Deep Learning

Andreas Eilschou

Hinnerup Net A/S
www.hinnerup.net

July 4, 2014

Introduction

Deep learning is a topic in the field of artificial intelligence (AI) and is a relatively new research area although based on the popular artificial neural networks (supposedly mirroring brain function). With the development of the perceptron in the 1950s and 1960s by Frank RosenBlatt, research began on artificial neural networks. To further mimic the architectural depth of the brain, researchers wanted to train a deep multi-layer neural network – this, however, did not happen until Geoffrey Hinton in 2006 introduced Deep Belief Networks [1].

Recently, the topic of deep learning has gained public interest. Large web companies such as Google and Facebook have a focused research on AI and an ever increasing amount of compute power, which has led to researchers finally being able to produce results that are of interest to the general public. In July 2012 Google trained a deep learning network on YouTube videos with the remarkable result that the network learned to recognize humans as well as cats [6], and in January this year Google successfully used deep learning on Street View images to automatically recognize house numbers with an accuracy comparable to that of a human operator [5]. In March this year Facebook announced their DeepFace algorithm that is able to match faces in photos with Facebook users almost as accurately as a human can do [9].

Deep learning and other AI are here to stay and will become more and more present in our daily lives, so we had better make ourselves acquainted with the technology. Let's dive into the deep water and try not to drown!

Data Representations

Before presenting data to an AI algorithm, we would normally prepare the data to make it feasible to work with. For instance, if the data consists of images, we would take each

image as a matrix of raw pixel values and extract edges to represent it at a higher level. We could also take this edge representation and represent it at a higher level by detecting local shapes. At an even higher level local shapes could be represented as object parts, and so on, until we get the understanding out of the data that we are interested in working with. See Figure 1.

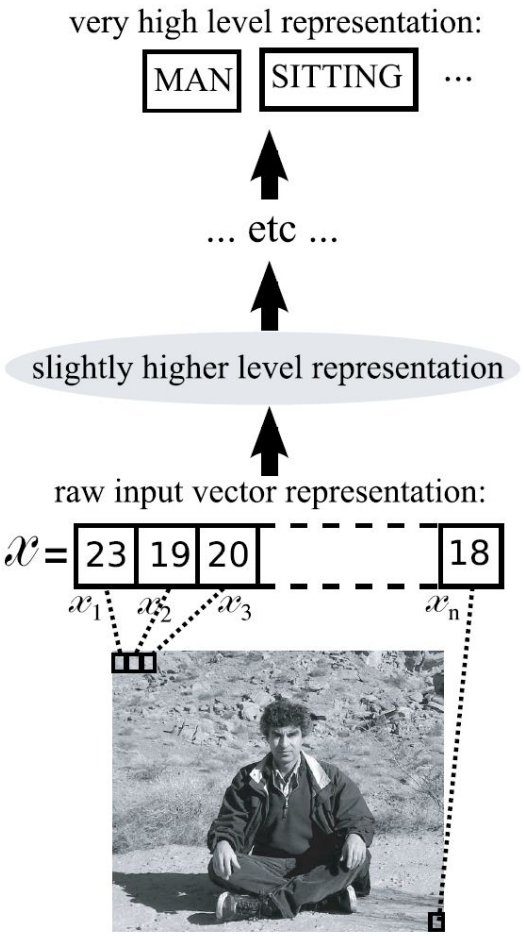


Figure 1: An image at different levels of representation. From raw input, to edges, to local shapes, to object parts, etc. This figure is taken from Figure 1.1 in Bengio's monograph [1].

Deep learning networks are able to learn intermediate data representations on their own. That is, presenting a trained deep learning network with an image of raw pixel values, each layer of neural units will represent the image at an increasingly higher level. For a classification task, the top layer could then be passed on as input to a classifier for training and testing.

Generative Model

Even though the deep architecture can be seen as an input to a different process, it holds very interesting qualities on its own. We can set up the deep architecture as a generative model whereby we can learn a distribution and generate samples from it. This means, training it with a certain dataset, we will be able to see if it has learned the characteristics of the dataset by having it generate new and previously unseen examples of data that could have come from that dataset.

