The Usefulness of Past Knowledge when Learning a New Task in Deep Neural Networks

Guglielmo Montone[*]
Laboratoire Psychologie de la Perception
Université Paris Descartes
75006 Paris, France
montone.guglielmo@gmail.com

J. Kevin O'Regan[*]
Laboratoire Psychologie de la Perception
Université Paris Descartes
75006 Paris, France
jkevin.oregan@gmail.com

Alexander V. Terekhov[*]
Laboratoire Psychologie de la Perception
Université Paris Descartes
75006 Paris, France
avterekhov@gmail.com

Abstract

In the current study we investigate the ability of a Deep Neural Network (DNN) to reuse, in a new task, features previously acquired in other tasks. The architecture we realized, when learning the new task, will not destroy its ability in solving the previous tasks. Such architecture was obtained by training a series of DNNs on different tasks and then merging them to form a larger DNN by adding new neurons. The resulting DNN was trained on a new task, with only the connections relative to the new neurons allowed to change. The architecture performed very well, requiring few new parameters and a smaller dataset in order to be trained efficiently and, on the new task, outperforming several DNNs trained from scratch.

1 Introduction

Deep Neural Networks (DNNs) have yielded impressive results during the last decade, in many cases outperforming other existing machine learning algorithms, bringing the dream of building intelligent machines closer to becoming a reality. But despite these impressive results, DNNs are still limited in their ability to replicate many characteristics of the human brain. The learning process in humans, for example, seems to involve the creation of abstract concepts and strategies. Such concepts and strategies can then be re-applied to new scenarios that share symbols or relations between symbols with familiar tasks. This capacity is accompanied by two other human abilities that it would be very useful to reproduce in DNNs: the ability to learn a new task from a very limited set of examples, and the ability to use a gradual and sequential learning process, where more complex tasks are presented only after the learner is familiar with simpler tasks.

The brain can be considered as a very large neural network where some areas are principally involved in one specific task or computation and others are used for multiple tasks. Two characteristics of the brain likely contribute to the development of abstraction abilities in humans. Specifically, the neural system is able to:

- store previously learned capacities,
- reuse areas used for solving previous tasks to solve a new task.

[*] lpp.psycho.univ-paris5.fr/feel/

One strategy for implementing such characteristics in a DNN could consist of a training algorithm that differentiates between the areas in the network that have been highly involved in solving a previous task from areas that were exploited less intensively. In this way could be possible to realize a training algorithm such that the areas involved in previous tasks would be slightly effected by the training while the areas less exploited previously would be more deeply modified.

It is likely that a network trained in such a way would be able not only to store previously learned tasks, but also to reuse some previously learned features and strategies when performing a new task. In this paper we present a step towards building such a network. To do so, we first trained several networks on different tasks. We then connected the networks together in a larger DNN, adding new groups of neurons. The resulting architecture was then trained on a new task, with only the newly introduced areas involved in the training. Our aim was to show that such an architecture is able to solve several tasks and to reuse areas trained on previous task to perform new tasks.

Many interesting attempts to implement these desired characteristics in a DNN can be found in the literature. The *pre-training* of a network is one example. In this technique, before a network is trained on the desired task, it is trained on a similar task in either a supervised or unsupervised fashion. This technique has been shown to work as a regularizer [8], and when used with the *dropout* [11] technique, has been shown to produce clear advantages compared to a network that is trained from scratch. Advantages that are particularly evident when the first and second tasks are functionally similar or when the tasks have a common input format [10]. However, training on a second task is often accompanied by *catastrophic forgetting* of the first task, with performance on the first task severely reduced by the new training.

Another technique, usually referred to as *multitask learning* [5], is also relevant to the construction of networks that benefit from learning several tasks. This technique consists of simultaneously training a DNN on several tasks. It has been recently applied in the domain of natural language processing [6, 7, 13]. Multitask learning has shown promising results. It seems that simultaneously learning on several tasks forces the network to learn more general features, which results in the network performing well even when trained on smaller datasets. However, the major disadvantage of this technique is that the network must be trained on all tasks in parallel, making it impossible to sequentially train the network with tasks of increasing difficulty. Although it has not yet been widely studied, a sequential learning strategy, similar to Bengio's *curriculum learning*, could be quite powerful in training DNN [3].

