Speech command recognition with deep learning
SOUNDS 4th Seasonal School

1 Preparations

1.1 Installations

- Python: please install the 3.10 version in https://www.python.org/downloads/release/python-3100/
- Anaconda: https://docs.anaconda.com/free/anaconda/install/index.html
- ffmpeg:
  1. For Windows: https://support.audacityteam.org/basics/installing-ffmpeg,
  2. For MacOS: same url as above and choose MacOS (you may need to first install homebrew).
- Visual Studio Code: https://code.visualstudio.com/, then please install the "Jupyter" and "Python" plugin as shown in Figure 1.
- Audacity [Optional]

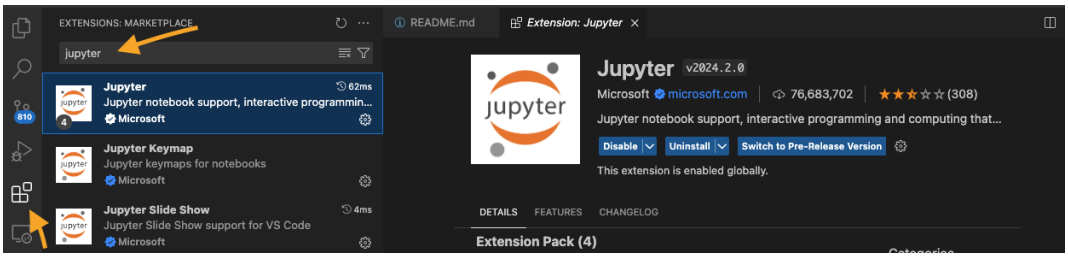

Figure 1: Install plugin.

1.2 Clone the repository

You can clone the code using the following git command:

```
git clone git@github.com:invincibleo/SOUNDS_4th_school_practical.git
cd SOUNDS_4th_school_practical
```

Or go to the repository's website: https://github.com/invincibleo/SOUNDS_4th_school_practical and click the green button "Code" then "Download ZIP" as shown in Figure 2.

1.3 Creation of virtual environment

Please run the following commands to create a conda virtual environment and install the necessary packages. (Please copy the following commands in the README in the code folder)

```
conda create --name sounds4th python==3.10 ipykernel notebook matplotlib tqdm scipy
    tensorboard
conda activate sounds4th
conda install 'ffmpeg<7' python-sounddevice scikit-learn seaborn pandas -c conda-forge
conda install pytorch::pytorch torchaudio ignite -c pytorch
```

2 Descriptions

Speech command recognition is one of the basic speech recognition tasks. Its applications spin around many aspects of daily life, such as triggering a speech assistant with a simple "Hey Siri" or controlling a smart home system through voice commands.

In this hands-on session, we will delve into the task of speech command classification using deep learning techniques. The dataset used in this exercise is the SpeechCommand dataset [1], featuring a collection of 35 distinct single-word commands spoken by various individuals.

The dataset is divided into three distinct partitions: the training set, validation set and testing set. The training set serves for model training, while the validation set aids in hyperparameters tuning. And the testing set allows us to assess the model's final performance on unseen data that was not part of the training process.

Throughout the exercises during this session, we will concentrate on 22 specific commands, representing typical words used for controlling various applications such as robots. Additionally, we will include an extra category for unknown words, bringing the total number of classes for classification to 23.

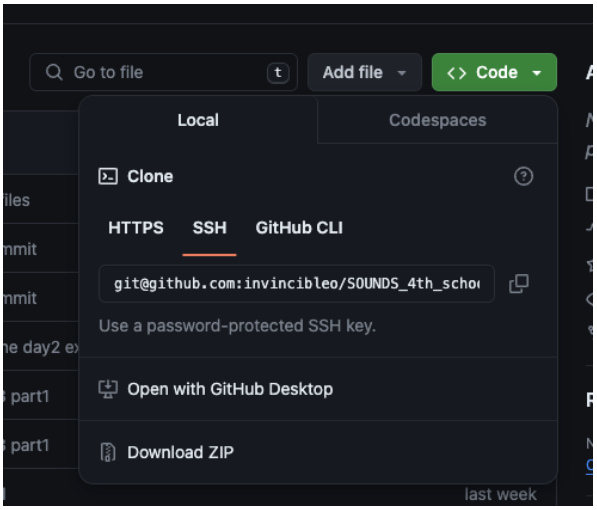

Figure 2: Clone the code.