Setting up a Deep Learning Network

In my search for a suitable library, I came across Theano [2] which is a Python library that allows to define, optimize and evaluate mathematical expressions efficiently. Among other features it provides us with numerical tricks for stability and allows transparent use of the GPU. And the best part, there is a section of deep learning tutorials to help get a deep learning network up and running.

Objective of my Deep Learning Network

In the tutorial on Deep Belief Networks, the network is first pre-trained using a greedy layer-wise unsupervised Restricted Boltzmann Machine training, and then fine-tuned using a Multilayer perceptron to have it act as a classifier.

Since a Deep Belief Network is a generative model, it would be of great interest to see some samples of what it has learned during training. I will undertake this job of setting up a generative Deep Belief Network that will learn the distribution of a dataset of images, and I will modify the code such that we can pull out samples to see whether it has learned anything about the image appearances.

The data used in the tutorials is a package of serialized modified MNIST images. I therefore also need to modify the code to be able to load in datasets from image files. In particular, I would like to see if I can teach the DBN what a human face looks like from example images of faces.

Theory

The following is the bare minimum of theory needed to explain how to sample from the generative Deep Belief Network. For more details, I refer the reader to Bengio's monograph *Learning Deep Architectures for AI* [1]. For an introduction to neural networks have a look at the online book Neural Networks and Deep Learning by Michael

Nielsen, and for sampling and machine learning in general Bishop [3] is a source of inspiration.

Deep Belief Network

A Deep Belief Network (DBN) is made up of a number of neural networks with a Restricted Boltzmann Machine (RBM) on top. In fact each of the neural networks are trained using additional RBMs with weights shared with the neural networks. The graphical model of a DBN is illustrated in Figure 2.

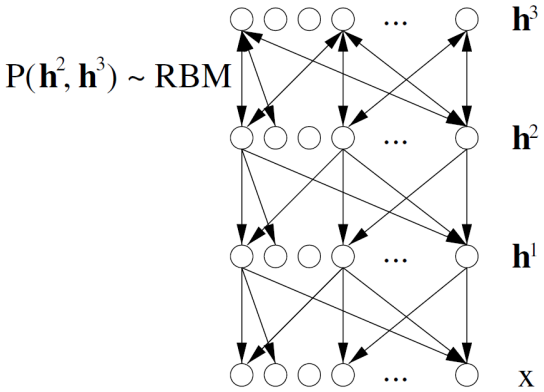


Figure 2: Graphical model of a Deep Belief Network with observed vector $\mathbf{x}$ and hidden layers $\mathbf{h}^1$, $\mathbf{h}^2$ and $\mathbf{h}^3$. This figure is taken from Figure 4.4 in Bengio's monograph [1].

For a DBN with layers $(\mathbf{x} = \mathbf{h}^0, \mathbf{h}^1, \ldots, \mathbf{h}^\ell)$ and binary units, the units of layer $\mathbf{h}^k, k \in \{0, 1, \ldots, \ell - 2\}$ are independent given the units in the layer above, $\mathbf{h}^{k+1}$. For each unit i the neuron activation function is given by

$$P(\mathbf{h}_i^k = 1 | \mathbf{h}^{k+1}) = \text{sigm}\left(\mathbf{b}_i^k + \sum_j W_{i,j}^{k+1} \mathbf{h}_j^{k+1}\right), \quad k \in \{0, 1, \ldots, \ell - 2\} \tag{1}$$

with $\mathbf{b}^k$ a vector of offsets, W^k a matrix of weights and the sigmoid function

$$\text{sigm}(u) = \frac{1}{1 + e^{-u}}. \tag{2}$$

The joint distribution of the top two layers $\mathbf{h}^{\ell-1}$ and $\mathbf{h}^\ell$ is a Restricted Boltzmann Machine

$$P(\mathbf{h}^{\ell-1}, \mathbf{h}^\ell) \propto e^{\mathbf{b}'\mathbf{h}^{\ell-1} + \mathbf{c}'\mathbf{h}^\ell + \mathbf{h}^{\ell\prime} W \mathbf{h}^{\ell-1}}. \tag{3}$$

The graphical model of an RBM is illustrated in Figure 3.

