Deep Learning Systems
(ENGR-E 533)
Homework 4

Instructions

Due date: Dec. 3, 2023, 23:59 PM (Eastern)

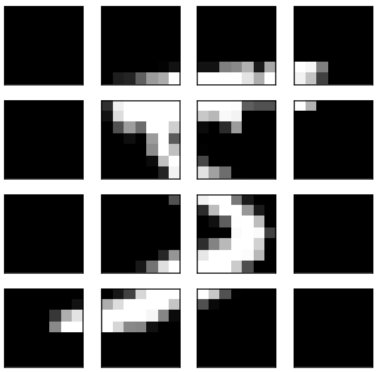
THERE IS NO GRACE PERIOD FOR THIS LAST HOMEWORK.

THE SUBMISSION SYSTEM WILL BE CLOSED AT THE DEADLINE.

- 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.

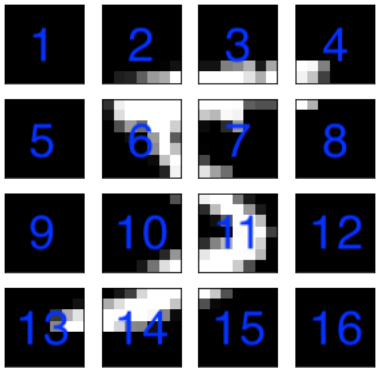
Problem 1: RNNs as a generative model [4 points]

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.

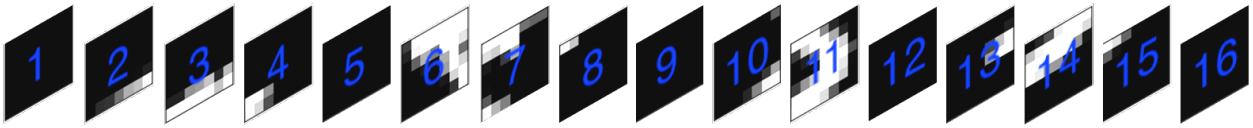


2. As the first step, let's divide every training image into 16 smaller patches. Since the original images are with 28×28 pixels, what I mean is that you need to chop off the image into 7×7 patches. There is no overlap between those patches. Above is an example from my implementation. It's an image of 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. While I'll keep the same order, 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 to keep the notation uncluttered.

7. Train an RNN out of these sequences. There must be 50,000 such sequences, or 500 minibatches 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:

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

   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:

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

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 $Y_{8,:}$:

$$(Y_{9,:}, C_{10,:}, H_{10,:}) = \text{LSTM}(Y_{8,:}, C_{9,:}, H_{9,:}) \tag{3}$$
$$(Y_{10,:}, C_{11,:}, H_{11,:}) = \text{LSTM}(Y_{9,:}, C_{10,:}, H_{10,:}) \tag{4}$$
$$(Y_{11,:}, C_{12,:}, H_{12,:}) = \text{LSTM}(Y_{10,:}, C_{11,:}, H_{11,:}) \tag{5}$$
$$\vdots \tag{6}$$
$$(Y_{15,:}, C_{16,:}, H_{16,:}) = \text{LSTM}(Y_{14,:}, C_{15,:}, H_{15,:}) \tag{7}$$

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.

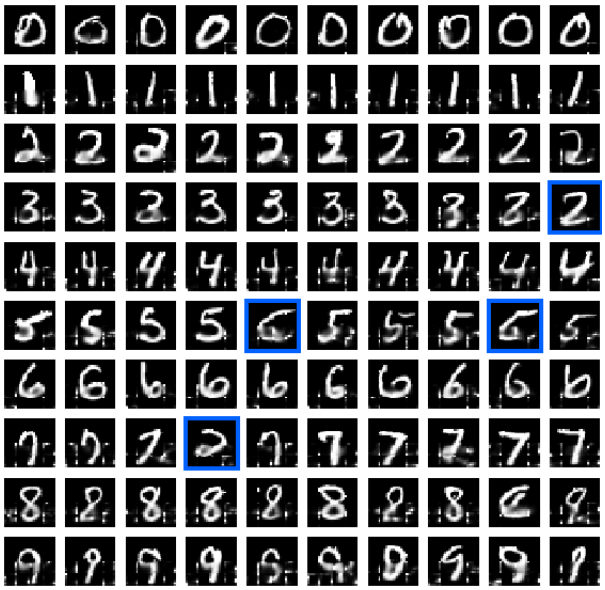
15. Combine the known top half $X_{1:8,:}$ and the predicted patches $Y_{8:15,:}$ into a sequence of 16 patches. We are curious of how the bottom half looks like, as they are the generated ones.

