Deep Learning Systems
(ENGR-E 533)
Homework 3

Instructions

Due date: Nov. 10, 2019, 23:59 PM (Eastern)

- Submit your pdf report and a zip file of source codes to Canvas
- No handwritten solutions
  - Go to `http://overleaf.com` ASAP and learn how to use LaTeX
- Start early if you're not familiar with the subject, 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.
- You can submit a `.ipynb` as a consolidated version of report and code if you want. But the math should be clear with LaTeX symbols and the explanations should be full by using text cells. Your sound clips embedded in there should be playable even before running all the codes (otherwise, submit the wavefile separately). Your images in there should be visible without running the code. Don't convert your `.ipynb` into PDF or HTML. We know how to read `.ipynb`, so PDF or HTML export will slow down the grading process and make the AIs grumpy.

Problem 1: Network Compression Using SVD [5 points]

1. Train a fully-connected net for MNIST classification. 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%. Let's call this network "baseline".
2. MNIST dataset can be loaded by using an API in TF:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

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):

```
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('/path/to/mnist/folder/',
                   train=True,
                   download=True,
                   transform=transforms.Compose([
                      transforms.ToTensor(),
                      transforms.Normalize((0.1307,), (0.3081,))
                  ])),
    batch_size=128, shuffle=True)
test_loader = torch.utils.data.DataLoader(
```

```
datasets.MNIST('/path/to/mnist/folder/',
                train=False,
                download=True,
                transform=transforms.Compose([
                   transforms.ToTensor(),
                   transforms.Normalize((0.1307,), (0.3081,))
                ])),
batch_size=128, shuffle=True)
```

3. You learned that Singular Value Decomposition (SVD) can compress the weight matrices (Module 6). You have 6 different weight matrices in your baseline network, i.e. $\boldsymbol{W}^{(1)} \in \mathbb{R}^{784\times1024}, \boldsymbol{W}^{(2)} \in \mathbb{R}^{1024\times1024}, \cdots, \boldsymbol{W}^{(5)} \in \mathbb{R}^{1024\times1024}, \boldsymbol{W}^{(6)} \in \mathbb{R}^{1024\times10}$. Run SVD on each of them, except for $\boldsymbol{W}^{(6)}$ which is too small already, to approximate the weight matrices:

$$\boldsymbol{W}^{(l)} \approx \widehat{\boldsymbol{W}}^{(l)} = \boldsymbol{U}^{(l)}\boldsymbol{S}^{(l)}\boldsymbol{V}^{(l)\top} \tag{1}$$

   For this, feel free to use whatever implementation you can find. `tf.svd` or `torch.svd` will serve the purpose. Note that we don't compress bias (just because we're lazy).

4. If you look into the singular value matrix $\boldsymbol{S}^{(l)}$, it should be a diagonal matrix. Its values are sorted in the order of their contribution to the approximation. What that means is that you can discard the least important singular values by sacrificing the approximation performance. For example, if you choose to use only D singular values and if the singular values are sorted in the descending order,

$$\boldsymbol{W}^{(l)} \approx \widehat{\boldsymbol{W}}^{(l)} = \boldsymbol{U}^{(l)}_{:,1:D}\boldsymbol{S}^{(l)}_{1:D,1:D}\left(\boldsymbol{V}^{(l)}{}_{:,1:D}\right)^{\top}. \tag{2}$$

   You may expect the $\widehat{\boldsymbol{W}}^{(l)}$ in (2) is a worse approximation of $\boldsymbol{W}^{(l)}$ than the one in (1) due to the missing components. But, by doing so you can do some compression.

5. Vary your D from 10, 20, 50, 100, 200, to D_{full}, where D_{full} is the original size of $\boldsymbol{S}^{(l)}$ (so $D = D_{\text{full}}$ means you use (1) instead of (2)). Now you have 6 differently compressed versions that are using $\widehat{\boldsymbol{W}}^{(l)}$ for feedforward. Each of the 6 networks are using one of the 6 D values of your choice. Report the test accuracy of the six approximated networks (perhaps a graph whose x-axis is D and y-axis is the test accuracy). You'll see that when $D = D_{\text{full}}$ the test accuracy is almost as good as the baseline, while $D = 10$ will give you the worst performance. Note, however, that $D = D_{\text{full}}$ doesn't give you any compression, while smaller choices of D can reduce the amount of computation during feedforward.