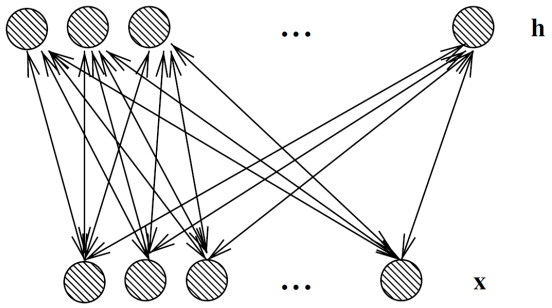


Figure 3: Graphical model of a Restricted Boltzmann Machine. There are no links between units of the same layer, thereby making the conditionals $P(\mathbf{h}|\mathbf{x})$ and $P(\mathbf{x}|\mathbf{h})$ factorize. This figure is taken from Figure 4.5 in Bengio's monograph [1].

The distribution of a Deep Belief Network with layers $(\mathbf{x} = \mathbf{h}^0, \mathbf{h}^1, \ldots, \mathbf{h}^\ell)$ and binary units is thus given by

$$P(\mathbf{h}^0, \mathbf{h}^1, \ldots, \mathbf{h}^\ell) = \left(\prod_{k=0}^{\ell-2} P(\mathbf{h}^k | \mathbf{h}^{k+1}) \right) P(\mathbf{h}^{\ell-1}, \mathbf{h}^\ell). \tag{4}$$

Sampling from a DBN

What to do in order to obtain a sample of the DBN generative model for $\mathbf{x}$ is outlined by Bengio [1]. First we need to sample a visible representation $\mathbf{h}^{\ell-1}$ from the RBM at the top layer. Due to the special structure of an RBM where the units of each layers are independent given the units of the other layer, we have that

$$P(\mathbf{h}^\ell | \mathbf{h}^{\ell-1}) = \prod_i P(\mathbf{h}_i^\ell | \mathbf{h}^{\ell-1}) \tag{5}$$

with

$$P(\mathbf{h}_i^\ell = 1 | \mathbf{h}^{\ell-1}) = \text{sigm}(\mathbf{c}_i + W_i \mathbf{h}^{\ell-1}). \tag{6}$$

Likewise for $P(\mathbf{h}^{\ell-1}|\mathbf{h}^\ell)$ we have that

$$P(\mathbf{h}^{\ell-1} | \mathbf{h}^\ell) = \prod_i P(\mathbf{h}_i^{\ell-1} | \mathbf{h}^\ell) \tag{7}$$

with

$$P(\mathbf{h}_j^{\ell-1} = 1 | \mathbf{h}^\ell) = \text{sigm}(\mathbf{b}_j + W_j' \mathbf{h}^\ell). \tag{8}$$

We can obtain a sample from an RBM by Gibbs sampling. We run a Gibbs chain for n steps starting with a data example $\mathbf{x}_1$ represented at the $\mathbf{h}^{\ell-1}$ layer by $\mathbf{h}_1^{\ell-1}$,

$$\begin{aligned} \mathbf{h}_1^{\ell} &\sim P(\mathbf{h}^{\ell}|\mathbf{h}_1^{\ell-1}) \\ \mathbf{h}_2^{\ell-1} &\sim P(\mathbf{h}^{\ell-1}|\mathbf{h}_1^{\ell}) \\ \mathbf{h}_2^{\ell} &\sim P(\mathbf{h}^{\ell}|\mathbf{h}_2^{\ell-1}) \\ &\vdots \\ \mathbf{h}_{n+1}^{\ell-1} &\sim P(\mathbf{h}^{\ell-1}|\mathbf{h}_n^{\ell}). \end{aligned} \tag{9}$$

For $k \in \{\ell-1, \ell-2, \ldots, 1\}$ we then sample $\mathbf{h}^{k-1}$ from $P(\mathbf{h}^{k-1}|\mathbf{h}^k)$. When we reach $\mathbf{x} = \mathbf{h}^0$ we have the DBN sample.

