Inspecting neural network internals using generated toy data

Gerben <gerben@treora.com>

6th April 2015

NOTE: This is a paper-compatible version of this publication. Due to physical limitations of paper, the included code examples cannot be executed. To reproduce and modify the experiments, access the full publication at http://archive.treora.com/2015/toydata.

Abstract

This student project report promotes a generic method for evaluating, and gaining insight in, neural network algorithms by using toy data sets for which descriptive 'desired' features have been predetermined. The desired features correspond to latent variables used to generate the data, the values of which are treated as 'hidden labels'. The idea is that a trained network's neural activations can be compared to the hidden labels, to give insight in what exactly the network learns, contrasted with what we expect or desire it to learn. The level of correspondence can be used as a quality measure for the learning algorithm. Bypassing the learning algorithm, the network's parameters can also be configured more directly using the hidden labels, in order to check whether the model actually possesses the representational power required to discriminate the desired features.

An important challenge is the design of toy data generators, which should produce toy data sets of low dimensionality while possessing properties found in real world data. Experiments have been performed with toy data consisting of 20-dimensional binary vectors, generated using two layers of latent variables. The data has been applied to a two-layer stacked RBM, and to an MLP, providing valuable insights in how they react to the statistics of their training data.

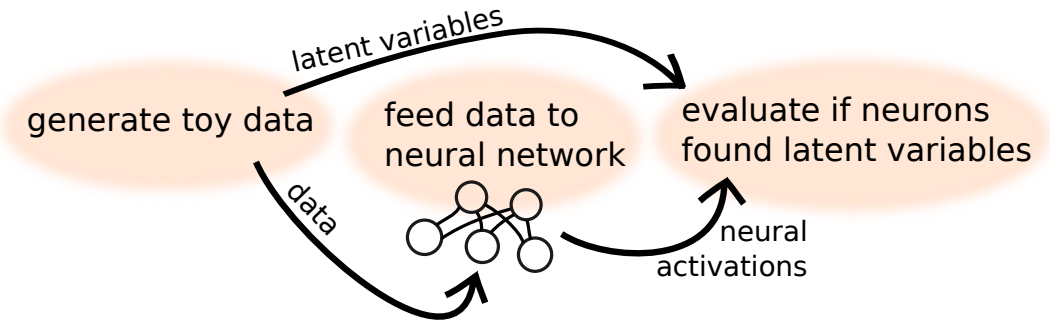

Figure 1: The general idea.

1 Introduction

In the development of neural network algorithms, simplified data sets are commonly used to evaluate an algorithm's performance. A well-known example is the MNIST data set, consisting of monochrome images each picturing one hand-written digit. However, even such a 'toy' data set is too complex and high-dimensional to fully understand the data: digits only look so simple to us because we already possess a superb visual system ourselves. It is difficult

to evaluate if the algorithm is learning things sensibly when lacking an understanding of what would be sensible. We can test whether the trained network correctly classifies the digits, but that is a crude measurement that gives no insight in the network's internal behaviour. We can inspect weight matrices, but we cannot do much more than visualising some features and mumbling "yes, that looks a bit like the corner of a seven..". Especially problematic are unsupervised algorithms, which are often evaluated by measuring whether a supervised algorithm initialised with the found weights will give good classification results. This shortage of ways to evaluate more precisely how well the algorithm is learning feeds the longing for having data sets for which we already know 'desired' features: features that describe the data well, and *that we expect a reasonably intelligent algorithm to learn to detect*. Training our algorithm on such a dataset would enable us to inspect the learned features and compare them to those we hoped to find. By keeping the dimensionality of this data small, it becomes feasible to manually track and understand the algorithm's behaviour, inspect weight matrices and neural activations, and possibly obtain valuable insights for improving the algorithm.

Of course, this method relies on many assumptions and in many situations it may not be applicable or useful. It only works for testing fundamental principles of an algorithm, and not at all for tweaking the algorithm for a specific problem, because the used data is simplified a lot. Also, it is only useful if one has presumptions about how an algorithm ought to react internally upon the given data. These conditions are met most fittingly when one is experimenting with ideas for relatively new types of network architectures and learning algorithms.

Generated toy data sets have been used more often in the field of machine learning, though commonly they do not provide latent variables except possibly a single class label or target value per sample[6]. For evaluating unsupervised learning algorithms, inspecting what the neurons have learned has also been done more often. For example, in the famous 'face and cat detector' paper[1], the researchers searched the network for neurons that activate when (cat) faces are visible in the input image. However, the key idea in my research is to combine the concepts of generating toy data from descriptive latent variables and assessing whether a neural network has learned desired features. Some searching and asking around did not lead to any existing publications in this direction.

Unable to find usable previous work, I decided to experiment with the idea myself by creating a toy data generator, applying its produced toy data to neural networks, and comparing the hidden layer activations with the hidden labels of the toy data. To try out the method, off-the-shelf neural network algorithms have been used, namely stacked Restricted Boltzmann Machines (RBM) to try it with unsupervised learning, and a Multilayer Perceptron Network (MLP) for supervised learning.

In the remainder of this paper, I will describe the method (section 2) and the considerations regarding and design of a toy data generator (3), and then describe the experiments with unsupervised (4) and supervised learning algorithms (5), followed by a conclusion and suggestions for future directions (6).

Reproducability and explorability

Before moving on, a small note about the form of this publication. This publication is intended to enable the reader to reproduce all results on the spot, and readers are encouraged to play around and modify the experiments, for example by changing the toy data generator or plugging in different learning algorithms. Most of the code has been written with adaptability in mind, making very few assumptions about input dimensions, numbers of layers, et cetera.

