Problem Set 2: Acoustic Models

The University of Texas at Austin
CS 378: Intro to Speech and Audio Processing
Instructor: David Harwath

This problem set consists of 2 problems (each with multiple parts), totaling 100 points.

You will explore:

1. Gaussian models and the E-M algorithm **(25 points)**
2. Implementing a deep neural network acoustic frame classifier in PyTorch. **(75 points)**

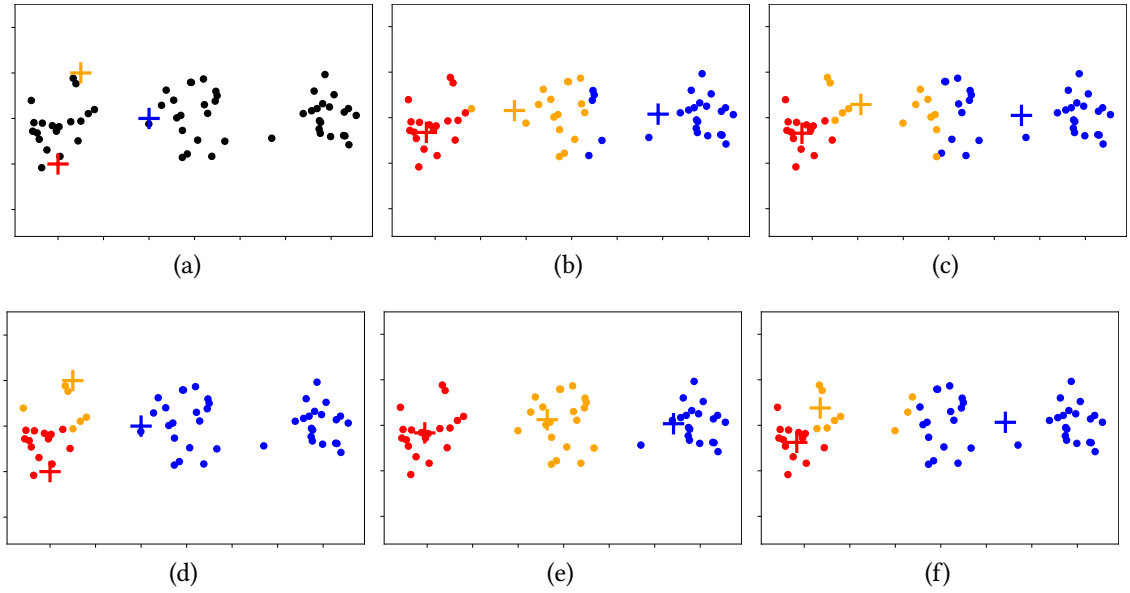
To turn in: You should submit a single .pdf writeup to Canvas, containing all of your deliverables for all parts of the problem set. For the written parts of the assignment, you must show your work. For the coding parts of the assignment, you must turn in all of the plots and code fragments that are specified as deliverables in the corresponding exercises.

Collaboration Policy: You may collaborate in groups of no larger than 3 students, and each student must turn in their individual writeup written in their own words. **You must clearly indicate all of your collaborators in the title block of your writeup.**

Late policy: Sometimes we have bad days, bad weeks, and bad semesters. In an effort to accommodate any unexpected, unfortunate personal crisis, per the course syllabus I have built 4 "slip days" into our course. You do not have to utilize this policy, but if you find yourself struggling with unexpected personal events, I encourage you to e-mail me as soon as possible to notify me that you are using our grace policy. Late assignments will otherwise not be accepted.

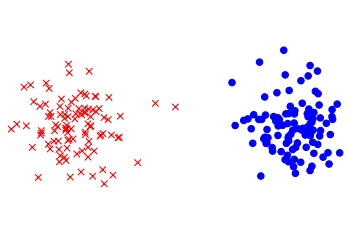
Exercise 1 (10 pts)

The following 6 plots represent intermediate states of running the K-means algorithm for $K = 3$ on a 2-dimensional dataset. Plot (a) represents the initial state, with black circles representing the datapoints, and the [red, yellow, blue] plus signs representing the initial cluster centroids. Plots (b) through (f) represent iterations 1 through 5 of the algorithm, *after points have been assigned to their nearest cluster centroids but before those centroids have been updated.* Write down the logical sequence of states that follow from (a).

