Deep Learning Systems (ENGR-E 533) Homework 3

Instructions

Due date: Oct. 31, 2021, 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: Data Augmentation [3 points]

1. CIFAR10 is a pretty straightforward image classification task, that consists of 10 visual object classes.
2. Download them from here[1] and be ready to use it. Both PyTorch and Tensorflow have options to conveniently load them, but I chose to download them directly and mess around because I found it easier.
3. Set aside 5,000 training examples for validation.
4. **Build your baseline CNN classifier.**
   (a) The images need to be reshaped into $32 \times 32 \times 3$ tensors.
   (b) Each pixel is an integer with 8bit encoding (from 0 to 255). Transform them down to a floating point with a range [0, 1]. 0 means a black pixel and 1 is a white one.
   (c) People like to rescale the pixels to [-1, 1] so that the input to the CNN is well centered around 0, instead of 0.5.

[1] `https://www.cs.toronto.edu/~kriz/cifar.html`

(d) I know you are eager to try out a fancier net architecture, but let's stick to this simple one:

```
1st 2d conv layer:  there are 10 kernels whose size is 5x5x3; stride=1
Maxpooling:  2x2 with stride=2
1st 2d conv layer:  there are 10 kernels whose size is 5x5x10; stride=1
Maxpooling:  2x2 with stride=2
1st fully-connected layer:  [flattened final feature map] x 20
2st fully-connected layer:  20 x 10 Softmax on the 10 classes
```

Let's stick to ReLU for activation and the Kaiming He's initializer.

(e) Train this net with an Adam optimizer with a default initial learning late (i.e. 0.001). Check on the validation accuracy at the end of every epoch. Report your validation accuracy over the epochs as a graph. This is the performance of your baseline system.

5. **Build another classifier using an augmented dataset.** Prepare four different datasets out of the original CIFAR10 training set (except for the 5,000 you set aside for validation):

   (a) I know you already changed the scale of the pixels from [0,255] to [-1,+1]. Let's go back to the intermediate range, [0,1].

   (b) **Augmented dataset #1**: Brighten every pixel in every image by 10%, e.g., by multiplying 1.1. Make sure though, that they don't exceed 1. For example, you may want to do something like this: `np.minimum(1.1*X, 1)`.

   (c) **Augmented dataset #2**: Darken every pixel in every image by 10%, e.g., by multiplying 0.9.

   (d) **Augmented dataset #3**: Flip all images horizontally (not upside down, but in the left-right direction). As if they are mirrored.

   (e) **Augmented dataset #4**: The original training set.

   (f) Merge the four augmented dataset into one gigantic training set. Since there are 45,000 images in the original training set (after excluding the validation set), after the augmentation you have 45,000×4=180,000 images. Each original image has four different versions: brighter, darker, horizontally flipped, and original versions. Note that the four share the same label: a darker frog is still a frog.

   (g) Don't forget to scale back to [-1,+1].

   (h) You better visualize a few images after the augmentation to make sure what you did is correct.

   (i) Train a fresh new network with the same architecture, but using this augmented dataset. Record the validation accuracy over the epochs.

6. Overlay the validation accuracy curve from the baseline with the new curve recorded from the augmented dataset. I ran 200 epochs for both experiments and was able to see convincing results (i.e., the data augmentation improves the validation performance).

7. In theory you have to conduct a test run on the test set, but let's forget about it.

Problem 2: Self-Supervised Learning via Pretext Tasks [4 points]

1. Suppose that you have only 50 labeled examples per class for your CIFAR10 classification problem, totaling 500 training images. Presumably it might be tough to achieve a high performance in this situation. Set aside those 500 examples from your training set (I chose the last 500 examples).

2. **The pretext task**:

   (a) We will try to improve this situation by learning from unlabeled dataset.

   (b) We will assume that the rest of the 49,500 training examples are *unlabeled*. We will create a bogus classification problem using them. Let this unlabeled examples (or the examples that you disregard their original labels) be "class 0".

   (c) "class 1": Create a new class, by vertically flipping all the images upside down.

   (d) "class 2": Create another class, by rotating the images 90 degree counter-clock wise.

   (e) Now you have three classes, each of which contains 49,500 examples that are with made-up labels.

   (f) This is not a classification problem anywhere near the original CIFAR10 problem. But, the idea here is that a classifier, that is trained to solve this problem, may need to learn some helpful features for the original CIFAR10 classification problem.

   (g) Train a network with the same setup/architecture described in Problem 3, to solve this *pretext* classification problem—a rotation estimator. In theory you need to validate every now and then to prevent overfitting, but who cares about this dummy problem? Let's forget about it and just run about a hundred epochs.

   (h) Store your model somewhere safe. Both TF and PT provide a nice way to save the net parameters.

3. **The baseline**:

   (a) Train a classifier from scratch on the 500 CIFAR10 dataset you set aside in the begining. Note that they are for the original 10-class classification problem, and you ARE doing the original CIFAR10 classification, except that you use a ridiculously small amount of dataset. Let's stick to the same architecture/setup. You may need to choose a reasonable initializer, e.g., the He initializer. You know, since the training set is too small, you may not even have to do batching.

   (b) Let's cheat here and use the *test set* of 10,000 examples as if they are our validation set. If you check on the test accuracy at every 100th epoch, you will see it overfit at some point. Record the accuracy values over iterations.

4. **The transfer learning task**:

   (a) Let's train our third classifier on the 500 CIFAR10 dataset you set aside in the begining. Again, note that they are for the original 10-class classification problem.

   (b) This time, instead of using an initializer, you will reload the weights from the pretext network you learned in P2.2. Yes, that's exactly the definition of transfer learning. But, because you learned it from an unlabeled set, and had to create a pretext task to do so, it falls in the category of *self-supervised learning*.

(c) Note that you can trasfer all the parameters in except for the final softmax layer, as the pretext task is only with 3 classes. Let's randomly initialize this last softmax layer parameters with He.

(d) Optimize the network as if you trained it from scratch. You need to reduce the learning rates for transfer learning in general. More importantly, for the ones you transfer in, they have to be substantially lower than 1×10^{-3}, e.g. $1 \times 10^{-5} or 1 \times 10^{-6}$. Meanwhile, the last softmax layer will prefer the default learning rate 1×10^{-3}, as it's randomly initialized.

(e) Report your test accuracy at every 100th epoch.

5. Draw two graphs from the two experiments, the baseline and the finetuning method, and compare the results. For your information, I ran both of them 10,000 epochs, and recorded the validation accuracy (actually, the test accuracy as I used the test set) at every 100th epoch. Of course, the point is that the self-supervised features should give improvement.

Problem 3: Speech Denoising Using RNN [3 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 perform 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:

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

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:

$$\boldsymbol{S}_{tr}^{(l)} \approx \hat{\boldsymbol{S}}_{tr}^{(l)} = \boldsymbol{M}^{(l)} \odot \boldsymbol{X}_{tr}^{(l)}. \tag{2}$$

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:

$$\boldsymbol{M}_{:,t}^{(l)} \approx \hat{\boldsymbol{M}}_{:,t}^{(l)} = \mathrm{RNN}\big(|\boldsymbol{X}_{tr}^{(l)}|_{:,1:t}; \mathbb{W}\big), \tag{3}$$

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 (note though RNNs don't always appreciate the dropout technique). 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 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.