Deep Learning Systems
(ENGR-E 533)
Homework 3

Instructions

Due date: Nov. 15, 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: Network Compression Using SVD [4 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.

Problem 2: Network Compression Using SVD [4 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 $D = 20$. Let's improve its performance.

2. Define a NEW network whose weight matrices $\boldsymbol{W}^{(l)}$ are factorized. You don't estimate $\boldsymbol{W}^{(l)}$ directly anymore, but its factor matrices 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 (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 [4 points]

1. In the second approach, you keep the weight matrix $\boldsymbol{W}^{(l)}$ as your parameter to update. So, your optimizer do need to update it. Initialize $\boldsymbol{W}^{(l)}$ using the "baseline" model. We will finetune it.

2. 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 $\widehat{\boldsymbol{W}}^{(l)} = \boldsymbol{U}^{(l)}_{:,1:20}\boldsymbol{S}^{(l)}_{1:20,1:20}{\boldsymbol{V}^{(l)}}^{\top}_{:,1:20}$.

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

4. 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.

5. 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 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 1x: `https://uoguelph-mlrg.github.io/tensorflow_gradients/`

   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 prefer this second option. I can boost the performance of the $D = 20$ compressed network up to

around 97%. Considering the amount of memory saving, this is a great way to compress your network.

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

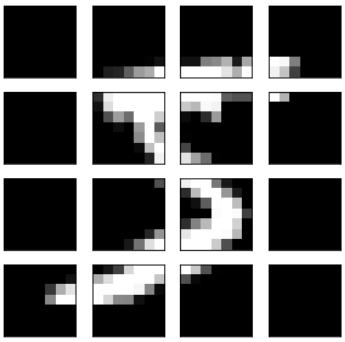
9. We'll test the performance of your network in terms of the test data. I provide some test signals in `te`, but not their corresponding sources. So, you can't calculate the SNR values for the test signals. Submit your recovered test speech signals in a zip file, which are the speech denoising results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

Problem 5: RNNs as a generative model [5 points]

1. We will train an RNN (LSTM or GRU; you choose one) that can predict the rest of the bottom half of an MNIST image given the top half. So, yes, this is a generative model that can "draw" handwritten digits.

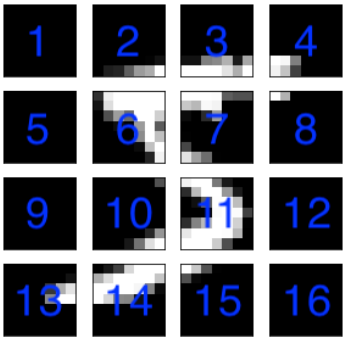
2. As the first step, let's divide every training image into 16 smaller patches. Since the original images are with 28×28 pixels, what I mean is that you need to chop off the image into

7×7 patches. There is no overlap between those patches. Below is an example from my implementation.

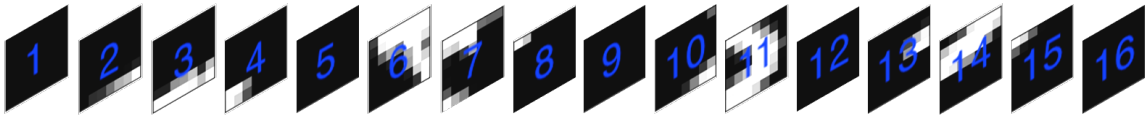


It's an image of number "5" obviously, but it's just chopped into 16 patches.

3. Let's give them an order. Let's do it from the top left corner to the bottom right corner. Below is going to be the order of the patches.



4. Now that we have an order, we'll use it to turn each MNIST image into a sequence of smaller patches. Although, below would be a potential way to turn these patches into a sequence,



I wouldn't use this way exactly, because then it is a sequence of 2d arrays, not vectors.

5. I'll keep the same order, but instead, to simplify our model architecture, we will vectorize each patch from 7×7 to a 49-dimensional vector. Finally, our sequence is a matrix $X \in \mathbb{R}^{16 \times 49}$, where 16 is the number of "time" steps. This is an input sequence to your RNN for training.

6. With a proper batch size, say 100, now an input tensor is defined as a 3D array of size $100 \times 16 \times 49$. But, I'll ignore the batch size in the equations below for keep the notation uncluttered.

7. Train an RNN out of these sequences. There must be 50,000 such sequences, or 500 mini-batches if your batch size is 100. I tried a couple of different model architectures but both worked quite well. The smallest one I tried was a 2×64 LSTM. I didn't do any fancy things like gradient clipping, as the longest sequence length is still just 16.

8. Remember to add a dense layer, so that you can convert whatever choice of the LSTM or GRU hidden dimension back to 49. You may also want to use an activation function for your output units so that the output is bounded.

9. You need to train your RNN in a way that it can predict the next patch out of the so-far-observed patches. To this end, the LSTM should predict the next patch in the following manner:

$$(Y_{t,:}, C_{t+1,:}, H_{t+1,:}) = \text{LSTM}(X_{t,:}, C_{t,:}, H_{t,:}), \tag{13}$$

where C and H denote the memory cell and hidden state, respectively (or with GRU C will be omitted), that are 0 when $t = 0$. To work as a predictive model, when you train, you need to compare $Y_{t,:}$ (the prediction) with $X_{t+1,:}$ (the next patch) and compute the loss (I used MSE as I'm lazy).

10. In other words, you will feed the input sequence $X_{1:15,:}$ (the full sequence except for the last patch) to the model, whose output $Y \in \mathbb{R}^{15\times 49}$ will need to be compared to $X_{2:16,:}$, vector-by-vector, to compute the loss:

$$\mathcal{L} = \sum_{t=2}^{16} \sum_{d=1}^{49} \mathcal{D}(X_{t,d} || Y_{t-1,d}), \tag{14}$$

where $\mathcal{D}(\cdot||\cdot)$ is a distance metric of your choice.

