Machine Learning for Signal Processing (ENGR-E 511) Homework 5

Due date: Apr. 10, 2022, 23:59 PM (US Eastern)

Instructions

- Submission format: Jupyter Notebook + HTML)
  - Your notebook should be a comprehensive report, not just a code snippet. Mark-ups are mandatory to answer the homework questions. You need to use LaTeX equations in the markup if you're asked.
  - Google Colab is the best place to begin with if this is the first time using iPython notebook.
  - Download your notebook as an .html version and submit it as well, so that the AIs can check out the plots and audio. Here is how to convert to html in Google Colab.
  - Meaning you need to embed an audio player in there if you're asked to submit an audio file
- Avoid using toolboxes.

P1: Kernel PCA [5 points]

1. `concetric.mat` contains 152 data points each of which is a 2-dimensional column vector. If you scatter plot them on a 2D space, you'll see that they form the concentric circles you saw in class.
2. Do kernel PCA on this data set to transform them into a new 3-dimensional feature space, i.e. $\boldsymbol{X} \in \mathbb{R}^{2\times 152} \Rightarrow$ Kernel PCA $\Rightarrow \boldsymbol{Z} \in \mathbb{R}^{3\times 152}$.
3. In theory, you have to do the centering and normalization procedures in the feature space, but for this particular data set, you can forget about it.
4. You remember how the after-transformation 3D features look like, right? Now your data set is linearly separable. Train a perceptron that does this linear classification. In other words, your perceptron will take the transformed version of your data (a 3-dimensional vector) and do a linear combination with three weights $\boldsymbol{w} = [w_1, w_2, w_3]^\top$ plus the bias term b:

$$y_i = \sigma\left(\boldsymbol{w}^\top \boldsymbol{Z}_{:,i} + b\right), \tag{1}$$

   where y_t is the scalar output of your perceptron and σ is the activation function you define. If you choose logistic function as your activation, then you'd better label your samples with 0 or 1. You know, the first 51 samples are for the inner circle, so their label will be 0, while for the other outer ring samples, I'd label them with 1.

5. You'll need to write a backpropagation algorithm to estimate your parameters $\boldsymbol{w}$ and b, which minimize the error between y_i and t_i. There are a lot of choices, but you can use the sum of the squared error: $\sum_i \frac{1}{2}(y_i - t_i)^2$. Note that this dummy perceptron doesn't have a hidden layer, because the nonlinearity is taken care of by the kernel method. Note that you may want to initialize your parameters with some small (uniform or Gaussian) random numbers centered around zero.

6. Your training procedure will be sensitive to the choice of the learning rate and the initialization scheme (the range of your random numbers). So, you may want to try out a few different choices. Good luck!

7. Do NOT use TensorFlow or PyTorch's automatic gradient computation feature for this. You have to write your own backpropagation routine in Numpy.

P2: Neural Networks [5 points]

1. Build a neural network that has a single hidden layer for the concentric circles problem. Instead of taking the kernel PCA results as the input, your neural network will take the raw $\boldsymbol{X}$ matrix as the input. Therefore, instead of a perceptron with 3 input units, now your neural network will have 2 input units as in the previous problem, each of which will take one of the coordinates of a sample.

2. As we skip kernel PCA, we need to implement the nonlinear part within the network. To this end, we will convert this 2D data point into a 3D feature vector first, a procedure that corresponds to the first layer of your neural network:

$$\boldsymbol{X}^{(2)}_{:,i} = \sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{X}^{(1)}_{:,i} + \boldsymbol{b}^{(1)}\right), \tag{2}$$

   where $\boldsymbol{X}^{(1)}_{:,i} \in \mathbb{R}^{2\times1}$ is the input to the network (i-th column vector of $\boldsymbol{X}$), while $\boldsymbol{X}^{(2)}_{:,i} \in \mathbb{R}^{3\times1}$ is the output of the hidden units. What that means is that the weight matrix $\boldsymbol{W}^{(1)} \in \mathbb{R}^{3\times2}$ and the bias vector $\boldsymbol{b}^{(1)} \in \mathbb{R}^{3\times1}$.

3. In the next (i.e., final) layer, you'll do the usual linear classification on $\boldsymbol{X}^{(2)}_{:,i}$, which will be similar to the perceptron you developed in Problem 3, although this time the input to the perceptron is $\boldsymbol{X}^{(2)}_{:,i}$, not the kernel PCA results:

$$y_i = \sigma\left(\left(\boldsymbol{w}^{(2)}\right)^\top \boldsymbol{X}^{(2)}_{:,i} + b^{(2)}\right), \tag{3}$$

   where $\boldsymbol{w}^{(2)}$ and $b^{(2)}$ correspond to $\boldsymbol{w}$ and b in Problem 1, respectively.

