Algorithms for Speech and Natural Language Processing - TP1

Cheng ZHANG

February 24, 2019

1 Classification of Single Voice Commands

In this part, we aim to train a simple speech command recognizer. We use the speech commands dataset [1] which contains 30 different single commands (*e.g.* up, down, go, zero, *etc.*). For each command there are several audio files and our goal is to recognize the command correspondent to an audio. The pipeline for this task involves the following steps: (i) data preprocessing; (ii) feature extraction; (iii) classifier training and parameters tuning. We will try different methods in each step and finally compare their performance.

Data preprocessing We have tried taking 500 instances in each class for training and 1000 instances for each to see if we can improve the performance by training on a bigger dataset. We also tried to randomly (with probability 0.8) add some background noise (with a decay of 0.1) to the training samples to see if we can improve the performance by data augmentation.

Feature extraction We used Mel-filterbanks function and Mel-frequency cepstrum coefficients (MFCC) function which are introduced in course for feature extraction. Concerning the parameters of MFCC, we use the following parameters:

- `nfilt = 40`: using 40 filters
- `ncep = 39, do_deltas = True, do_deltasdeltas = True`: 39 coefficients (containing 12 cepstral coefficients, 1 log energy, 13 first derivatives and 13 second derivatives) as discussed in course
- `lowerf = 20`: the lowest frequency that human can hear
- `upperf = 8000`: the upper frequency should less than the half of sampling frequency which is 16000
- `frate = 100`: stride of 10ms
- `wlen = 0.025`: window length of 25ms

and we thus get a feature space of 11817 dimension.

We also do the standard normalization to the extracted features before fitting the classifiers which is proved to be important to improve the performance.

Classifier training We trained three different classifiers: logistic regression, fully connected neural network and convolutional neural network. For logistic regression, we fine-tuned the coefficient of regularization. For fully connected neural network, we designed a neural network with just one hidden layer of 1000 neurons and we also fine-tuned the regularization. Finally, we designed a convolutional neural network and trained the model on GPU which greatly improved both the training time and the accuracy. The architecture of the convolutional neural network is as follows:

```
self.main = nn.Sequential(
    nn.Conv1d(1, 32, 90, stride=6), nn.ReLU(True), nn.BatchNorm1d(32),
```

	Logistic Regression		NN		CNN	
	T(s)	acc(%)	T(s)	acc(%)	T(s)	acc(%)
500/class	104.76	38.9	1972.19	66.5	**87.28**	**85.9**
1000/class	221.78	46.3	3223.77	73.1	**166.85**	**88.2**
1000/class + noise	172.31	38.1	6575.31	65.7	**171.34**	**85.6**

Table 1: Performance results of different classifiers

```
    nn.Conv1d(32, 64, 32, stride=6), nn.ReLU(True), nn.BatchNorm1d(64),
    nn.Conv1d(64, 128, 11, stride=3), nn.ReLU(True), nn.BatchNorm1d(128),
    nn.Conv1d(128, 256, 5, stride=2), nn.ReLU(True), nn.BatchNorm1d(256),
    nn.Conv1d(256, 512, 3, stride=1), nn.ReLU(True), nn.BatchNorm1d(512),
    nn.AvgPool1d(50)
)
self.fc = nn.Linear(512, 30)
```

The training time (T) and the accuracy (acc) of the best tuned model for each classifier on validation set are shown in Table 1.

We can see that the convolutional network produces the best results, including both the training time and the prediction accuracy in all three configurations. For all three classifiers, taking more training samples improves the performance however adding the background noise does not really improve the performance. This may because the probability of adding noise is too large or we do not decay enough the amplitude of background noise. These are the hyper-parameters we can still fine-tune. So we finally get the best CNN model by taking 1000 samples per class, which requires a training time of 166.85s and gives an accuracy of 88.2% on the validation set.

We then apply the best CNN model on the test set and get an accuracy of 83.0%. We also check explicitly the accuracy of each class on the test set. And we found that the accuracy of class "go" and "no" is extremely low (around 61% and 62%). This may because these two words are quite similar for the pronunciation. What's more, those words of one syllable (*e.g.* one, on) are also a bit difficult to recognize.

2 Classification of Segmented Voice Commands

In this part, we aim to predict a sequence of words given a sequence of audio commands. We denote the input sequence of speech features by $X_{1,\ldots,M}$ and the output sequence of words by $W_{1,\ldots,T}$ and we want to predict the most likely sequence of words which maximizes the probability of this sequence of words given the input sequence of speech features, *i.e.*

$$\text{argmax}_W P(W|X) \propto P(X|W)P(W)$$

in which $P(X|W)$ is the acoustic model and $P(W)$ is the language model.

The performance is evaluated by Word Error Rate (WER) which is defined as follows:

$$\text{WER} = 100\ \frac{S+D+I}{N}\%$$

where S is the number of substitutions, D is the number of deletions, I is the number of insertions and N is the total number of words in the ground truth sentence.

Question 2.1 Is it possible that WER < 0 and WER > 100?

Answer According to the definition of WER, we can easily conclude that WER ≥ 0 as S, D, I and N are all non-negative. We achieve the equality when we make an exact prediction

which means $S = D = I = 0$. And WER can be arbitrarily large, because the prediction can insert an arbitrary amount of words. So it is possible that WER > 100.

For simplification, we take the acoustic model from the discriminator we trained in the first part. As the discriminator has been trained with balanced dataset $P(W_i) =$ constant, we can model the acoustic model as follows:

$$P(X_i|W_i) \propto P_{discriminator}(W_i|X_i) \tag{1}$$

Question 2.2 Can you point which line in the code above approximated the prior probability of each word W_i be equal?