Throughout this report, code snippets will be given that can reproduce the shown results in a Python interpreter. To use them, start with importing the used functions from the accompanying code, and possibly fix the random number generator:

```
from experiments import *
seed = set_random_seed(123)
```

When using IPython Notebook, for which this publication is principally intended, run these lines to display figures inline and to automatically reload modules when you change the source files:

```
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

In other environments, running `fig.show()` may be required to show figures.

2 Method

The core principle of the method is to inspect whether a neural network learns in the way you expect or desire it to learn. The method can be seen as doing supervised evaluation not only on neurons that were trained to match a label, but also on all hidden neurons, by comparing them to known hidden labels (in a loose way, more on that below). This makes the method usable for (deep) supervised as well as for unsupervised learning algorithms, where in the latter case all neurons are hidden neurons and all labels are hidden labels.

Evaluating an algorithm would roughly take the following steps:

1. Create/obtain a toy data set (discussed in the next section), split into training and test parts
2. Train the algorithm with the training data, but withholding the hidden labels
3. Pass the test data to the network to transform it into neuron activations
4. Evaluate the correspondence between the activations and the data's hidden (& visible) labels
5. Visualise and inspect the correspondences, mismatches, neuron activations, and network weights

2.1 Feature comparison

An important question is of course *why* we want the algorithm to learn those desired features, and consider others to be inferior. If the algorithm finds other features, perhaps those are at least as good. Especially in the case of supervised learning, one could say that whichever features provide for the best end result are good features. There are several answers to this question.

Firstly, in many cases not just anything that works is okay. Checking for detection of very descriptive latent variables can be a way to test for overfitting. In the unsupervised case, it verifies that the algorithm learns to detect features that are actually descriptive of the data, and possibly useful for other tasks. In the supervised case it verifies that classifications are not based on the wrong reasons. When designing an algorithm, one probably has some conception of the kind of things it should learn, and this method seeks to provide a way to test these conceptions.

Secondly, there is much freedom in how comparison between neuron activations and hidden labels is done. It should probably not be a measurement of element-wise errors (e.g. $E(\boldsymbol{h}, \boldsymbol{l}) = \sum_i \left(h_{[i]} - l_{[i]}\right)^2$) as is usually done in supervised learning. At the very least, the ordering of features in a layer should normally not matter, so a better measure would be to take the best (element-wise) score among all permutations of $\mathbf{h}$ (e.g. $\min_{\boldsymbol{h_p} \in Sym(\boldsymbol{h})} E(\boldsymbol{h_p}, \boldsymbol{l})$). A comparison function can be designed to allow features to be distributed instead of represented by a single ('grandmother') neuron. For example, the distances between sample pairs in 'neural representation space' could be compared to distances of those in 'label space', to accept any hidden activation patterns, as long as similarly labelled items cause similar activations.[1]

Thirdly, maybe it is the case indeed that the network learns better features than those we desired it to learn. Though our evaluation results would then not be a useful quality measure, drawing this conclusion can give insight in what the network actually learns in contrast to our what we expected it to, and perhaps it gives new understanding of the data. Maybe the found features could be accounted for by changing the hidden labels, or by changing the comparison function.

[1] Yet untried but interesting. Inspired by [3].

2.2 Assessing representational power

If the network does not learn the features as desired, the cause can either lie in the learning algorithm not finding the desired model parameters, or it could be that there simply is no configuration of parameters that would give the desired results. One can attempt to falsify the latter hypothesis by exploring the representational power of the model, helped by the hidden labels of the toy data. How to do this depends on the model, but the idea is to do supervised training on what normally are the hidden layers of the network, using the hidden labels as targets for some chosen neurons. If the scale is small enough, one could even try to set the parameters manually. The thus obtained network can be evaluated as before, and if it performs better with these forged parameters than with ones it would normally learn, the representational power is sufficient and it is the learning algorithm that should be improved.

3 Toy data generation

An important aspect of this method is the availability of a data set with known descriptive features. A convenient way of obtaining such data is to generate it algorithmically as a function of a few random latent variables. The latent variables define the manifolds of the data, and their values can be used as the hidden labels. The intended challenge for the learning algorithm is thus to recover the latent variables by observing the generated data. An alternative way would be to label data using an exemplary algorithm (or human), but we only consider data generation here.

By creating data with low complexity and dimensionality, and reducing those of the neural network accordingly, inspection can be performed manually in order to obtain maximal insight. Also, this makes training and other computations fast, allowing for quick iterations and possibly for use of analysis methods that would be infeasible otherwise.

3.1 Applicability to real data

An important question is of course whether an algorithm that works well on such a toy dataset will also work well on real world data. There are two points to make about this.

Firstly, the wider applicability relies on the assumption that there are fundamental principles of intelligence that work for many types of data. This assumption is commonly made already, as a dataset like MNIST is also only intended as a proxy for many types of real world data (the demand for digit recognisers is not *that* big). Results on toy data should not be expected to map directly to performance on real data, but the fundamental principles (and limitations) may apply to both. For example, if an algorithm is incapable of solving a simple XOR-problem, neither will it work on real world data that has similar non-linearities.

Secondly, the solution (and big challenge in this method) lies in devising good data generators, whose data possesses properties found in real world data. Devising such generators may besides be a good exercise that forces us to think about the structure and properties of real data. Different generators with different non-linearities, statistics and such can be created, both to resemble different types of real data, and to test one's algorithm for its qualities in relative isolation (e.g. testing for second order relations separately from testing for bias-insensitivity).

3.2 Generator design

Generators could take many forms. the only requirement is that the latent variables provide a concise description of the data and you would expect a good learner to be able to recover them. An important point here is that the choice of hidden labels is strongly connected to what you want or expect your algorithm to learn.

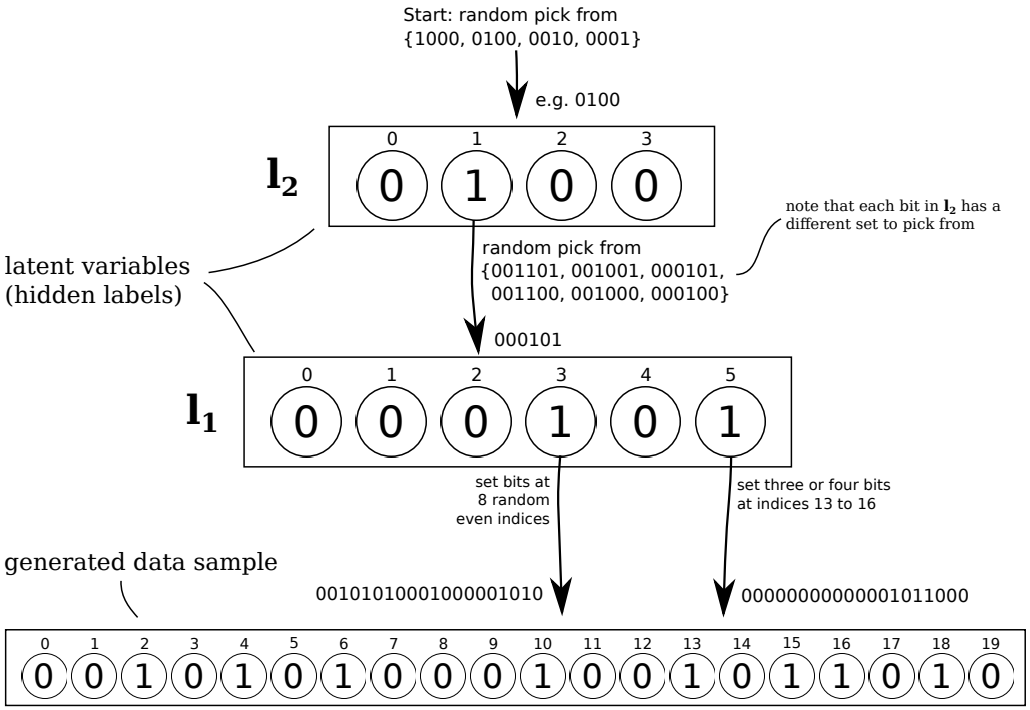

Figure 2: The toy data generation process used in the experiments.

In the experiments described below, toy data has been generated by a process using layers of latent variables. A value for the top layer is picked, and each layer determines the value of the (larger) layer below it, ending with the produced data sample at the bottom (see figure 2). The bits in a layer form a distributed representation, each bit standing for some modification it can apply to the layer below. For this experiment, each active bit picks a bit pattern from its own probability distribution, and the logical OR of these picked patterns constitutes the value of the next layer. For the top layer, we choose only one bit, which can be considered the 'class' of the data sample.

The idea behind the generation process is that the resulting data sample consists of a non-linear combination of a few factors, from which a neural network should first infer the middle-layer bits that produced this pattern, and from these inferences the network's next layer should be able to infer the original value of the top layer (the sample's 'class'). The intuition for this layer-wise inference originates from popular knowledge about visual perception, in which lower layers detect simple features, which are combined by higher layers to recognise more complex patterns.

The generation method seems a nice simplification of real world processes in which a few latent variables together cause a large-dimensional input. For example, the choice of which digit to draw determines (with some randomness) the pen's starting location and rough directions, which (again with some randomness) determine which pixels are coloured. The current approach involves a simple OR of bit patterns, but one can see how more interesting non-linearities could be created by assigning different functions to different bits. For example we could have one bit determine whether the left half next layer will be inverted. Or, by having some bits determine how many places to shift the final output bits, one can test if one's algorithm learns to be invariant to translation: if it does, it will still correctly infer the values of the other latent variables regardless of the presence of shift.

The choices about the number of variables in each layer and the patterns they create have been rather arbitrary. The plan was to not make it too difficult to recognise features; knowing the rules it should be easy to do by hand. At the same time, it should not be trivial either, so for example a single bit should never give enough information to infer the value of a latent variable, and a single intermediate latent variable (a bit in $\mathbf{l_1}$) should not pin down the top level latent variables ($\mathbf{l_2}$). For the complete rules, read the source in `toydata.py`. Some samples along with their latent variables can be generated like this:

```
from toydata import toydata
toydata(n_samples=5)

[['1000', '000011', '010100010101001111101'],
 ['0100', '000100', '100000101010000000010'],
 ['1000', '000011', '010001010001010111101'],
 ['1000', '000010', '000000010001010101000'],
 ['0010', '110100', '111110111110100011111']]
```

The effect of each feature can be inspected by manually fixing the value of one or both of the latent layers:

```
toydata(n_samples=5, latent_vars=['1000'])

[['1000', '000011', '000001000000101110001'],
 ['1000', '000011', '010101010100001110001'],
 ['1000', '000010', '010100010101000000000'],
 ['1000', '000011', '000001010100011111001'],
 ['1000', '000001', '000000000000001011000']]

toydata(n_samples=5, latent_vars=['0000', '100000'])  # (l2 is irrelevant now)

[['0000', '100000', '101111111000000000000'],
 ['0000', '100000', '010101111100000000000'],
 ['0000', '100000', '110100111100000000000'],
 ['0000', '100000', '110101011000000000000'],
 ['0000', '100000', '111101000100000000000']]
```

One remark to be made is that there are in fact more latent variables involved than are stored in the hidden labels. The information about which pattern is picked from a bit's probability distribution is not recorded, implying that the hidden labels do not provide a full description of the data. Also, in the other direction, the same sample could sometimes have been created by different choices of hidden labels, as information is lost in the OR function. Both the generation and (best possible attempt at) inference are thus probabilistic functions. This is not considered a problem, as it is often the case in real world situations too.

4 Unsupervised learning

Have described the general method and toy data generator, this section and the next will cover the experiments performed with this toy data. To start with, a training and a test set are generated, each containing 2000 samples.

```
trainingset = generate_dataset(N_train)
testset = generate_dataset(N_test)
```

In this section, the main question is whether a common unsupervised learning algorithm would find features that correspond to the latent variables used during generation. For this experiment, I chose to use Restricted Boltzmann Machines (RBM), for no specific reason. To make testing simple, two RBMs are used, one stacked upon the other, and the layer sizes are picked such that they match the number of hidden labels of the data.

