Journal of Machine Learning Research 17 (2016) 1-32 Submitted 7/13; Revised 10/14; Published 3/16

# Knowledge Matters: Importance of Prior Information for Optimization

**Çağlar Gülçehre** GULCEHRC@IRO.UMONTREAL.CA

**Yoshua Bengio** BENGIOY@IRO.UMONTREAL.CA
*Département d'informatique et de recherche opérationnelle*
*Université de Montréal, Montréal, QC, Canada*

**Editor:** Aaron Courville, Rob Fergus, and Christopher Manning

## Abstract

We explored the effect of introducing prior knowledge into the intermediate level of deep supervised neural networks on two tasks. On a task we designed, all black-box state-of-the-art machine learning algorithms which we tested, failed to generalize well. We motivate our work from the hypothesis that, there is a training barrier involved in the nature of such tasks, and that humans learn useful intermediate concepts from other individuals by using a form of supervision or guidance using a curriculum. Our results provide a positive evidence in favor of this hypothesis. In our experiments, we trained a two-tiered MLP architecture on a dataset for which each input image contains three sprites, and the binary target class is 1 if all of three shapes belong to the same category and otherwise the class is 0. In terms of generalization, black-box machine learning algorithms could not perform better than chance on this task. Standard deep supervised neural networks also failed to generalize. However, using a particular structure and guiding the learner by providing intermediate targets in the form of intermediate concepts (the presence of each object) allowed us to solve the task efficiently. We obtained much better than chance, but imperfect results by exploring different architectures and optimization variants. This observation might be an indication of optimization difficulty when the neural network trained without hints on this task. We hypothesize that the learning difficulty is due to the *composition* of two highly non-linear tasks. Our findings are also consistent with the hypotheses on cultural learning inspired by the observations of training of neural networks sometimes getting stuck, even though good solutions exist, both in terms of training and generalization error.

**Keywords:** deep learning, neural networks, optimization, evolution of culture, curriculum learning, training with hints

## 1. Introduction

There is a recent emerging interest in different fields of science for *cultural learning* (Henrich and McElreath, 2003) and how groups of individuals that communicates within each other can learn in ways superior to solely individual learning. This is further witnessed by the emergence of new research fields such as "Social Neuroscience". Learning from other agents in an environment by the means of cultural transmission of knowledge with a peer-to-peer communication is an efficient and natural way of acquiring or propagating common knowledge. A popular belief on how the information is transmitted between individuals is that bits of

information are transmitted by small units, called memes, which share some characteristics of genes, such as self-replication, mutation and response to selective pressures (Dawkins, 1976).

This paper investigates an aspect of the hypothesis (which is further elaborated in Bengio (2013a)) that human culture and the evolution of ideas have been crucial to counter an optimization issue: this difficulty would otherwise make it harder for human brains to capture high level knowledge of the world without the help of other educated humans. In this paper, machine learning experiments are used to explore some elements of this hypothesis by seeking answers for the following questions: are there machine learning tasks which are intrinsically hard for a lone learning agent, but those tasks may become easier when intermediate concepts are provided by another agent as an additional intermediate learning cues, in the spirit of curriculum learning (Bengio et al., 2009)? What makes learning such tasks more difficult? Can specific initial values of the neural-network parameters yield success when random initialization yield complete failure? Is it possible to verify that the problem being faced is an optimization problem or a regularization problem or does it influence both training and test behavior? These are the questions discussed (if not completely addressed) here, which relate to the following broader question: how can humans (and potentially one day, machines) learn complex concepts?

In the focus of this paper, results of different machine learning algorithms on an artificial learning task involving binary $64 \times 64$ images are presented. In that task, each image in the dataset contains 3 Pentomino tetris sprites (simple shapes). The task is to figure out if all the sprites in the image are the same or not. Several black-box state-of-the-art machine learning algorithms have been tested and none of them was able to perform better than a random predictor on the test set. Nevertheless, by providing hints about the intermediate concepts (the presence and location of particular sprite classes), can help a neural network to solve the task easily and fast. But the same model without the hints either fails to learn the task or learns it much more slowly and with substantially worse generalization accuracy. Surprisingly, our attempts at solving this problem with unsupervised pre-training algorithms also failed. For showing the impact of intermediate level guidance, we experimented with a two-tiered neural network, with supervised pre-training of the first part to recognize the category of sprites independently of their orientation and scale, at different locations, while the second part learns from the output of the first part and predicts the binary task of interest.

The objective of this paper is not to propose a novel learning algorithm or architecture, but rather to refine our understanding of the learning difficulties involved with composed tasks. Our results also bring empirical evidence in favor of some of the hypotheses from Bengio (2013a), as well as to introduce particular form of curriculum learning (Bengio et al., 2009) based on hints.

Building difficult AI problems has a long history in computer science. Hard AI problems are being studied to create CAPTCHAs that are easy to solve for humans, but hard to solve for machines (Von Ahn et al., 2003). In this paper we are investigating a difficult problem for the off-the-shelf black-box machine learning algorithms. The source code of some experiments presented in that paper is available at `https://github.com/caglar/kmatters`.

### 1.1 Curriculum Learning and Cultural Evolution for Effective Local Minima

What Bengio (2013a) calls an **effective local minimum**, is a point where iterative training stalls, either because of an actual local minimum or due to optimization algorithm is unable (in reasonable time) to find a descent path to a substantially better solution that exists. It is not yet clear why neural network training sometimes stalls in this way, although recent

work suggests that it would generally not be because of local minima (Dauphin et al., 2014; Choromanska et al., 2015). In this paper, we hypothesize that some more abstract learning tasks such as those obtained by composing simpler tasks are more likely to yield such effective local minima for neural networks, and are generally hard for general-purpose machine learning algorithms.

The idea that learning can be enhanced by guiding the learner through intermediate easier tasks is old, starting with animal training by *shaping* (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009). Bengio et al. (2009) introduce a computational hypothesis related to a presumed issue with effective local minima when directly learning the target task: good solutions correspond to hard-to-find-by-chance effective local minima, and intermediate tasks prepare the learner's internal configuration (parameters) in a way that is similar to continuation methods in global optimization. A continuation method would go through a sequence of intermediate optimization problems, starting with a convex one where local minima are no issue, and gradually morphing into the target task of interest.

In a related vein, Bengio (2013a) makes the following inferences based on experimental observations about training of deep neural networks:

Point 1: Training deep architectures is easier when some hints are given about the function that the intermediate levels should compute (Hinton et al., 2006; Weston et al., 2008; Salakhutdinov and Hinton, 2009; Bengio, 2009). *The experiments performed here expand in particular on this point.*

Point 2: It is much easier to train a neural network with supervision (where examples are provided to it when a concept is present and not present in a variety of examples) than to expect unsupervised learning to discover the concept (which may also happen but usually leads to poorer renditions of the concept). *The poor results obtained here with unsupervised pre-training reinforce that hypothesis.*

Point 3: Directly training all layers of a deep network together does not only makes it more difficult to exploit all the extra modeling power of a deeper architecture but in some cases it can yield worse results as the number of *required layers* is increased (Larochelle et al., 2009; Erhan et al., 2010). *The experiments performed here also reinforce that hypothesis.*

Point 4: Erhan et al. (2010) observed that no two training trajectories ended up in the same effective local minimum, out of hundreds of runs, even when comparing solutions as functions from input to output, rather than in parameter space (thus eliminating from the picture the presence of symmetries and multiple local minima due to relabeling and other reparametrizations). This suggests that the number of different effective local minima (even when considering them only in function space) must be huge.

Point 5: Unsupervised pre-training changes the initial conditions of the descent procedure and it can allow to reach substantially better effective local minima (in terms of generalization error). But as empirically observed by (Erhan et al., 2010), better local minima do not appear to be reachable by chance alone on the same architectures. *The experiments performed here provide another piece of evidence in favor of the hypothesis that random initialization can yield rather poor results while specifically targeted initialization can have a drastic impact, that is, the effective local minima are not just numerous but that some small subset of them are much better and hard to reach by chance.* Nevertheless, recent

work showed that rather deep feed-forward networks can be very successfully trained when large quantities of labeled data is available (Ciresan et al., 2010; Glorot et al., 2011; Krizhevsky et al., 2012). Nonetheless, the experiments reported here suggest that it all depends on the task and architecture being considered, since even with very large quantities of labeled examples, most the deep networks trained here were unsuccessful.

Based on the above points, Bengio (2013a) then proposed the following hypotheses regarding learning of high-level abstractions.

- **Deeper Networks are Harder Hypothesis:** Although solutions may exist, effective local minima are generally more likely to hamper learning as the required depth of the architecture increases.
- **Abstractions are Harder Hypothesis:** High-level abstractions are unlikely to be discovered by a single human learner by chance, because these abstractions are represented by a deep subnetwork of the brain, which learns by local descent, thus being sensitive to the issue associated with the above hypothesis.
- **Guided Learning Hypothesis:** A human brain can learn high level abstractions if guided by the signals produced by other agents that act as hints or indirect supervision for these high-level abstractions.
- **Memes Divide-and-Conquer Hypothesis:** Linguistic exchange, individual learning and the recombination of memes constitute an efficient evolutionary recombination operator in the meme-space. This helps human learners to *collectively* build better internal representations of their environment, including fairly high-level abstractions.

This paper particularly focuses on "*Point 1*" above and testing the "*Guided Learning Hypothesis*", using machine learning algorithms to provide experimental evidence. The experiments performed also provide evidence in favor of the "*Deeper Harder Hypothesis*" and are related to the "*Abstractions Harder Hypothesis*". Performance of machine learning algorithms are far from the current capabilities of humans on several tasks, and it is important to tackle the remaining obstacles to approach AI. For this purpose, the question to be answered is why on some tasks humans learn effortlessly from very few labeled examples, while machine learning algorithms fail miserably?

## 2. Optimization Difficulty of Learning High-level Concepts

As hypothesized in the "*Local Descent Hypothesis*", human brains would rely on a local approximate descent, just like a Multi-Layer Perceptron trained by a gradient-based iterative optimization. The main argument in favor of this hypothesis relies on the biologically-grounded assumption that although firing patterns in the brain change rapidly, synaptic strengths underlying these neural activities change only gradually, making sure that behaviors are generally consistent across time. If a learning algorithm is based on a form of local (for example gradient-based) descent, it can be sensitive to effective local minima (Bengio, 2013a).

When one trains a neural network, at some point in the training phase the evaluation of error seems to saturate, even if new examples are introduced. In particular Erhan et al. (2010) find that early examples have a much larger weight in the final solution. It looks like the learner

is stuck in or near a local minimum. But since it is difficult to verify if this is near a true local minimum or simply an effect of strong ill-conditioning or a saddle-point, we call such a "stuck" configuration an *effective local minimum*, whose definition depends not just on the optimization objective but also on the limitations of the optimization algorithm.

Erhan et al. (2010) highlighted both the issue of effective local minima and a regularization effect when initializing a deep network with unsupervised pre-training. Interestingly, as the network gets deeper the difficulty due to effective local minima seemed to be get more pronounced in these experiments. That might be because the number of effective local minima increases (more like an actual local minima issue), or maybe because the good ones are harder to reach (more like an ill-conditioning issue) and more work will be needed to clarify this question.