Modifications

Serializing trained parameters

Before we can sample from the DBN, we need to have the network trained. This training can take hours so we want to be able to save the learned parameters, such that we quickly can experiment with the sampling. With `cPickle` we can easily save the parameters:

```
# collect dbn weights
dbn_weights = []
for i in xrange(len(dbn.rbm_layers)):
    dbn_weights.append([dbn.rbm_layers[i].W,
                        dbn.rbm_layers[i].hbias,
                        dbn.rbm_layers[i].vbias])

# dump to file
pkl_gz_file = gzip.GzipFile(os.path.join(dbn_weights_folder,
                                         dbn_weights_file), 'wb')
cPickle.dump(dbn_weights, pkl_gz_file, -1)
pkl_gz_file.close()
```

and load them back in:

```
# load in weights from file
pkl_gz_file = gzip.open(os.path.join(dbn_weights_folder,
                                     dbn_weights_file), 'rb')
dbn_weights = cPickle.load(pkl_gz_file)
```

```
pkl_gz_file.close()

# restore weights
for i in xrange(len(dbn.rbm_layers)):
    dbn.rbm_layers[i].W     = dbn_weights[i][0]
    dbn.rbm_layers[i].hbias = dbn_weights[i][1]
    dbn.rbm_layers[i].vbias = dbn_weights[i][2]
```

Loading in image data

The data for the DBN should be provided as a 2-dimensional Theano symbolic data array of shape (number of data examples $\times$ number of entries in each data examples). Each entry is a 32-bit floating point between 0.0 and 1.0. 32-bit floats are used to be able to put computations on the GPU. The 0.0-1.0 interval is because each entry will be understood in the lowest layer neural network as the probability of this entry being a binary 1. Given that we have a folder with 8-bit gray-valued images, all images of the same size, we can load in the images as a dataset with this function:

```
def load_images(image_dir):
    image_arrays = []
    report_width_height = True

    for root, dirs, files in os.walk(image_dir):
        for file in files:
            image_file = os.path.join(root, file)
            image = Image.open(image_file)
            if report_width_height:
                (image_width, image_height) = image.size
                report_width_height = False
            image_array = numpy.array(image);
            image_array = image_array / 255.
            image_arrays.append(image_array.flatten())

    image_arrays = numpy.asarray(image_arrays,
                                 dtype=theano.config.floatX)
    return (theano.shared(image_arrays, borrow=True),
            image_width,
            image_height)
```

The `image_width` and `image_height` are returned as well to be used where the code was previously hardcoded to the shape of the MNIST dataset.

Sampling

I have restructured the sampling code for the RBM by dividing it into functions such that we can reuse some of the sampling functionality in the DBN.

Sampling from the RBM

We now sample from the RBM using the function

```
def sample_RBM(rbm, test_set_x, n_chains, n_samples, rng, plot_every,
               image_width, image_height, output_image_path):
    # select test set examples at layer
    persistent_vis_chain = random_data_samples(test_set_x,
                                               n_chains,
                                               rng)

    # run the Gibbs chain to sample a new visible vector
    vis_mfs, vis_samples = run_gibbs_chain(rbm,
                                           persistent_vis_chain,
                                           plot_every,
                                           n_samples)

    # save to image
    save_samples(vis_mfs, n_samples, n_chains,
                 image_width, image_height, output_image_path)
```