```
model = StackedRBMs(trainingset.layer_sizes)
```

Stacked RBMs have been used before for initialising weights of deep neural networks[2], after which inference can be performed by computing the conditional probabilities of the first layer's hidden units given the visible units ($\mathrm{p}\left(h_{[i]} = 1 \mid \mathbf{v}\right)$), and using these values as input for the next RBM to do the same trick again. In the ideal albeit unlikely case, these inferred conditional probabilities at the top layer would match exactly with the actual classes one was planning to predict, making it a perfect classifier after only unsupervised training. Note that because the conditional probabilities are not sampled from but are instead used as the next layer's input values directly, each layer after the first performs a mean-field approximation only. Also, the inference proceeds in a single upward pass, thus behaving much more like a multi-layer perceptron network than like a Boltzmann machine.

In this experiment, we wish to compare how well the inferred values of $\mathbf{h_1}$ and $\mathbf{h_2}$ will match with the data's hidden labels $\mathbf{l_1}$ and $\mathbf{l_2}$, respectively. Because any unit in a layer could have learned to detect any of the features, we first reorder the units of both activations $\mathbf{h_n}$ such that the units have maximal correlation with the hidden labels $\mathbf{l_n}$. Correlation here means the proportion of agreements round $\left(h_{n[i]}\right) = l_{n[i]}$, minus the proportion of disagreements, giving a score in the range $[-1 \ldots 1]$. If a column correlates strongly but negatively, it is also matched but the whole column is inverted ($h_{n[i]} \leftarrow 1 - h_{n[i]}$).

Apart from measuring these correlations, we also measure how well the model would classify samples. Because in our data set $\mathbf{l_2}$ always has exactly one active bit, $\mathbf{h_2}$ could be interpreted as a label prediction, and we measure its accuracy by comparing for which proportion of the samples $\arg\max_i h_{2[i]} = \arg\max_i l_{2[i]}$.

For further details regarding the comparison, read `evaluation.py`.

4.1 Learning

First, the stacked RBMs are trained with the generated training set. Next, the test set transformed by the RBMs to obtain $\mathbf{h_1}$ and $\mathbf{h_2}$, which are compared to the hidden labels.

```
train_unsupervised(model, trainingset)
results_rbm_learning = evaluate_model(model, testset)
fig = plot_correlations(results_rbm_learning['metrics']['correlations'])
```

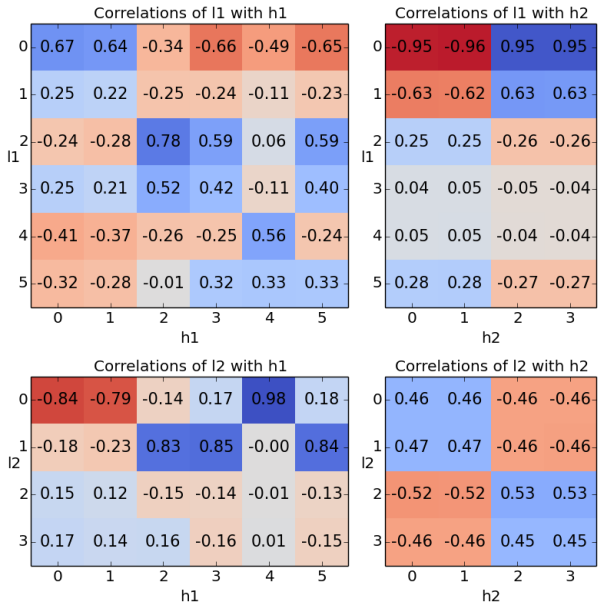

Figure 3: Correlations between hidden labels and RBM unit activations

```
fig = plot_weights(results_rbm_learning['ws'])
```

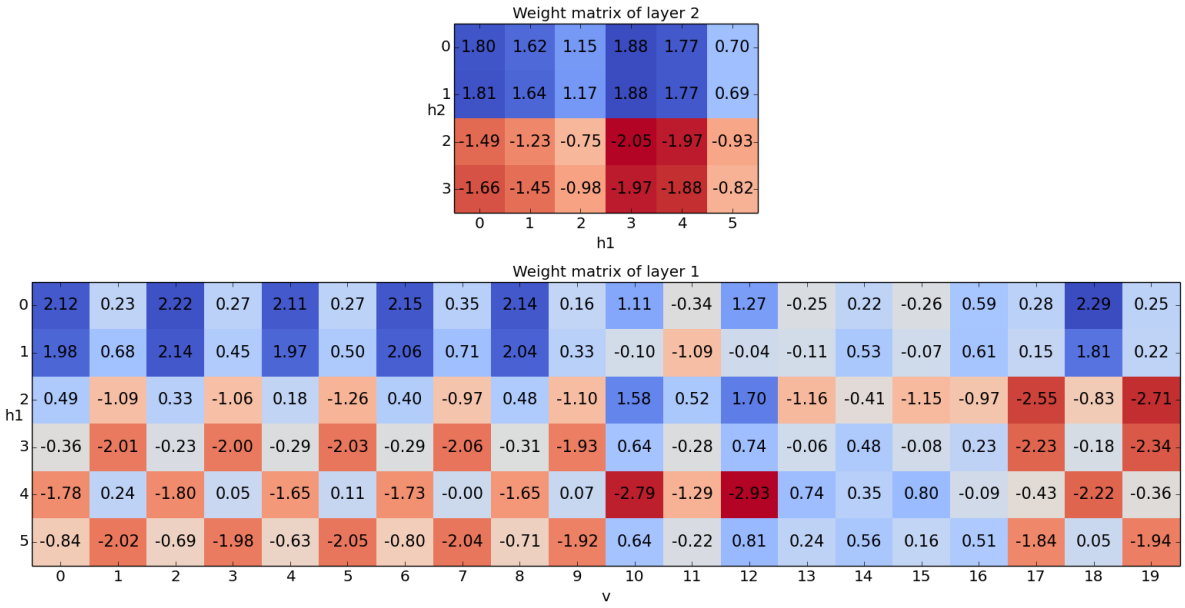

Figure 4: Weight matrices of the RBM's layers

For investigation of the results, it is most insightful to look at the activations of the first hidden layer ($\mathbf{h_1}$), as the second layer's performance is heavily influenced by that of the first. Although there certainly are some correlations between the units' activations and the hidden labels, most units do not reflect a single hidden label as much as I hoped for. Part of the reason (apart from my hopes simply being too high) may be that the toy data is still too complex for such a simple model. An indication of this is that the simpler features (e.g. number 2, always turning on the same three bits) are learned and detected quite well by a single unit in nearly every run of the experiment. Other features, that have more stochasticity, seem to be more easily mixed up by the units.

A look at the weight matrices suggests that each unit is quite sensitive to several features, as for example in most rows one can clearly see the chequered pattern that matches feature 3 and 4, which respectively activate some even or some odd bits. One explanation for this is that to detect one feature, one needs to ensure the feature's bits were not actually activated by another feature, so even if each unit would watch for one specific feature, the weight matrix would look like features are mixed up. This mixing of several patterns into each weight vector makes it more difficult to read the weight matrix.

Also from the correlation matrix between hidden labels and unit activations of the first layer (the top left matrix in figure 3), it can be seen that most units correlate with several hidden labels. Not coincidentally, the combinations of features seem to match somewhat with the patterns picked by the generator's top level latent variables. This is completely understandable, for two reasons:

1. Because in the *test* data the features occur in specific combinations, activations are expected to correlate accordingly.

2. Because in the *training* data the features occur in these combinations, their combinations are considered features themselves. Stacked RBMs are trained in a greedy layer-by-layer manner. The bottom RBM thus optimises its weights to mimic its input's probability distribution as well as possible, and learning to detect our desired features separately would ignore the correlations between those features.

We can modify the data generation for the test set to examine the influence of the first cause. To prevent the features from occurring only in particular combinations, the top layer in the toy data generation can be modified such that any feature co-occurs equally often with any other feature (using `generate_uncorrelated_dataset`). Even simpler data can be created by activating only a single feature per sample (using `generate_single_feature_dataset`). In either case, the correlations diminish somewhat but a significant part persists. As would be expected, the correlations also diminish when *training* the model with the 'uncorrelated' data, but a significant mixing of features still remains, presumably because of the mentioned reason of data complexity.

The second RBM, having four units, appears to usually end up with two units' activations correlating with the first two top level labels, the other two doing the same for the other two labels. The reason for this has not been investigated further.

4.2 Representational power

Given that the RBMs do not learn to detect the hidden labels as well as was hoped for, an interesting question is now whether it is possible at all to find model parameters that would give the desired results. To try this out, I modified the RBM's contrastive divergence training algorithm so that instead of using an inferred value of **h** to strengthen the weights in the positive phase, it is forced to use the hidden labels.