As a result of Point 4 we hypothesize that it is very difficult for an individual's brain to discover some higher level abstractions by chance only. As mentioned in the "*Guided Learning Hypothesis*" humans get hints from other humans and learn high-level concepts with the guidance of other humans. But some high-level concepts may also be hardwired in the brain, as assumed in the universal grammar hypothesis (Montague, 1970), or in nature vs nurture discussions in cognitive science. Curriculum learning (Bengio et al., 2009) and incremental learning (Solomonoff, 1989), are specific examples of this phenomena. Curriculum learning is done by properly choosing the sequence of examples seen by the learner, where simpler examples are introduced first and more complex examples shown when the learner is ready for them. One of the hypotheses on why a curriculum works states that curriculum learning acts as a continuation method that allows one to discover a good minimum: continuation methods first find a good minimum of a smoother objective function and then gradually change the objective function towards the desired one, while tracking local minima along the way. Recent experiments on human subjects also suggest that humans *teach* by using a curriculum strategy (Khan et al., 2011).

Some parts of the human brain are known to have a hierarchical organization (for example, the visual cortex) consistent with the deep architecture studied in the machine learning literature. As a stimulus is transmitted from the sensory level to higher levels of the visual cortex, higher level areas tend to respond to stimuli in a way corresponding to the detection of more concepts. This is consistent with the *Deep Abstractions Hypothesis*.

Training neural networks and machine learning algorithms by decomposing the learning task into sub-tasks and exploiting prior information about the task is well-established and in fact constitutes the main approach to solving industrial problems with machine learning. The contribution of this paper is rather on rendering explicitly, the effective local minima issue and providing evidence on the type of problems for which this difficulty arises. This prior information and hints given to the learner can be viewed as an inductive bias for a particular task, an important ingredient to obtain a good generalization error (Mitchell, 1980). An interesting earlier finding in that line of research was done with Explanation Based Neural Networks (EBNN) in which a neural network transfers knowledge across multiple learning tasks. An EBNN uses previously learned domain knowledge as an initialization or search bias (that is to constrain the learner in the parameter space) (O'Sullivan, 1996; Mitchell and Thrun, 1993).

Another related work in machine learning is mainly focused on reinforcement learning algorithms, based on incorporating prior knowledge in terms of logical rules to the learning algorithm as a prior knowledge to speed up and bias learning (Kunapuli et al., 2010; Towell and Shavlik, 1994). Also defining sub-tasks and sub-goals to guide the agent is a well-established principle in hierarchical reinforcement learning (Barto and Mahadevan, 2003).

## 3. Experimental Setup

Some tasks, which seem reasonably easy for humans to learn (keeping in mind that humans can exploit prior knowledge, either from previous learning or innate knowledge), are nonetheless appearing almost impossible to learn for current generic state-of-art machine learning algorithms.

Here we study more closely such a task, which becomes learnable if one provides hints to the learner about appropriate intermediate concepts. Interestingly, the task we used in our experiments is not only hard for deep neural networks but also for non-parametric machine learning algorithms such as SVMs, boosting and decision trees.

The result of the experiments for varying size of dataset with several off-the-shelf black box machine learning algorithms and some popular deep learning algorithms are provided in Table 4. The detailed explanations about the algorithms and the hyperparameters used for those algorithms are given in the Appendix Section B. We also provide some explanations about the methodologies conducted for the experiments at Section 3.3.

### 3.1 Pentomino Dataset

In order to test our hypothesis, we designed an artificial dataset for object recognition using 64×64 binary images. The source code for the script that generates the artificial Pentomino datasets (Arcade-Universe) is available at: `https://github.com/caglar/Arcade-Universe`. This implementation is based on Olivier Breuleux's bugland dataset generator. If the task is two tiered (i.e., with guidance provided), the task in the first part is to recognize and locate each Pentomino object class[1] in the image. The second part/final binary classification task is to figure out if all the Pentominoes in the image are of the same shape class or not. If a neural network learned to detect the categories of each object at each location in an image, the remaining task becomes an XOR-like operation over the detected object categories. The types of Pentomino objects that is used for generating the dataset are illustrated in Figure 1 and correspond to Pentomino sprites N, P, F, Y, J, and Q, along with the Pentomino N2 sprite (mirror of "Pentomino N" sprite), the Pentomino F2 sprite (mirror of "Pentomino F" sprite), and the Pentomino Y2 sprite (mirror of "Pentomino Y" sprite).

Figure 1: Different classes of Pentomino shapes used in our dataset.

As shown in Figures 2(a) and 2(b), the synthesized images are fairly simple and do not have any texture. Foreground pixels are "1" and background pixels are "0". Images of the training and test sets are generated iid. For notational convenience, we assume that the domain of raw input images is $X$, the set of sprites is $S$, the set of intermediate object categories for each possible location in the image is denoted as $Y$ and the set of possible final binary task outcomes is $Z = \{0, 1\}$. Two different types of rigid body transformation is performed: sprite rotation, $rot(X, \gamma)$ ,where the set of all rotations is $\Gamma = \{\gamma \colon (\gamma = 90 \times \phi) \wedge [(\phi \in \mathbb{N}), (0 \leq \phi \leq 3)]\}$ and sprite scaling, $scale(X, \alpha)$, where $\alpha \in \{1, 2\}$ is the scaling factor. The data generating procedure is summarized below.

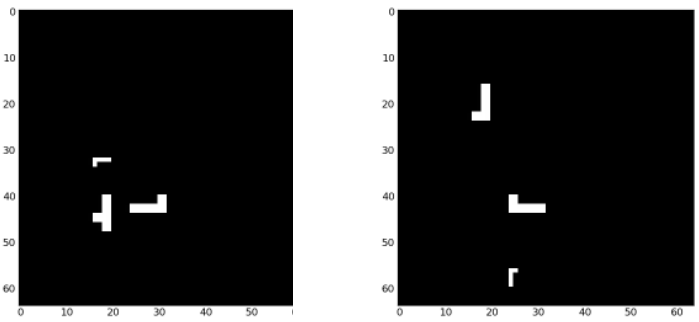
1. A human learner does not seem to need to be taught the shape categories of each Pentomino sprite in order to solve the task. On the other hand, humans have lots of previously learned knowledge about the notion of shape and how central it is in defining categories.

**Sprite transformations**: Before placing the sprites in an empty image, for each image $x \in X$, a value for $z \in Z$ is randomly sampled whether to have (or not) the same three Pentomino sprite shapes in the image. Conditioned on the constraint given by $z$, three sprites $s_{ij}$ at location $(i,\ j)$ are randomly selected from the set $S$ without replacement. Using a uniform probability distribution over all possible scales, a scale is chosen and accordingly each sprite in the image is scaled. Then each sprite is randomly rotated by a multiple of 90 degrees.

**Sprite placement**: Upon completion of sprite transformations, a $64 \times 64$ uniform grid is generated which is divided into $8 \times 8$ blocks, each block being of size $8 \times 8$ pixels, three different blocks are randomly selected from the $64 = 8 \times 8$ on the grid and the transformed objects are each placed into one of the three selected blocks. Consequently, the objects cannot overlap, by construction.

Each sprite is centered in the block in which it is located. Thus there is no object translation inside the blocks. The translation invariance can be achieved by being invariant to the location of the block inside the image.

A Pentomino sprite is guaranteed to not overflow the block in which it is located, and there are no collisions or overlaps between sprites, making the task simpler. The largest possible Pentomino sprite can actually be fit into an $8 \times 4$ mask.



(a) sprites, not all same type (b) sprites, all same type

Figure 2: Left (a): An example image from the dataset which has *a different sprite type* in it. Right (b): An example image from the dataset that has only one type of Pentomino object in it, but with different orientations and scales.

### 3.2 Control Experiments

To explore the effect of changing the complexity of the input representation on the difficulty of the task, a set of experiments have been designed with symbolic representations of the information in each patch. In all cases an empty patch is represented with a vector of all elements 0. In this section we are considering alternative representations to raw image representation for an abstract task similar to Pentomino. Basically we are seeking for an ideal representation that can be fed as an input to a regular feedforward MLP or another black-box classifier to perform well.

The following four experiments have been conducted, each one of them using a different input representation for each patch.

**Experiment 1: One-hot representation without transformations:** In this experiment several trials is done with a 10-input one-hot vector per patch. Each input corresponds to an object category given explicitly (corresponding to one input bit).

**Experiment 2: Disentangled representations:** In this experiment, we did trials with 16 binary inputs per patch, 10 one-hot bits for representing each object category, 4 for rotations and 2 for scaling, that is, the whole information about the input is given, but it is perfectly disentangled. This would ideally be reproduced by an unsupervised learning algorithm over each patch, if it was able to perfectly disentangle the image representation.

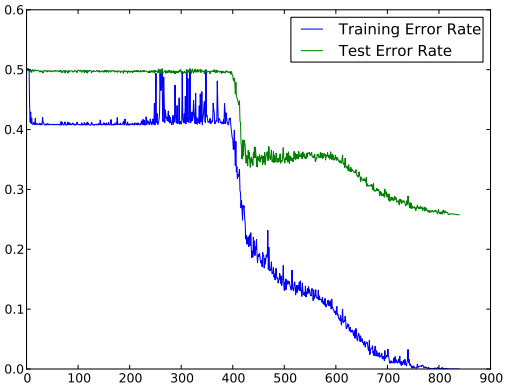
**Experiment 3: One-hot representation with transformations:** For each of the ten object types there are $8 = 4 \times 2$ possible transformations. Two objects in two different patches are the considered "the same" (for the final task) if their category is the same regardless of the transformations. The one-hot representation of a patch corresponds to the cross-product between the 10 object shape classes and the $4 \times 2$ transformations, that is, one out of $80 = 10 \times 4 \times 2$ possibilities represented in an 80-bit one-hot vector. This also contains all the information about the input image patch, but spread out in a kind of non-parametric and non-informative (not disentangled) way, like a perfect memory-based unsupervised learner (like clustering) could produce. Nevertheless, the shape class would be easier to read out from this representation than the image representation (it could be obtained by forming an OR over 8 of the bits).

**Experiment 4: One-hot representation with 80 choices:** This representation has the same 1 of 80 one-hot representation per patch but the target task is defined differently. Two objects in two different patches are considered the same iff they have exactly the same 80-bit one-hot representation (that are of the same object category with the same transformation applied).

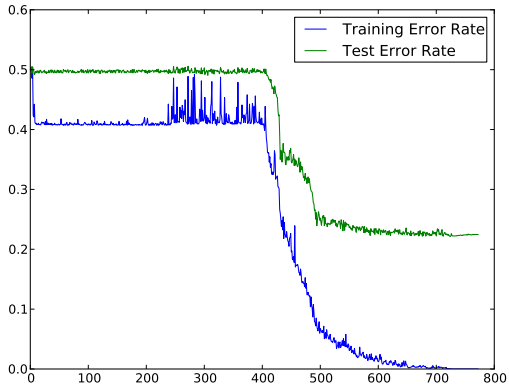
The first experiment is a sanity check. It was conducted with single hidden-layered MLPs using rectifier and tanh nonlinearity, and the task was learned perfectly (0 error on both training and test dataset) with very few training epochs.

The results of Experiment 2 are given in Table 1. To improve our results, we experimented with the Maxout non-linearity in a feedforward MLP (Goodfellow et al., 2013) with two hidden layers. Unlike the typical Maxout network mentioned in the original paper, regularizers have been deliberately avoided in order to focus on the optimization issue, i.e: no weight decay, norm constraint on the weights, or dropout. Although learning from a disentangled representation is more difficult than learning from perfect object detectors, it is feasible with some architectures such as the Maxout network. Note that this representation is the kind of representation that one could hope an unsupervised learning algorithm could discover, at best, as argued in Bengio et al. (2012).