6. Now you learned that the low rank approximation of $\boldsymbol{W}^{(l)}$ gives you some compression. However, you might not like the performance of the too small D values. From now on, fix your $D = 20$. Let's improve its performance. There are two different ways. Choose the one you feel like:

   (a) Initialize the weight matrices $\boldsymbol{W}^{(l)}$ of a NEW network with the SVD factorization $\boldsymbol{U}^{(l)}_{:,1:D}\bar{\boldsymbol{V}^{(l)}}^{\top}_{:,1:D}$ you learned in P1.5, where I combined $\boldsymbol{S}^{(l)}_{1:D,1:D}\boldsymbol{V}^{(l)\top}_{:,1:D}$ into a new matrix $\bar{\boldsymbol{V}^{(l)}}^{\top}_{:,1:D}$. Note that you don't replace $\boldsymbol{W}^{(l)}$ with the approximated matrix $\widehat{\boldsymbol{W}}^{(l)}$. Instead, you have to specifically use two variables $\boldsymbol{U}^{(l)}_{:,1:D}$ and $\bar{\boldsymbol{V}^{(l)}}^{\top}_{:,1:D}$ in your implementation as a new pair of parameter matrices. If you stop here, you'll get the same test performance as in P1.5. You can finetune this network though. Now this new network has new parameters to update, i.e. $\boldsymbol{U}^{(l)}_{:,1:D}$ and $\bar{\boldsymbol{V}^{(l)}}^{\top}_{:,1:D}$ (as well as the bias terms). Update them using BP. Since you initialized the new parameters with SVD, which is a pretty good starting point, you may want to use a smaller-than-usual learning rate. In this training job, you never use $\boldsymbol{W}^{(l)}$, but work on the compressed net with $\boldsymbol{U}^{(l)}_{:,1:D}$ and $\boldsymbol{V}^{(l)}_{:,1:D}$ as its parameters.

(b) Another technique you can use is to keep $\boldsymbol{W}^{(l)}$ as your parameter to update. Instead, you do SVD at every iteration and make sure the feedforward pass always uses $\widehat{\boldsymbol{W}}^{(l)} = \boldsymbol{U}^{(l)}_{:,1:D}\boldsymbol{S}^{(l)}_{1:D,1:D}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:D}$. What that means for the training algorithm is that you should think of the SVD procedure as an approximation function $\boldsymbol{W}^{(l)} \approx f(\boldsymbol{W}^{(l)}) = \boldsymbol{U}^{(l)}_{:,1:D}\boldsymbol{S}^{(l)}_{1:D,1:D}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:D}$ and the update for $\boldsymbol{W}^{(l)}$ involves the derivative $f'(\boldsymbol{W}^{(l)})$ due to the chain rule (See M6 S15 where I explained this in the quantization context). You can naïvely assume an identity here $f'(x) = x$ to streamline your BP. By doing so, you can feedforward using $\widehat{\boldsymbol{W}}^{(l)}$ while the updates are done on $\boldsymbol{W}^{(l)}$. As the feedforward is always using the SVD'ed version of the weights, the network is aware of the additional error introduced by the compression and can deal with it during training. The implementation of this technique could be tricky, because now you need to define a custom derivative of this SVD approximation function $f(\cdot)$ (an identity function). Both TF and PT give you this option. Take a look at these articles:

Tensorflow: `https://uoguelph-mlrg.github.io/tensorflow_gradients/`

PyTorch: `http://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html`

Although it takes more time to train (because you need to do SVD at every iteration), I prefer this second option. I can boost the performance of the $D = 20$ compressed network up to around 97%. You know, the compressed network is using only about 4% of the memory of the original network!

Problem 2: Speech Denoising Using RNN [5 points]

1. Audio signals natually contain some temporal structure to make use of for the prediction job. Speech denoising is a good example. In this problem, we'll come up with a reasonably complicated RNN implementation for the speech denoising job.

2. `homework3.zip` contains a folder `tr`. There are 1,200 noisy speech signals (from `trx0000.wav` to `trx1199.wav`) in there. To create this dataset, I start from 120 clean speech signal spoken by 12 different speakers (10 sentences per speaker), and then mix each of them with 10 different kinds of noise signals. For example, from `trx0000.wav` to `trx0009.wav` are all saying the same sentence spoken by the same person, while they are contaminated by different noise signals. I also provide the original clean speech (from `trs0000.wav` to `trs1199.wav`) and the noise sources (from `trn0000.wav` to `trn1199.wav`) in the same folder. For example, if you add up the two signals `trs0000.wav` and `trn0000.wav`, that will make up `trx0000.wav`, although you don't have to do it because I already did it for you.

3. Load all of them and convert them into spectrograms like you did in homework 1 problem 2. Don't forget to take their magnitudes. For the mixtures (`trxXXXX.wav`) You'll see that there are 1,200 nonnegative matrices whose number of rows is 513, while the number of columns depends on the length of the original signal. Ditto for the speech and noise sources. Eventually, you'll construct three lists of magnitude spectrograms with variable lengths: $|\boldsymbol{X}^{(l)}_{tr}|$, $|\boldsymbol{S}^{(l)}_{tr}|$, and $|\boldsymbol{N}^{(l)}_{tr}|$, where l denotes one of the 1,200 examples.