3 Day 1: First Model

On the first day of this practical session, we will guide you through the essential steps of feature extraction, model design, model training, and model deployment. By the end of this session, we will have our initial speech command recognition model capable of identifying specific speech commands. We're excited to test the model using our own voices!

To get started, please open *speech_command_recognition_day1.ipynb* in Visual Studio Code (VSC). Choose the appropriate kernel (matching the name of the conda environment, e.g., sounds4th) in the top-right corner as illustrated in Figure 3. You can then execute each code block by clicking the play button located on the left side of the code block, as demonstrated in Figure 4.

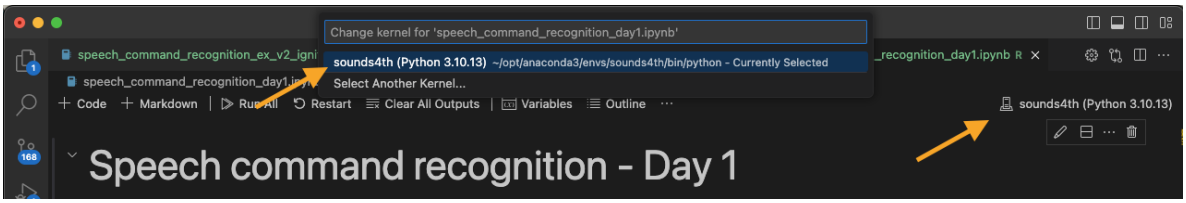

Figure 3: Choose the correct kernel to run the code.

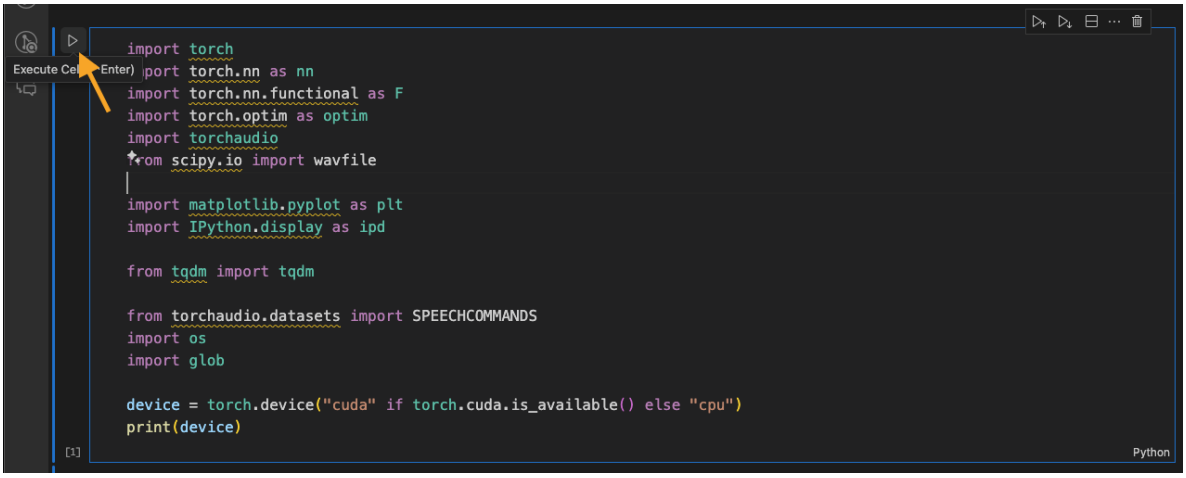

Figure 4: Run code blocks.

3.1 Feature Extraction

While an Artificial Neural Network (ANN) with more than 3 layers and non-linear activation functions has been shown to be a universal function estimator[2], a model performing well on raw input data can often become computationally demanding. Instead, employing hand-crafted features specifically tailored to the task can significantly reduce model complexity while maintaining strong generalization capabilities, especially for datasets of moderate to small sizes.

