Application of Autoencoders*

Ke Chen

To carry out this assessed coursework, you will need the Python notebook, `AE_app.ipynb`, which contains code to get you started with each of the assignments, and the several data files required by this coursework. The files, `MNIST1_train.npz` of 3,000 instances, `MNIST1_test_1.npz` of 100 instances and `MNIST1_test_2.npz` of 1,000 instances, contain handwritten digit images selected from the MNIST dataset[1]. Those data files are required by the assignments in Parts 1 and 3, respectively. `PATCHES.npy` is an image patch dataset[2], which contains 100,000 image patches of 8×8 size to be used in Part 2. Also, you are provided with two python files: `Network.py` and `RBM.py`, which enable you to implement various autoencoders and the hybrid learning strategy required by this coursework. All above are available in a zipped file on BlackBoard alongside this document.

In general, autoencoders (AEs) are one of the most important methods used in representation learning. In this coursework, you are asked to apply the provided implementations in Python to real datasets for data compression, spatial filter learning and visualization.

Supportive Software

To do this coursework, you are provided with our own implementation of deep neural networks in Python: neural network module, `Network.py`, and restricted Boltzmann machine (RBM) module, `RBM.py`. `Network.py` enables you to carry out traditional AE, denoising AE (DAE) and sparse AE (SAE) required for tasks specified in the following assignments, while `RBM.py` allows you to apply RBM to relevant tasks as described in the assignments. Both modules allow you to construct AE or RBM objects in a similar manner. In order to use AEs or RBM, you need to create a neural network or RBM object first by using one of the provided constructors properly.

To create a fully connected feedforward neural network object, the constructor is as follows:

```
Network(layer_units,activations=["activation"])
```

where `layer_units` is an array of integers used to specify a neural network architecture with input, hidden and output layers along with the number of units in those layers. For instance, the array "[10,2,10]" indicates a neural network of input, one hidden and output layers where there are 10, 2 and 10 units in input, hidden and output layers, respectively. array "[10,8,4,1]" specifies an MLP of two hidden layers and there are 10, 8, 4 and 1 units in input, the first hidden, the second and output layers, respectively. You can use this argument to specify a fully-connected neural network of any arbitrary architecture you want.
The second argument, `activation`, can be either a string or a list of strings of the same length as

*__Assessed Coursework__: the deadline and requirements can be found at the end of this document.

[1]The information on the MNIST dataset is available: `http://yann.lecun.com/exdb/mnist/`

[2]This dataset was adapted from the dataset provided by G. Hinton on his homepage: `http://www.cs.toronto.edu/~hinton/data/patches.mat`

`layer_units`. When this argument is set as a string, it means that the specified activation function is applied to all the layers. The options of activation function are "`sigmoid`", "`relu`", "`tanh`", "`linear`". When the argument is set as a list of strings of the same length as `layer_units` where each entry corresponds to the activation function used in a specific layer apart from the input layer where no activation function is needed. For instance, the argument is specified with the list of ["`sigmoid`", "`sigmoid`", "`linear`]", meaning that the neural network has sigmoid units in two hidden layers and the linear units in output layer.

To create a RBM object, the constructor is as follows:

```
RBM(n_vis,n_hid=1024, use_gaussian_visible_sampling=False,
use_gaussian_hidden_sampling=False, use_sample_vis_for_learning=False)
```

where `n_vis` and `n_hid` are used to specify the number of visible and hidden units in the RBM, respectively. The following two arguments specify whether the hidden or visible units use Gaussian or Bernoulli distribution for sampling. The last argument allows you to choose a way for collecting the statistics needed for learning[3].

During the object creation, all the weights are initialised automatically. Afterwards, you have to set hyperparameters used in the optimiser with a function for neural networks (AEs in this coursework) or RBM:

```
Network.set_lr(lr,lr_decay=0.0,momentum=0.0,weight_decay=0.0)
```

or

```
RBM.set_lr(lr,lr_decay=0.0,momentum=0.0,weight_decay=0.0)
```

where, in order, the arguments correspond to learning rate, learning rate decay, momentum and weight decay penalty, respectively.

