Homework V

STAT/CS 6685 - Fall semester 2022

Due: Friday, November 18, 2022 - 5:00 PM

Please upload all relevant files & solutions into Canvas. Include your answers (minus code) in a single PDF titled `<lastname and initals>_HW5.pdf` and any Python scripts specified in the form of .ipynb notebooks. Any requested plots should be sufficiently labeled for full points. **DO NOT UPLOAD COMPRESSED FILES (.zip,.7z, etc.)**

Unless otherwise stated, programming assignments should use built-in functions in Python, Tensorflow, and PyTorch. In general, you may use the `scipy` stack [1]; however, exercises are designed to emphasize the nuances of machine learning and deep learning algorithms - if a function exists that trivially solves an entire problem, please consult with the TA before using it.

By design some responses have been kept as open ended. This means that different students will explore the problems differently and come up with different results by the end of their exploration process. It is not expected for two students to have identical solutions to a given problem.

Point totals for each problem are tentative and may change slightly but should be approximately correct.

Problem 1 - 20 points

1. (4 pts) Describe the similarity between PCA and an autoencoder.
2. (4 pts) What are the differences between a convolutional autoencoder and a linear autoencoder?
3. (8 pts) Give an example of an application for each of the following potential RNN scenarios:
   (a) Many inputs to many outputs where the number of inputs and outputs is the same.
   (b) One input to many outputs.
   (c) Many inputs to one output.
   (d) Many inputs to many outputs where the number of inputs and outputs may not be the same.

4. (4 pts) Give an example of an application where it would make sense to apply an RNN (or some variation such as a transformer or LSTM) but not a bidirectional RNN.

Problem 2 - (5685 - 50 points, 6685 - 40 points)

Implement an undercomplete denoising convolutional autoencoder:
Undercomplete autoencoders are autoencoders whose code dimension is less than the input dimension. Training such an autoencoder forces it to capture the most salient features of the data. However, using a large encoder and decoder with few training examples allows the network to essentially memorize the examples without learning useful features.

Follow the steps detailed below to implement an undercomplete *denoising convolutional* autoencoder, and include your code. You may find starter code below:

`https://github.com/KevinMoonLab/STAT-6685/blob/master/HW-5/Autoencoder.ipynb`.

1. Import the MNIST dataset and normalize the data in such a way that the pixel values are between 0 and 1.
   (a) (4 pts) Corrupt the training images (`X_train`) with *salt and pepper* noise, and name this new data (`X_train_SP`). You may want to use the *skimage.util.random_noise* package.
   (b) (4 pts) Corrupt the training images (`X_train`) with Gaussian noise, and name this new data (`X_train_GN`).
   (c) (2pts) Plot 5 images of `X_train`, alongside their corrupted counterparts in `X_train_SP` and `X_train_GN`.
2. (15/10 pts) Create a convolutional autoencoder as described below.

   The first two layers of the encoder should have convolution operations, followed by ReLU activations, and finally max pooling after each layer. The last pooling layer should be followed by three fully connected layers (linear activations are fine although you can choose other activations if you wish) with the inner-most layer (the second fully-connected layer) having no more than 50 neurons. Finally, use 2 deconvolution layers (*nn.ConvTranspose2d*), along with unpooling layers as appropriate, to go back to the original data shape, in this case $28 \times 28 \times 1$ (height, width, and number of channels). **Make sure your dimensions match. This is not always trivial. Since we are not specifying the size of the kernels, the padding, the stride, or the number of filters, we expect to see different architectures from everyone. Since you are required to create your own design for this, it may take some time. So start early!**
   Hint: For those using Tensorflow, you will want to use the function in tf.contrib.layers for 2D convolution and 2D convolution transpose.

3. The idea is to train the autoencoder giving it as inputs the corrupted images, and computing the loss between the reconstructed image and their uncorrupted version. Thus, make sure to define a proper loss function.

   (a) (7.5/5 pts) Implement your own training routine (i.e. use your preferred approach for optimization and tuning the hyperparameters), and make sure you are computing the loss between the uncorrupted images and the outputs of the network. Select half of the test data for validation.

   (b) (7.5/5 pts) Train an instance of your network called `AutoNN_SP`, using as training images `X_train_SP`. Create a corrupted version of the validation data with salt and pepper noise in the same fashion as we did for the training data and report your final training and validation errors.

   (c) (2.5 pts) Corrupt 10 images from the remaining test data with salt and pepper noise in the same fashion as we did for the training data. Pass them through the already trained network `AutoNN_SP`, and plot the corrupted images alongside the reconstructed version given by the autoencoder. Report your test error for these 10 images. Is the autoencoder able to denoise the data?