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.

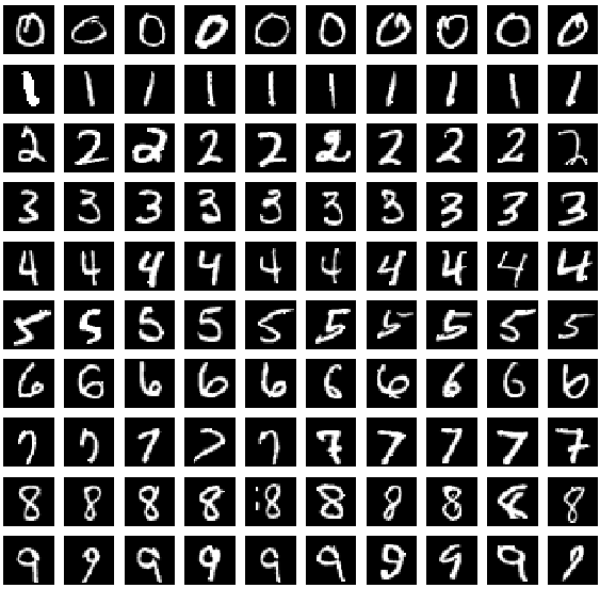
17. Repeat the experiment for all 10 digit classes. You will generate 100 images in total.

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 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 below) in terms of 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.



(a) LSTM generated images



(b) The original images

Problem 2: Variational Autoencoders on Poor Sevens [3 points]

1. `tr7.pkl` contains 6,265 MNIST digits from its training set, but not all ten digits. I only selected 7's. Therefore, it's with a rank-3 tensor of size $6,265 \times 28 \times 28$. Similarly, `te7.pkl` contains 1,028 7's.

2. The digit images in this problem are special, because I added a special effect to them. So, they are different from the original 7's in the MNIST dataset in a way. I want you to find out what I did to the poor 7's.

3. Instead of eyeballing all those images, you need to implement a VAE that finds out a few latent dimensions, one of which should show you the effect I added.

4. Once again, I wouldn't care too much about your network architecture. This could be a good chance for you to check out the performance of the CNN encoder, followed by a decoder with deconvolution layers (or transposed convolution layers), but do something else if you feel like. I found that fully-connected networks work just fine.

5. What's important here in the VAE is, as a VAE, it needs a hidden layer that is dedicated to learn the latent embedding. In this layer, each hidden unit is governed by a standard normal distribution as its *a priori* information. Also, be careful about the re-parameterization technique and the loss function.

6. You'll need to limit the number of hidden units K in your code layer (the embedding vector) with a small number (e.g. smaller than 5) to reduce your search space. Out of K, there must be a dimension that explains the effect that I added.

7. One way to prove that you found the latent dimension of interest is to show me the digits generated by the decoder. More specifically, you may want to "generate" new 7's by feeding a few randomly generated code vectors, that are the random samples from the K normal distributions that your VAE learned. But, they won't be enough to show which dimension takes care of my added effect. Therefore, your random code vectors should be formed specially.

8. What I'd do is to generate code vectors by fixing the $K-1$ dimensions with the same value over the codes, while varying only one of them.

9. For example, if $K = 3$ and you're interested in the third dimension, your codes should look like as follows:

$$\boldsymbol{Z} = \begin{bmatrix} 0.23 & -0.18 & -5 \\ 0.23 & -0.18 & -4.5 \\ 0.23 & -0.18 & -4.0 \\ 0.23 & -0.18 & -3.5 \\ 0.23 & -0.18 & -3.0 \\ & \vdots & \\ 0.23 & -0.18 & 4.5 \\ 0.23 & -0.18 & 5.0 \end{bmatrix} \tag{8}$$

   Note that the first two column vectors are once randomly sampled from the normal distributions, but then shared by all the codes so that the variation found in the decoded output relies solely on the third dimension.