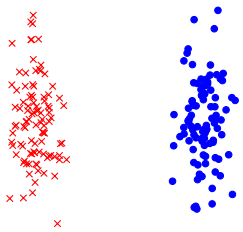


Exercise 2 (12 pts)

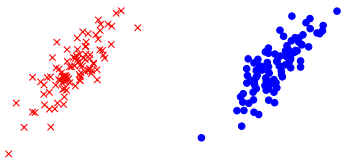
The following four plots represent four different scenarios in which we have a 2 classes, c_1 (red) and c_2 (blue). In each case, both classes have a 2-dimensional multivariate Gaussian distribution, given by $p(x|c_1) \sim \mathcal{N}(x|\mu_1, \Sigma_1)$ and $p(x|c_2) \sim \mathcal{N}(x|\mu_2, \Sigma_2)$. For each of the following modeling assumptions, list all of the scenarios that adhere to the assumption, or indicate "None" if the assumption does not describe any of the scenarios.



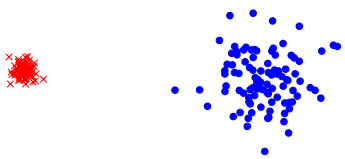
(g)



(h)



(i)

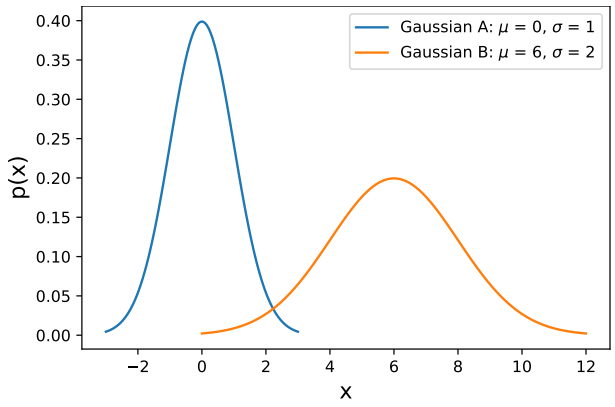


(j)

1. $\Sigma_1 = \Sigma_2$
2. For both Σ_1 and Σ_2, $j \neq k \implies \sigma_{jk} = 0$
3. $\Sigma_1 = \Sigma_2 = \sigma I$
4. $\Sigma_1 = \sigma_1 I$ and $\Sigma_2 = \sigma_2 I$ for some σ_1, σ_2

Exercise 3 (3 pts)

The figure below shows two Gaussian class densities in 1 dimension. Given the datapoint $x = -7$, is it more likely that the datapoint was generated by Gaussian A or by Gaussian B? Assume a uniform prior over the Gaussian components.



Exercise 4 (75 pts)

In this exercise, you will implement a deep neural network phoneme classifier in PyTorch. You will need to download two files from Canvas:

- `lab2_dataset.npz` : the dataset you will use to train and test your models
- `lab2.ipynb` : a Jupyter notebook containing some starter code

You will also need a Python environment with the `PyTorch` package available. Any of the UTCS lab machines should already have everything you need installed, but if you find it more convenient to use a different machine (such as a personal laptop) you should feel free to do so. I recommend using a Jupyter notebook for this exercise.

If you don't know how to use a particular feature, you can find all the documentation you should need at `https://pytorch.org/`. `PyTorch` is a very well documented library, but if you haven't used it before you might find it useful to read through this tutorial: `https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`.

At a high level, the outline of your code will look like this:

1. Load the provided dataset
2. Create the train and test data loader objects
3. Define the network architecture
4. Instantiate the model, a loss function, and optimizer
5. Train the model with stochastic gradient descent, iterating over the training dataset several times
6. Evaluate the model on the held-out test data.

We have provided some skeleton code for you to use in the `lab2.ipynb` Jupyter notebook. Your job is to fill in the model's class definition as well as the `train` function, run the training pipeline, do some analysis, and try to improve the model's performance.

4.1 Loading the Data

When you load the `lab2_dataset.npz` file, you will get a dictionary with 5 entries:

- `train_feats`: A numpy array of shape (44730, 11, 40). The first dimension indicates the number of training examples. The second dimension indicates the number of consecutive feature frames we've extracted for each example - one center frame in the middle of the phoneme, with 5 frames of context on both the left and the right for a total of 11 acoustic frames. The last dimension indicates the number of Mel frequency filter bins that each frame has, in this case 40.
- `test_feats`: A numpy array of shape (4773, 11, 40). The first dimension indicates the number of test examples, while the second and third dimensions are exactly the same as for the training features.
- `train_labels`: A numpy integer array of length 44730, where the i^{th} entry indicates the ground-truth class label for the i^{th} training example. The class labels here are only

integer indices; to obtain the string label, you can retrieve the corresponding element in the `phone_labels` array.