The method we propose in this paper consists of training a series of DNNs on different tasks. These networks are then connected with groups of new neurons, forming a bigger network to be trained on a task similar to the previous ones. Importantly, only connections relative to newly added neurons are learned during the new training. Using this method, the architecture does not suffer from catastrophic forgetting; and more importantly, as we will show, it can exploit the positive properties of pre-training and multitasking, namely:

- a smaller dataset is sufficient for training on a new task,
- fewer computational resources are needed in order to learn a new task.

In the next section, we illustrate the way in which we merge various DNNs into a single larger network. In section 1.2 we describe the artificially created tasks, and explain why they are particularly well suited for investigating the ability of the network to reuse previously learned features. In section 2 we give specific details about the DNNs used and the learning procedure. In section 3 we present the results. In the last section we suggest directions for further research.

1.1 Merging DNNs

Imagine that we have trained a DNN on several tasks at the same time, and that we have a learning algorithm which is able to differentiate between the areas of the network that were heavily used in any of the tasks and those that were less used. Now let us imagine training the network on a new task, and suppose that the training algorithm could be constructed so as to focus on the areas that were less used for the previous tasks. Such an architecture, we hypothesized, would be able to learn multiple tasks, even when presented in a sequential order, and would be able to *efficiently reuse previously trained areas*. The work presented here empirically confirms the previous hypothesis. The procedure used was as follows.

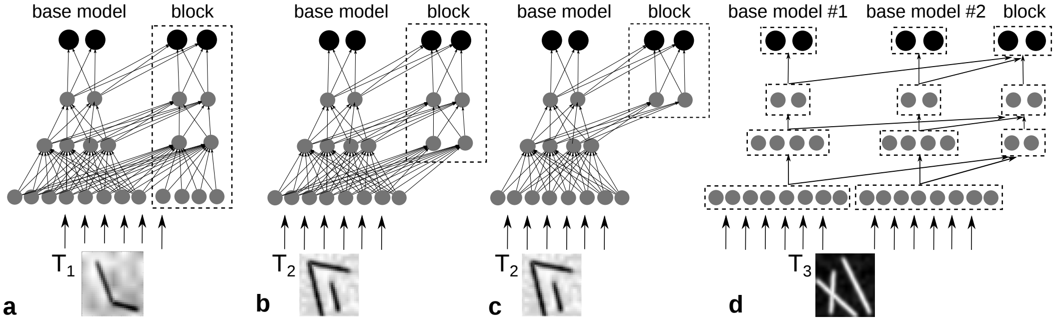

Figure 1: (a) The architecture built by adding a block neurons with three hidden layers to one base model. (b) Adding a block neurons with two hidden layers to one base model. (c) Adding a block neurons with one hidden layer to one base model. (d) Adding a block neurons to two base models. The dashed boxes indicate the layers of the two base models and the block neurons added. An arrow connecting two boxes indicates that all the neurons in the first box are connected to all the neurons in the second box.

We defined a set of tasks $T_1, \ldots, T_M$ and trained a DNN $N_1, \ldots, N_M$ on each task. The networks used were feed-forward DNNs with three hidden layers. After the first training phase, we used some of the trained networks, say $N_1, \ldots, N_m$, to build a *block architecture* that was then trained on one of the remaining tasks, say T_{m+1}. The block architecture was formed by adding a set of new neurons to the previously trained networks N_1,...,N_m (we refer to such networks as *base models* and to the added neurons as *block neurons*). The block neurons together comprised a DNN, which we connected to the base models as follows. The first hidden layer of the block neurons received the input for the task T_{m+1}. The same input was sent to all networks N_1,...,N_m. The second hidden layer was fully connected to both the first hidden layer of the block neurons and the first hidden layer of each network N_1,...,N_m. This pattern was repeated for the third hidden layer of the block neurons, which was fully connected to its own second hidden layer and to the second hidden layer of each network N_1,...,N_m. Finally, the output layer of the block neurons received the input coming from its own third layer and from the third layers of each network. Figure 1(a) illustrates this architecture when block neurons were added to only one base model. In this work, this architecture was tested along with two variations. In the first variation, figure 1(b), the block neurons did not have a first hidden layer. The second hidden layer was fully connected to the first hidden layer each network N_1,...,N_m. In the other variation, figure 1(c), the second layer of block neurons was also removed, with the only remaining hidden layer receiving input from the second layer of each network. When training on the task T_{m+1} *none of the parameters in the base model networks was allowed to change*. Training on task T_{m+1} was performed exclusively on the connections linking the base model to the layers of block neurons and on the connections between the different layers of block neurons.

