the connector weights are updated, the computed error is not used during the backward step for the other subnetworks. For better visualization, imagine the backpropagation step of each subnetwork like the roots of a tree, growing down and branching on its way. It’s important to note that each subnetwork has its own connector weights, thus the weight update of one subnetwork does not interfere with the weight update of the other subnetworks.

3.1 Hypothesis and intuition behind the ICNN

One might wonder why the interconnected neural network is structured the way it is. To better understand this, let's first look at the shortcomings of conventional MTL approaches:
Most conventional MTL approaches assume one of two things:

1) They either assume that tasks are of equal importance, without any discrimination between the shared information;
2) Or they assume prior knowledge, i.e. an architecture built in a specific way, that assumes knowledge about task relatedness.

In both cases, for MTL to succeed, it is necessary to figure out what set of tasks to train jointly. Often times this is not possible, and manual task selection has to be done. This can be both computationally expensive and time consuming. Many techniques try to solve this problem by following the second mentioned approach, where models are built specifically for a set of tasks [6], but this again assumes prior knowledge of the tasks. Furthermore the model architectures used for a set of tasks, might not be transferable to other sets of tasks.
On the other hand, for multitask problems with a diverse set of tasks, using the first mentioned approach, i.e. forcing information sharing between the whole set of tasks, might result in ignoring the relatedness of subsets of tasks. Some tasks might learn well with some tasks, but not that well with other tasks.
The initially proposed hypothesis was that these problems can be overcome by using an architecture like that of an interconnected neural network. There were two sub-hypotheses I am making by proposing the ICNN architecture, which will be tested during the Experiments section:

1) The proposed ICNN architecture performs as good as current MTL approaches with hard parameter sharing, when the set of tasks are related to one another.
2) The ICNN architecture performs better than MTL approaches with hard parameter sharing, when trained jointly on a set of tasks that has subsets of related and unrelated tasks.

Near the end of writing this thesis, I understood that the assumptions for the second hypothesis were wrong. There is no reason for the connector weights to regulate themselves depending on task relatedness. Because of this, we're just going to test in this thesis if the architecture learns well. An interesting addition to the ICNN idea, would be to at some kind of regularization at the connector weights, that additionally negatively weights the input of units that constantly contribute to a high error. That way, the second hypothesis might hold true. This could be further studied in another work.

4.0 Experiments

To test the idea of the interconnected neural network, the "Tox21 Data Challenge" dataset was used. The Tox21 challenge (also known as the "Toxicology in the 21st Century" initiative) was launched by the government agencies NIH, EPA and FDA, with the goal to predict the toxicity levels of various chemical compounds. Deep learning techniques showed tremendous success for this particular task [10], this is also why the Tox21 dataset was chosen for the experiments.
For simplicity, we will assume that each layer of an inter-layer will have the same number of units during the experiment. Note, this must not be the case. Different tasks could have layers of different sizes, but for simplicity, each layer of an inter-layer will have the same number of units.
To see if the ICNN architecture works, we are going to test it under two scenarios:

1) Four networks are used. One interconnected neural network, one normal neural network with hard parameter sharing for MTL (let's call it network M), and two sets of N single task feedforward neural networks (where N is the number of tasks), let's call them network A and B. Network A will have the same total number of units as the network M, and network B will have in total the same number of parameters, as there are root parameters in the ICNN model. The ICNN model on the other hand will have more parameters than network A and M.

The AUC (area under the curve) is used as a performance criterion. To see if the ICNN architecture works, we would expect the model to work better than network B (because of the connector weights), and since the tasks of Tox21 are already highly correlated, perform either similarly, or slightly better than network M, or A (because of the ICNN having more parameters).

2) In the second scenario, we have the same setup as before, but we introduce four new tasks. Those tasks (let's call them set B) are then attached to the to the original Tox21 tasks (set A). The labels of set B are randomly initialized, in a way so that the tasks inside B are correlated with other tasks inside B, but uncorrelated with the tasks of A.
   The hypothesis is that the performance of the ICNN will on average not decrease when introducing unrelated tasks (when compared to its performance in scenario 1).

