

Deep Learning Systems (ENGR-E 533) Homework 4

Instructions

Due date: Nov. 14, 2021, 23:59 PM (Eastern)

- Start early if you're not familiar with the subject, TF or PT programming, and L^AT_EX.
- 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 L^AT_EX 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. $\mathbf{W}^{(1)} \in \mathbb{R}^{784 \times 1024}$, $\mathbf{W}^{(2)} \in \mathbb{R}^{1024 \times 1024}$, \dots , $\mathbf{W}^{(5)} \in \mathbb{R}^{1024 \times 1024}$, $\mathbf{W}^{(6)} \in \mathbb{R}^{1024 \times 10}$. Run SVD on each of them, except for $\mathbf{W}^{(6)}$ which is too small already, to approximate the weight matrices:

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

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 $\mathbf{S}^{(l)}$, 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,

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

You may expect the $\widehat{\mathbf{W}}^{(l)}$ in (2) is a worse approximation of $\mathbf{W}^{(l)}$ 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 $\mathbf{S}^{(l)}$ (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 $\widehat{\mathbf{W}}^{(l)}$ 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 $\mathbf{S}^{(l)}$ 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 $\mathbf{W}^{(l)}$ 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 $\mathbf{W}^{(l)}$ are factorized. Again, this is a new one, different from your baseline in P1. In this new network, you don't estimate $\mathbf{W}^{(l)}$ directly anymore, but its factor matrices, to reconstruct $\mathbf{W}^{(l)}$ as follows: $\mathbf{W}^{(l)} = \mathbf{U}^{(l)} \mathbf{V}^{(l)\top}$.
3. In other words, the feedforward is now defined like this:

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

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

$$\mathbf{U}^{(l)} \leftarrow \mathbf{U}_{:,1:20}^{(l)}, \quad \mathbf{V}^{(l)\top} \leftarrow \mathbf{S}_{1:20,1:20}^{(l)} \mathbf{V}_{:,1:20}^{(l)\top} \quad (4)$$

5. Again, note that \mathbf{U} and \mathbf{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. $\mathbf{U}^{(l)}$ and $\mathbf{V}^{(l)}$ (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 performed once after the network training was completed. This time, we do SVD at every epoch.
2. Initialize $\mathbf{W}^{(l)}$ using the "baseline" model. We will finetune it.
3. This time, for the feedforward pass, you never use $\mathbf{W}^{(l)}$. Instead, you do SVD at every iteration and make sure the feedforward pass always uses $\widehat{\mathbf{W}}^{(l)} = \mathbf{U}_{:,1:20}^{(l)} \mathbf{S}_{1:20,1:20}^{(l)} \mathbf{V}_{:,1:20}^{(l)\top}$.
4. What that means for the training algorithm is that you should think of the low-rank SVD procedure as an approximation function $\mathbf{W}^{(l)} \approx f(\mathbf{W}^{(l)}) = \mathbf{U}_{:,1:20}^{(l)} \mathbf{S}_{1:20,1:20}^{(l)} \mathbf{V}_{:,1:20}^{(l)\top}$.

5. Hence, the update for $\mathbf{W}^{(l)}$ involves the derivative $f'(\mathbf{W}^{(l)})$ 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 $\widehat{\mathbf{W}}^{(l)}$ while the updates are done on $\mathbf{W}^{(l)}$:

Feedforward: (5)

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

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

Backpropagation: (8)

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

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

6. As the feedforward is always using the SVD'd 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 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 2x: https://www.tensorflow.org/api_docs/python/tf/custom_gradient

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 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 [3 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 $500 \times 16,180$ 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 $200 \times 22,631$ 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 fixed-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.