Our results from the first study on this kind of architecture have been published elsewhere [16]. In the present study, the DNNs were trained using a dataset half the size of that used in the previous work. Moreover, a much larger number of architectures was tested. In particular, we constructed architectures with up to 5 base models, while reducing the number of hidden layers in the block neurons, thereby reducing the number of parameters (weights and neurons) associated solely with the block neurons. Finally, we tested the capacity of the block architecture to learn from a dataset much smaller than the one used to train the base models.

1.2 The tasks

We developed six binary classification tasks, which the networks were trained on. The tasks all involved the concepts of line and angle. We wished to show that the networks $N_1, \ldots, N_m$, when trained on such tasks, would develop features that could be reused by the block architecture to solve another task involving the same concepts.

In each task the stimuli were gray scale images, 32 x 32 pixels in size. Each image contained two to four line segments, each at least 13 pixels long (30% of the image diagonal). The segments were white on a dark random background or black on a light random background. The distance between

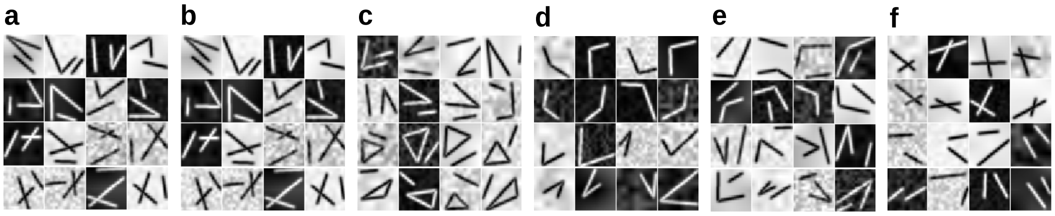

Figure 2: Examples of stimuli: (a) *ang_crs* – line segments forming an angle vs. two crossing line segments; (b) *ang_crs_ln* – same, with an additional non-crossing line segment; (c) *ang_tri_ln* – angle vs. triangle; (d) *blt_srp* – blunt angle vs. sharp angle; (e) *blt_srp_ln* – same, with a non-crossing line segment; (f) *crs_ncrs* – two crossing line segments vs. two non-crossing line segments.

the end points of each line segment and every other line segment was at least 4 pixels (10% of the image diagonal). In order to obtain anti-aliased images, the lines were first generated on a grid three times larger (96 x 96). The images were then filtered with a Gaussian filter with a sigma of 3 pixels, and down-sampled to the final dimensions. The 6 tasks were (see examples in figure 2):

ang_crs: requires classifying the images into those containing an angle (between 20° and 160°) and a pair of crossing line segments (the crossing point must lay between 20% and 80% along each segment's length).

ang_crs_ln: the same as *ang_crs*, but has an addition line segment crossing neither of the other line segments.

ang_tri_ln: distinguishes between images containing an angle (between 20° and 160°) and a triangle (with each angle between 20° and 160°); each image also contains a line segment crossing neither angle nor triangle.

blt_srp: requires classifying the images into those having blunt (between 100° and 160°) and those having sharp (between 20° and 80°) angles in them.

blt_srp_ln: the same as *blt_srp*, but has an additional line segment, crossing neither of the line segments forming the angle.

crs_ncrs: distinguishes between a pair of crossing and a pair of non-crossing lines (the crossing point must lay between 20% and 80% of each segment length).