4. Note that you train these two layers altogether. Your backpropagation should work from the final layer back to the first layer. Eventually, you want to estimate these parameters with your backpropagation: $\boldsymbol{W}^{(1)} \in \mathbb{R}^{3\times2}, \boldsymbol{w}^{(2)} \in \mathbb{R}^{3\times1}, \boldsymbol{b}^{(1)} \in \mathbb{R}^{3\times1}, b^{(2)} \in \mathbb{R}^{1}$.

5. Again, do NOT use any of the deep learning frameworks, such as TensorFlow or PyTorch, to get around the differentiation. Write up your own backpropagation routine and update algorithms.

P3: Spoken MNIST [5 points]

1. Download the spoken digit dataset from here: https://zenodo.org/record/1342401#.YjyUfi-B0UE.

2. In the `recordings` subfolder, you will see a bunch of `.wav` files. Its encoding schme is like this: `[digit class]_[subject name]_[example number].wav`. For example, `7_jackson_30.wav` means 30th recording of Jackson pronouncing number 7. There are four subjects, each speaks each digit 50 times. Today, we will focus on Jackson's recordings. Take a listen to them to see how they sound like.

3. Divide Jackson's recordings into training and testing sets. Since there are 50 recordings per digit, use 45 of them for training and 5 of them as testing. Then, there will be 450 training examples in total for all 10 digits and 50 for testing.

4. Train 10 HMM models, one for each digit class. You will use 45 recordings of one digit class to train one HMM model and so on.

5. This time, I wouldn't ask you to implement the whole HMM learning algorithms as they are so complicated. But I want you to know the basic mechanism how a small-scale speech recognition system works. So, let's use a Python HMM toolbox, which you can find here: https://hmmlearn.readthedocs.io/en/latest/.

6. There are various HMM versions you can choose from, but let's use `GMMHMM`, which might be the most popular choice anyway. I briefly mentioned about it at the end of the HMM module, but it's basically a variant of HMM whose *emission* probabilities are defined by a GMM. If you think of HMM as an elaborated GMM with complicated prior (i.e., the transition probabilities), GMM-HMM can be seen as a GMM model working on top of another GMM model, necessitating nested EM-like algorithms. Anyway, you don't have to worry about it today.

7. First, for digit class C, let's convert each of the 45 utterances into an MFCC spectrogram. You will use a toolbox for this: `librosa.feature.mfcc`. As a result, you produce $\boldsymbol{X}_i^{(C)} \in \mathbb{R}^{20\times T_i^{(C)}}$, where $T_i^{(C)}$ is the number of MFCC vectors of the i-th utterance in the digit class C. Store both the MFCC matrix $\boldsymbol{X}_i^{(C)}$ and length information $T_i^{(C)}$. Repeat this for all 45 utterances.

8. In theory, you can feed the 45 utterances one by one to your HMM training algorithm, but the HMM toolbox you use has a more elegant way to deal with the situation. You will concatenate all the MFCC matrices from 45 utterances into a one big matrix and feed it just once: $\boldsymbol{X}^{(C)} = [\boldsymbol{X}_1^{(C)}, \boldsymbol{X}_2^{(C)}, \ldots, \boldsymbol{X}_{45}^{(C)}]$. The only thing is, the algorithm doesn't know the boundaries between different utterances. Once again, the toolbox provides a nice way to inform the algorithm of this, by feeding a vector of per-utterance length information, i.e., $T^{(C)} = [T_1^{(C)}, T_2^{(C)}, \ldots, T_{45}^{(C)}]$. Take a look at the manual of the `fit` function of the GMMHMM class here: https://hmmlearn.readthedocs.io/en/latest/api.html#hmmlearn.hmm.GMMHMM.

9. So, you first need to initialize a GMMHMM class and call the `fit` function to train it. When you initialize it, you can choose to change a few hyperparameters, such as the number of iterations, the tolerance threshold (for the stopping criterion), the number of hidden states

and GMM latent components (small numbers just work fine for these), etc. I adjusted them to instruct the learning algorithm run for a long enough time to converge.

10. Repeat this process for all digit classes: $C = \{0, 1, \ldots, 9\}$. Now you have 10 HMM models trained from each of the digit classes.

11. Convert your test utterances into the MFCC matrices in the same way as above. Now you have 5 uttarences per class to test out. Feed all five of them (as a big matrix along with the length information) to each of the 10 HMM instances for testing. For this, you will use the `score` function. This will give you the "log-likelihood" score. For each digit-specific test set, you have 10 numbers from 10 HMM instances. The one that gives you the largest number is the digit class the test set belongs to. Report these numbers.

12. Draw a confusion matrix, whose x-axis is the ground-truth digit class label and y-axis is the prediction. Hence, for a given row of the confusion matrix, the brightest pixel should be on the diagonal element if the classification is accurate. Submit this confusion matrix plot, too.