The design of features aims to retain essential information within the data while filtering out less relevant or irrelevant details. However, some discarded information during feature extraction might still hold value in improving model performance. This balance between model computation complexity and the level of feature processing requires careful consideration in practice.

In this exercise, we will utilize Mel-Frequency Cepstral Coefficients (MFCC) features, which have been widely employed in speech related tasks. Prior to feature extraction, all speech recordings are standardized to a duration of 1 s by padding or trimming. Subsequently, we employ an Fast Fourier Transform (FFT) of size $0.03 \times 16000 = 480$ with a 50% overlap, Hamming window and 64 mel-scale filter banks to extract 16 MFCCs per time frame. This results in 65×16 features for a 1-second input sampled at 16 kHz.

The implementations are detailed under the MFCC feature extraction section. The function *collate_fn_extract_feature* is invoked by the data loaders, wherein the padding and MFCC calculations from the speech waveforms are executed. The resulting MFCC features are stored on disk to prevent redundant computation in each epoch.

3.2 Model Design

ANNs offer flexible architectures with various fundamental "building blocks", including dense layers, Convolutional Neural Network (CNN) layers, pooling layers, normalization layers, and Recurrent Neural Network (RNN) layers. In recent years, attention layers have gained popularity and serve as essential components in transformer-based models [3]. Each of these building blocks carries unique functionalities and properties. For instance, CNN layers resemble convolution operations in digital signal processing and can introduce non-linearity when combined with non-linear activation functions. On the other hand, RNN layers incorporate feedback from past outputs, making them ideal for capturing temporal dependencies in sequential data.

While machine learning theories and mathematics can offer insights into model design, the practical process often involves a significant degree of engineering. It entails intelligently combining these building blocks and iteratively testing the model. Despite the existence of automatic model architecture search algorithms, effective model design in practice often requires a wealth of experience.

Given that our task involves classification, a "softmax" activation function is typically applied to the outputs of the final layer's neurons. This choice is deliberate, as the "softmax" function ensures that

the outputs of the neurons sum to one, thereby interpreting each output as the posterior probability of a class given the observation.

> Question: What shape should the model output have?

Feel free to design your model from scratch or build upon the provided model in the "Model Design" section. It is important to thoroughly review the documentation of the layers you plan to use, paying particular attention to the input-output dimensions.

3.3 Training and Validation

For our classification task, we will utilize the cross-entropy loss function. Essentially, this loss aims to minimize the distance between the model's prediction distribution and the distribution of the training data assuming it is i.i.d sampled from an underlying distribution. For a more detailed explanation, you can refer to this post: What is the difference between Cross-entropy and KL divergence?

It's important to note that in our implementation, we employ PyTorch's Negative Log Likelihood (NLL) loss in combination with applying the model outputs to a log softmax function. This setup is equivalent to using the cross-entropy loss with softmax outputs, as outlined in the documentation: CROSSENTROPYLOSS.

During model training, we employ the Adam optimizer with an initial learning rate of 0.01, while setting the weight decay factor to 0.0001. Additionally, we incorporate a learning rate scheduler that decreases the learning rate by a factor of 10 every 20 epochs.

> Question: What is the weight decay factor? How it affects the training?

Next, we define the training loop, which involves loading batches of data from the *train_loader*, executing forward propagation, computing the loss, performing backward propagation, and calculating metrics. We track the training set's accuracy for each epoch.

Similarly, we establish the evaluation loop, which runs the model in inference mode on the validation set. We record the evaluation accuracy for each epoch as well.

Finally, we initiate the training by specifying a maximum number of epochs (e.g., 40). It is advisable to save the model checkpoint after each epoch to facilitate early stopping if needed, eliminating the need to rerun the entire training process.