Each stimulus was generated by randomly selecting the parameters describing each figure (length and orientation of straight segments, amplitude and orientation of the angles) and verifying that all conditions were satisfied. Four different types of random backgrounds were generated with four patterns changing with different velocities. For our experiments we generated 350,000 stimuli for each condition.

2 Network details

The networks used were feed-forward DNNs with three hidden layers. The number of hidden units in the different experiments varied, but the largest network had 200, 100, and 50 units in its first, second, and third hidden layers, respectively. We refer to these networks with the letters NN followed by the number of neurons in each layer. Thus, the name of the network just described would be NN-200-100-50.

We constructed different types of block architectures by varying the type of base models used, the number of base models used, and the number of neurons per layer in the block neurons. We refer to a block architecture obtained by connecting base models to a DNN with 100 units in the first hidden layer, 50 units in the second hidden layer, and 50 units in the third hidden layer as BA-100-50-50. We refer to an architecture with no hidden units in the first layer and 50 units in the other two hidden layers as BA-0-50-50, while an architecture with no units in the first and second hidden layers and 50 units in the third layer would be named BA-0-0-50.

All networks used a softmax logistic regression as the output layer. The activation functions of the neurons of the network were *rectified linear units* (ReLUs) [14]. Each network was trained to minimize a cost function which combines three terms: the negative log-likelihood of the prediction given the data, and two regularization terms. The first regularization term forces sparsity in the neurons activation and consists of the KL divergence between the mean activation of each neuron and a uniform probability distribution of value 0.05. The second regularization term is the squared sum of all the weights in the networks. The coefficient of the first regularization term was equal to 0.01. The weight-limiting coefficient was set to 0.0001. The weights of the k-th hidden level were initialized, according to [9], with random uniformly distributed values in the range $\pm\sqrt{6/(n_{(k-1)}+n_{(k)})}$, where $n_{(k)}$ is the number of neurons at the k-th level and $n_{(0)}$ is equal to the number of inputs. The biases were all initialized to 0.

The total dataset was split into training (330,000 samples), validation (10,000 samples), and test (10,000 samples) datasets. All of the architectures were trained on the entire training dataset using mini-batch gradient descent learning with a batch size of 20. The initial learning rate for the gradient descent was set to 0.01, and it decreased by a factor 0.985 after every epoch. We used early stopping of the training process if the error on the validation dataset did not decrease after 5 epochs. The test score corresponding to the minimal validation error is presented as the performance of the network. The initial learning rate was selected using a human-guided search. Different values of the initial learning rate were tested, with uniform steps for the logarithm of the tested values taken over the interval $[log(0.1), log(10^{-6})]$.

All code was written in python using Theano [1, 4]. Source files are available online: `https://github.com/feel-project/abstraction`

3 Results

In this section, we first report the results obtained by training a DNN on each of the previously described tasks. Then we report the results of training different block architectures on the same tasks. We present the results for block architectures with one or two base models, then the results for those with three, four, or five base models. The number of possible architectures that can be built by changing the base models, the number of block neurons and the task on which the block architecture is trained, is very large, and exploring all possibilities was not feasible. Instead, we tried to select the conditions that best allowed us to obtain an understanding of the performance of the block architectures. The performance was evaluated by computing the percentage of misclassified samples on the test dataset. Each architecture was trained five times, randomly initializing its weights. The mean performance over the five repetitions and the best and worst performance are reported in the tables.

Original networks

Prior to building block architectures, we trained a DNN on each task. The networks used were of type NN-200-100-50, with 200, 100, and 50 nodes in the first, second, and third layers, respectively. Networks of this type were used as base models for all of the block architectures. The percentages of misclassified test examples for these networks are shown in table 1 together with the results for other architectures, namely NN-30-20-10, NN-60-40-30 and NN-70-50-30. Each of these last three networks had approximately the same number of parameters (weight of the networks) of some of the block architectures, making interesting performance comparisons possible.

One and two base-model architectures

The percentages of misclassified test examples for block architectures with one and two base models are presented in the tables 2 and 3, respectively. The architectures that performed better than (or equal to) the network NN-200-100-50, which was trained from scratch, are shown in bold. In these tables, the tasks on which the block architectures were trained are listed together with the tasks on which the base models were trained (in parentheses).