To see if the assumptions about connector weights converging to small values for unrelated tasks really holds true, we will design a heatmap between the connector weights of all the tasks. If the absolute sum of connector weights between the tasks of set A (or B) are significantly higher than the absolute sum between the connector weights of A and B together, we can deduce that the assumption was correct.

5 Results

5.1 Results on the first scenario

Tasks	ICNN MTL AUC	NN MTL AUC	NN STL AUC	NN2 STL AUC
NR.AhR	0.81	0.77	0.78	0.65
NR.AR	0.71	0.66	0.64	0.46

NR.AR.LBD	0.52	0.54	0.56	0.47
NR.Aromatase	0.71	0.69	0.71	0.59
NR.ER	0.71	0.75	0.74	068
NR.ER.LBD	0.63	0.61	0.61	0.53
NR.PPAR.gamma	0.71	0.62	0.59	0.49
SR.ARE	0.7	0.67	0.7	0.64
SR.ATAD5	0.72	0.62	0.71	0.43
SR.HSE	0.63	0.55	0.57	0.49
SR.MMP	0.8	0.76	0.77	0.67
SR.p53	0.75	0.69	0.73	0.48

Table 1. The areas under the curve (AUC) for each task in the Tox21 dataset, for four neural network architectures. The results for the "ICNN MTL AUC" column were produced by using an interconnected neural network. The ones for the "NN MTL AUC" used a feedforward neural network with hard parameter sharing. Whereas "NN STL AUC" and "NN2 STL AUC" both used a set of N single task learning feedforward neural networks, i.e. no information sharing between the tasks. Each architecture was trained three times, and the AUCs for each task were averaged. All the models used the same hyperparameters, however the number of units for the ICNN is not the same to that of the other models.

It is important to note that the above table is *not* proof that the ICNN architecture works better than the other models. One would have to search through various combinations of hyperparameters and parameters to be able to make such a conclusion, which was not done in this thesis. The purpose of the table is for the reader to be able to compare the performance of each model with the performances of the models in *Table 2*. The model for the "NN2 STL AUC" column, has the same number of *root* parameters as the ICNN model, showing that information sharing between the tasks made a difference in performance. Such a fair comparison on the other

hand, was not possible between the ICNN architecture, and the models trained for the “NN MTL AUC” and “NN STL AUC” columns, as one would have to train many models, with many hyperparameter / parameter combinations to be able to make an unbiased statement.
A single inter-layer in an ICNN requires $n \cdot m \cdot N^2$ parameters (where n is the number of units for the previous inter-layer, m is the number of units for the next inter-layer, and N is the number of tasks). Because of that, the space complexity for an ICNN increases drastically for each added task to the task set. Thus if we have a layer with dimension 800x500 as an example, and 12 tasks, the inter-layer for an ICNN would require $800 \cdot 500 \cdot 12^2 = 57600000$ parameters. Using that many parameters for a single layer for a normal MTL or STL feedforward network seems unrealistic, as one would probably use a different architectural configuration (deeper networks, instead of wider), making it an unfair comparison. Because of this reason (and because of the computational cost of training so many large networks), I decided to compare the architectures in terms of performance difference when introducing unrelated tasks, as described in Table 2, instead of comparing the models to one another.

5.2 Results on the second scenario

For table 2, four new tasks with randomly sampled labels were introduced. The labels for the four tasks were drawn, so that they would be correlated to one-another, but uncorrelated to the 12 other tasks of the Tox21 dataset.
The next table (table 2) shows the results on the four models with 33% more parameters than before (as there were 4 new tasks added).