Answer The line of code corresponding to Eq.(1) is

```
posterior = model_predict_proba_function(features_input)
```

Question 2.3 Can you detail the computations of the WER for the example: true sentence is `go marvin one right stop` and predicted sentence is respectively `happy five one two two` (the default result) and `go marvin one off stop` (the CNN result)?

Answer We can see that for the default case, the true sentence can be obtained by substituting the "happy", "five", "two" and "two" in the predicted sentence by "go", "marvin", "right" and "stop". Thus we have $S = 4, D = 0, I = 0$ and $N = 5$ and WER $= 80\%$. For the CNN case, the true sentence can be obtained by simply substituting the "off" in the predicted sentence by "right", so we have $S = 1, D = 0, I = 0$ and $N = 5$, so WER $= 20\%$.

Question 2.4 Write the Bigram approximation formula of the language model.

Answer From the chain rule we get the formula for the language model as follows:

$$\begin{aligned} P(W_{1,\dots,T}) &= P(W_1)P(W_2|W_1)P(W_3|W_{1,2})\dots P(W_T|W_{1,\dots,T-1}) \\ &= P(W_1)\prod_{k=1}^{T-1} P(W_{k+1}|W_{1,\dots,k}) \end{aligned} \tag{2}$$

And in the Bigram approximation, we consider only a window of 2 words and assume that $P(W_{k+1}|W_{1,\dots,k}) = P(W_{k+1}|W_k)$, thus the formula becomes

$$P(W_{1,\dots,T}) = P(W_1)\prod_{k=1}^{T-1} P(W_{k+1}|W_k)$$

Question 2.5 Explain briefly your implementation choices.

Answer We choose to apply Bigram model for mainly three reasons. First, it is simpler than other N-gram models for larger N and our dataset is simple enough which only contains 30 words. Second, the generated sentences are not too long, mainly containing less than 10 words, thus we consider that Bigram model will be enough. Finally, compared with Uni-gram model, Bigram model can retain relationship between words.

Question 2.6 What are the advantages and drawbacks to increase the N?

Answer By increasing N, we can model more complicated relationship between N words which will also approximate more accurately to Eq.(2). However, the dimension of the transition matrix increases exponentially with N so a large N will cause great computational complexity. What's more, a larger N will require longer sentences in the dataset.

Question 2.7 What is the complexity of the Beam-Search algorithm?

Answer Suppose that the input posterior probability matrix is of size $l \times N$ where l is the number of words in the sequence and N is the number of single command classes (*i.e.* 30 in our case). We also suppose that we store k candidates in the memory (when $k = 1$ it is equivalent to greedy decoder and we take $k = 5$ or 10 in general). The complexity of Beam-Search algorithm is thus $O(l \times N \times k)$.

	Unigram	Bigram		
		Greedy	Beam-Search	Viterbi
WER (sub training set)(%)	14.14	15.36	**7.35**	8.35
WER (test set)(%)	20.23	18.54	10.67	**10.28**
T(s)	33.54	34.05	33.59	33.89

Table 2: Performance results of different decoders and different language models

Question 2.8 In Viterbi algorithm, what is the relationship between the probability to be in state j at step k, and the probabilities to be in state j' at step $k-1$? What is the final complexity of the Viterbi algorithm?

Answer We denote the probability to be in in state j at step k as $p(j,k)$ and the probabilities to be in state j' at step $k-1$ as $p(j',k-1)$, then in Viterbi algorithm, we have

$$p(j,k) = \max_{j'} p(j',k-1) p_{transition}(j|j') p_{posterior}(j|k)$$

Using the same notations as in the Question 2.7, we conclude that the complexity of Viterbi algorithm is $O(l \times N^2)$.

Question 2.9 Can you spot systematic errors due to language model you derived from the training sequences? Provide us some examples of these errors.

Answer There may be some words or some pairs of words (as we consider the Bigram model) which do not exist in the training sequences however they appear in the test sequences.

Question 2.10 Can you implement some backoff strategies to face rare seen words (or sequence of words) and out of vocabulary words? Does it improve your Word Error Rate?

Answer We can implement some smoothing methods to avoid this issue. We implemented the additive smoothing method which adds a constant to all words or all pairs of words and then calculates the probability. This technique avoids the errors for unseen words and slightly improves the Word Error Rate.

As mentioned previously, we choose the best CNN model for the discriminator and the Bigram model for the language model and implement greedy decoder, Beam-Search decoder and Viterbi decoder. We also apply add-1 smoothing technique to avoid unseen vocabulary issue and we get the results shown in Table 2. Notice that if we use Unigram model, different decoders should give the same results as the default greedy decoder since the transition matrix for Unigram model is constant everywhere.

We can see that the best result on the test set is given by the Viterbi decoder and the Bigram model. There is no great different between Beam-Search and Viterbi which may because our sentences are too simple. We believe that for a larger and more complex dataset, Viterbi performs better than Beam-Search as the former finds the exact solution. One thing to notice is that for the greedy decoder, even though the bigram model gives worse result than unigram model on training set, the performance on test set is inverse. So we consider that bigram model in fact improves the performance. There is no great difference of time consumption.

Question 2.11 How would you optimize jointly an acoustic model and language model?

Answer We can take the sequence of speech feature as input and the sequence of words as output and train an end-to-end RNN network.

References

[1] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, abs/1804.03209, 2018.