Architectures with just one base model (table 2) rarely outperformed the original network, NN-200-100-50. This only happened when the task on which the block architecture was trained was similar to the one used to train the base model, namely for the conditions *ang_crs_ln* (*ang_crs*) and

Table 1: Original networks result.

condition	200-100-50	70-50-30	60-40-20	30-20-10
ang_crs	5.5 (5.4–5.9)	8.6 (8.1–9.3)	9.4 (8.9–9.8)	13.3 (12.8–15.0)
ang_crs_ln	13.6 (12.5–15.2)	17.2 (16.5–17.4)	18.3 (16.7–18.8)	21.9 (21.5–23.5)
ang_tri_ln	6.1 (5.5–6.8)	9.7 (8.8–10.1)	11.4 (10.6–14.0)	14.1 (13.4–15.2)
blt_srp	2.0 (1.8–2.3)	3.4 (3.1–3.8)	3.7 (3.4–4.2)	5.9 (5.1–6.3)
blt_srp_ln	6.5 (6.4–6.9)	10.7 (10.0–13.7)	12.5 (11.6–14.1)	17.2 (16.6–19.5)
crs_ncrs	2.8 (2.3–2.9)	3.7 (3.2–4.3)	4.5 (4.1–5.2)	5.5 (4.6–6.1)

The numbers correspond to the median (min–max) percentage of misclassified examples.

blt_srp (*blt_srp_ln*). Even when a second base model was added, *ang_crs_ln* and *blt_srp* remained the only two tasks on which the block architecture outperformed the original network, NN-200-100-50. It is interesting to compare the results of the architecture BA-100-50-50 with one base model to those of the architecture BA-0-50-50 with two base models. In this example, even though it has fewer parameters, the architecture with two base models outperforms that with one in three of the four tasks on which it was tested, namely *blt_srp*, *blt_srp_ln* and *ang_crs*. This result seem to suggests that increasing the number of base models yields better results than increasing the number of neurons in the block architecture. We decided to further study this possibility by adding even more base models.

Table 2: Adding blocks to one base model.

condition	0-50-50	0-100-50	100-50-50
ang_crs_ln (*ang_crs*)	13.8(13.7–14.3)	**13.4 (13.1–13.6)**	**13.2 (13.1 – 14.0)**
blt_srp_ln (*blt_srp*)	9.2 (8.8–9.4)	8.4(8.3–8.6)	8.1 (8.0–8.6)
ang_crs (*ang_crs_ln*)	6.0 (5.9 – 6.2)	5.8 (5.7–6.1)	6.3 (6.0–6.4)
blt_srp (*blt_srp_ln*)	**1.8(1.7–1.9)**	**1.6(1.5–1.9)**	**1.8 (1.5–2.3)**
ang_tri_ln (*ang_crs_ln*)	11.2 (11.1–11.7)	10.8 (10.2–11.1)	7.5(7.1–8.0)
blt_srp_ln (*ang_crs_ln*)	11.7 (11.3–12.3)	10.9 (10.5–11.2)	8.0 (7.6–8.7)
ang_crs_ln (*ang_tri_ln*)	17.3 (16.6–18.2)	16.6 (15.9–17.2)	14.4 (13.4–14.7)
ang_crs_ln (*blt_srp_ln*)	18.1 (17.2–18.6)	17.1 (16.2–17.3)	14.5 (14.3–14.9)

The tasks on which the block architectures were trained are listed together with the tasks on which the base models were trained (in parentheses). The architectures that performed better than (or equal to) the networks NN-200-100-50 trained from scratch are shown in bold. This is also the case for Tables 3-8.