Tasks	ICNN MTL AUC	NN MTL AUC	NN STL AUC	NN2 STL AUC
NR.AhR	0.79	0.78	0.78	0.65
NR.AR	0.7	0.69	0.64	0.46

NR.AR.LBD	0.62	0.59	0.56	0.47
NR.Aromatase	0.72	0.65	0.71	0.59
NR.ER	0.72	0.74	0.74	068
NR.ER.LBD	0.63	0.6	0.61	0.53
NR.PPAR.gamma	0.69	0.66	0.59	0.49
SR.ARE	0.7	0.68	0.7	0.64
SR.ATAD5	0.71	0.72	0.71	0.43
SR.HSE	0.65	0.6	0.57	0.49
SR.MMP	0.79	0.76	0.77	0.67
SR.p53	0.71	0.73	0.73	0.48

Table 2. The areas under the curve (AUC) for each task in the Tox21 dataset, for the four neural network architectures that were used for table 1. as well. Four new random targets were introduced, to see how the performance of each model changes. The performance of the STL networks is of course unchanged. For the MTL networks, adding noise actually made the performance of some tasks better.

As we can see, adding noise did not significantly change the performance of both the MTL neural networks. This is probably because both networks received a 33% increase in the number of parameters. This increase of parameters was performed to the neural network with hard parameter sharing, so that the comparison to the ICNN remains fair (as the ICNNs number of parameters by default increases with every additional task).

In these two experiments, I was not able to prove if an ICNN works better than a neural network with hard parameter sharing (because of time constraints on performing a high number of random searches), but I've shown that the architectural idea of an ICNN does work.

5.3 Observations during the experiments

Since the performance of the ICNN showed to be quite good, I assumed that the heatmap of the absolute sum of connector weights of an interconnected neural network will result in a visualization similar to that of a correlation matrix of the tasks. That however was not the case. The heatmap turns out to look quite random.