The only results obtained on the validation set for Experiment 3 and Experiment 4 are shown respectively in Table 2 and Table 3. In these experiments a tanh MLP with two hidden layers have been tested with the same hyperparameters. In experiment 3 the complexity of the problem comes from the transformations ($8 = 4 \times 2$) and the number of object types.




(a) Training and Test Errors for Experiment 4




(b) Training and Test Errors for Experiment 3

Figure 3: tanh MLP training curves. Left (a): The training and test errors of Experiment 3 over 800 training epochs with 100k training examples using tanh MLP. Right (b):The training and test errors of Experiment 4 over 700 training epochs with 100k training examples using tanh MLP.

But in experiment 4, the only source of complexity of the task comes from the number of different object types. These results are in between the complete failure and complete success observed with other experiments, suggesting that the task could become solvable with better training or more training examples. Figure 3 illustrates the progress of training a tanh MLP, on both the training and test error, for Experiments 3 and 4. Clearly, MLPs were able to make significant progress on the task, but the task is far from being solved completely. On experiment 3 for both maxout and tanh, there was a long plateau where the training error and objective stays almost same. Maxout did just chance on the experiment for about 120 iterations on the training and the test set. But after $120^{\text{th}}$ iteration, the training and test error started to decline and eventually it was able to solve the task. Moreover as seen from the curves in Figure 3(a) and 3(b), the training and test error curves are almost the same for both tasks. This implies that for one-hot inputs, whether you increase the number of possible transformations for each object or the number of object categories, as soon as the number of possible configurations is same, the complexity of the problem is almost the same for the MLP.

| **Learning Algorithm** | **Training Error** | **Test Error** |
|---|---|---|
| SVM | 0.0 | 35.6 |
| Random Forests | 1.29 | 40.475 |
| Tanh MLP | 0.0 | 0.0 |
| Maxout MLP | 0.0 | 0.0 |

Table 1: Performance of different learning algorithms on disentangled representation in Experiment 2.

| Learning Algorithm | Training Error | Test Error |
|---|---|---|
| SVM | 11.212 | 32.37 |
| Random Forests | 24.839 | 48.915 |
| Tanh MLP | 0.0 | 22.475 |
| Maxout MLP | 0.0 | 0.0 |

Table 2: Performance of different learning algorithms using a dataset with one-hot vector and 80 inputs as discussed for Experiment 3.

| Learning Algorithm | Training Error | Test Error |
|---|---|---|
| SVM | 4.346 | 40.545 |
| Random Forests | 23.456 | 47.345 |
| Tanh MLP | 0 | 25.8 |

Table 3: Performance of different algorithms using a dataset with one-hot vector and 80 binary inputs as discussed in Experiment 4.

### 3.3 Learning Algorithms Evaluated

Initially the models are cross-validated by using 5-fold cross-validation. With $40,000$ examples, this gives $32,000$ examples for training and $8,000$ examples for testing. For training neural networks, we used stochastic gradient descent (SGD). The following standard learning algorithms were first evaluated: decision trees, SVMs with Gaussian kernel, ordinary fully-connected Multi-Layer Perceptrons, Random Forests, k-Nearest Neighbors, Convolutional Neural Networks, and Stacked Denoising Auto-Encoders with supervised fine-tuning. More details of the configurations and hyper-parameters for each of them are given in Appendix Section B. We obtained significantly better than chance results with only variations of the Structured Multi-Layer Perceptron that is described in this paper.

#### 3.3.1 Structured Multi-Layer Perceptron (SMLP)

The neural network architecture that is used to solve this task is called the SMLP (Structured Multi-Layer Perceptron), a deep neural network with two parts as illustrated in Figure 4 and 6.

The bottom part, P1NN (*Part 1 Neural Network*, as it is called in the rest of the paper), has shared weights and local connectivity, with one identical MLP instance of the P1NN for each patch of the image, and typically an 11-element output vector per patch (unless otherwise noted). The idea is that these 11 outputs per patch could represent the detection of the sprite shape category (or the absence of sprite in the patch). The upper part, P2NN (*Part 2 Neural Network*) is a fully connected one hidden layer MLP that takes as input the concatenation of the outputs of the patch-wise P1NNs over all the patches. Note that the first layer of P1NN is similar to a convolutional layer but where the stride equals the kernel size, such that windows do not overlap, that is, P1NN can be decomposed into separate networks sharing the same parameters but applied on different patches of the input image, so that each network can

actually be trained patch-wise in the case where a target is provided for the P1NN outputs. The P1NN output for patch $\mathbf{p_i}$ which is extracted from the image $\mathbf{x}$ is computed as follows:

$$\mathrm{f}_\theta(\mathbf{p_i}) = \mathrm{g}_2(\mathbf{V}\mathrm{g}_1(\mathbf{U}\mathbf{p_i} + \mathbf{b}) + \mathbf{c}) \tag{1}$$

where $\mathbf{p_i} \in \mathcal{R}^d$ is the input patch/receptive field extracted from location $i$ of a single image. $\mathbf{U} \in \mathcal{R}^{d_h \times d}$ is the weight matrix for the first layer of P1NN and $\mathbf{b} \in \mathcal{R}_h^d$ is the vector of biases for the first layer of P1NN. $\mathrm{g}_1(\cdot)$ is the activation function of the first layer and $\mathrm{g}_2(\cdot)$ is the activation function of the second layer. In many of the experiments, best results were obtained with $\mathrm{g}_1(\cdot)$ a rectified linear unit as non-linearity (a.k.a. as ReLU), which is $max(0,\ X)$ (Jarrett et al., 2009; Nair and Hinton, 2010; Glorot et al., 2011). $\mathbf{V} \in \mathcal{R}^{d_h \times d_o}$ is the second layer's weights matrix and $\mathbf{c} \in \mathcal{R}_{d_o}$ are the biases of the second layer of the P1NN, with $d_o$ expected to be smaller than $d_h$.

In this way, $\mathrm{g}_1(\mathbf{U}\mathbf{p_i} + \mathbf{b})$ is an overcomplete representation of the input patch that can potentially represent all the possible Pentomino shapes for all factors of variations in the patch (rotation, scaling and Pentomino shape type). On the other hand, when trained with hints, $\mathrm{f}_\theta(\mathbf{p_i})$ is expected to be the lower dimensional representation of a Pentomino shape category invariant to scaling and rotation in the given patch.

In the experiments with SMLP trained with hints the P1NN is applied to each $8 \times 8$ patch and expected to predict it shape category (or the absence of a Pentomino object in the patch). The input representation of P2NN is obtained from the concatenated output of P1NN across all the 64 patch locations:

$\mathbf{h_o} = [\mathrm{f}_\theta(\mathbf{p_0}), \cdots, \mathrm{f}_\theta(\mathbf{p_i}), \cdots, \mathrm{f}_\theta(\mathbf{p_N}))]$ where $N$ is the number of patches and the $\mathrm{h}_o \in \mathcal{R}^{d_i}, d_i = d_o \times N$. $\mathbf{h_o}$ is the concatenated output of the P1NN at each patch.

$\mathbf{h_o}$ is standardized for each training and test sample separately, so that the per-feature mean and variance across examples are respectively 0 and 1. That standardization layer seems to be crucial for successfully training the SMLP. Details of the standardization layer are given in Section A.

The concatenated output of P1NN is fed as an input to the P2NN. P2NN is a feedforward MLP with a sigmoid output layer using a single rectifier hidden layer. The task of P2NN is to perform a nonlinear logical operation on the representation provided at the output of P1NN.

#### 3.3.2 Structured Multi Layer Perceptron Trained with Hints (SMLP-hints)

The SMLP-hints architecture exploits a hint about the presence and category of Pentomino objects, specifying a semantics for the P1NN outputs. P1NN is trained with the intermediate target $Y$, specifying the type of Pentomino sprite shape present (if any) at each of the 64 patches ($8 \times 8$ non-overlapping blocks) of the image. Because a possible answer at a given location can be "none of the object types" that is, an empty patch, $y_p$ (for patch $p$) can take one of the 11 possible values, 1 for rejection and the rest for the Pentomino shape classes, illustrated in Figure 1:

$$y_p = \begin{cases} 0 & \text{if patch } p \text{ is empty} \\ s \in S & \text{if the patch } p \text{ contains a Pentomino sprite .} \end{cases}$$

A similar task has been studied by Fleuret et al. (2011) (at SI appendix Problem 17), who compared the performance of humans vs computers.

The SMLP-hints architecture takes advantage of dividing the task into two subtasks during training with prior information about intermediate-level relevant factors. Because the sum of the training losses decomposes into the loss on each patch, the P1NN can be pre-trained patch-wise. Each patch-specific component of the P1NN is a fully connected MLP with $8 \times 8$ inputs and 11 outputs with a softmax output layer. SMLP-hints uses the standardization given in Equation 8 but with $\epsilon = 0$.

In general, standardization layer for SMLP seems to make the training much easier and the distribution of activations of P1NN becomes much more peaky.

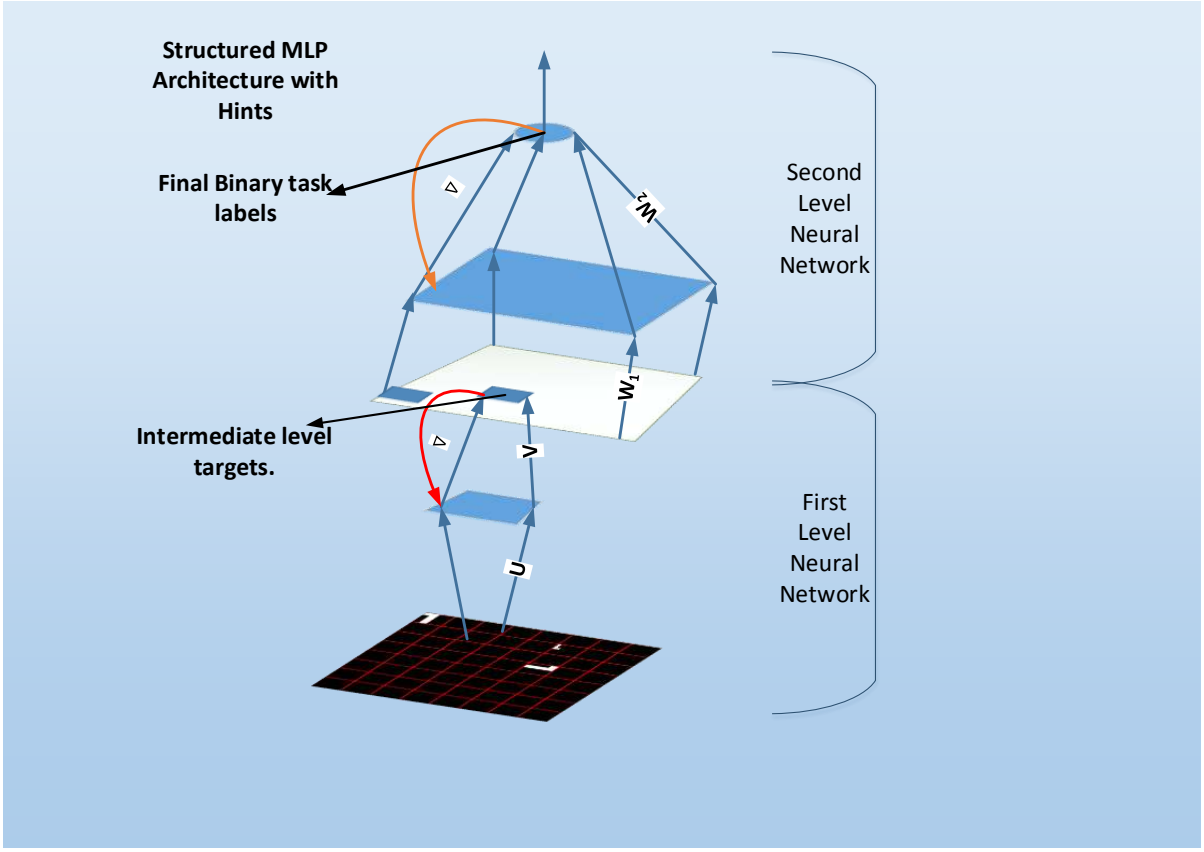



Figure 4: Structured MLP architecture, used with hints (trained in two phases, first P1NN, bottom two layers, then P2NN, top two layers). In SMLP-hints, P1NN is trained on each $8x8$ patch extracted from the image and the softmax output probabilities of all 64 patches are concatenated into a $64 \times 11$ vector that forms the input of P2NN. Only $\mathbf{U}$ and $\mathbf{V}$ parameters are learned in the P1NN and its output on each patch is fed into P2NN. The first level and the second level neural networks are trained separately, not jointly.

