Deep Learning Systems
(ENGR-E 533)
Homework 3

Instructions

Due date: Nov. 5, 2023, 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 2.x 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 as well, 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: Network Compression Using SVD [2 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". You can reuse the one from the previous homework if its accuracy is good enough. Otherwise, this would be a good chance for you to improve your "baseline" MNIST classifier.

2. 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).

3. 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.

4. 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)). For example, $D_{\text{full}} = 784$ when $l = 1$ and 1024 when $l > 1$. 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.

5. Report your test accuracies of the six SVDed versions along with your baseline performance. Report the number of parameters of your SVDed networks and compare them to the baseline's. Be careful with the $\boldsymbol{S}^{(l)}$ matrices: they are diagonal matrices, meaning that there are only D nonzero elements.

6. Note that you don't have to run the SVD algorithm multiple times to vary D. Run it once, and extract different versions by varying D. That's what's good about SVD.

Problem 2: Network Compression Using SVD [2 points]

1. 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 $D = 20$ and let's improve its performance.

2. Define a NEW network whose weight matrices $\boldsymbol{W}^{(l)}$ are factorized. Again, this is a new one, different from your baseline in P1. In this new network, you don't estimate $\boldsymbol{W}^{(l)}$ directly anymore, but its factor matrices, to reconstruct $\boldsymbol{W}^{(l)}$ as follows: $\boldsymbol{W}^{(l)} = \boldsymbol{\mathcal{U}}^{(l)}{\boldsymbol{\mathcal{V}}^{(l)}}^{\top}$.

3. In other words, the feedforward is now defined like this:

$$\boldsymbol{x}^{(l+1)} \leftarrow g\left(\boldsymbol{\mathcal{U}}^{(l)}{\boldsymbol{\mathcal{V}}^{(l)}}^{\top}\boldsymbol{x}^{(l)} + \boldsymbol{b}^{(l)}\right) \tag{3}$$

4. But instead of randomly initializing these factor matrices, initialize them using the P1 SVD results of the $D = 20$ case:

$$\boldsymbol{\mathcal{U}}^{(l)} \leftarrow \boldsymbol{U}^{(l)}_{:,1:20}, \qquad {\boldsymbol{\mathcal{V}}^{(l)}}^{\top} \leftarrow \boldsymbol{S}^{(l)}_{1:20,1:20}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:20} \tag{4}$$

5. Again, note that $\boldsymbol{\mathcal{U}}$ and $\boldsymbol{\mathcal{V}}$ are the new variables that you need to estimate via optimization. They are fancier though, because they are initialized using the SVD results. If you stop here, you'll get the same test performance as in P1.

6. Finetune this network. Now this new network has new parameters to update, i.e. $\boldsymbol{\mathcal{U}}^{(l)}$ and $\boldsymbol{\mathcal{V}}^{(l)}$ (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.

7. Report the test-time classification accuracy.

Problem 3: Network Compression Using SVD [3 points]

1. Another way to improve our $D = 20$ case is to inform the training process of the SVD approximation. It's a different method from P1, where SVD was performend once after the network training was completed. This time, we do SVD at every epoch.

2. Initialize $\boldsymbol{W}^{(l)}$ using the "baseline" model. We will finetune it.

3. This time, for the feedforward pass, you never use $\boldsymbol{W}^{(l)}$. Instead, you do SVD at every iteration and make sure the feedforward pass always uses $\widehat{\boldsymbol{W}}^{(l)} = \boldsymbol{U}^{(l)}_{:,1:20}\boldsymbol{S}^{(l)}_{1:20,1:20}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:20}$.

4. What that means for the training algorithm is that you should think of the low-rank SVD procedure as an approximation function $\boldsymbol{W}^{(l)} \approx f(\boldsymbol{W}^{(l)}) = \boldsymbol{U}^{(l)}_{:,1:20}\boldsymbol{S}^{(l)}_{1:20,1:20}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:20}$.

5. Hence, 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 that your SVD approximation is near perfect (although it's not). Then, at least for the BP, you don't have to worry about the gradients as the derivative will be just one everywhere, because $f(x) = x$. By doing so, you can feedforward using $\widehat{\boldsymbol{W}}^{(l)}$ while the updates are done on $\boldsymbol{W}^{(l)}$:

$$\text{Feedforward:} \tag{5}$$

$$\text{Perform SVD: } \boldsymbol{W}^{(l)} \approx \boldsymbol{U}^{(l)}_{:,1:20}\boldsymbol{S}^{(l)}_{1:20,1:20}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:20} \tag{6}$$

$$\text{Perform Feedforward: } \boldsymbol{x}^{(l+1)} \leftarrow g\left(\boldsymbol{U}^{(l)}_{:,1:20}\boldsymbol{S}^{(l)}_{1:20,1:20}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:20}\boldsymbol{x}^{(l)} + \boldsymbol{b}^{(l)}\right) \tag{7}$$

$$\text{Backpropagation:} \tag{8}$$

$$\text{Update Parameters: } \boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \eta\frac{\partial\mathcal{L}}{\partial f(\boldsymbol{W}^{(l)})}\frac{\partial f(\boldsymbol{W}^{(l)})}{\partial \boldsymbol{W}^{(l)}} \tag{9}$$

Note that $\frac{\partial f(\boldsymbol{W}^{(l)})}{\partial \boldsymbol{W}^{(l)}} = 1$ everywhere due to our identity assumption.

