IT3030 - Deep Learning

Assignment 2: Semi-Supervised Learning with Deep Nets

Feb. 2020

This documents describes the second assignment in IT3030, where you will implement semi-supervised learning with a deep neural network. By doing this project, you will: a) gain experience with one deep-learning approach to semi-supervised learning, and b) learn to develop neural networks using one of the two popular machine-learning packages: Tensorflow or PyTorch.

The rules are as follows:

- The task should be solved individually.
- It should be solved in Python. Preferably,Tensorflow or PyTorch.
- Your solution will be given between 0 and 20 points; the rules for scoring are listed in the last section (Deliverables).
- Deadline for **demonstration** is March 13th, during the announced lab-hours on that day. Upload the code to Blackboard before you demonstrate your code to an evaluator.
- Please note that grading depends on how you demonstrate your code to the evaluator; submission of code is just to lock your work for the event.
- Make sure code is commented properly and submitted to blackboard well before the demonstration starts, to avoid any confusion.
- Lab hours/demonstration hours schedule will be published (announcement section) on Blackboard every week.

1 Introduction

The vast majority of machine learning applications involve supervised learning, and success typically hinges upon a large set of **labeled** cases: features plus the correct class. Unfortunately, data labeling can be a time-consuming and expensive process. One route to exploiting the power of deep networks for supervised learning in the absence of many labeled cases is semi-supervised learning, in which only some cases have labels.

The semi-supervised learning system (SSL) built for this project uses standard, fully-labeled datasets, but it processes them in a semi-supervised fashion. The process involves two key steps: 1) Using only the features (no labels) of many data cases to train an autoencoder, and b) Reusing the encoder half of the autoencoder as part of a classifier, which is trained on only a small fraction of labeled data. In theory (at least), the resulting classifier should then be capable of performing well on the test data, i.e., the remainder of the labeled data.

This exercise uses standard, benchmark datasets to highlight a choice that a machine learning expert might encounter in a real-world setting. Assume that the expert has access to a large dataset, D, where $D1 \subset D$ consists of unlabeled cases, while $D2 = D - D1$ contains labeled cases. Furthermore, $\|D2\| \ll \|D1\|$. The first approach is to train a supervised classifier network (SCN) using D2 and to then trust the generality of SCN and use it to predict the classes for every member of D1. This is a simple, but quite risky, process. The second approach attempts to mitigate that risk by first training an autoencoder on D1, thus encoding knowledge of D1's structure in the semi-supervised network (SSN). Next, SSN is reconfigured (replacing the decoder with a classifier head) and then trained in a supervised manner on D2. In theory, SSN should a) be able to train more quickly on D2 due to the knowledge transfer from the autoencoder stage, and b) have a better chance of correctly handling cases from D1. In short, the extra effort needed to train the autoencoder on D1 should pay off in terms of both ease of training and better generality of the final classifier network.

You will use the benchmark datasets to systematically explore the choice between these two alternative approaches. The evaluation of this project is based on the overall design and functionality of your code, including its ability to seamlessly run on multiple datasets, all of which contain 2-dimensional images as their features.

2 The Neural Networks

Figure 1 portrays the two deep networks used in the semi-supervised learning system: the autoencoder and the classifier, which share the encoder module. The detailed architectures of these two networks will vary, and you will need to design them yourself. Furthermore, the *optimal* architectures will normally depend upon the datasets, but you only need to implement one architecture for each component and just insure that it can successfully process each of your 4 datasets (see below for more on datasets). Optimality is not an issue in this project.

Although a single architecture for the autoencoder and classifier should be sufficient, you will have to adjust the size of the input image and the length of the classifier's output layer based on the current dataset. Your code should easily handle this on the fly. The datasets included in Tensorflow and PyTorch include critical information such as image size, number of classes, etc., which your code can easily access (and thus use to dimension your input and output layers).

2.1 The Autoencoder Module

Most autoencoders begin with large layers – i.e. dense layers containing many neurons or convolution layers housing many filters. Layer sizes (neurons, filters) normally decrease in successive layers until reaching the middle of the autoencoder, where (for this project) you will need to have a single dense layer of **latent units/neurons**. These latents serve as the output of the encoder and the input of the decoder.