```
model = StackedRBMs(trainingset.layer_sizes)
train_with_hidden_labels(model, trainingset)
results_rbm_power = evaluate_model(model, testset)
fig = plot_correlations(results_rbm_power['metrics']['correlations'])
```

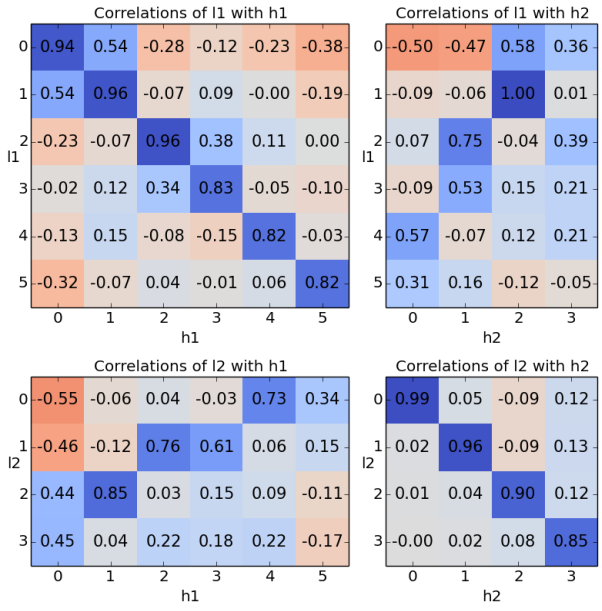

Figure 5: Correlations for RBMs with (close to) ideal parameters

```
fig = plot_weights(results_rbm_power['ws'])
```

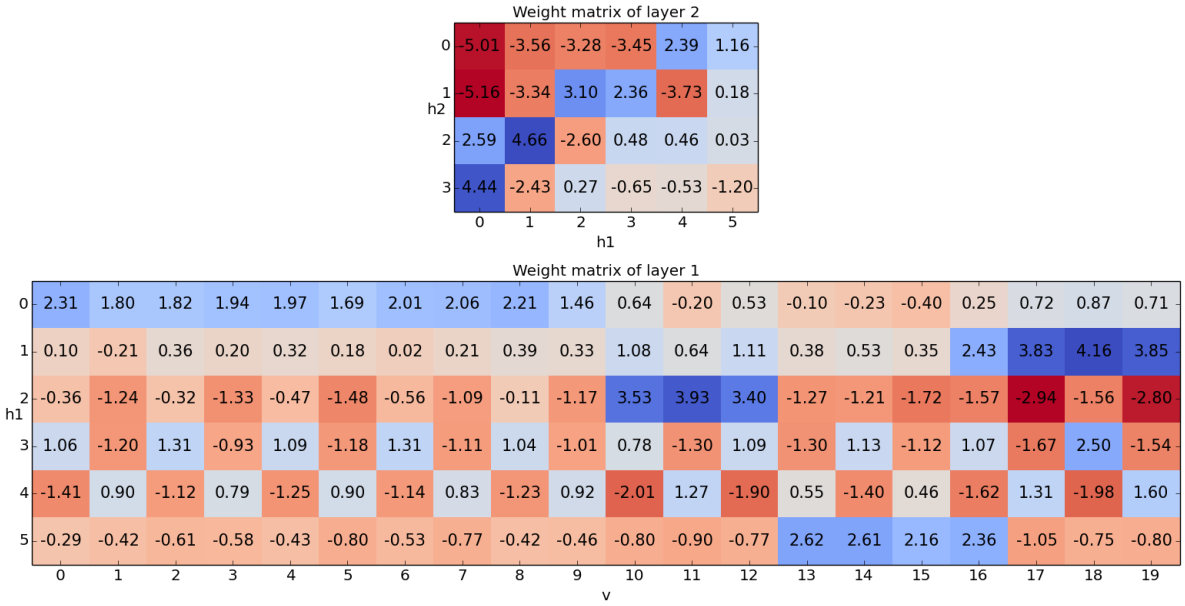

Figure 6: Weights of RBMs with (close to) ideal parameters

As is clear from figure 5, the results of this check are much better than those of unsupervised learning, giving correlations between hidden labels and unit activations around 0.9, and similar numbers for classification accuracy every run of the experiment:

```
[experiment_rbm_power()['metrics']['class_accuracy'] for _ in range(5)]

[0.9415, 0.9495, 0.928, 0.9395, 0.9285]
```

Strong and seemingly undesired correlations are still visible between $h_{1[0]}$ and $h_{1[1]}$, and also $h_{1[2]}$ and $h_{1[3]}$, but they do not differ much from the correlations in the test data itself (inspectable with `experiment_data_autocorrelation`), so that could largely account for these correlations.

Though this result shows that the model, given the right parameters, is capable of detecting our desired features reasonably, it does not yet tell why the learning algorithm does not find these desired parameters. It could be that this parameter setting forms another optimum, and given some lucky initialisation parameters the model would end up there, or it could be that the desired result does not satisfy what the model wants to learn. To test which is the case, after having performed the modified training to find the desired parameters, we train the model again but with normal unsupervised training.