Finally, you can use a fit function to train a neural network (AEs in this coursework) or RBM:

```
Network.fit(x,y_true,x_val,y_true_val,batch_size,epochs,patience)
```

or

```
RBM.fit(x, x_val,batch_size,epochs,patience)
```

where `x`, `x_val` are training and validation datasets and have to be organised in the numpy matrix format where each row refers to the feature vector of a data point. `batch_size`, `epochs` specify the number of training instances in a batch and the maximum number of epochs for training. `patience` is a hyperparameter used to specify a period measured usually in epochs for early stopping. For example, if `patience` is set to 25 epochs, early stopping will occur if **no** performance improvement on validation dataset in last 25 training epochs. If `patience=-1`, then this early stopping criterion will be disabled and the training will be terminated by the maximum number of epochs specified in the `epochs` argument.

To see the complete summary of all the learnable parameters in a trained neural network after applying `Network.fit`, you are provided with a function, `Network.get_summary()`.

[3] For details, see Section 3.3 in the RBM guide: `http://www.cs.toronto.edu/~hinton/absps/guideTR.pdf`.

For test, you can use the `Network.predict(x)` and `Network.evaluate(x,y_true)` functions to achieve the prediction of `x` and the validated performance of a trained neural network. For RBM, you can use the function: `RBM.reconstruct(x,force_sample_visible=True)` to produce the input reconstruction. The second argument specifies whether the reconstructed entity yielded from hidden to visual layer is a realisation via sampling (`True`) or in a probability format (`False`).

More functions are provided for you to do different assignments. The details of those functions can be found in the relevant assignments.

To help you familiar with the provided supportive software, a complete example on a synthetic XOR dataset is offered in the notebook where you should be able to see how to use the appropriate functions to specify and train a neural network.

1 Data Compression

It is well known that AEs of a bottleneck code layer can fulfill data compression. In this work, you are asked to apply three different AEs, traditional AE, DAE and RBM, to the MNIST subset for data compression.

In general, the performance of data compression is evaluated compression ratio defined as p/d, d and p are the dimensions of raw data and its code (low-dimensional representation of raw data), respectively, and the information loss in reconstruction measured by the mean squared error (MSE). In **Assignments 1-3** described below in this section, you are asked to train different shallow AEs[4], respectively, with 3,000 training images in `MNIST1_train.npz` on different code dimensions (the number of hidden units) in the range, {40,80,160,320}, for data compression. Then, you can test your trained models on 100 images in `MNIST1_test_1.npz` As a good practice, you are suggested pre-processing data via the normalisation before training AEs.

Assignment 1 [3 marks] Apply the traditional AE of tied weights to the MINST data files described above for data compression. In your experiment, set `batch_size=100`, `momentum=0.0`, `lr=1.0`, `patience=25` and `epochs=150` (stop training when either of two termination conditions is satisfied) in common for different code dimensions. To specify the tied weights in the traditional AE, you can use the function: `Network.tie_layer_weights(layer_1_index, layer_2_index)`. In your case, set `layer_1_index=1` and `layer_2_index=3`.

Assignment 2 [3 marks] Apply the denoising AE (DAE) of tied weights to the MINST data files described above for data compression. In your experiment, use the same manner for specifying the tied weights in the DAE and the same hyperparameters given in **Assignment 1**. For training, add Gaussian noise sampled from $\mathcal{N}(0, 0.1)$ (zero mean, 0.1 standard deviation) to generate noisy data. (**Hint**: you need to clip noisy data to ensure its value range to be the same as that of clean data).

Assignment 3 [3 marks] Apply the Gaussian-Bernoulli RBM to the MINST data files described above for data compression. In your experiment, specify the Gaussian-Bernoulli RBM by setting the parameter `use_sample_vis_for_learning=False`, set the hyperparameters `lr=0.01`, `momentum=0.5`, `epochs=1,500` and keep others identical to those given in **Assignment 1**.

[4] For details of different shallow AEs, see "Shallow Autoencoder" lecture note.

For **each** of those three assignments described above, in your answer notebook, (a) describe how you train the AE with a provided function on the training dataset in a mark-up cell; (b) display the MSEs on 100 test images of AEs with a heat map (each row corresponding to a digit label) for the code dimension that leads to the minimum averaged MSE. To generate and display a heat map, you are provided a function, `plot_results(model,x_test,hidden_units)`, which returns the averaged MSE on all the images in `x_test` where `model` is a trained AE with `Network.fit` or `RBM.fit` function and `hidden_units` refers to a code dimension (e.g., `plot_results(NN,x_test,160)` indicates an trained model, `NN`, that has 160 hidden units); and (c) display the reconstructed images of 100 test instances created with the best AE of the minimum averaged MSE by arranging images in a 10 grid where each row corresponds to a digit label.