A reverse process then occurs in the decoder, which gradually expands the latents through larger and larger dense (or transposed-convolution) layers, before outputing a reconstructed image with the same dimensions as the input image.

You will need to experiment with different encoder and decoder architectures. For image processing, a

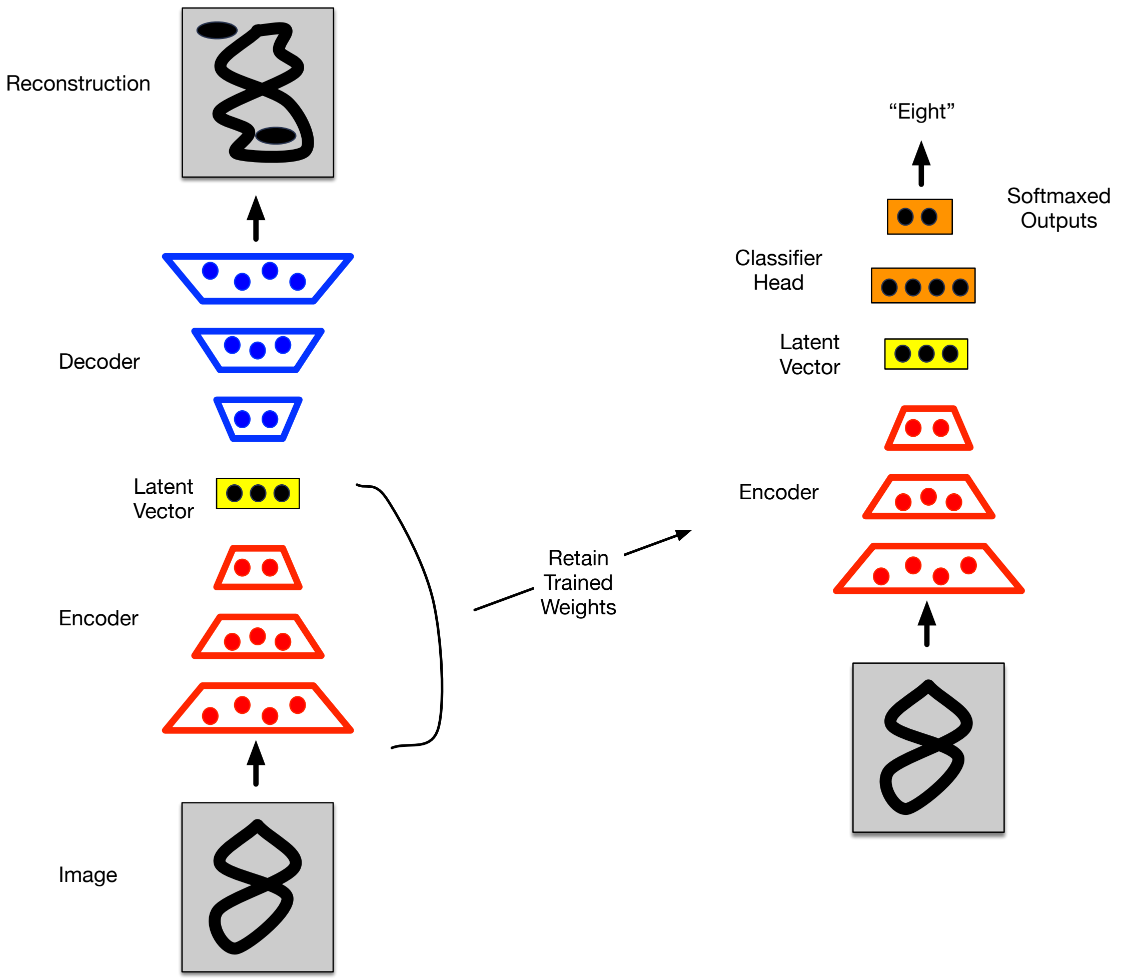

Figure 1: The two neural networks constituting a semi-supervised learner (SSL): an autoencoder (left) and a classifier (right). Note that the encoder module of the autoencoder gets reused in the classifier, with weights learned during autoencoding taken as the starting point for learning in the classifier.

standard approach incorporates a few convolution layers during encoding and then transposed convolution layers in the decoder. Both Tensorflow and PyTorch include primitives for convolution and transposed-convolution layers, so you may want to read up on them. However, many of the benchmark datasets, such as MNIST and FASHION-MNIST can be learned using only dense (non-convolution) layers.