Table 3: Adding blocks to two base models.

condition	0-50-50	0-100-50	50-50-50
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*)	13.7 (13.1–14.2)	**13.2 (12.9–13.3)**	**13.1(12.7–13.8)**
ang_crs_ln (*ang_tri_ln*+*blt_srp_ln*)	14.2(13.8–14.6)	**13.6(12.9–13.8)**	**13.5(12.9–14.2)**
blt_srp (*ang_tri_ln*+*crs_ncrs*)	**1.9 (1.8–2.3)**	**1.6(1.5–1.7)**	**1.8(1.6–2.1)**
blt_srp (*ang_tri_ln*+*ang_crs_ln*)	**2.0(1.7–2.3)**	**1.5(1.4–1.8)**	**1.7(1.5–1.7)**
blt_srp (*ang_tri_ln*+*blt_srp_ln*)	**1.5 (1.3–1.7)**	**1.3 (1.3–1.6)**	**1.4 (1.2–1.7)**
blt_srp_ln (*ang_tri_ln*+*ang_crs_ln*)	7.6 (7.5–7.8)	7.3 (7.1–7.8)	6.9 (6.8–7.3)
blt_srp_ln (*ang_tri_ln*+*crs_ncrs*)	7.6 (7.5–7.8)	7.3 (7.1–7.8)	6.9 (6.8–7.3)
ang_crs (*ang_tri_ln*+*blt_srp_ln*)	6.2 (6.0–6.3)	5.7 (5.6–5.9)	5.6 (5.5–5.7)
ang_crs (*ang_tri_ln*+*ang_crs_ln*)	6.2 (6.0–6.3)	5.7 (5.6–5.9)	5.6 (5.5–5.7)

Architectures with more than two base models

In this paragraph we present the results obtained adding three, four and five base-models to two kinds of block-architectures: BA-0-50-50 and BA-0-0-50. In table 4 are presented the percentages of misclassified test examples obtained using three base-models. The results for the architecture BA-0-50-50 were better than those obtained for the architecture NN-60-40-20 that has the same number of parameters(approximately 60,000 weights), and better, for nearly all tasks, than those for the architecture NN-200-100-50. This suggest that the block architecture highly benefits from

Table 4: Adding blocks to three base-models.

condition	0-50-50	0-0-50
ang_crs (*ang_tri_ln*+*crs_ncrs*+*blt_srp*)	**5.5 (5.2–5.8)**	6.0 (5.7–6.1)
ang_crs (*ang_tri_ln*+*ang_crs_ln*+*crs_ncrs*)	**4.0 (3.9–4.1)**	**4.5 (4.2 – 4.6)**
ang_crs (*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*)	**4.6 (4.3–4.8)**	5.9 (5.6 – 6.0)
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*)	**11.3 (10.9–11.8)**	15.0 (14.1 – 16.3)
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*+*blt_srp*)	**12.0 (11.6–12.5)**	15.2 (15.0–15.9)
blt_srp (*ang_crs*+*ang_tri_ln*+*crs_ncrs*)	**1.4 (1.4–1.6)**	2.2 (2.0 –2.3)
blt_srp (*ang_crs_ln*+*ang_tri_ln*+*crs_ncrs*)	**1.5 (1.3–1.8)**	2.4 (2.0–3.0)
blt_srp_ln (*ang_crs_ln*+*ang_tri_ln*+*crs_ncrs*)	7.0 (6.7–7.4)	10.2(9.9 –10.5)
blt_srp_ln (*ang_crs*+*ang_tri_ln*+*crs_ncrs*)	6.9 (6.7–7.4)	9.8 (9.3–10.1)

being constituted by several base models trained on different tasks. Eliminating the second hidden layer from the block architecture yielded much poorer results. For example, the architecture BA-0-0-50 outperformed NN-200-100-50 on only one task. However, the results for BA-0-0-50 improved significantly when the number of base models was increased to four (Table 5). This performance was comparable to that of BA-0-50-50 with three base models. However, at the same time, it seems that for the architecture BA-0-0-50 with four base models the choice of the base models has a big influence on the final performance. This can be observed, for example, in the task *ang_crs_ln* where using to performance drops from 11.4 to 14.1 when just one of the base model is changed.