```
# Note: still using the same model
trainingset2 = generate_dataset(N_train)
train_unsupervised(model, trainingset2)
results_rbm_power_stability = evaluate_model(model, testset)
fig = plot_correlations(results_rbm_power_stability['metrics']['correlations'])
```

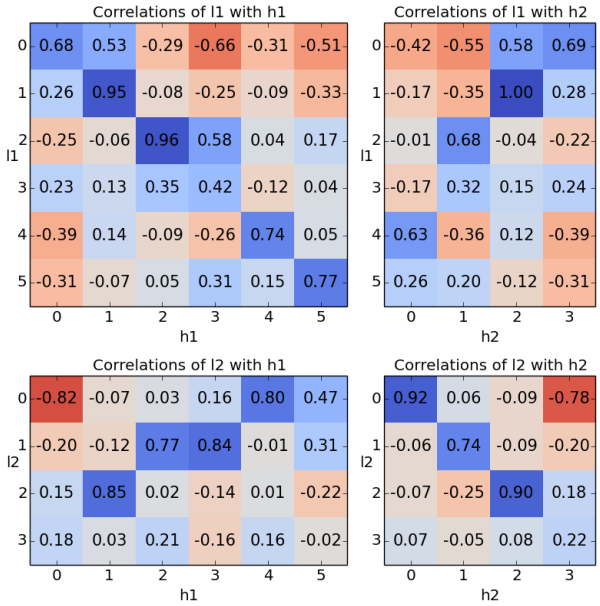

Figure 7: Correlations matrices showing that unsupervised training steers away from 'ideal' parameter setting

The result shows that the RBMs tend to move away from the desired parameters and become less performant by our metrics, so our desired setting is simply not what the learning algorithm tries to find. This is of course not a big surprise, as the RBM's learning algorithm was not designed for the type of task we are giving it here.

4.3 Experiment conclusion

Apparently, stacked RBMs are not really the right tool for our purpose. Firstly, RBMs are not made for the job. Looking at the correlations between hidden labels and unit activations, it becomes clear how the first-layer units learn combinations of frequently co-occurring features, rather than the individual features we would like it to detect. Inspecting the weight matrix gives impression too, although is seems more difficult to read. The learning algorithm optimises for a goal different from ours. Secondly, the simplistic way of stacking RBMs is unsatisfactory. The layers are not 'aware' of each other, and different training schemes should be used to improve their teamwork, as has also been pointed out by others[4]. Perhaps using a Deep Boltzmann Machine[5] instead could have been a solution here.

Apart from concluding that this algorithm is inadequate for our goal, we should also consider the possibility that it 'outsmarts' us and we should actually revise the goal instead; for example I may have been too predetermined about which features should appear in each layer. Given the observations, it seems worth to rethink how an intelligent algorithm should handle correlating but distinct features. My intuition after this experiment is that we should search for an algorithm that behaves a bit more like Independent Component Analysis, in that it tries to discriminate separate phenomena.

5 Supervised learning

While in the previous section we assessed whether an unsupervised learning algorithm starts to distinguish the latent variables that generated the data, the question is now whether a supervised learning algorithm learns to distinguish the *intermediate* latent variables ($\mathbf{l_1}$) when it is trained to classify the *top* level latent variables ($\mathbf{l_2}$).

Most of the experiment setup remains the same, except that instead of a stack of two RBMs of six and four units, a Multi-Layer Perceptron (MLP) with four (softmax) outputs and a hidden layer of six sigmoid units is used.

```
trainingset = generate_dataset(N_train)
testset = generate_dataset(N_test)
model = MLP(trainingset.layer_sizes)
```

5.1 Learning

Learning is done using standard backpropagation learning with a small L2 regularisation. The exact hyperparameters are not important for this experiment, and can be found in the code (see `mlp_theano.py`).

```
train_supervised(model, trainingset)
results_mlp_learning = evaluate_model(model, testset)
fig = plot_correlations(results_mlp_learning['metrics']['correlations'])
```

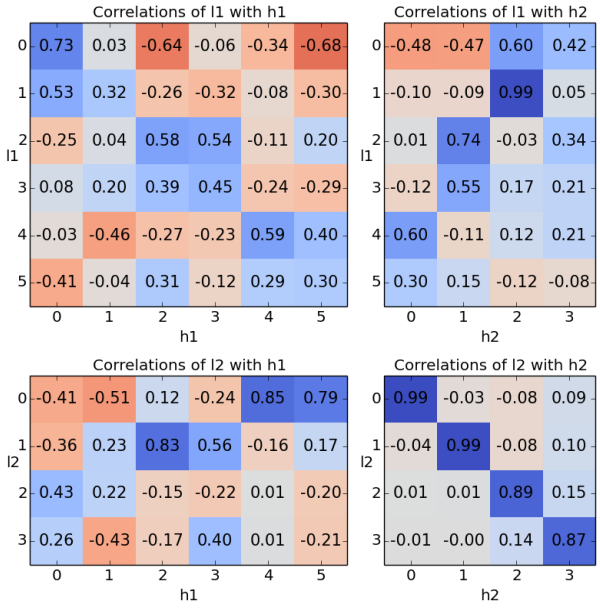

Figure 8: Correlations between (partially hidden) labels and MLP unit activations.