The recursive nature of modules comes into play here, since the autoencoder must be implemented as a *model* object (in Tensorflow or Pytorch), and it must contain two *model* objects: the encoder and decoder. Then, the input to the autoencoder will also serve as input to the encoder, while the output of the decoder also constitutes the output of the autoencoder. Modularity then enables straightforward reuse of the encoder model within the classifier, also implemented as a model containing two sub-models.

2.2 The Classifier Module

The classifier combines the encoder model with a classifier-head model, which transforms the latent vector into a probability distribution over the image classes. Once again, the details of the classifier may vary: connecting the latent inputs directly to the classifier's output layer may work fine for some datasets, whereas an intermediate dense layer or two may give better results for more complex datasets.

3 Network Training and Testing

Initially, you will need to open a complete dataset and load in some fraction of its contents. Let us denote that subset DSS. Working with the entire dataset may be too time consuming, so feel free to choose any sized DSS that suits your computing resources. However, be sure to include a balanced sample of cases in DSS. So, for example, in MNIST, the DSS should contain approximately the same number of images for each of the 10 digits.

As depicted in Figure 2, the basic semi-supervised learning procedure is as follows:

1. Divide the DSS into subsets D1 and D2.
2. Train the autoencoder using all of the images in D1.
3. Reusing the encoder in the classifier (C1), train C1 using only a fraction of the cases (FC) in D2.
4. Test C1 on the remaining cases (D2 -FC) and record the accuracy.

To compare the semi-supervised approach to the purely supervised approach:

1. Build a second classifier (C2) and train it, from scratch (i.e. no autoencoder pre-training), on the cases in FC.
2. Test C2 on D2 - FC and record the accuracy.
3. Finally, test both C1 and C2 on all of D1 and compare the accuracies.

A comparison of the training and test results of C1 to those of C2 should indicate whether or not autoencoder pre-training provided any advantage over standard supervised learning (of a small fraction of labeled cases).

Your semi-supervised classifier will begin training while using the encoder weights that were learned during autoencoder training. A simple *freeze* flag in your system will govern whether or not those weights will be frozen (and thus not modified) during classifier training. During the demonstration session, it must be possible to run the system in either frozen or unfrozen mode. Tensorflow and PyTorch include simple mechanisms for freezing weights.[1]