3.4 Check list

- ☐ Training data loader
- ☐ Evaluation data loader
- ☐ Model
- ☐ Loss
- ☐ Optimizer
- ☐ [Optional]Scheduler
- ☐ Training loop
- ☐ Validation loop
- ☐ Main loop

4 Day 2: Model Evaluation

4.1 Monitoring Training Progress

Deep Neural Networks (DNNs) employ the back-propagation algorithm alongside stochastic gradient descent algorithms to tune their model parameters. Given the high computational demands of modern

DNNs, they often require a substantial amount of time to converge. Thus, it becomes crucial in practice to monitor the training process for debugging and parameter adjustment purposes.

Tensorboard offers an excellent visualization interface and serves as a convenient solution for monitoring model training progress. It provides visualizations such as evolving loss, evaluation metrics, optimizer parameters (e.g., learning rate), distribution of model weights, intermediate embeddings, and more.

In this exercise, we will utilize Tensorboard in conjunction with the PyTorch Ignite package for automated training and monitoring. A comprehensive tutorial on how to transform pure PyTorch code to Ignite code can be found at https://pytorch-ignite.ai/how-to-guides/02-convert-pytorch-to-ignite/. We have already transformed the Day 1 code into Ignite code and implemented it in *speech_command_recognition_day2.ipynb*

During this exercise, we will log various metrics and visualize them using Tensorboard. To launch Tensorboard, simply use the following command:

```
tensorboard --logdir=[Root of your logdirs]
```

Then, navigate to the URL (typically http://localhost:6006/) displayed in the command line window using your web browser. In the subsequent section, we will define several metrics to monitor throughout the training process.

4.2 Evaluation Metrics

4.2.1 Precision, Recall, and F1 Score

While prediction accuracy is a straightforward evaluation metric for classification tasks, it can be misleading when dealing with imbalanced data. Consider a binary classification scenario where 95% of examples belong to the negative class and only 5% to the positive class. A no-skill classifier that always predicts the negative class will achieve 95% accuracy on this dataset, even though it offers no meaningful prediction.

To address this issue, precision and recall metrics are commonly used for evaluating classifiers on imbalanced datasets. In the case of the aforementioned no-skill classifier, both precision and recall will be zero. A well-performing classifier will exhibit high values for both precision and recall. The F1 score, which is the harmonic mean of precision and recall, provides a single metric to assess classifier performance on imbalanced datasets. (For further reading: Tour of Evaluation Metrics for Imbalanced Classification)

4.2.2 Confusion Matrix

The confusion matrix offers a direct visualization for assessing per class classification performance. Typically, the rows of the confusion matrix represent true class labels, while the columns represent predicted class labels. As a result, the diagonal cells of this matrix represent the true positives for each class.

4.2.3 Logging Metrics with Tensorboard Logger

Ignite provides interfaces for integrating with Tensorboard. In the "Attach evaluation metrics" section of the example code *speech_command_recognition_day2.ipynb*, we demonstrate how to log metrics such as loss value, accuracy, precision, recall, and the confusion matrix to Tensorboard. You can execute these examples and monitor the training progress in Tensorboard.

4.3 Generalization Capability

DNNs are powerful and flexible non-linear models. However, it is essential to ensure that these models do not become over-trained or over-fit to the training data. In essence, our goal is to find a model with good generalization capabilities rather than one that merely excels on the training data.

The assessment of a model's generalization capability is typically performed on a separate dataset that was not utilized during training, known as the validation set. A well-designed validation set should accurately represent the real-world characteristics of the data to which the model will be applied. In practice, the validation set can be tailored to evaluate specific aspects of the model's performance. For instance, in tasks such as speech emotion recognition, a desirable model should perform independently

of speaker identification. Consequently, the validation set might exclusively comprise speech recordings from speakers not present in the training set.

In the speech command dataset used for our exercises, a validation set is provided. This set possesses characteristics similar to the training set, including similar sound artifacts, speakers, and volume levels. Based on this setup, we can draw the following conclusions:

1. If the model performs well on the training set and equally well on the validation set, it indicates good generalization capability.
2. Conversely, if the model achieves high performance on the training set but significantly underperforms on the validation set, it might be an indication of over-fitting.