Table 5: Adding blocks to four base models.

condition	0-50-50	0-0-50
ang_crs (*ang_tri_ln*+*crs_ncrs*+*blt_srp*+*blt_srp_ln*)	**4.9 (4.8–5.6)**	**5.3 (5.0–5.7)**
ang_crs (*ang_tri_ln*+*ang_crs_ln*+*crs_ncrs*+*blt_srp_ln*)	**3.6 (3.4–3.8)**	**4.3 (4.0–4.7)**
ang_crs (*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*+ *ang_crs_ln*)	**3.8 (3.5–4.0)**	**4.2 (4.0–4.4)**
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*+*ang_crs*)	**9.7 (9.3–10.1)**	**11.4 (11.1–12.1)**
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*+*blt_srp*+*blt_srp_ln*)	**10.9 (10.6–11.1)**	14.1 (13.6–14.4)
blt_srp (*ang_crs*+*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*)	**1.1 (1.0–1.2)**	**1.3 (1.1–1.5)**
blt_srp (*ang_crs_ln*+*ang_tri_ln*+*crs_ncrs*+*ang_crs*)	**1.2 (1.0–1.4)**	**1.8 (1.7–2.0)**
blt_srp_ln (*ang_crs_ln*+*ang_tri_ln*+*crs_ncrs*+*ang_crs*)	**5.9 (5.6–6.2)**	10.2 (10.0–10.5)
blt_srp_ln (*ang_crs*+*ang_tri_ln*+*crs_ncrs*+*ang_crs_ln*)	**5.8 (5.6–6.0)**	9.9 (9.3–10.1)

In Table 6, we list the results obtained using five base models. In this case, for each task, all of the networks trained on each of the other tasks were used. Both architectures BA-0-50-50 and BA-0-0-50 outperformed the architecture NN-70-50-30, which had the same number of parameters, as well as the architecture NN-200-100-50. It is interesting to note here that the performance on the task *ang_crs_ln* (11.3) is near to the best of the performance obtained using 4 base models. This probably suggest that the architecture has been able to select among the base models the ones more useful for the new task.

Table 6: Adding blocks to five base models.

condition	0-50-50	0-0-50
ang_crs (all models used except *ang_crs*)	**3.6 (3.3–3.8)**	**4.1 (3.6–4.3)**
ang_crs_ln (all models used except *ang_crs_ln*)	**9.5 (9.3–9.9)**	**11.3 (11.0–11.8)**
blt_srp (all models used except *blt_srp*)	**1.0 (0.9–1.3)**	**1.2 (1.0–1.3)**
blt_srp_ln (all models used except *blt_srp_ln*)	**4.7 (4.4–4.9)**	**6.5 (6.2–7.0)**
crs_ncrs (all models used except *crs_ncrs*)	**1.1 (1.0–1.1)**	**1.1 (1.0–1.2)**
ang_tri_ln (all models used except *ang_tri_ln*)	**4.9 (4.6–5.0)**	7.6 (7.4–7.8)

On a smaller dataset

As mentioned in the introduction, we also wished to verify that the proposed architecture could be trained with a smaller dataset than a network trained from scratch. In this paragraph we present the results obtained by training the same architectures presented in Tables 5 and 6 with a dataset of 200,000 examples. The architectures in Tables 5 and 6 were trained on a dataset of 350,000 examples. The percentages of misclassified examples, presented in Tables 7 8, showed that even with a much smaller training dataset, with four or five base models the architecture BA-0-50-50 outperformed the network trained from scratch, NN-200-100-50. The other architecture, BA-0-0-50, had worse results then NN-200-100-50, especially for the more complex tasks, namely *blt_srp_ln* and *ang_tri_ln*.

Table 7: Adding blocks to four base models. Dataset of 200.000 examples.