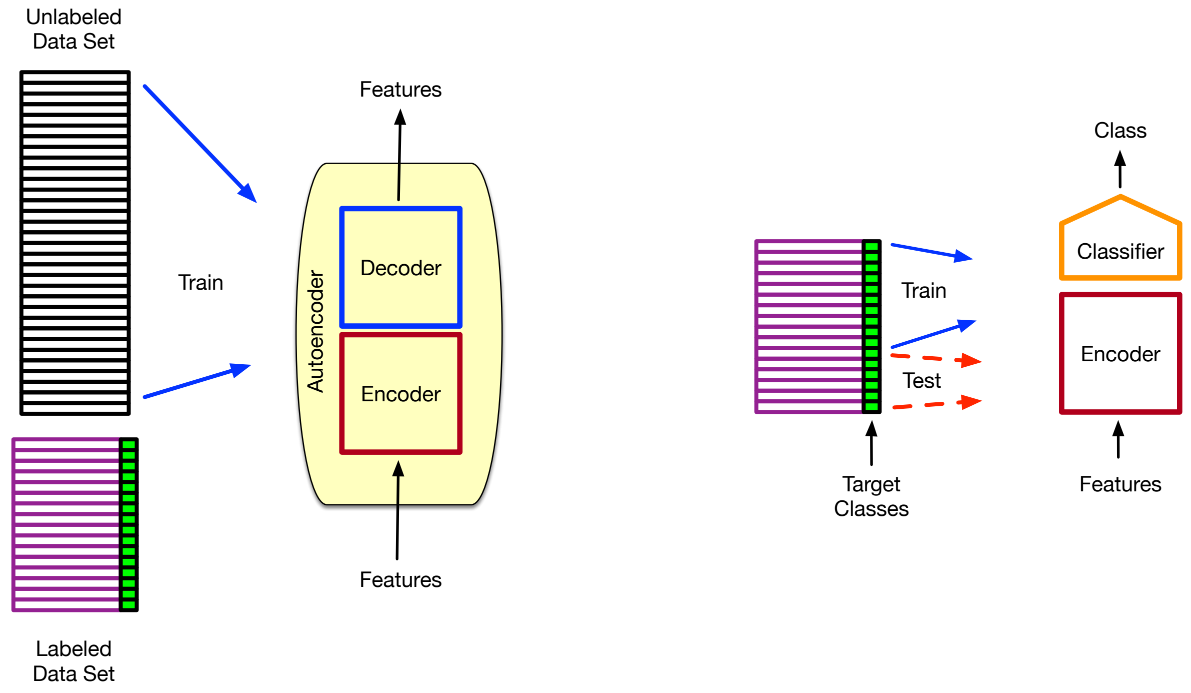

Figure 2: Two main processes in the semi-supervised learning of this project: (Left) Divide the dataset into a larger (black) and smaller (purple) set, D1 and D2, respectively. Ignore the labels of D1 and use it to train the autoencoder. (Right) Reuse the encoder module of the autoencoder and combine it with a classifier module. Train this classifier net using a subset of D2, and then test it using the remainder of D2.

4 Visualization

Your system must display three main graphics: the evolution of loss and accuracy during training, sample reconstructions produced by the autoencoder, and the clustering of latent vectors. Each is explained below.

4.1 Graphs of Learning Progress

Plots of the evolution of loss and (for classification) accuracy are essential tools for assessing the progress of learning. For this project, three main plots are required:

1. For autoencoder training, plot the loss for both the training set and the validation set, both as a function of the training epoch.

[1]In Keras, the weights and biases of a layer can be frozen by setting the *trainable* property of a layer to False. However, the Keras model containing that layer must be compiled AFTER modifying *trainable* in order for the change to take effect.

2. For the semi-supervised classifier training, plot the accuracy as a function of the training epoch for both the training and the validation sets.
3. For training of the purely supervised classifier, plot the accuracy as a function of the training epoch for both the training and validation sets.

For ease of comparison, plots 2 and 3 (above) should be on the same graph. This comparison should enable the user to quickly assess whether or not semi-supervised learning offered any advantage over basic supervised learning. Figure 3 illustrates each of these plots.

On the right of Figure 3, note that the semi-supervised run achieved a high accuracy (for training and validation sets) much faster than the purely supervised run. This is the expected result, though certainly not that which occurs everytime. In general, the pre-trained layers provided a good head start for classifier training (which, in a practical setting involving a large, complex dataset) should allow the system to achieve good performance with a minimum of supervised training.

However, making a fair comparison can be difficult. The pre-trained net was able to use a higher learning rate (by an order of magnitude) than the purely-supervised net. The graph of Figure 3 is an attempt at comparing each net using the best supervised learning rate for each. When given the higher learning rate, the purely-supervised net is unstable, and when given the lower learning rate, the semi-supervised net behaves similar to the other net.

In this project, try to find your own comparisons that highlight the differences between the two approaches, and be very clear about the assumptions (i.e. modeling choices) underlying the results.