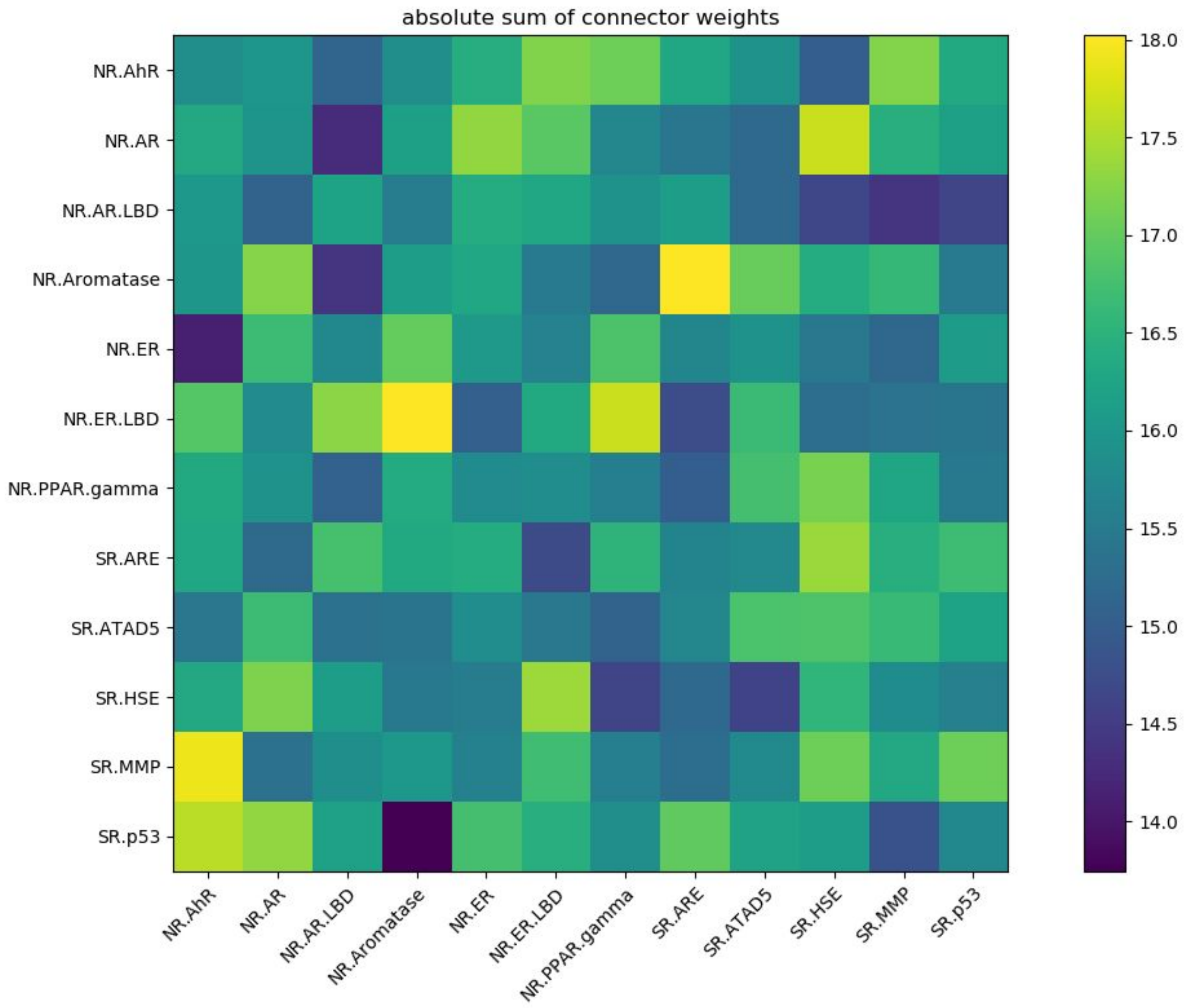

Figure 5. A heatmap of the absolute sum of the last layer of connector weights of an interconnected neural network, for the tasks of the Tox21 dataset. Each weight matrix connecting a task to another task had the same dimensions.