By default, the SMLP uses rectifier hidden units as activation function: we found a significant boost by using rectifier compared to hyperbolic tangent and sigmoid activation functions. The P1NN has a highly overcomplete architecture with 1024 hidden units per patch, and L1 and L2 weight decay regularization coefficients on the weights (not the biases) are respectively $1e-6$ and $1e-5$. The learning rate for the P1NN is 0.75. 1 training epoch was enough for the P1NN to learn the features of Pentomino shapes perfectly on the 40000 training examples. The

P2NN has 2048 hidden units. L1 and L2 penalty coefficients for the P2NN are 1e-6, and the learning rate is 0.1. These were selected by trial and error based on validation set error. Both P1NN (for each patch) and P2NN are fully-connected neural networks, even though P1NN globally is a special kind of convolutional neural network.

Filters of the first layer of SMLP are shown in Figure 5. These are the examples of the filters obtained with the SLMP-hints trained with 40k examples, whose results are given in Table 4. Those filters look very noisy but they work perfectly on the Pentomino task.

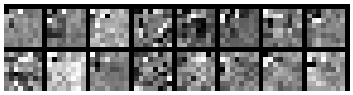


Figure 5: Filters of Structured MLP architecture, trained with hints on 40k examples.

#### 3.3.3 Deep and Structured Supervised MLP without Hints (SMLP-nohints)

SMLP-nohints uses the same connectivity pattern (and deep architecture) that is also used in the SMLP-hints architecture, but without using the intermediate targets ($Y$). It directly predicts the final outcome of the task ($Z$), using the same number of hidden units, the same connectivity and the same activation function for the hidden units as SMLP-hints. 120 hyperparameter values have been evaluated by randomly selecting the number of hidden units from $[64, 128, 256, 512, 1024, 1200, 2048]$ and randomly sampling 20 learning rates uniformly in the log-domain within the interval of $[0.008, 0.8]$. Two fully connected hidden layers with 1024 hidden units (same as P1NN) per patch is used and 2048 (same as P2NN) for the last hidden layer, with twenty training epochs. For this network the best results are obtained with a learning rate of 0.05. The source code of the structured MLP is available at the GitHub repository: `https://github.com/caglar/structured_mlp`.

We decided to experiment with various SMLP-nohint architectures and optimization procedures, trying unsuccessfully to achieve as good results with SMLP-nohint as with SMLP-hints.

*Rectifier Non-Linearity* A rectifier nonlinearity is used for the activations of MLP hidden layers. We observed that using piecewise linear nonlinearity activation function such as the rectifier can make the optimization more tractable.

*Intermediate Layer*

The output of the P1NN is considered as an intermediate layer of the SMLP. For the SMLP-hints, only softmax output activations have been tried at the intermediate layer, and that sufficed to learn the task. Since things did not work nearly as well with the SMLP-nohints, several different activation functions have been tried: softmax$(\cdot)$, tanh$(\cdot)$, sigmoid$(\cdot)$ and linear activation functions.

*Adaptive Learning Rates* We have experimented with several different adaptive learning rate algorithms. We tried RMSprop (Tieleman and Hinton, 2012), Adadelta (Zeiler, 2012), Adagrad (Duchi et al., 2011) and a linearly $(\frac{1}{t})$ decaying learning rate (Bengio, 2013b). For the SMLP-nohints with sigmoid activation function we have found Adadelta(Zeiler, 2012) converging faster to an effective local minimum and usually yielding better generalization error compared to the others.

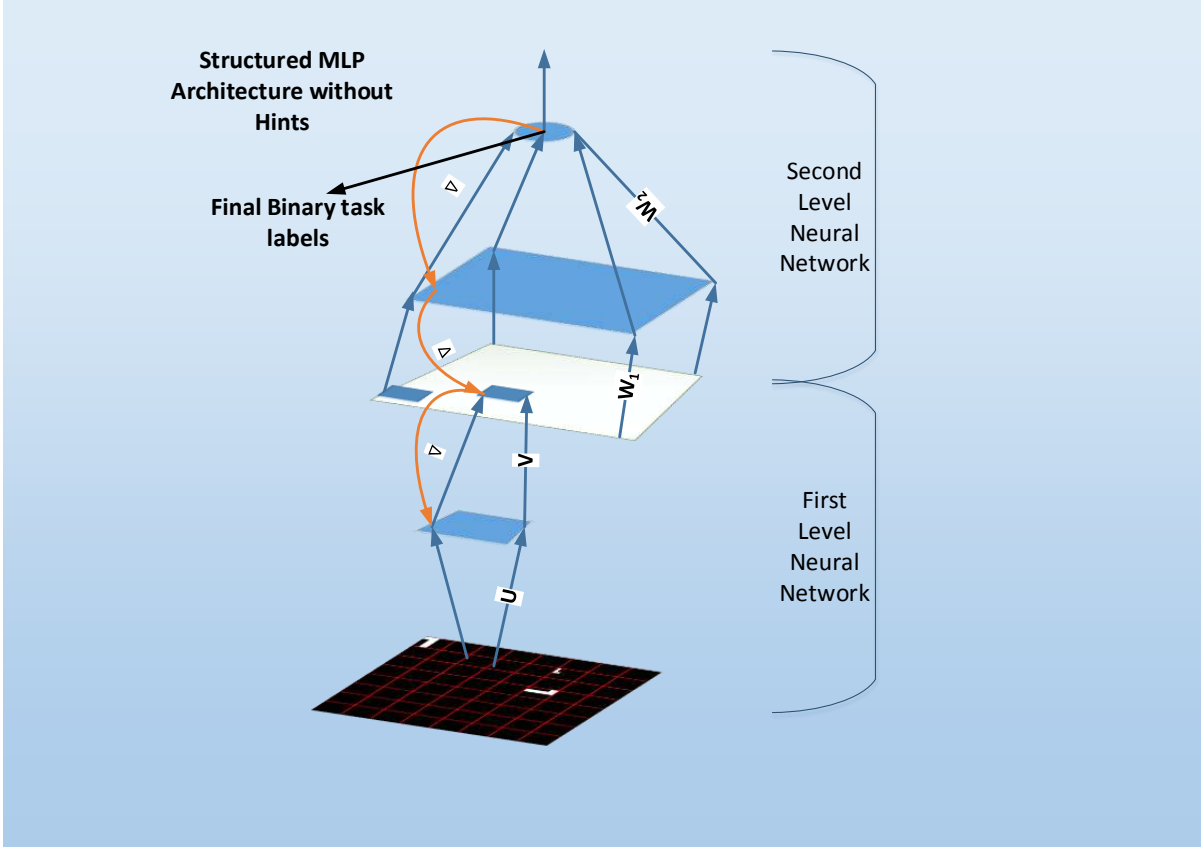



Figure 6: Structured MLP architecture, used without hints (SMLP-nohints). It is the same architecture as SMLP-hints (Figure 4) but with both parts (P1NN and P2NN) trained jointly with respect to the final binary classification task.

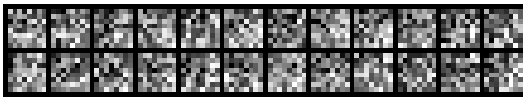


Figure 7: First layer filters learned by the Structured MLP architecture, trained without using hints on 447600 examples with online SGD and a sigmoid intermediate layer activation.

#### 3.3.4 Deep and Structured MLP with Unsupervised Pre-Training

Several experiments have been conducted using an architecture similar to the SMLP-nohints, but by using unsupervised pre-training of P1NN, with Denoising Auto-Encoders (DAE) and/or Contractive Auto-Encoders (CAE). Supervised fine-tuning proceeds as in the deep and structured MLP without hints. Because an unsupervised learner may not focus the representation just on the shapes, a larger number of intermediate-level units at the output of P1NN has been explored: previous work on unsupervised pre-training generally found that larger hidden layers were optimal when using unsupervised pre-training, because not all unsupervised features will be relevant to the task at hand. Instead of limiting to 11 units per patch, we experimented with networks with up to 20 hidden (i.e., code) units per patch in the second-layer patch-wise auto-encoder.

In Appendix B we also provided the result of some experiments with binary-binary RBMs trained on $8 \times 8$ patches from the 40k training dataset.

For the second P1NN layer, a DAE with rectifier hidden units was trained with L1 sparsity and weight decay on the weights of the auto-encoder. The greedy layerwise unsupervised pre-training procedure is used to train the deep auto-encoder architecture (Bengio et al., 2007). In unsupervised pretraining experiments, tied weights have been used. Different combinations of CAE and DAE for unsupervised pretraining have been tested, but none of the configurations tested managed to learn the Pentomino task, as shown in Table 4. In terms of generalization, without hints for the permutation-invariant Pentomino dataset, an MLP using $L_p$ units introduced in (Gulcehre et al., 2013) is the best performing model in Table 4.

| **Algorithm** | **20k dataset** | | **40k dataset** | | **80k dataset** | |
|---|---|---|---|---|---|---|
| | **Training Error** | **Test Error** | **Training Error** | **Test Error** | **Training Error** | **Test Error** |
| **SVM RBF** | 26.2 | 50.2 | 28.2 | 50.2 | 30.2 | 49.6 |
| **K Nearest Neighbors** | 24.7 | 50.0 | 25.3 | 49.5 | 25.6 | 49.0 |
| **Decision Tree** | 5.8 | 48.6 | 6.3 | 49.4 | 6.9 | 49.9 |
| **Randomized Trees** | 3.2 | 49.8 | 3.4 | 50.5 | 3.5 | 49.1 |
| **MLP** | 26.5 | 49.3 | 33.2 | 49.9 | 27.2 | 50.1 |
| **Convnet/Lenet5** | 50.6 | 49.8 | 49.4 | 49.8 | 50.2 | 49.8 |
| **Maxout Convnet** | 14.5 | 49.5 | 0.0 | 50.1 | 0.0 | 44.6 |
| **2 layer sDA** | 49.4 | 50.3 | 50.2 | 50.3 | 49.7 | 50.3 |
| **Supervised SMLP-nohints** | 0.0 | 48.6 | 0.0 | 36.0 | 0.0 | 12.4 |
| **Large SMLP-nohints** | - | - | - | - | 5.3 | 6.7 |
| **SMLP-nohints+CAE Supervised Finetuning** | 50.5 | 49.7 | 49.8 | 49.7 | 50.3 | 49.7 |
| **SMLP-nohints+CAE+DAE, Supervised Finetuning** | 49.1 | 49.7 | 49.4 | 49.7 | 50.1 | 49.7 |
| **SMLP-nohints+DAE+DAE, Supervised Finetuning** | 49.5 | 50.3 | 49.7 | 49.8 | 50.3 | 49.7 |
| **$L_p$ Units** | 0.0 | 50.5 | 0.0 | 45.0 | 0.4 | 31.85 |
| **SMLP with Hints** | **0.21** | **30.7** | **0** | **3.1** | **0** | **0.01** |

Table 4: The error percentages with different learning algorithms on Pentomino dataset with different number of training examples.