```
fig = plot_weights(results_mlp_learning['ws'])
```

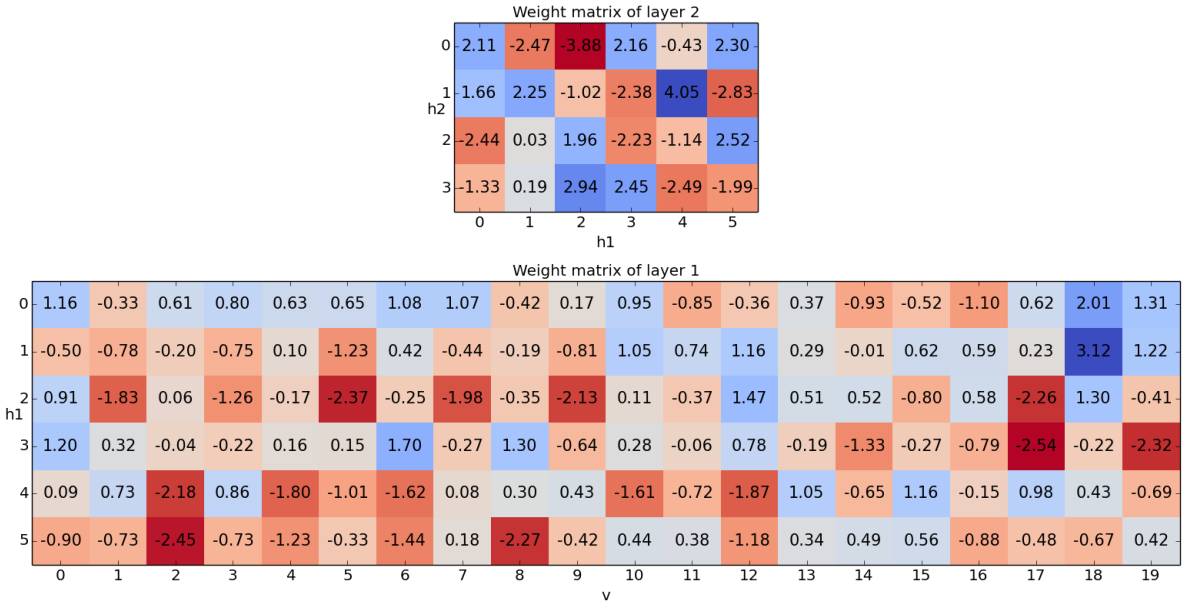

Figure 9: Weights of the trained MLP

An indication of the classification accuracy is given by the values on the diagonal of the correlation matrix of $\mathbf{l_2}$ and $\mathbf{h_2}$ (the bottom right matrix in figure 8). The data seems simple enough to classify quite accurately, with little variation among experiment runs:

```
[experiment_mlp_learning()['metrics']['class_accuracy'] for _ in range(5)]

[0.947, 0.9325, 0.9335, 0.9515, 0.941]
```

As with the RBMs, we can see from the correlation matrices of $\mathbf{h_1}$ (the left ones in figure 8) that each unit correlates with several hidden features. Again, some are much more indicative of the top level latent variables ($\mathbf{l_2}$) than of the bottom ones ($\mathbf{l_1}$). This may be even more expectable than before, as the network has this time been specifically trained to recognise the $\mathbf{l_2}$ variables.

5.2 Representational power

Like before, we can check how close to our goal the model could come when the parameter settings are set to an ideal configuration rather than learned from training data. The 'ideal' configuration is found by using logistic regression to train each unit in the MLP's hidden layer to predict the normally hidden labels $\mathbf{l_1}$ when given the input data, and then likewise train the output layer to predict $\mathbf{l_2}$ given $\mathbf{l_1}$.[2]

```
model = MLP(trainingset.layer_sizes)
train_with_hidden_labels(model, trainingset)
results_mlp_power = evaluate_model(model, testset)
fig = plot_correlations(results_mlp_power['metrics']['correlations'])
```

[2] In hindsight, a more logical method would be to simply add the error between $\mathbf{h_1}$ and $\mathbf{l_1}$ to the training cost function.

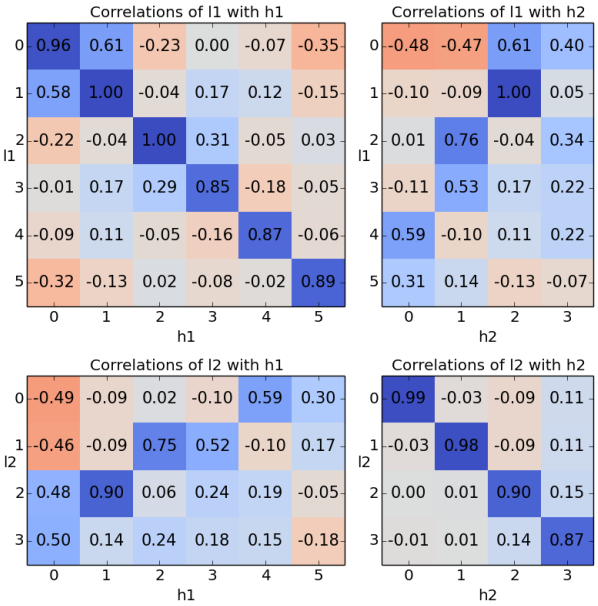

Figure 10: Correlations of MLP with (close to) ideal parameters