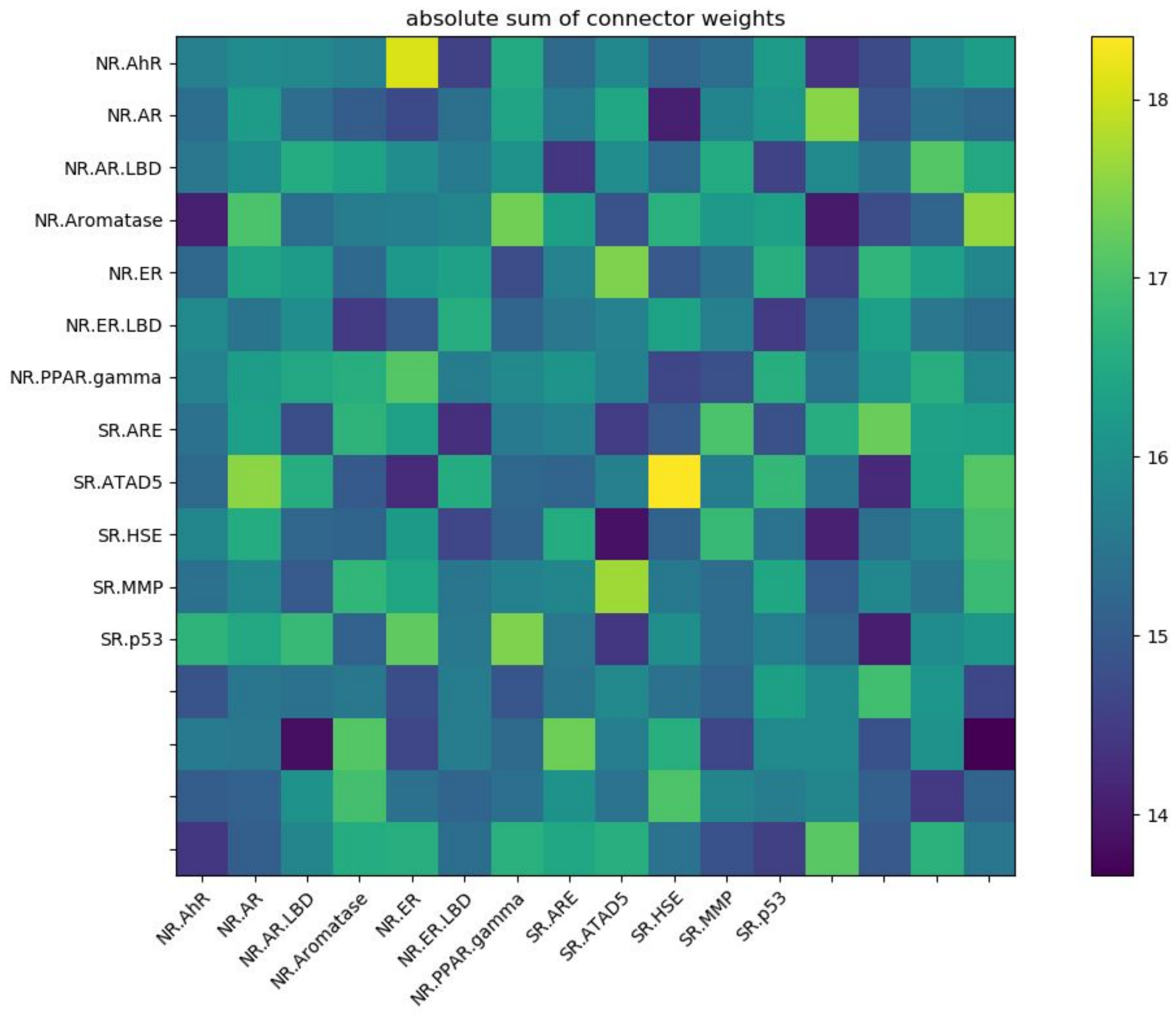

Figure 6. A heatmap of the absolute sum of the last layer of connector weights of an interconnected neural network, for the tasks of the Tox21 dataset, where four random tasks were introduced. Each weight matrix connecting a task to another task had the same dimensions.

Furthermore, each time the models were re-trained, the heatmaps were different. This fact made me judge my assumptions. Something seemed not right. The ICNN performance was significantly better than the single task network, that had the same number of parameters as root parameters in the ICNN, meaning the input from the other subnetworks actually increased performance of each task. But the region of the heatmap of the four additional random tasks looked just as random as that of the other tasks. After seeing that, I decided to add one more experiment to this thesis: The Tox21 dataset contains twelve tasks. I wanted to test what happens if we take one task of the dataset, and generate eleven fake tasks.