10. You'll want to examine all the K dimensions by generating samples from each of them. Show me the ones you like. They should show a bunch of **similar-looking** 7's but with **gradually changing effect** on them. The generated samples that show a gradual change of the thickness of the stroke, for example, are not a good answer, because that's not the one I added, but something that was there in the dataset already.

11. Submit your notebook with figure and code.

Problem 3: Conditional GAN [3 points]

1. Let's develop a GAN model that can generate MNIST digits, but based on the auxiliary input from the user indicating which digit to create.

2. To this end, the generate has to be trained to receive two different kinds of input: the random vector and the class label.

3. The random vector is easy to create. It should be a d-dimensional vector, sampled from a standard normal distribution $\mathcal{N}(0, 1)$. $d = 100$ worked just fine for me.

4. As for the *conditioning* vector, you somehow need to inform the network of your intention. For example, if you want to generate a "0", you need to give that information to the generator. There are many different ways to condition a neural network at various stages. But, this time, let's use a simple one. We will convert the digit label into a one-hot vector. For example, if you want to generate a "7" the conditioning vector is $[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$.

5. Then, we need to combine these two different kinds of information. Again, there are many different ways, but let's just stick to a simple solution. We will concatenate the d-dimensional random vector and the 10-dimensional one-hot vector. Therefore, the input to your generator is with $d + 10$ dimensions. If your $d = 100$, the input dimension is 110.

6. You are free to choose whatever network architecture you want to practice with. Here's the fully-connected one I found as a good starting point: $110 \times 200 \times 400 \times 784$. I used ReLU as the activation function, but as for the last layer, I used tanh. It means that I'll interpret -1 as the black pixel while $+1$ being the white pixel.

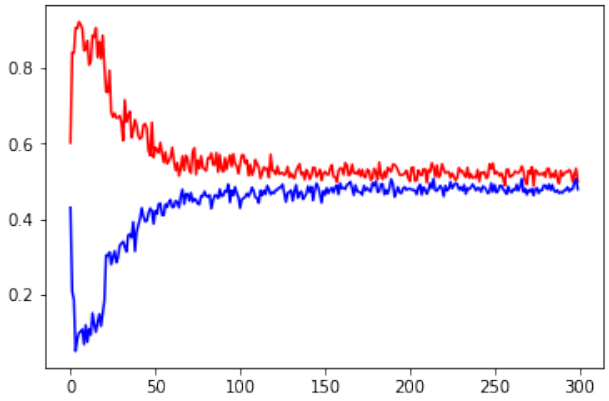
7. The discriminator has a similar architecture: $794 \times 400 \times 200 \times 100 \times 1$. The reason why it takes a 794-dim vector is that it wants to know what the image sample is conditioned on. Also note that it does binary classification to discern whether the conditioned input image is a real or fake example, i.e., you will need to set up the last layer as a logistic regression function.

8. To train this GAN model, sample a minibatch of B examples from your MNIST dataset. These are your real examples. But, instead of feeding them directly to your discriminator, you'll append their label information by turning it into the one-hot representation. Don't forget to match the scale: it has to be from -1 to $+1$ instead of $[0, 1]$ as that's how the generator defines the pixel intensity.

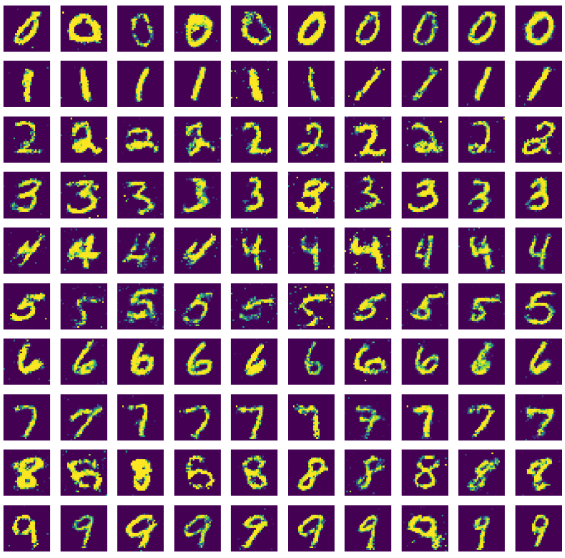
9. Accordingly, generate a class-balanced set of fake examples by feeding B random vectors to your generator. Again, each of your random vectors needs to be appended by a randomly chosen one-hot vector. For example, if your $B = 100$, you may want to generate ten ones, ten twos, and so on. Although the generated images are not with any label information anymore, you know that each should belong to a particular digit class based on your conditioning vector. Therefore, when you feed these fake examples to the discriminator, you need to append the one-hot vectors once again. Of course, the one-hot vectors should match the ones you used to inform the generator as input.