where we have the functions

```
def random_data_samples(samples, n_rand, rng):
    # find out the number of samples
    n_samples = samples.get_value(borrow=True).shape[0]

    # pick random examples, with which to
    # initialize the persistent chain
    idx = rng.randint(n_samples - n_rand)
    return theano.shared(samples.get_value(borrow=True)[idx:idx + n_rand])
```

```
def run_gibbs_chain(rbm, persistent_vis_chain, plot_every, n_samples):
    # define one step of Gibbs sampling (mf = mean-field) define a
    # function that does `plot_every` steps before returning the
    # sample for plotting
    [presig_hids, hid_mfs, hid_samples, presig_vis,
     vis_mfs, vis_samples], updates =  \
                        theano.scan(rbm.gibbs_vhv,
                                outputs_info=[None,  None, None, None,
                                              None, persistent_vis_chain],
                                n_steps=plot_every)

    # add to updates the shared variable that takes care of our persistent
    # chain.
    updates.update({persistent_vis_chain: vis_samples[-1]})
    # construct the function that implements our persistent chain.
    # we generate the "mean field" activations for plotting and the actual
    # samples for reinitializing the state of our persistent chain
    sample_fn = theano.function([], [vis_mfs[-1], vis_samples[-1]],
                                updates=updates,
                                name='sample_fn')

    vis_mfs = []
    vis_samples = []

    for idx in xrange(n_samples):
        print ' ... obtaining sample ', idx
        # generate `plot_every` intermediate samples that we discard,
        # because successive samples in the chain are too correlated
        vis_mf, vis_sample = sample_fn()
        vis_mfs.append(vis_mf)
        vis_samples.append(vis_sample)

    return (vis_mfs, vis_samples)
```

```
def save_samples(vis_mfs, n_samples, n_chains,
                 img_w, img_h, output_image_path):
    # create a space to store the image for plotting
    # (we need to leave room for the tile_spacing as well)
    image_data = numpy.zeros(((img_h + 1) * n_samples + 1,
                              (img_w + 1) * n_chains - 1),
                             dtype='uint8')
    for idx in xrange(n_samples):
```

```
        # generate `plot_every` intermediate samples that we discard,
        # because successive samples in the chain are too correlated
        vis_mf = vis_mfs[idx]
        print ' ... plotting sample ', idx
        image_data[(img_h + 1) * idx:(img_h + 1) * idx + img_h, :] = \
            tile_raster_images(
                X=vis_mf,
                img_shape=(img_h, img_w),
                tile_shape=(1, n_chains),
                tile_spacing=(1, 1))
    # construct image
    image = PIL.Image.fromarray(image_data)
    image.save(output_image_path)
```

Sampling from the DBN

We can now sample from the DBN by calling the function

```
def sample_DBN(dbn, test_set_x, n_chains, n_samples, rng, plot_every,
               image_width, image_height, output_image_path):
    # select test set examples at layer x=h^0
    bottom_vis_examples = random_data_samples(test_set_x, n_chains, rng)

    # obtain a representation h^{l-1} by sampling
    # through the layers h^0 -> h^1 -> ... -> h^{l-1}
    pre_sigmoid_h, h_mean, h_sample = \
        move_sample_from_bottom_to_top(dbn, bottom_vis_examples)
    move_sample_from_bottom_to_top_theano = \
        theano.function(inputs=[], outputs=h_sample)
    top_vis_samples = theano.shared(move_sample_from_bottom_to_top_theano())

    # run the Gibbs chain at the representation h^{l-1} to sample
    # a new visible vector h^{l-1} from the top-level RBM
    vis_mfs, vis_samples = run_gibbs_chain(dbn.rbm_layers[-1],
                                           top_vis_samples,
                                           plot_every,
                                           n_samples)

    # obtain a representation x=h^0 by sampling
    # through the layers h^{l-1} -> h^{l-2} -> ... -> h^0
    pre_sigmoid_vs, v_means, v_samples = \
```

```
        move_samples_from_top_to_bottom(dbn, vis_samples, n_samples)
    move_sample_from_top_to_bottom_theano = \
        theano.function(inputs=[], outputs=v_means)
    bottom_vis_samples = move_sample_from_top_to_bottom_theano()

    # save to image
    save_samples(bottom_vis_samples, n_samples, n_chains,
                 image_width, image_height, output_image_path)
```

where we have the additional functions

```
def move_sample_from_bottom_to_top(dbn, sample):
    for k in xrange(len(dbn.rbm_layers)-1):
        pre_sigmoid_h, h_mean, h_sample = \
            dbn.rbm_layers[k].sample_h_given_v(sample)
        sample = h_sample
    return [pre_sigmoid_h, h_mean, h_sample]
```

```
def move_sample_from_top_to_bottom(dbn, sample):
    for k in reversed(xrange(len(dbn.rbm_layers)-1)):
        pre_sigmoid_v, v_mean, v_sample = \
            dbn.rbm_layers[k].sample_v_given_h(sample)
        sample = v_sample
    return [pre_sigmoid_v, v_mean, v_sample]
```

```
def move_samples_from_top_to_bottom(dbn, samples, n_samples):
    pre_sigmoid_vs = []
    v_means = []
    v_samples = []
    for i in xrange(n_samples):
        pre_sigmoid_v, v_mean, v_sample = \
            move_sample_from_top_to_bottom(dbn, samples[i])
        pre_sigmoid_vs.append(pre_sigmoid_v)
        v_means.append(v_mean)
        v_samples.append(v_sample)
    return [pre_sigmoid_vs, v_means, v_samples]
```