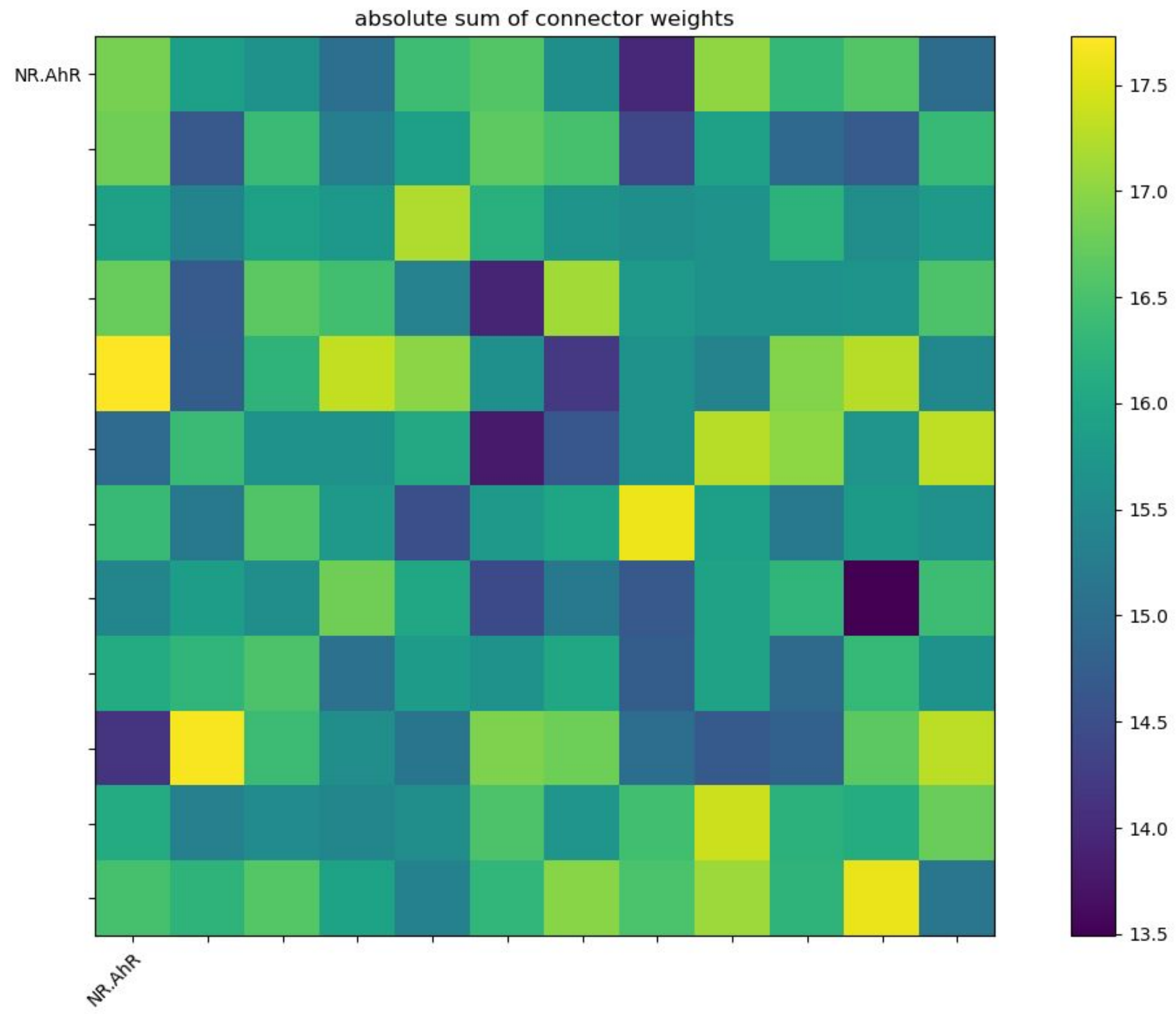

Figure 7. A heatmap of the absolute sum of the last layer of connector weights of an interconnected neural network, for the first tasks of the Tox21 dataset. The Eleven other tasks were just random noise, fake tasks that are correlated to each-other, but not correlated to the tasks of Tox21.

I repeated this for each task, picking one task, and generating eleven fake ones. After training the ICNN models and MTL neural nets on the new tasks, I got identical results to table 1. This proves that there is no task relatedness contributing to the good performances. The multitask learning models, at least for the tox21 dataset, perform that well not because of information sharing between related tasks, but because of noise injection. Thus the tasks that are trained jointly do not help each-other in the way one might think, it’s the noise that is contributing to the performance of each individual task, not the task relatedness. My assumption is that the extra noise added during training serves as a regularization technique to make the model generalize more.

6 Uses, modifications, and pitfalls of ICNNs

As we saw from the experiments, the ICNN architecture has potential to be used for many multi task learning problems. Requiring no prior knowledge for task relatedness can save a lot of time. By looking at the connector weights, the ICNN architecture can also be used to better understand task relatedness, further helping researchers to develop better models. The ICNN architecture, in itself however, i.e. without any modifications to the algorithm, is not a scalable architecture. The number of parameters needed per added task explodes rapidly. One might argue that this is a technological problem, and that future computers will have vaster memories, but the argument still stands - an interconnected neural network’s space complexity is a problem.
Because of this, I would like to propose a few ideas that might help lower this complexity, while still keeping a robust model. These ideas were not tested in this thesis, but they could lead to interesting future research.