6. 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 requires you to define an SVD routine running in the middle of the feedforward process. Both TF and PT provide their SVD implementations you can use:

   Tensorflow 2x: `https://www.tensorflow.org/api_docs/python/tf/linalg/svd`

   PyTorch: `https://pytorch.org/docs/stable/generated/torch.svd.html`

   Although it takes more time to train (because you need to do SVD at every iteration), I like it as I can boost the performance of the $D = 20$ compressed network up to around 97%. Considering the amount of memory saving (i.e., after the compression it uses only about 2%!), this is a great way to compress your network.

Problem 4: Speaker Verification [4 points]

1. In this problem, we are going to build a speaker verification system. It takes two utterances as input, and predicts whether they were spoken by the same speaker (positive class) or not (negative class).

2. `trs.pkl` contains an 500×16,180 matrix, whose row is a speech signal with 16,180 samples. They are the returned vectors from the `librosa.load` function. Similarly, `tes.pkl` holds a 200×22,631 matrix.

3. The training matrix is ordered by speakers. Each speaker has 10 utterances, and there are 50 such speakers (that's why there are 500 rows). Similarly, the test set has 20 speakers, each of which is with 10 utterances.

4. Randomly sample L pairs of utterances from the ten utterance of the first speaker. In theory, there are $\binom{10}{2} = 45$ pairs you can sample from (the order of the two utterances within a pair doesn't matter). You can use all 45 of them if you want. These are the *positive* examples in your first minibatch.

5. Let's construct L *negative* pairs as well. First, randomly sample L utterances from the 49 training speakers. Second, randomly sample another L utterances from the first speaker (the speaker you sampled the positive pairs from). Using these two sets, each has L examples, form another set of L pairs. If $L > 10$, you'll need to repeatedly use the first speaker's utterance (i.e. sampling with replacement). This set is your *negative* examples, each of whose pair contains an utterance from the first speaker and a random utterance spoken by a different speaker.

6. The L positive pairs and L negative pairs form your first minibatch. You have $2L$ pairs of utterances in total.

7. Repeat this process for the other training speakers, so that each speaker is represented by L positive pairs and L negative pairs. By doing so, you can form 50 minibatches with a balanced number of positive and negative pairs.

8. Train a Siamese network that tries to predict 1 for the positive pairs and 0 for the negative ones. In a minibatch, since you have L positive and L negative pairs, respectively, your net must predict L ones and L zeros, respectively.

9. I found that STFT on the signals serves the initial feature extraction process. Therefore, your Siamese network will take as input TWO spectrograms, each of which is of size $513 \times T$. I wouldn't care too much about your choice of the network architecture this time (if it works anyway), but it has to somehow predict a fixed-length feature vector for the given sequence of spectra (consequently, TWO fixted-length vectors for the pair of input spectrograms). Using the inner product of the two latent embedding vectors as the input to the sigmoid function, you'll do a logistic regression. Use your imagination and employ whatever techniques you learned in class to design/train this network.

10. Construct similar batches from the test set, and test the verification accuracy of your network. Report your test-time speaker verification performance. I was able to get a decent result ($\sim$ 70%) with a reasonable network architecture (e.g., a GRU working on STFT), which converged in a reasonable amount of time (i.e. in an hour).

11. Submit your code and accuracy on the test examples.

Problem 5: Speech Denoising Using RNN [4 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 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: $|\boldsymbol{X}_{tr}^{(l)}|$, $|\boldsymbol{S}_{tr}^{(l)}|$, and $|\boldsymbol{N}_{tr}^{(l)}|$, where l denotes one of the 1,200 examples.

4. The $|\boldsymbol{X}_{tr}^{(l)}|$ 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:

$$\boldsymbol{M}_{f,t}^{(l)} = \begin{cases} 1 & \text{if } |\boldsymbol{S}_{tr}^{(l)}|_{f,t} > |\boldsymbol{N}_{tr}^{(l)}|_{f,t} \\ 0 & \text{if } |\boldsymbol{S}_{tr}^{(l)}|_{f,t} \leq |\boldsymbol{N}_{tr}^{(l)}|_{f,t} \end{cases} \tag{10}$$

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:

$$\boldsymbol{S}_{tr}^{(l)} \approx \hat{\boldsymbol{S}}_{tr}^{(l)} = \boldsymbol{M}^{(l)} \odot \boldsymbol{X}_{tr}^{(l)}. \tag{11}$$

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:

$$\boldsymbol{M}_{:,t}^{(l)} \approx \hat{\boldsymbol{M}}_{:,t}^{(l)} = \mathrm{RNN}\big(|\boldsymbol{X}_{tr}^{(l)}|_{:,1:t}; \mathbb{W}\big), \tag{12}$$

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.