condition	0-50-50	0-0-50
ang_crs (*ang_tri_ln*+*crs_ncrs*+*blt_srp*+*blt_srp_ln*)	**5.0 (4.8–5.2)**	5.8 (5.4 – 6.3)
ang_crs (*ang_tri_ln*+*ang_crs_ln*+*crs_ncrs*+*blt_srp_ln*)	**4.3 (4.0–4.5)**	**4.6 (4.0 – 5.0)**
ang_crs (*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*+ *ang_crs_ln*)	**4.3 (4.1–4.8)**	**4.7 (4.3–5.5)**
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*+*ang_crs*)	**10.7 (10.4–11.3)**	**12.0 (11.5–12.4)**
ang_crs_ln (*ang_tri_ln*+*crs_ncrs*+*blt_srp*+*blt_srp_ln*)	**12.4 (12.0–12.6)**	15.1 (14.6–15.5)
blt_srp (*ang_crs*+*ang_tri_ln*+*crs_ncrs*+*blt_srp_ln*)	**1.2 (1.1–1.4)**	**1.4 (1.3–1.5)**
blt_srp (*ang_crs_ln*+*ang_tri_ln*+*crs_ncrs*+*ang_crs*)	**1.8 (1.7–2.0)**	2.1 (1.7 – 2.4)
blt_srp_ln (*ang_crs_ln*+*ang_tri_ln*+*crs_ncrs*+*ang_crs*)	**6.4 (6.3–6.6)**	9.7(9.2–10.6)
blt_srp_ln (*ang_crs*+*ang_tri_ln*+*crs_ncrs*+*ang_crs_ln*)	**6.5 (6.3–6.8)**	9.8(9.4–10.3)

Table 8: Adding blocks to five base models. Dataset of 200.000 examples.

condition	0-50-50	0-0-50
ang_crs (all models used except *ang_crs*)	**4.3 (4.0–4.7)**	**4.4 (3.9–4.7)**
ang_crs_ln (all models used except *ang_crs_ln*)	**10.6 (10.4 – 10.8)**	**11.7 (11.2 – 12.1)**
blt_srp (all models used except *blt_srp*)	**1.2 (0.9 – 1.9)**	**1.4 (1.1 – 1.8)**
blt_srp_ln (all models used except *blt_srp_ln*)	**5.6 (5.2 – 5.9)**	7.2 (6.8 – 8.0)
crs_ncrs (all models used except *crs_ncrs*)	**1.2 (1.0–1.3)**	**1.2 (1.0–1.3)**
ang_tri_ln (all models used except *ang_tri_ln*)	**5.8 (5.7–6.0)**	8.6 (8.3–9.0)

4 Conclusions

DNNs are known to have some abstraction abilities [15]. These abstractions, however, are very limited compared to those typical of humans, where different concepts recall each other through a vast network of relationships. One possible aspect of the creation of such powerful abstractions could be that humans are continuously engaged in different tasks which are each solved with a limited set of resources that must be shaped in order to optimize performance on the largest number of tasks.

In this paper we built a DNN by training several DNNs on different tasks and then merging them with a group of added neurons. The resulting architecture was capable of performing several tasks. Moreover, it was very efficiently trained, using a dataset smaller than that used for the original DNNs and adding only a small number of new neurons. Our results are consistent with previous experimental observations that training deep architectures is easier when cues to the function that intermediate levels should compute are provided ([2, 17]) and when training is not performed on the whole network at the same time [12].

Our study can be considered a first step toward the construction of DNN architectures which are able to use areas trained on previous tasks when learning a new task. This type of procedure should focus on training areas of the network that have not previously been used, while only slightly modifying areas already trained on previous tasks. Such an architecture would maintain the positive characteristics described for our architectures, and would be capable of sequential learning.

Acknowledgments

This work was funded by the European Research Council (FP 7 Program) ERC Advanced Grant “FEEL” to KO’R

References

[1] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[2] Yoshua Bengio. Evolving culture versus local minima. In *Growing Adaptive Machines*, pages 109–138. Springer, 2014.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[5] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[6] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[7] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoffrey Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

[8] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[10] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgeting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[11] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[12] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *The Journal of Machine Learning Research*, 10:1–40, 2009.

[13] Xiaodong Liu, Jianfeng Gao, Xiaodong He, Li Deng, Kevin Duh, and Ye-Yi Wang. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. *Proc. NAACL, May 2015*.

[14] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[15] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. *arXiv preprint arXiv:1412.1897*, 2014.

[16] Alexander V Terekhov, Guglielmo Montone, and J Kevin ORegan. Knowledge transfer in deep block-modular neural networks. In *Biomimetic and Biohybrid Systems*, pages 268–279. Springer, 2015.

[17] Jason Weston, Frédéric Ratle, Hossein Mobahi, and Ronan Collobert. Deep learning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pages 639–655. Springer, 2012.