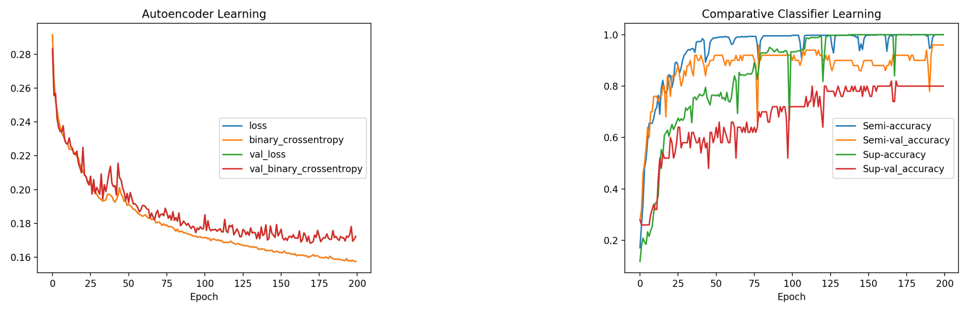

Figure 3: Graphs of learning progress on MNIST cases. (Left) Training-set and validation-set loss as a function of epoch during autoencoder training. The *loss* and *binary-cross-entropy* plots are identical in this diagram; the latter denotes the evaluation metric used during autoencoder training, which equals the loss in this case. (Right) Comparative evolution of accuracy on the training and validation sets for the semi-supervised classifier and the conventional (supervised only) classifier. Accuracy is simply the fraction of correctly-classified images. Note that during supervised training, the semi-supervised network could exploit a high learning rate (0.01), while the purely supervised network was dependent upon a lower rate (0.001), since higher rates caused dramatic instability.

4.2 Autoencoder Reconstructions

After autoencoder training, simply send a small sample of (10-20) cases through the autoencoder and record the reconstructions. Then display each pair (input image, output reconstruction) either as one diagram per pair or as two diagrams: one with all inputs and the other with all reconstructions. Figure 4 shows the two-diagram option for 16 MNIST cases. These diagrams provide a nice supplement to the evolution of autoencoder training loss, giving a more tangible indication of autoencoding success or failure.

Figure 4: Visualization of a trained autoencoder's performance. (Left) 16 original images from the MNIST dataset sent to the autoencoder. (Right) 16 corresponding reconstructions, most of which are accurate, though confusion between 4, 7 and 9 still occurs.

4.3 Latent Vector Clusters

A well-trained classifier net achieves good separation between the internal neural patterns produced by inputs of different classes. Hence, when trained on MNIST data, the internal patterns for different instances of a 7 should be quite different from the internal patterns for instances of a 6. Furthermore, most instances of 7 should produce similar internal patterns.

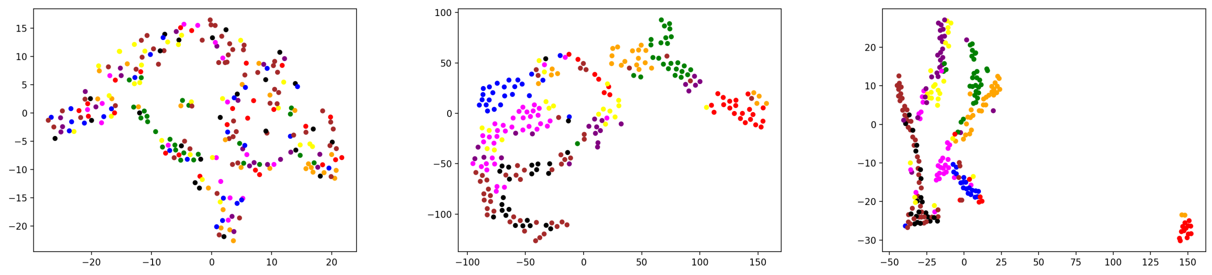

Figure 5: Three tSNE plots of 250 latent vectors produced by the encoder trained on MNIST cases. Dot colors denote digit class: 0...9 (Left) Prior to any training (Middle) Immediately after autoencoder training, (Right) After autoencoder and classifier training. Note the stronger clustering after each round of training (i.e. moving left to right).

To investigate this effect, get a hold of the encoder model (which should be a component of the autoencoder model) and run it as a stand-alone network that inputs images and outputs a latent vector. Run many (e.g., 100-200) cases through this network and record pairs consisting of a) the class of the input image, and b) the resulting latent vector. Next, send all of this data (latent vectors and their corresponding labels) to a tSNE plotting routine - SCIKIT-LEARN has a nice one. The resulting graphic displays each vector as a color-coded point (with colors corresponding to class labels) in two dimensions.[2] In a well-trained network, all or most points of the same color will form relatively tight clusters, with good separation from the other

[2]tSNE plots in 3 dimensions are possible, but they are not appropriate for this assignment.

colored clusters. Figure 5 shows tSNE plots of many latent vectors recorded at three stages of the training process, illustrating the gradual pattern-separating effect of learning.