Results

I have trained the Deep Belief Network on two different face datasets

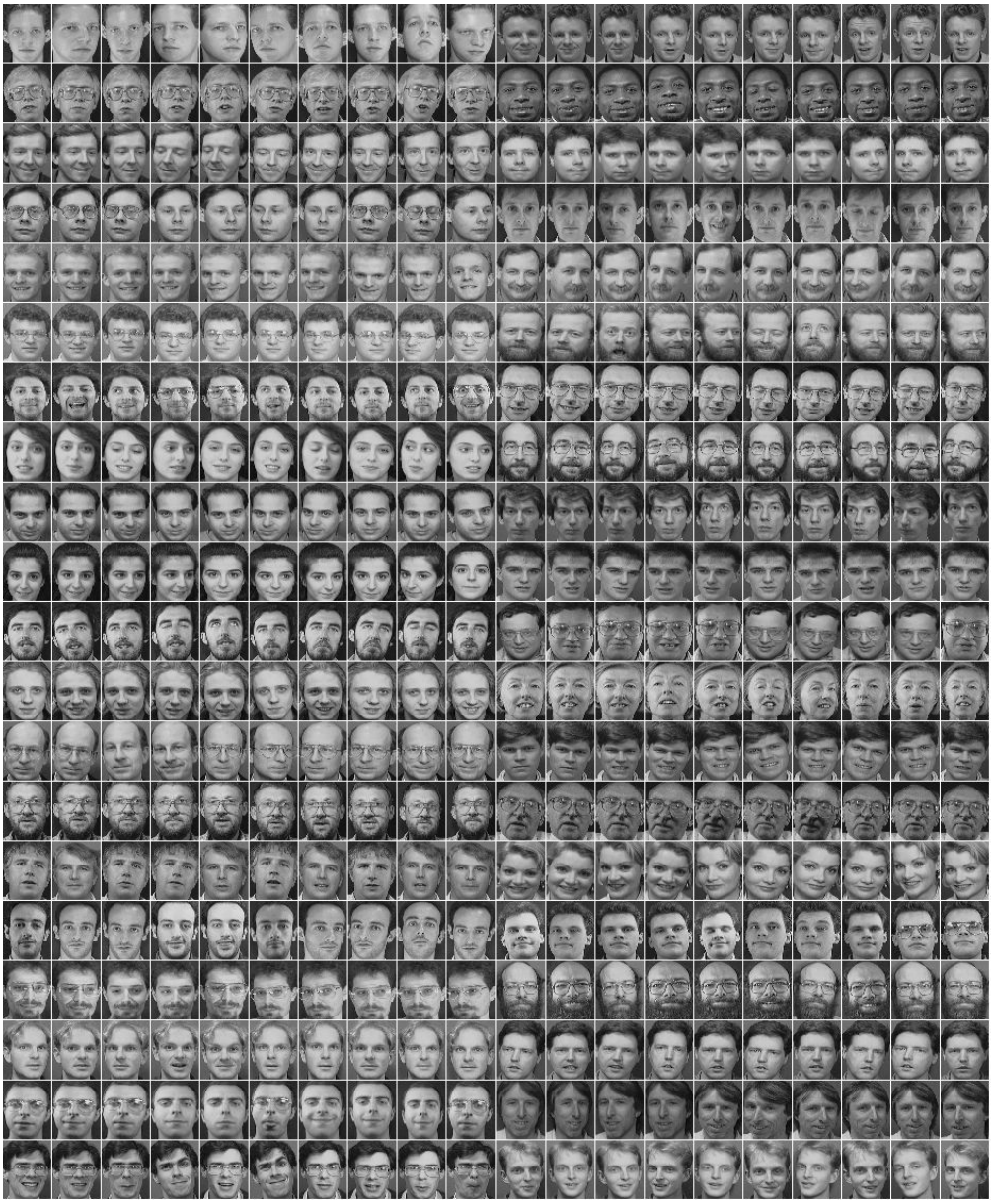
- The ORL Database of Faces
- The Extended Yale Face Database B

The Deep Belief Network was set up to have an input layer of units equal to the image resolution, and keeping the default configuration the DBN was created with 3 hidden layers, each consisting of 1000 units.

