Deep Learning Systems (ENGR-E 533) Homework 1

Instructions

Due date: Oct. 4, 2020, 23:59 PM (Eastern)

- Start early if you're not familiar with the subject, TF or PT programming, and LaTeX.
- Do it yourself. Discussion is fine, but code up on your own
- Late policy
  - If the sum of the late hours (throughout the semester) $<$ seven days (168 hours): no penalty
  - If your total late hours is larger than 168 hours, you'll get only 80% of all the late-submitted homework.
- I ask you to use either PyTorch or Tensorflow running on Python 3.
- Submit a `.ipynb` as a consolidated version of your report and code snippets. But the math should be clear with LaTeX symbols and the explanations should be full by using text cells. In addition, submit an `.html` version of your notebook where you embed your sound clips and images. For example, if you have a graph as a result of your code cell, it should be visible in this `.html` version before we run your code. Ditto for the sound examples.

Problem 1: A Detailed View to MNIST Classification [5 points]

1. Train a fully-connected net for MNIST classification (sorry, no CNN please, yet). It should be with 5 hidden layers each of which is with 1024 hidden units. Feel free to use whatever techniques you learned in class. You should be able to get the test accuracy above 98%.
2. MNIST dataset can be loaded by using an API in TF version 1.X:

   ```
   %tensorflow_version 1.x <-- To make sure Google Colab operates with version 1.x
   from tensorflow.examples.tutorials.mnist import input_data
   mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
   ```

   Or, in TF 2.x, you can do something like this:

   ```
   mnist = tf.keras.datasets.mnist
   ```

   In PT, you can use these lines of commands (don't worry about the batch size and normalization–you can go for your own option for them):

```
import torchvision
mnist_train=torchvision.datasets.MNIST('mnist',
    train=True,
    download=True,
    transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.1307,), (0.3081,))
    ]))
mnist_test=torchvision.datasets.MNIST('mnist',
    train=False,
    download=True,
    transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.1307,), (0.3081,))
    ]))
```

3. Once you're done with training, as a starter, do a feedforward step on your test samples, a thousand of them. Capture the output of the softmax layer, which will be a 10-dim probability vector per sample. In other words, each output dimension has 1,000 predictions corresponding to the 1,000 examples. For each 10-d output vector, find the dim with the maximum probability (which will eventually decide the class label). Plot the input image associated with that in a grid of subplots. For example, you can create a 10×10 grid of subplots, whose first row plots first ten input images that produced the highest probabilities for the first dim (which corresponds to "0"). Eventually, if your classification was near perfect, you'll see ten 0's in the first row, ten 1's in the second, and so on.

4. Repeat the procedure in Problem 1.3 for your second to the last layer output. This time, you should have 1024-dim vector per sample. Choose 10 random dimensions of interest and repeat the procedure in 1.3 as if the 10 out of 1024 dimensions are your output vectors. Note that there can be some dimensions that are with less than 10 images associated, because they are not popular. In your 10×10 grid, now there must be some rows that are not with enough number of images or even an empty rows. Explain your observation compared with the results from 1.3. What can you see? What would have been the ideal situation for this second-to-the-last layer? Feel free to investigate the other layers if you want, but I wouldn't care because we have a better way.

5. t-Stochastic Neighbor Embedding (tSNE) or Principal Component Analysis (PCA) are useful tools to reduce the dimension of your data and visualize. By using them, you can reduce the dimension of your data, for example, down to 2D space, so that you can scatter plot your data samples. Feel free to use whatever implementation you can find for tSNE and PCA. I'd use the one in scikit-learn.

6. First, take a thousand test samples from your MNIST dataset. Apply tSNE and PCA on the flattend 784-dim pixels. Now you have 2×1000 (or 1000×2 if you transposed the data) matrix from each of the dim reduction algorithms. Scatter plot the data samples. USE THE LABELS OF THE DATA SAMPLES SO THAT EACH SET OF SAMPLES FROM THE SAME CLASS ARE REPRESENTED WITH THE SAME COLOR. OVERLAY THE CLASS

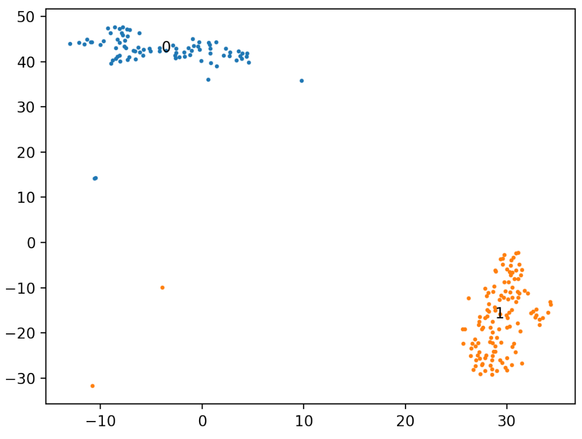

Figure 1: tSNE on two classes

LABEL ON TOP OF THE MEAN OF THE CLASS. By doing so, you can examine if your data is easy to classify or not. Do you think this raw image samples are easy to classify? For your information I share my scatter plot of the first two classes in Figure 1. It looks easy because there are only two classes, but with all 10 classes the situation will be different. Your plot should be similar to this but with all 10 classes.