10. To summarize, the input to your generator is a $d + 10$-dim vector. The last 10 elements should be copied to augment your fake example, generated from the generator, to construct a 794-dim vector. You have B fake examples as such. The real examples are with the same size, but their first 784 elements are from the real MNIST images, accompanied by the last 10 elements representing the class, to which the image belongs.

11. Train this GAN model. I used Adam with lower-than-usual learning rates. Dropout helped the discriminator. Below is the figure that shows the change of the classification accuracy over the epochs (red for real and blue for fake examples). I can see that it converged to the Nash equilibrium, as the discriminator seems to be confused.



12. Below is the test examples that I generated by feeding new random vectors (plus the intended class labels). I placed ten examples per class in a row. These are of course not the best MNIST digits I can imagine, but they look fine given the simple structure and algorithm I used.



13. Please feel free to use whatever other things you want to try out, such as WGAN, but if your results are decent (like mine) we'll give away the full score.

14. Report both the convergence graph as well as the generated examples.

Problem 4: Missing Value Imputation Using Conditional GAN [5 points]

1. We've already seen that LSTM can act like generative model that can "predict future patches" given the "past patches" in P1.

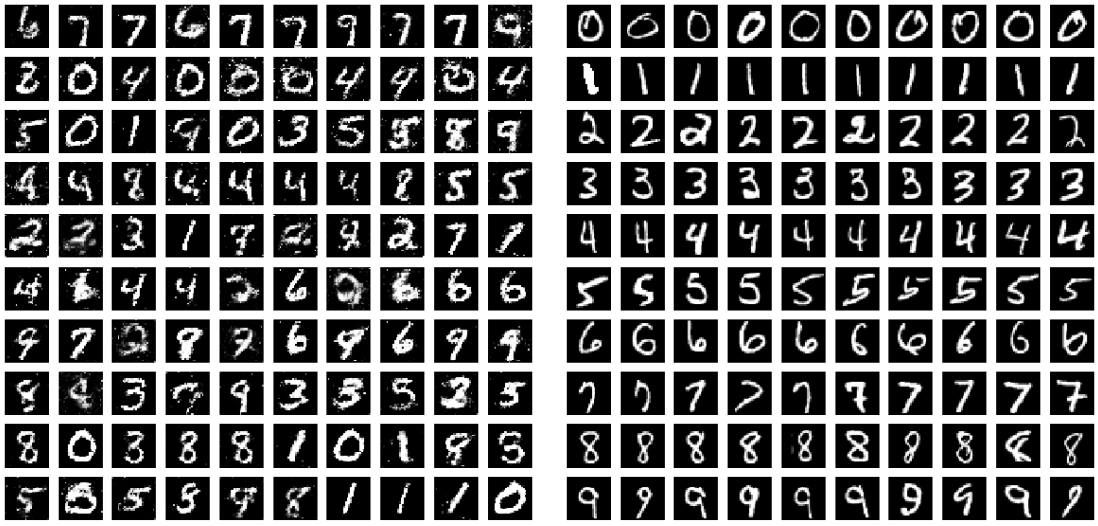
2. This time, we'll do something similar but by using GAN. This time, it works like a missing value imputation system. We assume that only the center part of the image is known, while the generator has to predict what the other surrounding pixels are.