- `test_labels`: A numpy integer array of the same format as `train_labels`, but with 4773 examples used for testing.
- `phone_labels`: A length 48 array of strings, where the i^{th} entry is the name of the i^{th} class.

4.2 Defining your model

We have provided a skeleton class definition for your model. Following the PyTorch examples shown in class and in the official documentation, complete the class definition by declaring the layer objects you wish to use and implementing the model's forward pass. Keep in mind that the dataloader will provide 3-D tensors of size ($batch\_size$, 11, 40), where $batch\_size$ is by default set to 8, 11 represents the number of consecutive spectral frames extracted for each phoneme example, and 40 is the number of Mel filterbanks belonging to each frame. Depending on whether you choose to use convolutional, recurrent, or feedforward layers in your model, you'll need to make sure that your input features are sized and shaped properly. The PyTorch tensor `.view(), .reshape(), .transpose(), and .permute()` functions will be useful here, as will the layer documentation found at `https://pytorch.org/docs/stable/nn.html`. Remember that there are 48 different phoneme classes, so your network's output should be a 48-dimensional vector containing the class scores. Note that for numerical stability, PyTorch's `nn` library provides the `CrossEntropyLoss` criterion (PyTorch lingo for an object that represents a loss function), which combines both the softmax normalization and the cross-entropy loss into one; therefore, if you use this criterion, you don't need to include a final softmax layer in your model definition.

4.3 Implementing the training loop

Next, you should fill in the `train_model(...)` function. Recall that during the training loop for stochastic gradient descent, we iterate over the entire training dataset multiple times (5-10 epochs is a good starting point for this project). Each of these iterations of the outer loop is called a *training epoch*. During each epoch, the inner loop cycles through small subsets of the training data called *minibatches*.

One of PyTorch's most important features is the `autograd` module. This module keeps track of every operation performed on all torch `tensor` objects that are registered with it, which is accomplished by specifying the `requires_grad=True` keyword argument when the tensor is instantiated. When `.backward()` is called on a tensor, it backpropagates its gradient into the tensors that were involved in its computation history, calling `.backward()` on them which continues until there are no more gradients to be computed.

This means that gradients in PyTorch are computed numerically (as opposed to symbolically), automatically (you don't need to manually specify or compute the gradient), and dynamically (we don't need to pre-compile a network architecture or computation graph). All you have to worry about is coding the forward pass for your network and computing the value of the loss function; calling `.backward()` on the loss will take care of backpropagation for you. For more information, see `https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html`.

Once your model's gradient has been computed via backpropagation, you'll need to actually perform the parameter update. PyTorch also provides the `optim` module, which contains implementations of all of the popular optimization algorithms used in deep learning. The starter code contains a declaration of an SGD optimization object, and also registers your model's parameters with that object by passing them as an argument to the model's init function. All you need to do is call `.step()` on the optimizer after the gradient has been computed. One thing to keep in mind is that gradients in PyTorch are implemented as accumulators - they'll keep adding up gradients computed via successive calls to `.backward()` until you reset them to zero. Therefore, the very first thing you'll want to do inside your inner training loop is make a call to the optimizer's `.zero_grad()` function.

Finally, you may also find it helpful to review the example PyTorch code we went over in lecture.

4.4 Evaluating and improving the model's performance

We've provided the `test_model(...)` function for you to test your model's performance on the held-out evaluation dataset. **For full credit, your model should achieve at least 55% accuracy on this evaluation set, however an accuracy in excess of 65% should be possible . See how high of an accuracy you can achieve!** Things to try include changing the model architecture, as well as changing the training hyperparameters such as the number of epochs, optimizer, learning rate, etc.

Once your model is achieving an overall accuracy you are happy with, answer the following analysis questions:

1. Report the final accuracy achieved by your model.
2. Write some code that computes the accuracy for each different phoneme class individually. What are the 3 phoneme classes that your model predicts with the highest accuracy?
3. What about the 3 classes that have the lowest accuracy?
4. For phoneme segments that have the ground-truth label 'sh', what other phoneme class are they most commonly mis-classified as? Does this make sense? Why or why not?
5. Repeat the previous question for the 'p', 'm', 'r', and 'ae' phoneme classes.