### 3.4 Does the Effect Persist with Larger Training Set Sizes?

The results shown in this section indicate that the problem in the Pentomino task clearly is not just a regularization problem, but also involves an optimization difficulty. This is suggested by the experiments in which the training set size is increased (eventually to the point of doing online learning and never repeating the same example twice), without solving the problem. In the online mini-batch setting, parameter updates are performed as follows:

$$\theta_{t+1} = \theta_t - \Delta_{\theta_t}, \tag{2}$$

$$\Delta_{\theta_t} = \epsilon \frac{\sum_i^N \nabla_{\theta_t} \mathcal{L}(x_i, \theta_t)}{N}. \tag{3}$$

where $L(x_i, \theta_t)$ is the loss incurred on the $i$-th example $x_i$ of the minibatch, with parameters $\theta_t$, $t \in \mathcal{Z}^+$, $N$ is the number of examples in the mini-batch, and $\epsilon$ is the learning rate.

Ordinary batch learning algorithms converge linearly to the optimum $\theta^*$, however the noisy gradient estimates in the online SGD will cause parameter $\theta$ to fluctuate near the local optima. On the other hand, online SGD directly optimizes the expected risk, because the examples are drawn iid from the ground-truth distribution (Bottou, 2010). Thus:

$$\mathcal{L}_\infty = \mathrm{E}[\mathcal{L}(\mathbf{x}, \theta)] = \int_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \theta) p(\mathbf{x}) d\mathbf{x}, \tag{4}$$

where $\mathcal{L}_\infty$ is the generalization error. Therefore online SGD is trying to minimize the expected risk with noisy updates. Those noisy updates can have regularization effect:

$$\Delta_{\theta_t} = \epsilon \frac{\sum_t^N \nabla_{\theta_t} \mathcal{L}(x_t, \theta_t)}{N} = \epsilon \nabla_{\theta_t} \mathcal{L}(\mathbf{X}, \theta_t) + \epsilon \xi_t. \tag{5}$$

where $\nabla_{\theta_t} \mathcal{L}(\mathbf{X}, \theta_t)$ is the true gradient and $\xi_t$ is the zero-mean stochastic gradient "noise" due to computing the gradient over a finite-size mini-batch sample.

An SMLP-nohints model was trained by online SGD with a randomly generated online Pentomino data stream. We used Adadelta (Zeiler, 2012) on mini-batches of 100 examples. In the online SGD experiments, two SMLP-nohints that are trained with and without standardization at the intermediate layer have the same hyperparameters. The SMLP-nohints P1NN patch-wise submodel has 2048 hidden units and the SMLP intermediate layer has $1152 = 64 \times 18$ units. The intermediate layer nonlinearity is sigmoid. P2NN has 2048 hidden units.

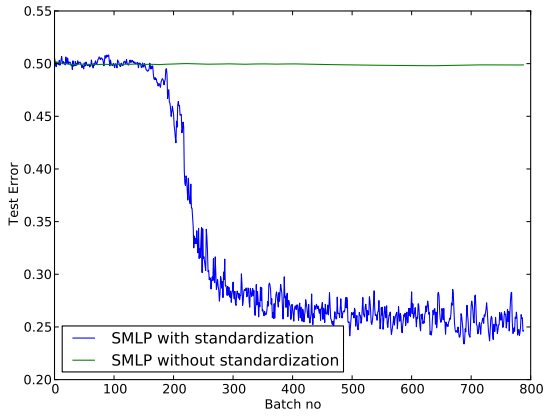
We trained SMLP-nohints both with and without standardization at the intermediate layer. The experiments illustrated in Figures 8(a) and 8(b) are using the same architecture as in the SMLP experiments reported in Table 4. The figures show the training and test results when training on the stream of 545400 randomly generated Pentomino samples.

Training of SMLP-nohints is done with mini-batch Adadelta and standardization in the intermediate layer, on 1046000 training examples from a randomly generated Pentomino data stream. At the end of the training, test error went down to 27.5%.

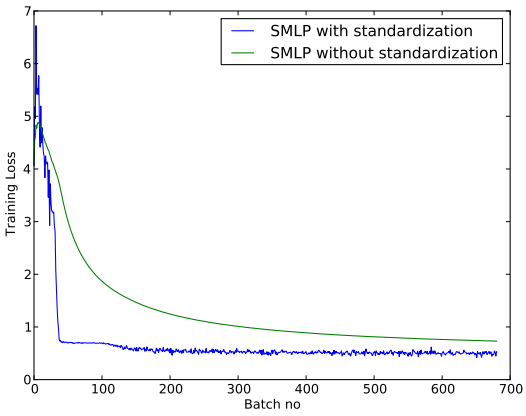
In SMLP-nohints experiment *without standardization*, the model is trained with the 1580000 Pentomino examples using online mini-batch SGD. P1NN has 2048 hidden units and 16 sigmoid output units per patch. P2NN has 1024 hidden units for the hidden layer. We used Adadelta to automatically adapt the learning rate. At the end of training this SMLP, the test error remained stuck, at 50.1%.

#### 3.4.1 Experiments with Larger Training Sets

We consider the effect of training different learners with different numbers of training examples. For the experimental results shown in Table 4, we used 3 training set sizes of 20k, 40k and 80k examples. We generated each dataset with different random seeds (so they do not overlap). Figure 9 also shows the error bars for an ordinary MLP with three hidden layers, for a larger range of training set sizes, between 40k and 320k examples. The number of training epochs is 8 (more did not help), and there are 3 hidden layers with 2048 feature detectors. Learning rate




(a) (i) Test Error of SMLP-nohints




(b) (ii) Training Error of SMLP-nohints

Figure 8: Online training and test errors of SMLP-nohints with and without standardization in the intermediate layer. Sigmoid nonlinearity has been used as an intermediate layer activation function. The x-axis is in units of blocks of 400 examples in the training set.

is 0.01 with tanh activation function. We used both L1, L2 penalty on weights and they are both $1e-6$.

Decision trees and SVMs can overfit the training set but they could not generalize on the test set. Note that the results reported in Table 4 are with hyper-parameters selected based on validation set error, and with early-stopping. Lower training errors are possible by avoiding regularizations and using large enough models. On the training set, an MLP with two large hidden layers (several thousands) could reach to almost 0% training error, but still it did not manage to achieve a good test error.

In the experimental results shown in Figure 9, we evaluated the impact of adding more training data for the fully-connected MLP. In those experiments, we used an MLP with three hidden layers where each layer has 2048 hidden units. The $\tanh(\cdot)$ activation function is used with 0.05 learning rate and mini-batches of size 200.

As seen on Figure 9, adding more training examples did not help for both training and test error. This is reinforcing the hypothesis that the difficulty encountered is one of optimization, not of regularization.

### 3.5 Experiments on Effect of Initializing with Hints

Initialization of the parameters in a neural network can have a big impact on the learning and generalization (Glorot and Bengio, 2010). Previously Erhan et al. (2010) showed that initializing the parameters of a neural network with unsupervised pretraining guides the learning towards basins of attraction of effective local minima that provide better generalization. In this section we analyze the effect of hints to initialize the SMLP, continuing training without hints. The SMLP is pretrained for 1 epoch using the hints and then for 60 epochs without hints on the 40k examples of training set. We also compared the same architecture with the same hyperparameters, against the SMLP-nohints trained for 61 iterations on the same dataset. After one iteration of hint-based training SMLP obtained 9% training error and 39% test error.

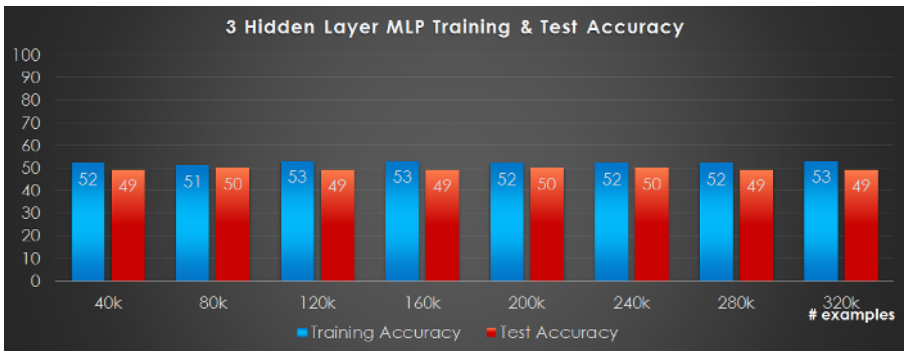



Figure 9: Training and test error bar charts for a regular MLP with 3 hidden layers. There is no significant improvement on the generalization error of the MLP as the new training examples are introduced.

Following the hint based training, SMLP is trained without hints for 60 epochs, but at epoch 18, it already got 0% training and 0% test error. The hyperparameters for this experiment and the experiment that the results shown for the SMLP-hints in Table 4 are the same. Figure 10 suggests that initializing with hints can give the same generalization performance but training takes longer.

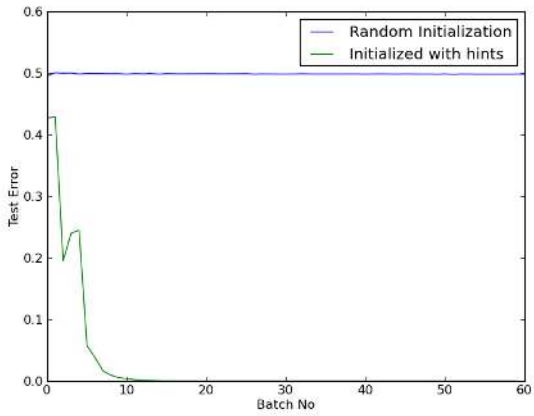



Figure 10: Plots showing the test error of SMLP with random initialization vs initializing with hint based training.

### 3.6 Experiments on Optimization for Pentomino Dataset

With extensive hyperparameter optimization and using standardization in the intermediate level of the SMLP with softmax nonlinearity, SMLP-nohints was able to get 5.3% training and 6.7% test error on the $80k$ Pentomino training dataset. The key ingredient was the use of a larger hidden layer. We used 2050 hidden units in the P1NN, 11 softmax outputs per patch, and 1024 hidden units in the P2NN. The network was trained with a learning rate 0.1 without using any adaptive learning rate. The SMLP uses a rectifier nonlinearity for hidden layers of both P1NN and P2NN. We also applied a small amount of $L_1$ and $L_2$ regularization on the weights of the network.

An MLP with 2 hidden layers, each 1024 rectifier units, was trained using **LBFGS** (the implementation from the scipy.optimize library) on $40k$ training examples, with gradients computed on batches of 10000 examples at each iteration. However, after convergence of training, the MLP was still doing chance on the test dataset.