3. We'll formulate this as a conditional GAN. First, take a batch of MNIST images. Take their center 10×10 patches, and then flatten it. This is your 100-dimensional conditioning vector. Since there are 28×28 pixels in each image, you'll do something like this, `X[:,10:19,10:19]`, to take the center patch. This will form a $B \times 100$ matrix, for your batch of B conditioning vectors.

4. Append this matrix to your random vectors of 100 dimensions drawn from the standard normal distribution. This $B \times 200$ matrix is the input to your generator.

5. The generator takes this 200 dimensional vectors and synthesizes MNIST-looking digits. You will need to prepare another set of B real examples. Eventually, you feed $2B$ examples in total to your discriminator as a minibatch.

6. If both discriminator and generator are trained properly, you can see that the results are some MNIST-looking digits. But, I found that the generator simply ignores the conditioning vector and generate whatever it wants to generate. They all certainly look like MNIST digits, but the conditioning part doesn't work. Below is the generated images (left) and the ground-truth images that I extracted the center patches from (right).

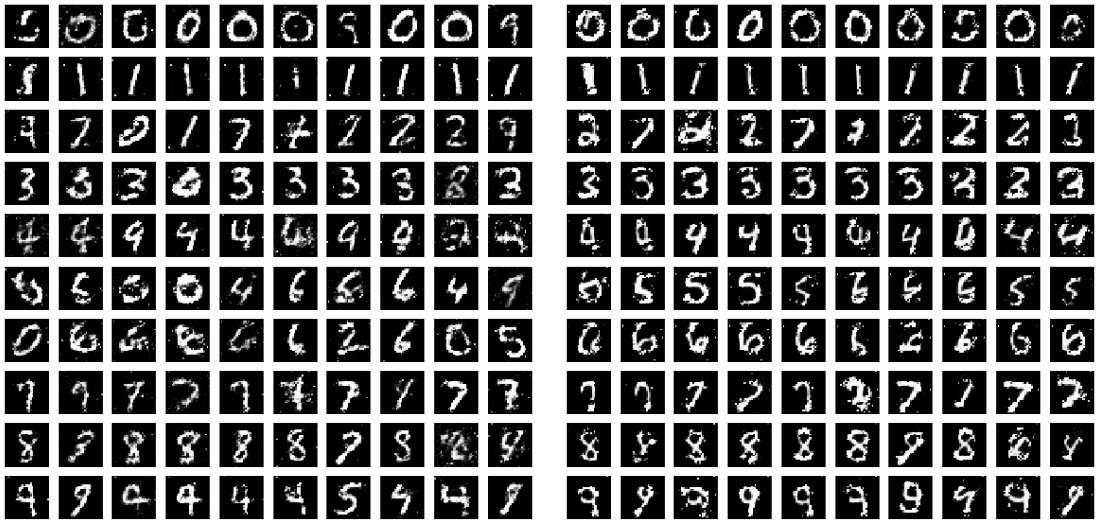


They are completely different from each other.

7. So, even though I did feed the center patch as the conditioning vector to the generator, it ignores it and generate something totally different. It's because, I think, the generator has no way to know the conditioning vector is actually the center patch of the digit that it must generate. In other words, the generator is generating the whole image, although it doesn't have to generate the center patch, which is known to me. Instead, I wanted it to generate the surrounding pixels, that are the missing values.

8. As a remedy, I added another regularizer to my generator so that it functions as an autoencoder at least for the center pixels. You know, in an ordinary GAN setup, the generator loss has to penalize the discriminator's decision that classifies the fake examples into the fake class (i.e., when the generator fails to fool the discriminator). On top of this ordinary generator loss, I add a simple mean squared error term that penalizes the difference between the conditioning vector and the center patch of the generated image, as they have to be the same, essentially.

9. Since it's a regularizer, I needed to investigate different λ values to control its contribution to the total loss of the generator. It turned out that the generator is not too sensitive to this choice, although it does generate a "less conditioned" example when it comes to a too small λ. Below is the two sets of examples when I set $\lambda = 0.1$ (left) and $\lambda = 10$ (right).



10. Replicate what I did with the regularized model and submit your code and generated examples (i.e., you don't have to replicate my failed model with no regularization). Once again, you can try some other fancy models and different ways to condition the model. But we'll give you a full score if your results are as good as mine.