11. Let's use the test set to validate the model at every epoch, to see if it overfits. If it starts to overfit, stop the training process early. It took from a few to tens of minutes to train the network.

12. Once your net converges, let's move on to the fun "generation" part. Pick up a test image that belongs to a digit class, and feed its first 8 patches to the trained model. It will generate eight patches ($Y \in \mathbb{R}^{8\times 49}$), and two other vectors as the last memory cell and hidden states: $C_{9,:}, H_{9,:}$. Note that the dimension of C and H vectors depends on your choice of model complexity.

13. Then, run the model frame-by-frame, by feeding the last memory cell states, last hidden states and the *last predicted output as if it's the new input*. You will need to run this 7 times using a for loop, instead of feeding a sequence. Remember, for example, you don't know what to use as an input at $t = 9$, because we pretend like we don't know $X_{9,:}$, until you predict $Y_{8,:}$:

$$(Y_{9,:}, C_{10,:}, H_{10,:}) = \text{LSTM}(Y_{8,:}, C_{9,:}, H_{9,:}) \tag{15}$$
$$(Y_{10,:}, C_{11,:}, H_{11,:}) = \text{LSTM}(Y_{9,:}, C_{10,:}, H_{10,:}) \tag{16}$$
$$(Y_{11,:}, C_{12,:}, H_{12,:}) = \text{LSTM}(Y_{10,:}, C_{11,:}, H_{11,:}) \tag{17}$$
$$\vdots \tag{18}$$
$$(Y_{15,:}, C_{16,:}, H_{16,:}) = \text{LSTM}(Y_{14,:}, C_{15,:}, H_{15,:}) \tag{19}$$

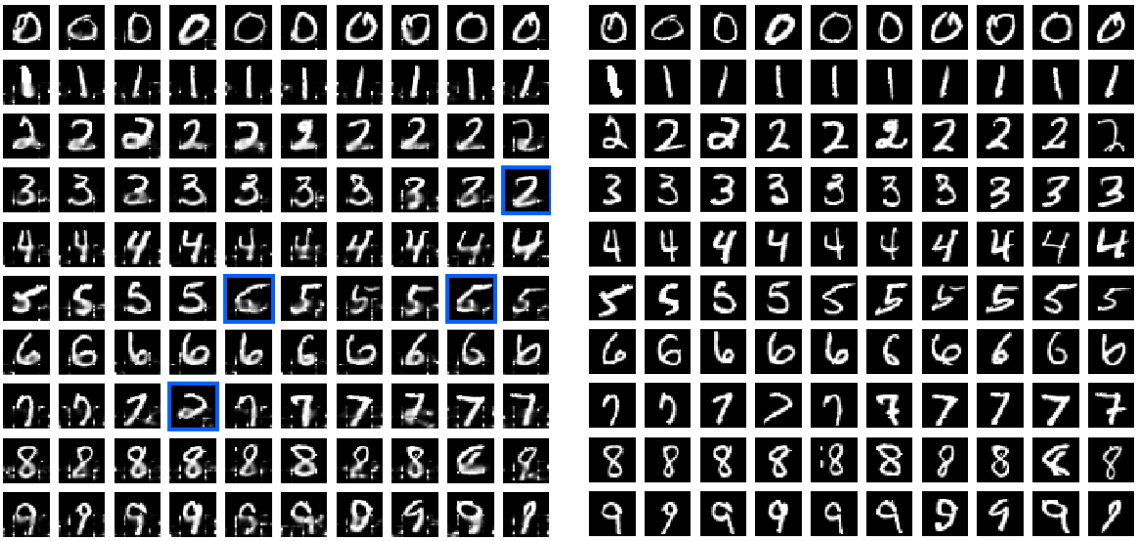
14. Note that $Y_{15,:}$ is the prediction for your 16-th patch and, e.g., $Y_{8,:}$ is the prediction for your 9-th patch, and so on. We will discard $Y_{1:7,:}$, as they are the predictions of patches that are already given (i.e., $t < 9$). Once again, you know, we pretend like the top half (patch 1 to 8) are given, while the rest (patch 9 to 16) are *NOT* known.

15. Combine the known top half $X_{1:8,:}$ and the predicted patches $Y_{8:15,:}$ into a sequence of 16 patches. We are curious of how the bottom half looks like, as they are the generated ones.

16. Reshape the synthesized matrix $[X_{1:8,:}, Y_{8:15,:}]$ back into a 28×28 image. Repeat this experiment on 10 chosen images from the same digit class.

17. Repeat the experiment for all 10 digit classes. You will generate 100 images in total.

18. Below are examples from my model. On the left, you see the examples whose bottom half is "generated" from the LSTM model, while the right images are the original (whose top half was fed to the LSTM). I can see that the model does a pretty good job, but at the same time



(a) LSTM generated images

(b) The original images

there are some interesting failure cases (blue boxes). For example, if the upper arch of "3" is too large, LSTM thinks it's 2 and draws a 2. Or, for some reason, if the some 5's are not making a sharp corner on its top left, LSTM thinks it's 6. Same story for tilted 7 that LSTM thinks it's 2. So, my point is, if *I* had to guess the botton half of these images, I'd have been confused as well.

19. Submit your 10×10 images that your LSTM generated. Submit their original images as well. Your figures should look like mine (the two figures shown above) in terms or quality. Feel free to do it better and embarrass me but you'll get a full mark if the generated images look like mine. Note that these have to be sampled from your *test* set, not the training set.