4. The $|\boldsymbol{X}^{(l)}_{tr}|$ matrices are your input to the RNN for training. An RNN (either GRU or LSTM is fine) will consider it as a sequence of 513 dimensional spectra. For each of the spectra, you want to do a prediction for the speech denoising job.

5. The target of the training procedure is something called Ideal Binary Masks (IBM). You can easily construct an IBM matrix per spectrogram as follows:

$$\boldsymbol{M}^{(l)}_{f,t} = \begin{cases} 1 & \text{if } |\boldsymbol{S}^{(l)}_{tr}|_{f,t} > |\boldsymbol{N}^{(l)}_{tr}|_{f,t} \\ 0 & \text{if } |\boldsymbol{S}^{(l)}_{tr}|_{f,t} \leq |\boldsymbol{N}^{(l)}_{tr}|_{f,t} \end{cases} \tag{3}$$

IBM assumes that each of the time-frequency bin at (f, t), an element of the $|\boldsymbol{X}_{tr}|^{(l)}$ matrix, is from either speech or noise. Although this is not the case in the real world, it works like charm most of the time by doing this operation:

$$\boldsymbol{S}_{tr}^{(l)} \approx \hat{\boldsymbol{S}}_{tr}^{(l)} = \boldsymbol{M}^{(l)} \odot \boldsymbol{X}_{tr}^{(l)}. \tag{4}$$

Note that masking is done to the complex-valued input spectrograms. Also, since masking is element-wise, the size of $\boldsymbol{M}^{(l)}$ and $\boldsymbol{X}_{tr}^{(l)}$ is same. Eventually, your RNN will learn a function that approximates this relationship:

$$\boldsymbol{M}_{:,t}^{(l)} \approx \hat{\boldsymbol{M}}_{:,t}^{(l)} = \mathrm{RNN}\big(|\boldsymbol{X}_{tr}^{(l)}|_{:,1:t}; \mathbb{W}\big), \tag{5}$$

where $\mathbb{W}$ is the network parameters to be estimated.

6. Train your RNN using this training dataset. Feel free to use whatever LSTM or GRU cells available in Tensorflow or PyTorch. I find dropout helpful, but you may want to be gentle about the dropout ratio. I didn't need too complicated network structures to beat a fully-connected network.

7. Implementation note: In theory you must be able to feed the entire sentence (one of the $\boldsymbol{X}_{tr}^{(l)}$ matrices) as an input sequence. You know, in RNNs a sequence is an input sample. On top of that, you still want to do mini-batching. Therefore, your mini-batch is a 3D tensor, not a matrix. For example, in my implementation, I collect ten spectrograms, e.g. from $\boldsymbol{X}_{tr}^{(0)}$ to $\boldsymbol{X}_{tr}^{(9)}$, to form a $513 \times T \times 10$ tensor (where T means the number of columns in the matrix). Therefore, you can think that the mini-batch size is 10, while each example in the batch is not a multidimensional feature vector, but a sequence of them. This tensor is the mini-batch input to my network. Instead of feeding the full sequence as an input, you can segment the input matrix into smaller pieces, say $513 \times T_{trunc} \times N_{mb}$, where T_{trunc} is the fixed number to truncate the input sequence and N_{mb} is the number of such truncated sequences in a mini-batch, so that the recurrence is limited to T_{mb} during training. In practice this doesn't make big difference, so either way is fine. Note that during the test time the recurrence works from the beginning of the sequence to the end (which means you don't need a truncation for testing and validation).

8. I also provide a validation set in the folder `v`. Check out the performance of your network on this dataset. Of course you'll need to see the validation loss, but eventually you'll need to check out the SNR values. For example, for a recovered validation sequence in the STFT domain, $\hat{\boldsymbol{S}}_{v}^{(l)} = \hat{\boldsymbol{M}}^{(l)} \odot \boldsymbol{X}_{v}^{(l)}$, you'll perform an inverse-STFT using `librosa.istft` to produce a time domain wave form $\hat{s}(t)$. Normally for this dataset, a well-tuned fully-connected net gives slightly above 10 dB SNR. So, your validation set should give you a number larger than that. Once again, you don't need to come up with a too large network. Start from a small one.

9. We'll test the performance of your network in terms of the test data. I provide some test signals in `te`, but not their corresponding sources. So, you can't calculate the SNR values for the test signals. Submit your recovered test speech signals in a zip file, which are the speech denoising results on the signals in `te`. We'll calculate SNR based on the ground-truth speech we set aside from you.