In order to observe the effect of introducing the intermediate level hints into the SMLP for optimization, we compared the learning curves of the same SMLP model trained with and without hints. To train the SMLP with hints, we jointly optimize the loss for hints $\mathcal{L}_{\text{hints}}(\mathbf{X}; \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2\})$, and for the Pentomino task $\mathcal{L}_{\text{pentomino}}(\mathbf{X}; \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3\})$ together as a joint loss $\mathcal{L}_{\text{joint}}(\mathbf{X}; \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3\})$ as in Eqn 6:

$$\mathcal{L}_{\text{joint}}(\mathbf{X}; \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3\}) = \lambda \mathcal{L}_{\text{hints}}(\mathbf{X}; \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2\}) + \ (1 - \lambda)\mathcal{L}_{\text{pentomino}}(\mathbf{X}; \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_3\}). \tag{6}$$

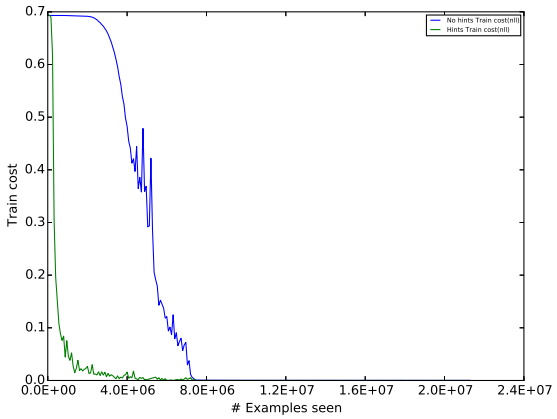



Figure 11: Plot showing the training learning curve of SMLP from random initialization when training with hints and without hints. SMLP training with hints converges faster than the training without hints. This means that the optimizing the training objective with hints is easier than optimizing without hints.

As seen on Figure 11 and 12, hint based training converges faster and the training of SMLP without hints overfits. This aligns with the hypothesis that hints make the optimization easier. But in the mean time, hints cost, puts the model into a basin of attraction to where a solution on the test set can easily be found. For the experiments in these plots, we cross-validated momentum, learning rate, standard deviation of the initialization and minibatch size. We trained the models with rectifier activation function using 512 hidden units at each layer. We trained the first level MLP on patches. It outputs 11 features for each such patch, with a softmax$(\cdot)$ activation function over the 11 units. We ran 64 different trials for the hyperparameters of the SMLP-hints. The learning rate is randomly sampled in log-space from the range $[5e-5, 1e-2]$, momentum is sampled from uniform distribution in the range $[0.25, 0.99]$, batch sizes and standard deviations of the weights are sampled from the grids $\{100, 200, 300, 400\}$ and $\{0.01, 0.025, 0.05\}$. We ran 150 different trials to explore hyperparameters for the SMLP-nohints. The learning rate is randomly sampled in log-space from the range $[6e-5, 8e-2]$, momentum is chosen randomly from the range $[0.25, 0.99]$, batch sizes and standard deviations of the weights are sampled from the grids $\{100, 200, 300, 400\}$ and $\{0.01, 0.025, 0.038, 0.05\}$ respectively for the without hints training. At the end of the hyper-parameter search, we chose the

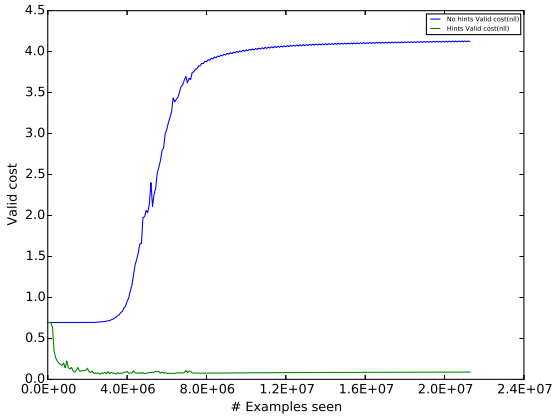



Figure 12: Plot showing validation learning curve of SMLP from random initialization when training with and without hints. Training of SMLP without hints overfits.

best hyper-parameters based on the training cost after 20000 updates. The codes to reproduce these experiments are available at `https://github.com/caglar/PentominoExps`.

We used a soft-curriculum strategy, for training our models. We started the training by only optimizing the easy hint objective function, namely setting $\lambda$ to 1 in the beginning of the training and linearly annealing it towards to 0.1 in 10000 updates.

## 4. Conclusion and Discussion

In this paper we have shown an example of a toy task which seems to be quiet difficult to solve by standard black-box machine learning algorithms, but it can be solved almost perfectly when one encourages a particular semantics on the intermediate-level representation which is guided by the prior knowledge about the task. Both tasks have the particularity that they are defined by the composition of two non-linear sub-tasks (for example, object detection on one hand, and a non-linear logical operation similar to XOR on the other hand).

What is interesting is that for neural networks, we can compare two networks with exactly the same architecture but a different pre-training, one of which uses the known intermediate concepts to teach an intermediate representation to the network. With enough capacity and training time they can overfit. But they failed solve the task, as seen by test set performances.

We know that a structured deep network can learn the task, if it is initialized in the right place, and it can do that from a relatively small number of examples. Furthermore we have shown that if one pre-trains the SMLP with hints for only one epoch, eventually it is able to solve the task. But the same architecture trained from random initialization failed to generalize as well as the pretrained network.

Considering the fact that even SMLP-nohints with standardization after being trained using online SGD over 1046000 generated examples and still gets 27.5% test error. This suggests that in addition to the optimization effect of hints, there is also a regularization effect, since a much larger training set allowed to reduce test error, albeit not as much as using hints and a much smaller training set.

We also investigated the effect of introducing hints on the convergence of the optimization of SMLP in Figure 11. After hyperparameter optimization, we observed that SMLP trained with hints converges on training set faster and achieves lower validation error as well. This supports our claims that the optimizing with hints cost is easier.

What we hypothesize is that for most initializations and architectures (in particular the fully-connected ones), although it is possible to find a *good effective local minimum of training error* when enough capacity is provided, it is difficult (without the proper initialization) to find a good local minimum of generalization error. On the other hand, when the network architecture is constrained enough, even though a good solution exists (e.g. with the SMLP architecture), it seems that the optimization problem can still be difficult and even training error remains stuck high without the standardization layer. Standardization makes the training objective of the SMLP easier to optimize and helps it to find at least a *better effective local minimum of training error*. This finding suggests that by using specific architectural constraints and sometimes domain specific knowledge about the problem, one can alleviate the optimization difficulty that generic neural network architectures face.

It could be that the combination of the network architecture and training procedure produces a training dynamics that tends to yield into these minima that are poor from the point of view of generalization error, even when they manage to slowly nail training error by providing enough capacity. Of course, as the number of examples increases, we would expect this discrepancy to decrease, but then the optimization problem could still make the task unfeasible in practice. Note however that our preliminary experiments with increasing the training set size (8-fold) for MLPs did not reveal signs of potential improvements in test error yet, as shown in Figure 9. Even using online training on 545400 Pentomino examples, the SMLP-nohints architecture was still doing far from perfect in terms of generalization error (Figure 8(a)).

These findings bring supporting evidence to the "Guided Learning Hypothesis" and "Deeper Harder Hypothesis" from Bengio (2013a): higher level abstractions, which are expressed by composing simpler concepts are more difficult to learn (with the learner often getting in an effective local minimum ), but that difficulty can be overcome if another agent provides hints about intermediate-level abstractions which are relevant to the task.

Many interesting questions remain open. Would a network without any guiding hint eventually find the 0% error solution provided enough training time and/or with alternate parameterizations? To what extent is ill-conditioning a core issue? The results with LBFGS were disappointing but changes in the architectures (such as standardization of the intermediate level) seem to make training much easier. Clearly, one can reach good solutions from an appropriate initialization, pointing in the direction of an issue with local minima, but it may be that good solutions are also reachable from other initializations, albeit going through a ill-conditioned path in parameter space. Why did our attempts at learning the intermediate concepts in an unsupervised way fail? Are these results specific to the task we are testing or a limitation of the unsupervised feature learning algorithm tested? Trying with many more unsupervised variants and exploring explanatory hypotheses for the observed failures could help us answer that. Finally, and most ambitious, can we solve these kinds of problems if we allow a community of learners to collaborate and collectively discover and combine partial solutions in order to obtain solutions to more abstract tasks like the one presented here? Indeed, we would like to discover learning algorithms that can solve such tasks without the use of prior knowledge as specific and strong as the one used in the SMLP here. These experiments could be

inspired by and inform us about potential mechanisms for collective learning through cultural evolutions in human societies.

## Acknowledgments

We would like to thank to the ICLR 2013 reviewers for their insightful comments, and NSERC, CIFAR, Compute Canada and Canada Research Chairs for funding.

## Appendix A. Standardization Layer

Normalization has been used occasionally in convolutional neural network to encourage the competition between the hidden units. (Jarrett et al., 2009) used a local contrast normalization layer in their architecture which performs subtractive and divisive normalization. A local contrast normalization layer enforces a local competition between adjacent features in the feature map and between features at the same spatial location in different feature maps. Similarly, (Krizhevsky et al., 2012) observed that using a local response layer that enjoys the benefit of using local normalization scheme aids generalization.

Standardization has been observed to be crucial for SMLP trained either with or without hints. In both SMLP-hints and SMLP-nohints experiments, the neural network was not able to generalize or even learn the training set without using standardization in the SMLP intermediate layer, doing just chance performance. More specifically, in the SMLP-nohints architecture, standardization is part of the computational graph, hence the gradients are being backpropagated through it. The mean and the standard deviation is computed for each hidden unit separately at the intermediate layer as in Equation 9. But in order to prevent numerical underflows during the backpropagation, we used $\epsilon = 1e - 8$ (Equation 8).

The benefit of having sparse activations may be specifically important for the ill-conditioned problems, for the following reasons. When a hidden unit is "off", its gradient (the derivative of the loss with respect to its activation before the non-linearity is applied) is usually close to 0 as well. That means that all off-diagonal second derivatives involving that hidden unit (for example, its input weights) are also near 0. This is basically like removing some columns and rows from the Hessian matrix associated with a particular example. It has been observed that the condition number of the Hessian matrix (specifically, its largest eigenvalue) increases as the size of the network increases (Dauphin and Bengio, 2013), making training considerably slower and inefficient. Hence one would expect that as sparsity of the gradients (obtained because of sparsity of the activations) increases, training would become more efficient, as if we were training a smaller sub-network for each example, with shared weights across examples, as in dropouts (Hinton et al., 2012).

Standardization layer is on top of the output of P1NN which centers the activations and performs divisive normalization by dividing by the standard deviation over a mini-batch of the activations of that layer. We denote the standardization function $z(\cdot)$. Standardization makes use of the mean and standard deviation computed for each hidden unit such that each hidden unit of $\mathbf{h_o}$ will have 0 activation and unit standard deviation on average over the mini-batch. $X$ is the set of pentomino images in the mini-batch, where $X \in R^{d_{in} \times N}$ is a matrix with $N$ images. $h_o^{(i)}(\mathbf{x_j})$ is the vector of activations of the $i$-th hidden unit of hidden layer $h_o(\mathbf{x_j})$ for the $j$-th example, with $x_j \in X$.