5 The System Interface

Your system must be designed such that many runs can be done quickly, without wasting time hunting through the source code to make simple modifications. Therefore, many pivotal parameters of the system need to be determined in an *interface*, where this term is used very loosely.

You are welcome (but certainly not required) to build a nice graphical user interface (GUI) for entering these parameters. An easier alternative is a configuration file (of any format you choose) housing values for the pivotal parameters. Another option is to simply code up a main routine that takes all of the pivotal parameters as arguments.

Any of these choices are acceptable, and none is worth any more or less points than another. However, points will be lost if defining a new scenario requires you to make changes in many source files (or many distant places in the same source file). In short, insure that all pivotal parameters can be set in ONE PLACE.

The pivotal parameters for this project are the following:

1. The name of the dataset.
2. Learning rates for the autoencoder and classifier (they may vary).
3. Loss functions for the autoencoder and classifier (they will probably vary).
4. Optimizers for the autoencoder and classifier (they may vary).
5. Size of the latent vector. Use the same size for both autoencoder and classifier.
6. Number of training epochs for the autoencoder and classifier (they may vary).
7. Fraction determining how DSS will be split into D1 and D2.
8. Two fractions determining how D2 will be divided into training, validation and test sets.
9. A flag indicating whether or not to freeze the weights of the encoder module when using it in the classifier.
10. The number of autoencoder reconstructions (and corresponding input images) to display at the end of the run.
11. A flag indicating whether or not tSNE plots of latent vectors will be shown at the 3 stages of training.

You are free to (but not required to) include other parameters in the interface, such as the activation function(s) used in the two networks, the number of layers and their sizes in each network, the number and dimensions of filters in the convolution layers, etc.

6 Modularity of the Code

Modularity has several different levels in this project. First, the code must be object-oriented such that each semi-supervised learner is encapsulated in a (complex) object. Thus, it should be no problem to generate several learners during a single run of your system. Second, both Tensorflow and PyTorch include useful *model* objects that you must use for implementing the modules discussed above. These permit a recursive model structure – i.e., models inside of models – while also allowing models to be combined and recombined. This supports a very straightforward implementation of an autoencoder that reconfigures into a classifier.

7 Mandatory and Optional Datasets

Your system needs to run on both the MNIST and FASHION MNIST datasets, along with TWO other datasets (containing 2-dimensional images) of your own choosing. Both Tensorflow and PyTorch contain handlers for easily downloading and preprocessing images and labels from a wide host of datasets (including MNIST and FASHION MNIST).

The dataset **must** be one of the many arguments to your system such that, during the demonstration session, you can run a series of datasets (as requested by the examiner) without requiring changes to the source code between each dataset. Failure to satisfy this elementary requirement will result in a loss of points.

8 Deliverables

1. Object-oriented code that encapsulates each semi-supervised learner, consisting of an autoencoder and classifier, in an object. (**4 points**).
2. Proper implementation of (Tensorflow or PyTorch) models, such that each of the following is a model: encoder, decoder, autoencoder, classifier head, and classifier. (**4 points**).
3. The ability to **seamlessly** run any of the 4 datasets (described above) through your semi-supervised learner and to compare its performance to the purely supervised learner. (**4 points**).
4. The ability to display the two graphs of learning progress: autoencoder learning, and comparative classifier learning (as shown above) at the end of any run. (**2 points**).
5. The ability to display any number (between 5 and 20, as specified by the project reviewer) autoencoder reconstructions (and the corresponding input images). (**3 points**).
6. The ability to display tSNE plots (of at least 100 cases) at these three timepoints: a) prior to autoencoder training, b) immediately after autoencoder training, and c) after classifier training. (**3 points**).

WARNING: Failure to properly explain ANY portion of your code (or to convince the reviewer that you wrote the code) can result in the loss of 5 to 20 points, depending upon the seriousness of the situation. This is an individual exercise in programming, not in downloading nor copying.

A zip file containing your commented code must be uploaded to BLACKBOARD directly following your demonstration. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

The 20 total points for this project are 20 of the 100 points that are available for the entire semester.