```
fig = plot_weights(results_mlp_power['ws'])
```

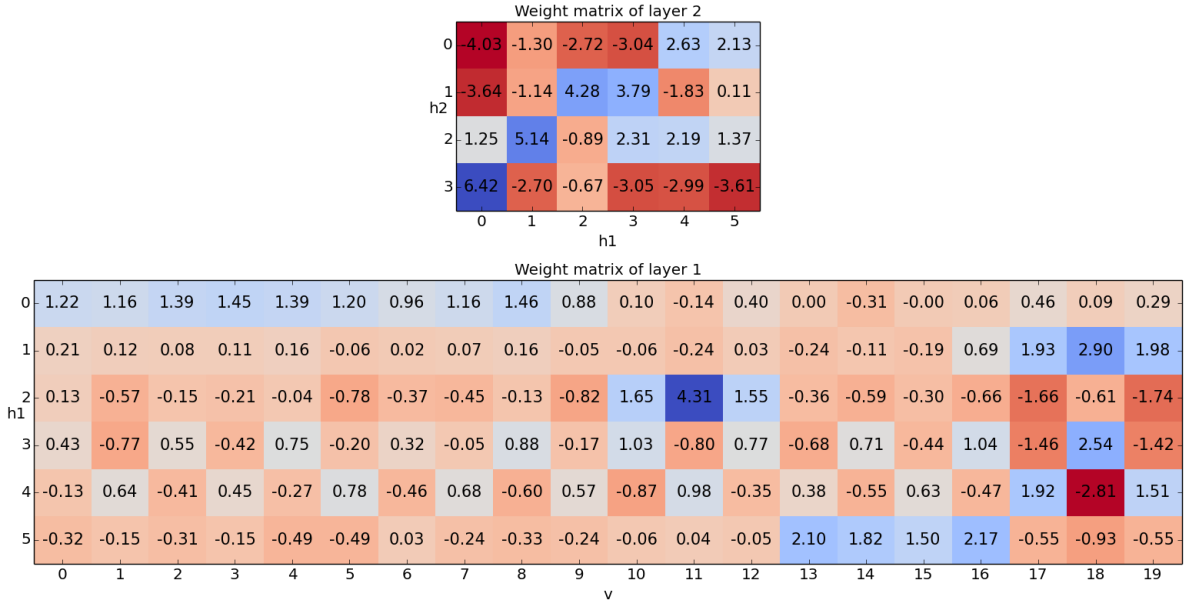

Figure 11: Weight matrices of MLP with (close to) ideal parameters

As performing inference in the MLP is practically the same process as in the stacked RBMs (apart from the softmax output), it should not be surprising that its representational power is quite similar. The small differences that do show up here must then have come from the different methods used to find the 'ideal' parameters.

An interesting question is of course whether the classification accuracy has improved or diminished:

```
[experiment_mlp_power()['metrics']['class_accuracy'] for _ in range(5)]

[0.9315, 0.943, 0.9405, 0.9415, 0.9285]
```

Apparently, the classification accuracy is nearly as good as with normal supervised training. Although perhaps slightly sub-optimal, first inferring the hidden labels thus seems quite a decent way to infer the class label for this data set.

5.3 Experiment conclusion

Apart from the obvious difference that the activations $\mathbf{h_2}$ match the top level labels $\mathbf{l_2}$ much better, as is to be expected from supervised learning, the correlations between activations and hidden labels seem to form rather similar patterns to those of the RBM. The patterns visible in the hidden layer's weight matrix seem somewhat different however, and also differ between runs of the experiment. They seem to mix up the different features a little bit less, but still often take correlating features together to recognise them as one.

This behaviour of combining features that frequently co-occur is probably the most interesting insight from both experiments. Conceptually, it relates to the hypothesis that the brain works with a "first forest, then trees" approach: in a given picture, we would first see a forest (the higher level class), and only after that we discern the individual trees (intermediate features). However, in creating the toy data generator and its hidden labels, I presumed that a feed-forward neural network should first distinguish the individual features and combine those to infer the class, making it see "first trees, then forest". In a way, from the results of this experiment we could say that the MLP follows a "first forest, *no* trees" approach, as it (to some extent) only detects combinations of features. Though this may be an efficient approach in the task it is trained for, my intuition is that distinguishing individual features is ulmitately more helpful, for example to reuse the features for other tasks (transfer learning) and maintaining performance when features or other circumstances change.

6 Conclusion

This report has covered two things: Firstly, it introduced a method for evaluating neural network algorithms, and secondly it covered initial experiments using this method. Despite the experiments taking up the bigger amount of pages, they mostly just served as a proof of concept. With more attention and scientific scrutiny they could probably lead to more valuable results, as quite some decisions have been taken somewhat arbitrarily, especially regarding the toy data generation. Also the measurements for evaluation have been a bit ad-hoc. For example, correlation (adapted to count binary matches) has been used primarily, which is sensitive to the bias of the data: two sparse features always correlate well because both are mostly zero.

However, the main purpose of this research was to try out if the method has a potential value in the development of learning algorithms. Although more experimentation and real usage are needed to find that out, I think it looks promising enough to pursue. At least for me it provided insight in how RBMs and MLPs learn. It may yet have to be found out in which situations the method is useful, and what types of data to generate.

The essence of the method is to reduce a real world problem to one at a toy scale. A fundamental requirement for this reduction is that you understand your problem, in order to retain its properties on smaller scale. As you have to provide the desired features or hidden labels yourself, the method seems fit for cases where you actually understand *what* you want your algorithm to learn, but just do not know *how* to make an algorithm that learns it.

An important question is how tightly the type of comparison, the toy data generator, and the type of learning algorithm are tied together. The comparison function may have to be designed specifically for a toy data generator, because for different types of data there may be different desired properties. Likewise, the comparison function may have to be designed specifically for the internal representations of the algorithm under test, for example to make it

invariant to permutations, biases or other properties. Imagine the toy data having latent variables representing a number in binary encoding, whereas the neural network produces a one-hot encoding: the comparison needs to know about both sides. If it is possible to sufficiently decouple the design of the data generator, comparison function and learning algorithm, it would enable easy reuse across algorithms, saving effort and enabling standardised comparisons between algorithms.

Future work

The most important thing to do next is to design toy data generators, and investigate their correspondence with types of real world data. It would be a good indication when algorithms with a good performance on the toy data set would also score well on the real world data set.

Different types of generators could perhaps test for different properties properties of algorithms in relative isolation, such as bias or translation invariance, use different types of non-linearities and relations, test for generalisation, et cetera. If it is possible to standardise the experiment setup, it may be worthwhile to create a collection of toy data generators, providing an easy way to test a newly designed architecture for several properties.

Regarding evaluation, there could be better ways for measuring the algorithm's performance on a toy data set. In these experiments, neural activations have been compared to hidden labels, after reordering neurons to find the best matches. Many other ways of comparison could be possible however. Instead of using the activations, the network's weights or its mapping function could be accessed more directly, removing influence from particularities of the test set from the evaluation.

References

[1] Quoc V. Le et al. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[2] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006. See section 4.

[3] Christopher Olah. Visualizing Representations: Deep Learning and Human Beings.

[4] Nicolas Le Roux and Yoshua Bengio. Representational power of restricted Boltzmann machines and deep belief networks. *Neural Computation*, 20(6):1631–1649, 2008. See section 3.2.

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

[6] Scikit-learn. `sklearn.datasets` sample generators. Some example toy data generators.