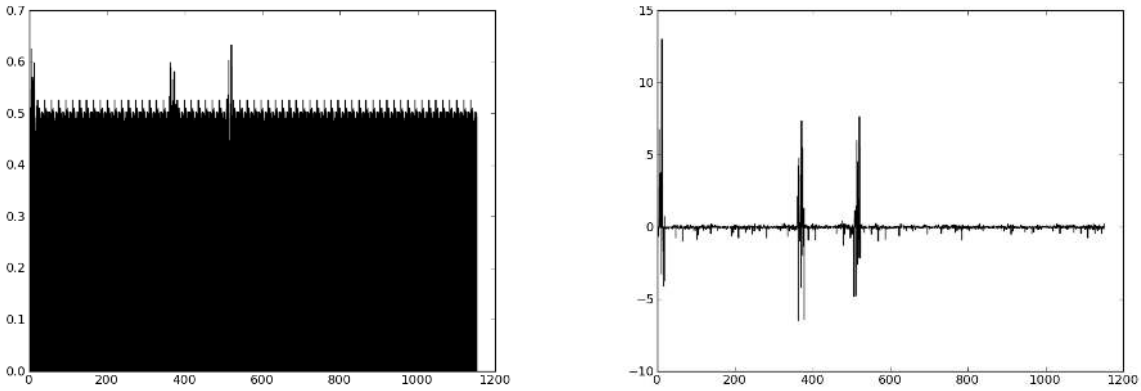
$$\mu_{h_o^{(i)}} = \frac{1}{N} \sum_{\mathbf{x_j} \in X} h_o^{(i)}(\mathbf{x_j}), \tag{7}$$

$$\sigma_{h_o^{(i)}} = \sqrt{\frac{\sum_j^N (h_o^{(i)}(\mathbf{x_j}) - \mu_{h_o^{(i)}})^2}{N} + \epsilon}, \tag{8}$$

$$z(h_o^{(i)}(\mathbf{x_j})) = \frac{h_o^{(i)}(\mathbf{x_j}) - \mu_{h_o^{(i)}}}{\max(\sigma_{h_o^{(i)}}, \epsilon)}. \tag{9}$$

where $\epsilon$ is a very small constant, that is used to prevent numerical underflows in the standard deviation. P1NN is trained on each $8 \times 8$ patches extracted from the image.

In Figure 13, the activation of each hidden unit in a bar chart is shown: the effect of standardization is significant, making the activations sparser.




(a) Before standardization.

(b) After standardization.

Figure 13: Activations of the intermediate-level hidden units of an SLMP-nohints for a particular examples (x-axis: hidden unit number, y-axis: activation value). Left (a): before standardization. Right (b): after standardization.

In Figure 15, one can see the activation histogram of the SMLP-nohints intermediate layer, showing the distribution of activation values, before and after standardization. Again the sparsifying effect of standardization is very apparent.

In Figures 15 and 13, the intermediate level activations of SMLP-nohints are shown before and after standardization. These are for the same SMLP-nohints architecture whose results are presented on Table 4. For that same SMLP, the Adadelta (Zeiler, 2012) adaptive learning rate scheme has been used, with 512 hidden units for the hidden layer of P1NN and rectifier activation function. For the output of the P1NN, 11 sigmoid units have been used while P2NN had 1200 hidden units with a rectifier activation function. The output nonlinearity of the P2NN is a sigmoid and the training objective is the binary cross-entropy.

## Appendix B. Binary-Binary RBMs on Pentomino Dataset

We trained binary-binary RBMs (both visible and hidden are binary) on 8×8 patches extracted from the Pentomino Dataset using PCD (stochastic maximum likelihood), a weight decay of .0001 and a sparsity penalty[2]. We used 256 hidden units and trained by SGD with a batch size of 32 and an annealing learning rate (Bengio, 2013b) starting from 1e-3 with annealing rate 1.000015. The RBM is trained with momentum starting from 0.5 to 0.9. The biases are initialized to -2 in order to get a sparse representation. The RBM is trained for 120 epochs (approximately 50 million updates).

After pretraining the RBM, its parameters are used to initialize the first layer of an SMLP-nohints network. As in the usual architecture of the SMLP-nohints on top of P1NN, there is an intermediate layer. Both P1NN and the intermediate layer have a sigmoid nonlinearity, and the intermediate layer has 11 units per location. This SMLP-nohints is trained with Adadelta and standardization at the intermediate layer [3].

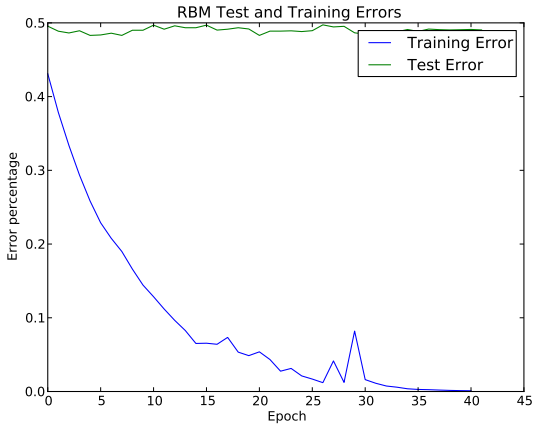


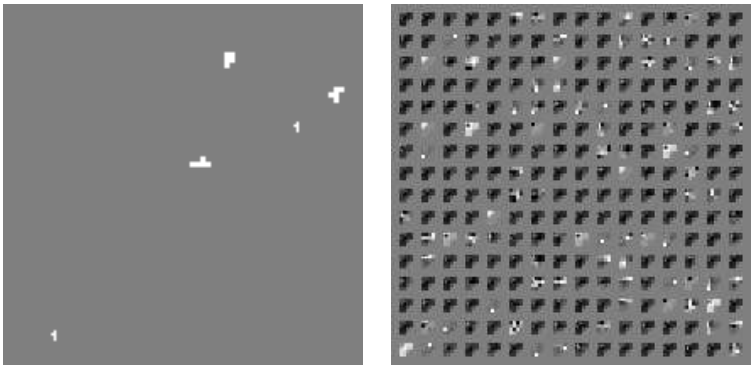

Figure 14: Training and test errors of an SMLP-nohints network whose first layer is pre-trained as an RBM. Training error reduces to 0% at epoch 42, but test error is still chance.

## Experimental Setup and Hyper-parameters

### B.1 Decision Trees

We used the decision tree implementation in the scikit-learn (Fabian Pedregosa, 2011) python package which is an implementation of the CART (Regression Trees) algorithm. The CART algorithm constructs the decision tree recursively and partitions the input space such that the samples belonging to the same category are grouped together (Breiman et al., 1984). We used The Gini index as the impurity criteria. We evaluated the hyper-parameter configurations with a grid-search. We cross-validated the maximum depth (*max_depth*) of the tree (for preventing the algorithm to severely overfit the training set) and minimum number of samples required to create a split (*min_split*). 20 different configurations of hyper-parameter values were evaluated. We obtained the best validation error with $max\_depth = 300$ and $min\_split = 8$.

2. implemented as TorontoSparsity in pylearn2, see the yaml file in the repository for more details
3. In our auto-encoder experiments we directly fed features to P2NN without standardization and Adadelta.



(a) Samples of RBM (b) Filters learned by RBM

Figure 15: Inspecting the representation learned by RBM. Left (a): 100 samples generated from trained RBM. All the generated samples are valid Pentomino shapes. Right (b): 100 Binary-Binary RBM Pentomino negative samples.

### B.2 Support Vector Machines

We used the "Support Vector Classifier (SVC)" implementation from the scikit-learn package which in turn uses the libsvm's Support Vector Machine (SVM) implementation. Kernel-based SVMs are non-parametric models that map the data into a high dimensional space and separate different classes with hyperplane(s) such that the support vectors for each category will be separated by a large margin. We cross-validated three hyper-parameters of the model using grid-search: $C$, $\gamma$ and the type of kernel($kernel\_type$). $C$ is the penalty term (weight decay) for the SVM and $\gamma$ is a hyper-parameter that controls the width of the Gaussian for the RBF kernel. For the polynomial kernel, $\gamma$ controls the flexibility of the classifier (degree of the polynomial) as the number of parameters increases (Hsu et al., 2003; Ben-Hur and Weston, 2010). We evaluated forty-two hyper-parameter configurations. That includes, two kernel types: $\{RBF,\ Polynomial\}$; three gammas: $\{1e-2,\ 1e-3,\ 1e-4\}$ for the RBF kernel, $\{1, 2, 5\}$ for the polynomial kernel, and seven $C$ values among: $\{0.1, 1, 2, 4, 8, 10, 16\}$. As a result of the grid search and cross-validation, we have obtained the best test error by using the RBF kernel, with $C = 2$ and $\gamma = 1$.

### B.3 Multi Layer Perceptron

We have our own implementation of Multi Layer Perceptron based on the Theano (Bergstra et al., 2010) machine learning libraries. We have selected 2 hidden layers, the rectifier activation function, and 2048 hidden units per layer. We cross-validated three hyper-parameters of the model using random-search, sampling the learning rates $\epsilon$ in log-domain, and selecting $L1$ and $L2$ regularization penalty coefficients in sets of fixed values, evaluating 64 hyperparameter values. The range of the hyperparameter values are $\epsilon \in [0.0001, 1]$, $L1 \in \{0., 1e-6, 1e-5, 1e-4\}$ and $L2 \in \{0, 1e-6, 1e-5\}$. As a result, the following were selected: $L1 = 1e-6$, $L2 = 1e-5$ and $\epsilon = 0.05$.

### B.4 Random Forests

We used scikit-learn's implementation of "Random Forests" decision tree learning. The Random Forests algorithm creates an ensemble of decision trees by randomly selecting for each tree a subset of features and applying bagging to combine the individual decision trees (Breiman, 2001). We have used grid-search and cross-validated the $max\_depth$, $min\_split$, and number of trees ($n\_estimators$). We have done the grid-search on the following hyperparameter values, $n\_estimators \in \{5, 10, 15, 25, 50\}$, $max\_depth \in \{100, 300, 600, 900\}$, and $min\_splits \in \{1, 4, 16\}$. We obtained the best validation error with $max\_depth = 300$, $min\_split = 4$ and $n\_estimators = 10$.

### B.5 k-Nearest Neighbors

We used scikit-learn's implementation of k-Nearest Neighbors (k-NN). k-NN is an instance-based, lazy learning algorithm that selects the training examples closest in Euclidean distance to the input query. It assigns a class label to the test example based on the categories of the $k$ closest neighbors. The hyper-parameters we have evaluated in the cross-validation are the number of neighbors ($k$) and $weights$. The $weights$ hyper-parameter can be either "uniform" or "distance". With "uniform", the value assigned to the query point is computed by the majority vote of the nearest neighbors. With "distance", each value assigned to the query point is computed by weighted majority votes where the weights are computed with the inverse distance between the query point and the neighbors. We have used $n\_neighbours \in \{1, 2, 4, 6, 8, 12\}$ and $weights \in \{"uniform", "distance"\}$ for hyper-parameter search. As a result of cross-validation and grid search, we obtained the best validation error with $k = 2$ and $weights$="uniform".

### B.6 Convolutional Neural Nets

We used a Theano (Bergstra et al., 2010) implementation of Convolutional Neural Networks (CNN) from the deep learning tutorial at `deeplearning.net`, which is based on a vanilla version of a CNN LeCun et al. (1998). Our CNN has two convolutional layers. Following each convolutional layer, we have a max-pooling layer. On top of the convolution-pooling-convolution-pooling layers there is an MLP with one hidden layer. In the cross-validation we have sampled 36 learning rates in log-domain in the range $[0.0001, 1]$ and the number of filters from the range $[10, 20, 30, 40, 50, 60]$ uniformly. For the first convolutional layer we used $9\times9$ receptive fields in order to guarantee that each object fits inside the receptive field. As a result of random hyperparameter search and doing manual hyperparameter search on the validation dataset, the following values were selected:

- The number of features used for the first layer is 30 and the second layer is 60.
- For the second convolutional layer, $7\times7$ receptive fields. The stride for both convolutional layers is 1.
- Convolved images are downsampled by a factor of $2\times2$ at each pooling operation.
- The learning rate for CNN is 0.01 and it was trained for 8 epochs.