Assignment 4 [1 mark] In your answer notebook, draw a plot of $\log(\text{Compression Ratio})$ versus $\log(\text{Averaged MSE Error})$ (compression ratio in x-axis and averaged MSE error in y-axix) on all test results yielded by traditional AE, DAE and RBM. Based on results shown in this plot, state which AE performs the best in your answer notebook.

Assignment 5 [2 marks] For the Gaussian-Bernoulli RBM of a given code dimension, how can you significantly improve its performance reported in **Assignment 3** in any manner you can figure out? In your answer notebook, it is essential to describe your idea clearly and provide the solid experimental evidence against those achieved in **Assignment 3** for justification of your idea.

2 Spatial Filter Learning

In visual perception, sparse coding forms spatial filters, which plays a crucial role in extracting non-trivial features for visual recognition. Sparse AE (SAE) provides a manner to learn such spatial filters encoded in its weights from images. In this work, you are asked to apply the SAE with KL-sparsity penalty[5] to the image patch dataset for spatial filter learning. To set up the hyperparameters, ρ and λ, in SAE, you are provided a function, `Network.layers[i].enable_sparsity(lambda,rho)`, in the `Network.py` module. This function allows for applying the KL-sparsity penalty to the ith hidden layer of the sigmoid units with the specified hyperparameters ($i = 1$ corresponding to the first hidden layer will be used in this coursework).

For visualisation of weights required in the above two assignments, you are provided functions in the `Network.py` module. To read out weights and bias associated with the ith layer ($i = 1$ corresponds to the first hidden layer) in a trained neural network, you can use the provided functions, `Network.layers[i].w` and `Network.layers[i].b`.

Assignment 6 [4 marks] Apply the SAE of **no** tied weights with the architecture of encoder, **64-100**, to `PATCHES.npy` for spatial filter learning. In your experiment, set `batch_size=500`, `momentum=0.5`, `lr=1.0`, `patience=10` and `epochs=60` (stop training when either of two termination conditions is satisfied). In your experiment, you need to tune two hyperparameters in the KL-sparsity, ρ and λ, in the given ranges, $0.01 < \rho < 0.4$ and $0.001 < \lambda < 1.0$, to learn proper spatial filters from data. In your answer notebook, (a) implement a function that normalises the grey-level intensity

[5] For details of SAE and KL-sparsity penalty, see the "Autoencoder" lecture note.

of pixels to the range [0,1] within **each** patch image; (b) complete the implementation of a function, `display_filters()`, to visualise the weights associated with hidden units in the SAE; i.e., learned spatial filters, where the function must display the weight vector associated with each of all 100 hidden neurons in the 8×8 image format and arrange 100 spatial filter images in a 10×10 grid; (c) with your implemented function, `display_filters()`, display weight vectors associated with all 100 hidden neurons. You need to display 3 results produced by using the different ρ and λ values (explicitly give two hyperparameter values linked to each result) and point out which one corresponds to the best learned spatial filters.
Hints: (1) make a function to display the mean activation of hidden units, which allows you to check the sparse status of the hidden layer; (2) use the display function made by yourself to help you choose a λ value that can make **all** of the mean activations close to a specified ρ value.

Assignment 7 [1 mark] Apply traditional AE of tied weights with the architecture of encoder, **64-100**, to the dataset, `PATCHES.npy`, for over-complete representation learning. In your experiment, you should use the hyperparameter values identical to those given in **Assignment 6**. In your answer notebook, display weight vectors associated with 100 hidden neurons with your implemented function, `display_filters()`, in the same format as specified in **Assignment 6**.

3 Visualisation

Deep autoencoders (AEs) have turned out to be an effective nonlinear dimension reduction method. In this work, you are asked to apply deep AEs to the MNIST data files, `MNIST1_train.npz` and `MNIST1_test_2.npz`, for visualisation. Also, you are asked to compare a deep AE trained with the hybrid learning strategy based on RBMs to another deep AE trained with random initialisation directly for the same task.