   (d) (5 pts) Train an instance of your network called `AutoNN_GN`, using as training images `X_train_GN`. Create a corrupted version of the validation data with Gaussian noise in the same fashion as we did for the training data and report your final training and validation errors.

   (e) (2.5 pts) Corrupt 10 images from your test data with Gaussian noise in the same fashion we did for the training data. Pass them through the already trained network `X_train_GN`, and plot the corrupted images alongside the reconstructed version given by the autoencoder. Report your test error for these 10 images. Is the autoencoder able to denoise the data?

Problem 3 - 6685 only - 20 points

Autoencoder for matrix imputation

In some cases our data contains missing values. In this exercise we will implement an autoencoder to impute the missing values where the missing values are represented using NaNs.
As shown in the figure, the autoencoder receives as input the incomplete data and retrieves an imputed version after training. The autoencoder is trained in a similar way as usual, but we need to mask the NaN values in the loss function $\mathcal{L} = \sum_{i,j:(i,j)\in S}(X_{ij} - \hat{X}_{ij})^2$. Notice that if your pixels are normalized between 0 and 1 you can also use binary cross entropy. The set S are all the (i,j) pairs with known entries. You have to mask some entries of the loss. Thus, you need to find the NaN entries and use their indices. Also make sure you use a loss that allows you to mask entries. The NaN values in your input matrix should be transformed to zeros since the layers in torch do not handle NaN entries.

Follow the steps detailed in the starter code to implement a *matrix imputation* autoencoder. We are our comparing autoencoder to some sklearn methods for imputing the missing data. Most of the code has been written for you at the starter code link below. You only need to finish the parts marked with ***. Include

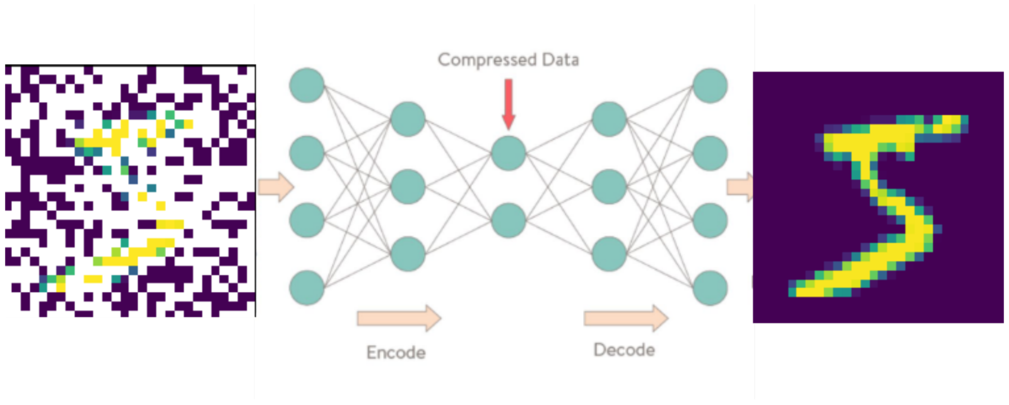

Figure 1: Autoencoder for matrix imputation.

your code.

`https://github.com/KevinMoonLab/STAT-6685/blob/master/HW-5/Matrix_Completion_AE.ipynb`

1. (10 pts) Complete the part marked with *** in *class AE_module()*, i.e. Complete the *init* method and the forward pass. Note that the *input_dim* is the number of features of your data, *hidden_dim* is a list containing the inner layers dimensions as specified above, *z_dim* is the dimension of the bottleneck.
2. (5 pts) Complete the part marked with *** in *class AE_MI()*, i.e. define the reconstruction criterion and implement the loss.
3. (5 pts) Create and fit the model. Compute the MSE of the reconstruction using *data.true_data* and the output of the 3 sklearn methods and your autoencoder. Briefly comment on these results.

Problem 4 - (5685 - 30 points, 6685 - 20 points)

Semi-supervised learning with geometry regularized neural networks (GRNN)
In this exercise we will reproduce the following architecture and employ it in a semi-supervised classification problem. Each of the inner layers is mapped to a reference representation of the data given by a dimensionality reduction method. In this way the network is regularized. The loss function is thus $\mathcal{L} = E[-log(\hat{y})] + \sum_{i=1}^{Nh} ||\hat{g}^i - \mathcal{E}||_F^2$. The first term is the usual cross entropy loss for classification and $\hat{y}$ is the output of the last green layer in the figure. The second term is the geometry regularization where g^i is the mapping from the inner layer i to the red representations, and $\mathcal{E}$ is a reference embedding that we will see next.