Following the sampling procedure of the RBM tutorial, we choose 20 different test examples to start 20 simultaneous Gibbs chains. Each chain is run 10 times each time taking 1000 steps, and the 20 samples that are obtained for each run of the Gibbs chains will be depicted row-wise.

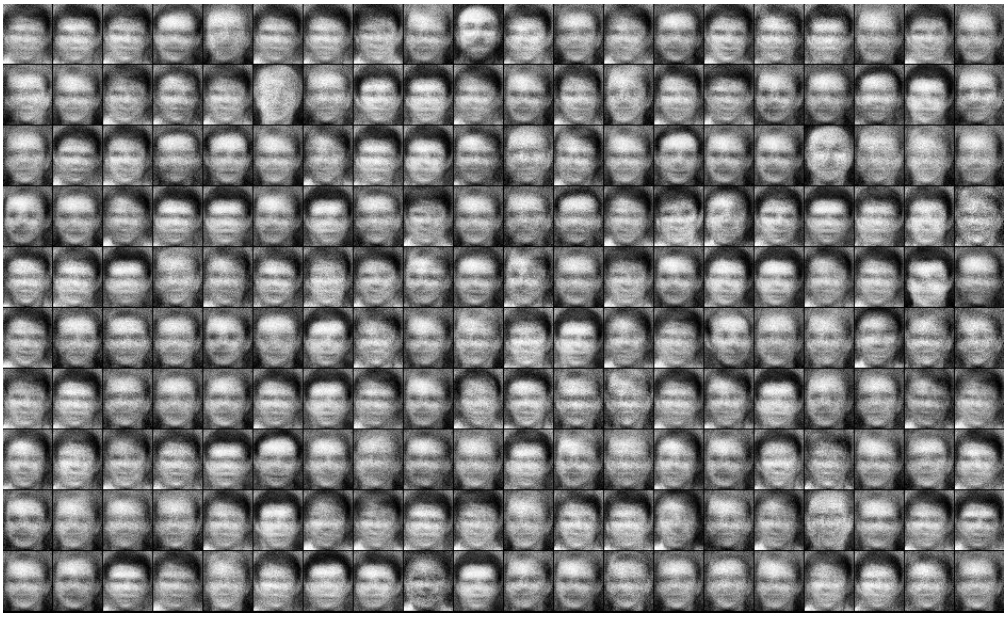
The ORL Database of Faces

The ORL Database of Faces [8] provided by AT&T Laboratories Cambridge is a small dataset of 400 faces (10 different images of each of 40 distinct subjects).



We notice that there is quite a lot of variation in the head position and facial expression, so it will be interesting to see what the DBN learns from this dataset.

Let's pull out some samples:



Due to the relatively small dataset and the observed variations, the sampled faces are a bit blurry, but nonetheless, the DBN definitely has learned something about faces!

The Extended Yale Face Database B

The Extended Yale Face Database B [4] contains 16128 images of 28 human subjects under 9 poses and 64 illumination conditions.

We will run the DBN on the Cropped version [7] which consists of 2414 selected images (excluding ambient images) that are cropped to unify the faces.

Let's see some samples:

Great! Due to the unified faces we have a lot more details in our samples, and we can conclude that the DBN has indeed learned a distribution of faces. The very dark / half-dark images are not errors, they are a result of having different lighting conditions in the dataset. For the DBN they are just as valid faces as the ones in good lighting conditions.

References

[1] 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.

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

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[4] A.S. Georghiades, P.N. Belhumeur, and D.J. Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 23(6):643–660, 2001.

[5] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082*, 2013.

[6] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In John Langford and Joelle Pineau, editors, *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, ICML '12, pages 81–88, New York, NY, USA, July 2012. Omnipress.

[7] K.C. Lee, J. Ho, and D. Kriegman. Acquiring linear subspaces for face recognition under variable lighting. *IEEE Trans. Pattern Anal. Mach. Intelligence*, 27(5):684–698, 2005.

[8] Ferdinando Samaria and Andy Harter. Parameterisation of a stochastic model for human face identification. In *Proceedings of 2nd IEEE Workshop on Applications of Computer Vision, Sarasota FL, December 1994*.

[9] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.