Please review the Tensorboard logs and consider the following questions:

> Question 1: Do you observe signs of overfitting?
> Question 2: What insights can you derive from examining the confusion matrices?

5 Day 3: Model Improvement

5.1 Data Augmentation

Enhancing the generalization capabilities of data-driven models such as DNNs often involves acquiring more annotated data, which is considered the most effective solution. However, in practice, this can be challenging due to factors such as the nature of the data source, data acquisition costs, and annotation expenses. An accessible approach to address this challenge is through data augmentation, which involves making slight transformations to existing training data without altering its actual annotations.

In *speech_command_recognition_day3_part1.ipynb*, we have included several examples of speech data augmentation techniques:

- Adding existing noise: involves mixing the speech waveform with recorded background noises. The target Signal to Noise Ratio (SNR) is set within the range of [-5, 10] dB.
- Time shifting: entails shifting the speech waveform to the left (i.e., reversing time direction) or to the right (i.e., forward time direction). The maximum shift is set to be 0.3 s.
- Speed perturbation: includes speeding up or slowing down the speech recording. The speed-up or slow-down factor is set to 10%.
- Volume adjustment: involves decreasing or increasing the volume of the speech recording. The volume gain is set within the range of [-10, 10] dB.
- Adding white noise: similar to mixing existing noise, but the noise type is constrained to white noise. The target SNR is within the range of [-5, 10] dB.
- Time masking: randomly masks out a segment of the recording in the time domain. The maximum duration to mask out is set to be 0.05 s.
- Frequency masking: randomly masks out a segment of frequency bins in the frequency domain. The maximum bins to mask out is set to be 5. Note that we apply the frequency masking after the 64 mel-scale filter banks.
- Adding reverberation: artificially introduces reverberation to the speech recording. The room impulse responses (Room Impulse Responses (RIRs)) are pre-recorded and then convolved with the speech recording. For detailed information about the RIR dataset, please refer to Aachen Impulse Response Database.

We invite you to explore the implementation and listen to the resulting outputs.

> Question: What are your thoughts on the effectiveness and feasibility of the provided data augmentation techniques?

5.2 Addressing Class Imbalance

One potential issue that may arise is evident in the confusion matrix, where numerous misclassifications are predicted to the "unknown" class. This could be attributed to the fact that our training set contains a significantly larger number of examples in the "unknown" class compared to the other classes. Ideally, a training set should reflect examples drawn from the underlying class multinomial distribution. When one class is disproportionately represented in the training data, it might suggest a higher probability of this class in the underlying distribution, thereby influencing data-driven models. Hence, it is crucial to align the training data distribution with the real data distribution. However, achieving this alignment can be challenging in practice and may require domain-specific knowledge.

In this exercise, we assume equal probabilities for all classes to be drawn. To achieve this balance, a straightforward solution involves randomly oversampling the minority class examples to ensure all classes have an equal number of training data.

In *speech_command_recognition_day3_part2.ipynb*, we have provided a random sampler class. This class will calculate weights for each training example based on their label and oversample the minority class.

5.3 Further Enhancing Model Performance

The final topic in this practical session involves strategies to further enhance the model's performance. This area offers considerable flexibility and exploration. While acquiring more annotated data is always beneficial, additional avenues for improvement include increasing the model's capacity, incorporating data structures into the model, tuning hyperparameters, using efficient data augmentation techniques, constructing enhanced loss functions, utilizing pre-trained models, and more.

However, in many practical applications, constraints such as memory and computational resources limit the feasibility of increasing the model's capacity (e.g., model depth and width). Thus, there may be trade-offs to consider between model complexity and performance.

> Question: Please attempt to design a larger model and potentially integrate Long Short-term Memory (LSTM) layers to capture temporal dependencies in the data.

Please use *speech_command_recognition_day3_part2.ipynb* to try out utilizing data augmentation and data over-sampler during the training. An example of a more complex model with LSTM layer is at the code appendix.

> Question 1: Do you observe a smaller gap between the training and evaluation metrics?
> Question 2: What are your thoughts why the training metrics are getting lower compared to the training without data augmentation? How would you improve this?

References

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

[2] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.