Consider the following semisupervised setting: We have a dataset for which only a (usually small) portion of the data is labeled. We could simply train a classifier just with the labeled observations. But, semi-supervised learning techniques include the unlabeled data to improve the performance. In this approach, we first learn a reference embedding $\mathcal{E}$ by applying a manifold learning method (such as PHATE) to all of the data. The GRNN is able to leverage this information via the geometry regularization term in which the neural network is forced to learn the structure of the data.

Note that we are implicitly assuming that knowing the general structure of the data will help us in our classification task. In other words, we are assuming that the classification function is relatively smooth, which is a common assumption.

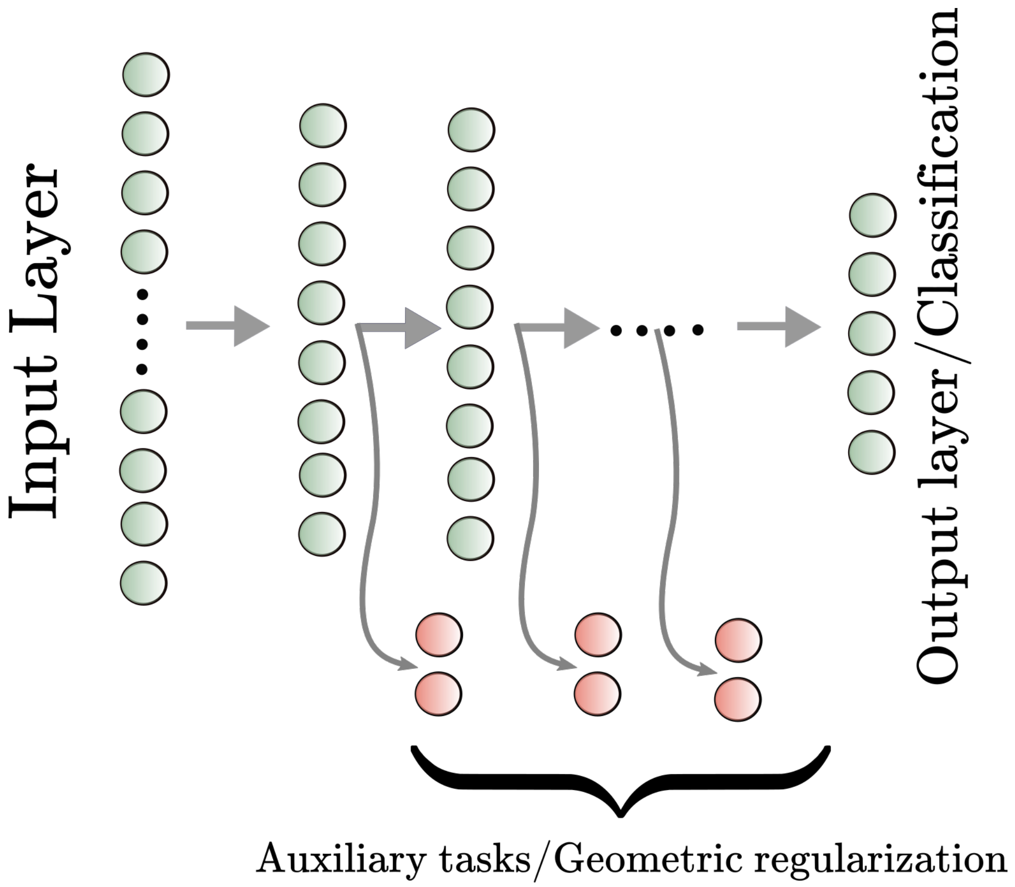
Input Layer
Output layer/Classification
Auxiliary tasks/Geometric regularization

Figure 2: Semi-supervised learning with GRNN.

Follow the steps detailed in the starter code to implement a *geometric regularization neural network*. Most of the code has been written for you in the starter code below. You only need to finish the parts marked with ***. Include your code.

`https://github.com/KevinMoonLab/STAT-6685/blob/master/HW-5/GRNN.ipynb`

1. (20/15 pts) Complete the part marked with *** in *class GRNN()*. I.e. define the classification criterion and complete the computation of the loss.
2. (10/5 pts) Run the code to train the model with $\lambda = 0$ and $\lambda = 1$. Note that $\lambda = 0$ means that the geometry regularization plays no role and we simply use a vanilla NN. Run the code below this part to

generate the confusion matrices for the vanilla NN and the regularized NN and include these in your submission. Briefly comment on what you observe.

References

[1] "The scipy stack specification." [Online]. Available: https://www.scipy.org/stackspec.html