### B.7 Maxout Convolutional Neural Nets

We used the pylearn2 (`https://github.com/lisa-lab/pylearn2`) implementation of maxout convolutional networks (Goodfellow et al., 2013). There are two convolutional layers in the selected architecture, without any pooling. In the last convolutional layer, there is a maxout non-linearity. The following were selected by cross-validation: learning rate, number of channels for the both convolution layers, number of kernels for the second layer and number of units and pieces per maxout unit in the last layer, a linearly decaying learning rate, momentum starting from 0.5 and saturating to 0.8 at the 200'th epoch. Random search for the hyperparameters was used to evaluate 48 different hyperparameter configurations on the validation dataset. For the first convolutional layer, 8×8 kernels were selected to make sure that each Pentomino shape fits into the kernel. Early stopping was used and test error on the model that has the best validation error is reported. Using norm constraint on the fan-in of the final softmax units yields slightly better result on the validation dataset.

As a result of cross-validation and manually tuning the hyperparameters we used the following hyperparameters:

- 16 channels per convolutional layer. 600 hidden units for the maxout layer.
- 6x6 kernels for the second convolutional layer.
- 5 pieces for the convolution layers and 4 pieces for the maxout layer per maxout units.
- We decayed the learning rate by the factor of 0.001 and the initial learning rate is 0.026367. But we scaled the learning rate of the second convolutional layer by a constant factor of 0.6.
- The norm constraint (on the incoming weights of each unit) is 1.9365.

Figure 16 shows the first layer filters of the maxout convolutional net, after being trained on the 80k training set for 85 epochs.

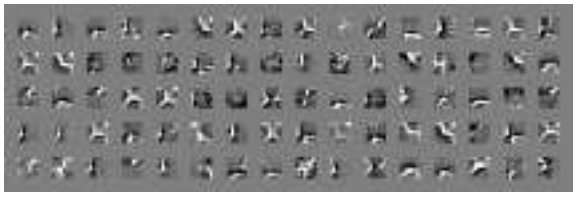


Figure 16: Maxout convolutional net first layer filters. Most of the filters were able to learn the basic edge structure of the Pentomino shapes.

### B.8 Stacked Denoising Auto-Encoders

Denoising Auto-Encoders (DAE) are a form of regularized auto-encoder (Bengio et al., 2013). The DAE forces the hidden layer to discover more robust features and prevents it from simply learning the identity by reconstructing the input from a corrupted version of it (Vincent et al., 2010). Two DAEs were stacked, resulting in an unsupervised transformation with two hidden layers of 1024 units each. Parameters of all layers are then fine-tuned with supervised fine-tuning using logistic regression as the classifier and SGD as the gradient-based optimization

algorithm. The stochastic corruption process is binomial (0 or 1 replacing each input value, with probability 0.2). The selected learning rate is $\epsilon_0 = 0.01$ for the DAe and $\epsilon_1 = 0.1$ for supervised fine-tuning. Both L1 and L2 penalty for the DAEs and for the logistic regression layer are set to 1e-6.

#### B.8.1 CAE+MLP with Supervised Finetuning:

A regularized auto-encoder which sometimes outperforms the DAE is the Contractive Auto-Encoder (CAE), (Rifai et al., 2012), which penalizes the Frobenius norm of the Jacobian matrix of derivatives of the hidden units with respect to the CAE inputs. The CAE serves as pre-training for an MLP, and in the supervised fine-tuning state, the Adagrad method was used to automatically tune the learning rate (Duchi et al., 2011).

After training a CAE with 100 sigmoidal units patch-wise, the features extracted on each patch are concatenated and fed as input to an MLP. The selected Jacobian penalty coefficient is 2, the learning rate for pre-training is 0.082 with batch size of 200 and 200 epochs of unsupervised learning are performed on the training set. For supervised finetuning, the learning rate is 0.12 over 100 epochs, L1 and L2 regularization penalty terms respectively are 1e-4 and 1e-6, and the top-level MLP has 6400 hidden units.

#### B.8.2 Greedy Layerwise CAE+DAE Supervised Finetuning:

For this experiment we stack a CAE with sigmoid non-linearities and then a DAE with rectifier non-linearities during the pre-training phase. As recommended by Glorot et al. (2011) we have used a softplus nonlinearity for reconstruction, $softplus(x) = log(1 + e^x)$. We used an L1 penalty on the rectifier outputs to obtain a sparser representation with rectifier non-linearity and L2 regularization to keep the non-zero weights small.

The main difference between the DAE and CAE is that the DAE yields more robust reconstruction whereas the CAE obtains more robust features (Rifai et al., 2011).

As seen on Figure 6 the weights U and V are shared on each patch and we concatenate the outputs of the last auto-encoder on each patch to feed it as an input to an MLP with a large hidden layer.

We used 400 hidden units for the CAE and 100 hidden units for DAE. The learning rate used for the CAE is 0.82 and for DAE it is 9*1e-3. The corruption level for the DAE (binomial noise) is 0.25 and the contraction level for the CAE is 2.0. The L1 regularization penalty for the DAE is 2.25*1e-4 and the L2 penalty is 9.5*1e-5. For the supervised finetuning phase the learning rate used is 4*1e-4 with L1 and L2 penalties respectively 1e-5 and 1e-6. The top-level MLP has 6400 hidden units. The auto-encoders are each trained for 150 epochs while the whole MLP is fine-tuned for 50 epochs.

#### B.8.3 Greedy Layerwise DAE+DAE Supervised Finetuning:

For this architecture, we have trained two layers of denoising auto-encoders greedily and performed supervised finetuning after unsupervised pre-training. The motivation for using two denoising auto-encoders is the fact that rectifier nonlinearities work well with the deep networks but it is difficult to train CAEs with the rectifier non-linearity. We have used the same type of denoising auto-encoder that is used for the greedy layerwise CAE+DAE supervised finetuning experiment.

In this experiment we have used 400 hidden units for the first layer DAE and 100 hidden units for the second layer DAE. The other hyperparameters for DAE and supervised finetuning are the same as with the *CAE+DAE MLP Supervised Finetuning* experiment.

## References

*Journal of Machine Learning Research*, -1.

Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379, 2003.

Asa Ben-Hur and Jason Weston. A user's guide to support vector machines. *Methods in Molecular Biology*, 609:223–239, 2010.

Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009. Also published as a book. Now Publishers, 2009.

Yoshua Bengio. Evolving culture vs local minima. In *Growing Adaptive Machines: Integrating Development and Learning in Artificial Neural Networks*, number also as ArXiv 1203.2990v1, pages T. Kowaliw, N. Bredeche & R. Doursat, eds. Springer-Verlag, March 2013a. URL `http://arxiv.org/abs/1203.2990`.

Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In K.-R. Müller, G. Montavon, and G. B. Orr, editors, *Neural Networks: Tricks of the Trade*. Springer, 2013b.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In Bernhard Schölkopf, John Platt, and Thomas Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS'06)*, pages 153–160. MIT Press, 2007.

Yoshua Bengio, Jerome Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In Léon Bottou and Michael Littman, editors, *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML'09)*. ACM, 2009.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. Technical Report arXiv:1206.5538, U. Montreal, 2012. URL `http://arxiv.org/abs/1206.5538`.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Unsupervised feature learning and deep learning: A review and new perspectives. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 2013.

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)*, 2010.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. Classification and regression trees. *Belmont, Calif.: Wadsworth*, 1984.

Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks. *AISTATS 2015, Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 192–204, 2015.

Dan C. Ciresan, Ueli Meier, Luca M. Gambardella, and Jürgen Schmidhuber. Deep big simple neural nets for handwritten digit recognition. *Neural Computation*, 22:1–14, 2010.

Yann Dauphin and Yoshua Bengio. Big neural networks waste capacity. Technical Report arXiv:1301.3583, Universite de Montreal, 2013.

Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS'2014*, 2014.

Richard Dawkins. *The Selfish Gene*. Oxford University Press, 1976.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? In *Journal of Machine Learning Research* JML (-1), pages 625–660.

Alexandre Gramfort Vincent Michel Bertrand Thirion Olivier Grisel Mathieu Blondel et al Fabian Pedregosa, Gal Varoquaux. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

Franois Fleuret, Ting Li, Charles Dubout, Emma K. Wampler, Steven Yantis, and Donald Geman. Comparing machines and humans on a visual categorization test. *Proceedings of the National Academy of Sciences*, 108(43):17621–17625, 2011.

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

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, April 2011.

Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In *ICML'2013*, 2013.

Caglar Gulcehre, Kyunghyun Cho, Razvan Pascanu, and Yoshua Bengio. Learned-norm pooling for deep neural networks. *arXiv preprint arXiv:1311.1780*, 2013.

Joseph Henrich and Richard McElreath. The evolution of cultural evolution. *Evolutionary Anthropology: Issues, News, and Reviews*, 12(3):123–135, 2003.

Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. Technical report, arXiv:1207.0580, 2012.

Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification, 2003.

Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*, pages 2146–2153. IEEE, 2009.

Faisal Khan, Xiaojin Zhu, and Bilge Mutlu. How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems 24 (NIPS'11)*, pages 1449–1457, 2011.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'2012)*. 2012.

Kai A. Krueger and Peter Dayan. Flexible shaping: how learning in small steps helps. *Cognition*, 110:380–394, 2009.

Gautam Kunapuli, Kristin P. Bennett, Richard Maclin, and Jude W. Shavlik. The adviceptron: Giving advice to the perceptron. *Proceedings of the Conference on Artificial Neural Networks In Engineering (ANNIE 2010)*, 2010.

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

Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Tom M. Mitchell. *The Need for Biases in Learning Generalizations*. Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ., 1980.

Tom M. Mitchell and Sebastian B. Thrun. Explanation-based neural network learning for robot control. *Advances in Neural information processing systems*, pages 287–287, 1993.

Richard Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

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.

Joseph O'Sullivan. Integrating initialization bias and search bias in neural network learning, 1996.

Gail B. Peterson. A day of great illumination: B. F. Skinner's discovery of shaping. *Journal of the Experimental Analysis of Behavior*, 82(3):317–328, 2004.

Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of theTwenty-eight International Conference on Machine Learning (ICML'11)*, June 2011.

Salah Rifai, Yoshua Bengio, Yann Dauphin, and Pascal Vincent. A generative process for sampling contractive auto-encoders. In *Proceedings of the Twenty-nine International Conference on Machine Learning (ICML'12)*. ACM, 2012. URL `http://icml.cc/discuss/2012/590.html`.

Ruslan Salakhutdinov and Geoffrey E Hinton. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 448–455, 2009.

Burrhus F. Skinner. Reinforcement today. *American Psychologist*, 13:94–99, 1958.

Ray J. Solomonoff. A system for incremental learning based on algorithmic probability. In *Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*, pages 515–527. Citeseer, 1989.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 2012.

Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial intelligence*, 70(1):119–165, 1994.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. In *Journal of Machine Learning Research* JML (-1), pages 3371–3408.

Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. Captcha: Using hard ai problems for security. In *Advances in CryptologyEUROCRYPT 2003*, pages 294–311. Springer, 2003.

Jason Weston, Frédéric Ratle, and Ronan Collobert. Deep learning via semi-supervised embedding. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *Proceedings of the Twenty-fifth International Conference on Machine Learning (ICML'08)*, pages 1168–1175, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390303.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.