6.1 An interconnected model for more efficient single task learning

In the earlier proposed architecture, we said that an interconnected neural network has N separate neural networks (Where N is the number of tasks), a.k.a. subnetworks, and that each subnetwork has connector weights, connecting each layer of one subnetwork, with the next layers of the other subnetworks.
For many problems however, we do not care about the performances of each individual task, but rather want a good performance on a single task (or a small set of tasks), while using the

advantages of multitask learning. In such cases, instead of connecting each subnetwork to the other subnetworks, resulting in a high number of parameters, one can pick the subnetwork(s) of interest, and connect only those ones to the other subnetworks. As a result, instead of having N subnets, where each layer of a subnet has N-1 weight matrices, we have only one subnetwork with that many connections.

That lowers the total number of weights for an inter-layer from:

$$n \cdot m \cdot N^2$$

(Where n is the number of rows, m is the number of columns, and N the number of tasks),

To:

$$n \cdot m \cdot 2N$$

(If we are interested in the output of only one task). This lowers the number of parameters significantly. The technique makes use of the advantages of multitask learning, i.e. the inductive bias of other models, while not requiring to know the relatedness between the tasks.

6.2 Pruning

Pruning would not solve the issue of having too many parameters during training, but it would solve the issue of having too many parameters during testing. Having compact models with capabilities for quick predictions is also an important attribute for real-world applications. We can assume that most of the parameters in an ICNN are useless. Pruning the connector weights would lower the models memory consumption significantly, while also removing some unnecessary computations, i.e. speeding up predictions.

7 Related work

The most related work I could find to that of the interconnected neural network idea, was the progressive neural network [11]. Similar to the ICNN idea, the progressive neural network uses lateral connections between individual neural networks to leverage prior knowledge. One difference between the progressive neural network, and the interconnected network, is that the former network trains each network individually, and then transfers the knowledge while training the model of interest. The progressive neural network also uses adapters to augment the layers of lateral connections, and does not perform backpropagation the same way as in an interconnected neural network (because the single task neural networks are trained separately).

8 Conclusion

As we have proven that task relatedness is not the reason why the performance of multitask models is higher for the tox21 dataset, and that the ICNN model needs slight modifications to work for the second postulated hypothesis in section 4.1, it is necessary to do further studies before saying something decisive about the proposed architecture. The interconnected neural network performed overall well, but for an unbiased statement, further studies are needed. It will be very interesting to see how the interconnected neural network will compare to the conventional hard parameter sharing technique, once some kind of connector weight regularization is added to the model. We will explore such ideas, as well as ideas of "neural plasticity" to keep the number of parameters lower, in future work.

References

1. Caruana, R. Multitask Learning. in *Learning to Learn* (eds. Thrun, S. & Pratt, L.) 95–133 (Springer US, 1998).

2. Mayr, A., Klambauer, G., Unterthiner, T. & Hochreiter, S. DeepTox: Toxicity Prediction using Deep Learning. *Front. Environ. Sci. Eng. China* **3,** 80 (2016).
3. Girshick, R. Fast R-CNN. *arXiv [cs.CV]* (2015).
4. Collobert, R. & Weston, J. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. in *Proceedings of the 25th International Conference on Machine Learning* 160–167 (ACM, 2008).
5. Deng, L., Hinton, G. & Kingsbury, B. New types of deep neural network learning for speech recognition and related applications: an overview. in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* 8599–8603 (2013).
6. Ruder, S. An Overview of Multi-Task Learning in Deep Neural Networks. *arXiv [cs.LG]* (2017).
7. Anandkumar, A. & Ge, R. Efficient approaches for escaping higher order saddle points in non-convex optimization. *arXiv [cs.LG]* (2016).
8. Hochreiter, S. Untersuchungen zu dynamischen neuronalen Netzen.
9. Mitchell, T. M. The Need for Biases in Learning Generalizations. *citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.5466* (1980).
10. Unterthiner, T., Mayr, A., Klambauer, G. & Hochreiter, S. Toxicity Prediction using Deep Learning. *arXiv [stat.ML]* (2015).
11. Rusu, A. A. *et al.* Progressive Neural Networks. *arXiv [cs.LG]* (2016).