7. Do a feedforward using your classifier. Capture the output of your first hidden layer, which will give you 1024×1000 matrix. What that means is that you transformed your input data into a 1024-dim space. You may hope that this makes your classification easier. Check it out by doing tSNE and PCA on this matrix, which will once again give you 2×1000 matrix. Scatter plot and check out if this layer gives you a better representation in terms of classification.

8. Repeat this procedure for all your layers including the last one. Explain your observation.

Problem 2: Speech Denoising Using Deep Learning [5 points]

1. *If you took my MLSP class, you may think that you've seen this problem. But, it's actually somewhat different from what you did before, so read carefully. And, this time you SHOULD implement a DNN with at least two hidden layers. So, don't reuse your legacy MATLAB code for this problem.*

2. When you attended IUB, you took a course taught by Prof. K. Since you really liked his lectures, you decided to record them without the professor's permission. You felt awkward, but you did it anyway because you really wanted to review his lectures later.

3. Although you meant to review the lecture every time, it turned out that you never listened to it. After graduation, you realized that a lot of concepts you face at work were actually

covered by Prof. K's class. So, you decided to revisit the lectures and study the materials once again using the recordings.

4. You should have reviewed your recordings earlier. It turned out that a fellow student who used to sit next to you always ate chips in the middle of the class right beside your microphone. So, Prof. K's beautiful deep voice was contaminated by the annoying chip eating noise.

5. But, you vaguly recall that you learned some things about speech denoising and source separation from Prof. K's class. So, you decided to build a simple deep learning-based speech denoiser that takes a noisy speech spectrum (speech plus chip eating noise) and then produces a cleaned-up speech spectrum.

6. Since you don't have Prof. K's clean speech signal, I prepared this male speech data recorded by other people. `train_dirty_male.wav` and

   `train_clean_male.wav` are the noisy speech and its corresponding clean speech you are going to use for training the network. Take a listen to them. Load them and covert them into spectrograms, which are the matrix representation of signals. To do so, you'll need to install `librosa` and use it by using the following codes:

   ```
   !pip install librosa # in colab, you'll need to install this
   import librosa
   s, sr=librosa.load('train_clean_male.wav', sr=None)
   S=librosa.stft(s, n_fft=1024, hop_length=512)
   sn, sr=librosa.load('train_dirty_male.wav', sr=None)
   X=librosa.stft(sn, n_fft=1024, hop_length=512)
   ```

   which is going to give you two matrices $\boldsymbol{S}$ and $\boldsymbol{X}$ of size 513×2459. This procedure is something called Short-Time Fourier Transform.

7. Take their magnitudes by using `np.abs()` or whatever suitable method, because $\boldsymbol{S}$ and $\boldsymbol{X}$ are complex valued. Let's call them $|\boldsymbol{S}|$ and $|\boldsymbol{X}|$.

8. Train a fully-connected deep neural network. A couple of hidden layers would work, but feel free to try out whatever structure, activation function, initialization scheme you'd like. The input to the network is a column vector of $|\boldsymbol{X}|$ (a 513-dim vector) and the target is its corresponding one in $|\boldsymbol{S}|$. You may want to do some mini-batching for this. Make use of whatever functions in Tensorflow or Pytorch.

9. But, remember that your network should predict nonnegative magnitudes as output. Try to use a proper activation function in the last layer to make sure of that. I don't care which activation function you use in the middle layers.

10. `test_01_x.wav` is the test noisy signal. Load them and apply STFT as before. Feed the magnitude spectra of this test mixture $|\boldsymbol{X}_{test}|$ to your network and predict their clean magnitude spectra $|\hat{\boldsymbol{S}}_{test}|$. Then, you can recover the (complex-valued) speech spectrogram of the test signal in this way:

$$\hat{\boldsymbol{S}} = \frac{\boldsymbol{X}_{test}}{|\boldsymbol{X}_{test}|} \odot |\hat{\boldsymbol{S}}_{test}|, \tag{1}$$

which means you take the phase information of the input noisy signal $\frac{\boldsymbol{X}_{test}}{|\boldsymbol{X}_{test}|}$ and use that to recover the clean speech. $\odot$ stands for the Hadamard product and the division is element-wise, too.