Both deep AEs have the same architecture of encoder, **784-500-500-250-2**, all units with the sigmoid activation function apart from linear units used in the bottleneck (code) layer. In your experiment, set `batch_size=100`, `momentum=0.5`, `patience=25` and `epochs=500` in fine-tuning and direct training of the deep AE as specified in the following two assignments, but you need to tune the learning rate `lr` by yourself to achieve an acceptable performance. For early stopping, you can use the **held-out** validation on the training set of 3,000 instances in `MNIST1_train.npy`. For visualisation, you are asked to project all 1,000 test instances in `MNIST1_test_2.npy` onto the 2-D latent space with the encoder of two trained deep AEs, respectively.

Assignment 8 [5 marks] Use RBMs as the building blocks in the hybrid learning strategy[6] to construct the deep AE stated above. For the greedy layerwise pre-training, you are provided a function in in the `Network.py` module, which enable to pre-train a RBM-based deep AE with a pre-specified architecture. In your experiment, you can pre-train the RBM-based deep AE by using the function in the manner as follows:

```
pretrain_autoencoder(net, x, x_val, rbm_lr=0.1, rbm_use_gauss_visible=False,
rbm_use_gauss_hidden=False, rbm_mom=0.5, rbm_weight_decay=0.0, rbm_lr_decay=0.0,
rbm_batch_size=100, rbm_epochs=500,rbm_patience=25)
```

where `net` is a Network object (see the Supportive Software section for details) and needs to be set

[6] For details of the hybrid learning strategy, see the "Deep Autoencoder" lecture note.

to the deep AE architecture including both encoder and decoder. Apart from training and validation data settings, other hyperparameters required in the pre-training stage for construction and training of RBMs have been set up for this assignment so you can pre-train the deep AE with no change on those hyperparameters related to RBM (see the Supportive Software section for the details of those RBM hyperparameters). In your answer notebook, (a) describe your held-out validation setting in detail and report the minimum averaged reconstruction error achieved on your validation set; (b) with those hyperparameters leading to the minimum averaged reconstruction error, plot the training and validation learning curves in the fine-tuning stage and describe when you stop the training (you can use the objects, `Network.train_err_hist` and `Network.val_err_hist`, to achieve the losses of a deep AE on training and validation sets during training); and (c) display the 2-D representations of 100 test images coloured with their digit labels (the colour scheme for visualisation needs to appear in the plot).

Assignment 9 [3 marks] Train the deep AE of the architecture stated above with random initialisation. You must use the exactly same held-out validation procedure used in **Assignment 8**. In your answer notebook, (a) describe your investigation in hyperparameter tuning and explicitly list those hyperparameter values that lead to the minimum averaged reconstruction error; (b) with those hyperparameter values leading to the minimum averaged reconstruction error, plot the learning curves during training and describe when you stop the training (again, you can use the objects, `Network.train_err_hist` and `Network.val_err_hist`, to achieve the losses of a deep AE on training and validation sets during training); and (c) display the 2-D representations of 100 test images coloured with their digit labels (the colour scheme for visualisation needs to appear in the plot).

Requirement: Before starting working on this assessed coursework, you need to

1. download all the files required by this coursework from Blackboard as specified at the beginning of this document;
2. unzip the file then you should be see a Jupyter notebook file named `AE_app.ipynb` and two sub-directories named `Data` and `Code`, respectively (you must keep this directory/sub-directory structure and their names unchanged when you work on this coursework);
3. rename `AE_app.ipynb` in the directory as `yourfullname_AE_app.ipynb`. For instance, if your name is "John Smith", your filename should be `john_smith_AE_app.ipynb`. This file will be your answer notebook to be submitted for marking, so you must include everything required by the coursework in this Jupyter notebook.

Deliverable: Only your answer notebook, `yourfullname_AE_app.ipynb`, which should include all your code, output, answers and your interpretation/justification. In this Jupyter notebook, all assignments have been separated with the clear delimiters. You must put your stuff regarding an assignment in those cells related to this assignment and, if necessary, create new cells within the delimiters of this assignment.

Your answer notebook, `yourfullname_AE_app.ipynb`, must be submitted via the Blackboard.

Marking: Marking will be on the basis of (1) correctness of results and quality of comments on your code; (2) rigorous experimentation; (3) how informative and clear your results and answers are presented in your answer book; and (4) your knowledge exhibited, interpretation and justification.

Late Submission Policy: The default departmental late submission policy is applied to this coursework.

Deadline: 23:30, 19th January 2021 (Tuesday)