11. Recover the time domain speech signal by applying an inverse-STFT on $\hat{\boldsymbol{S}}_{test}$, which will give you a vector. Let's call this cleaned-up test speech signal $\hat{\boldsymbol{s}}_{test}$. I'll calculate something called Signal-to-Noise Ratio (SNR) by comparing it with the ground truth speech I didn't share with you. It should be reasonably good. You can actually write it out by using the following code:

    ```
    librosa.output.write_wav('test_s_01_recons.wav', sh_test, sr)
    ```

    or

    ```
    import soundfile as sf
    sf.write('test_s_01_recons.wav', sh_test, sr)
    ```

12. Do the same testing procedure for `test_02_x.wav`, which actually contains Prof. K's voice along with the chip eating noise. Enjoy his enhanced voice using your DNN.

13. Note: You cannot compute SNR without knowing the ground-truth source. So, you may wonder if your result is good enough or not. I deliberately don't share the ground-truth clean source with you, because if so, your model will overfit the test example's performance. There are a couple of ways to evaluate the performance of your system. First, you can just go ahead and listen to the input noisy signal and compare it with the processed version. You know, you can actually go ahead and listen to the result generated from the training examples, too, although they tend to be too good compared to the test examples. Second, you can also set aside a small number of training examples for validation, i.e., you don't include them during training. Once the training process is done, you apply your model to this validation set, and compute the SNR values. That way, you can *simulate* the testing environment, although it doesn't guarantee that the model will work well on the test example, because the validation set can be different from the test set. The second approach is related to the early stopping technique explained in M03 S37 (which you can actually implement if you want). For those who want to try out the second method, SNR is defined as follows:

$$\text{SNR} = 10\log_{10}\frac{\sum_t s^2(t)}{\sum_t \big(s(t)-\hat{s}(t)\big)^2}, \tag{2}$$

    where $s(t)$ and $\hat{s}(t)$ are the ground-truth clean speech and the recovered one in the time domain, respectively. Be careful with the division and logarithm: you don't want your denominator to be zero or anything inside the log function zero. Adding a very small number, e.g., $1e^{-20}$, is a good idea to prevent it.

Problem 3: Adult Optimization [6 points]

Replicate the figures in M03 Adult Optimization, slide 33 and 34 using the details as follows:

1. Use the same network architecture and train five different network instances in five different setups. The architecture has to be a fully connected network (a regular network, not a CNN or RNN) with five hidden layers, 512 hidden units per layer.
2. Create five different networks that share the same architecture as follows:
   (a) Activation function: the logistic sigmoid function; initialization: random numbers generated from the normal distribution ($\mu = 0$, $\sigma = 0.01$)
   (b) Activation function: the logistic sigmoid function; initialization: Xavier initializer
   (c) Activation function: ReLU; initialization: random numbers generated from the normal distribution ($\mu = 0$, $\sigma = 0.01$)
   (d) Activation function: ReLU; initialization: Xavier initializer
   (e) Activation function: ReLU; initialization: Kaiming He's initializer
3. You don't have to implement your own initializer. Both TF and PT come with pre-implemented initializers.
4. When you train them you have two optimizer options: the traditional SGD and Adam. Do not improve SGD by introducing momentum or any other advanced stuff. Your goal is to replicate the figures in 33 and 34. Feel free to use pre-implemented optimizers. Both TF and PT come with those optimizers.
5. In practice, you will need to investigate different learning rate as for SGD, which will give you different convergence behaviors. Let's fix it to 1.0 for this homework. As for Adam, the default setup usually works just fine.
6. Don't worry if your graphs are slightly different from mine. We will give a full mark if your graphs show the same trend.

Problem 4: Dropout [5 points]

Replicate the figures in M03 Adult Optimization, slide 40 using the details as follows:

1. For this one you will train four network instances with the same network architecture. Let's use a larger one this time, to maximize the effect of dropout: 1024×5. It has to be a fully connected network again (a regular network, not a CNN or RNN).
2. Create four different networks that share the same architecture as follows:
   (a) Activation function: the logistic sigmoid function; initialization: Xavier initializer; **no dropout**
   (b) Activation function: the logistic sigmoid function; initialization: Xavier initializer; **with dropout rate: 0.2 for the first layer and 0.5 for the other hidden layers**
   (c) Activation function: ReLU; initialization: Kaiming He's initializer; **no dropout**
   (d) Activation function: ReLU; initialization: Kaiming He's initializer; **with dropout rate: 0.2 for the first layer and 0.5 for the other hidden layers**

3. You don't have to implement your own dropout function. Both TF and PT come with pre-implemented initializers. Note that you apply dropout to the *input* of each layer (i.e. the output of its previous layer).

4. Let's just use the pre-implemented Adam optimizer with the default setup this time. Both TF and PT come with those optimizers.

5. Don't worry if your graphs are slightly different from mine. We will give a full mark if your graphs show the same trend.