Japanese Vowel speaker Classification: A Traditional Machine Learning and a Deep Learning Approach

Carmen Jica s3935841
Mirko Borszukovszki s4745000
Levente Kis s4765389
Haoran Peng p312334

***Abstract*—Speaker recognition is a staple challenge in the field study of machine learning. Particularly, we tackle this endeavor in the context of Japanese vowel classification facilitated by sequences of captured speech. We employ two-dimensional singular value decomposition (2dSVD) for feature extraction, followed by a k-nearest model for classification. A grid search procedure is integrated in order to determine a best performing model.**

***Index Terms*—Multivariate timeseries classification, kNN, CNN**

Speaker recognition is a foundational challenge in machine learning and artificial intelligence, with applications ranging from security systems to personalized user interfaces [4]. It enables systems to identify or verify individuals based on their vocal characteristics, playing a pivotal role in biometric authentication, voice-controlled applications, and forensic analysis. However, achieving accurate and efficient speaker recognition remains a complex task due to variations in speech patterns caused by linguistic, environmental, and individual factors.

A critical component of speaker recognition is vowel classification. Vowels are central to human speech and exhibit distinct spectral features, making them essential for capturing the unique vocal characteristics of individuals [2]. By accurately classifying vowels, speaker recognition systems can improve their ability to differentiate between individuals, even in challenging scenarios such as noisy environments or overlapping speech.

In this study, we aim to address these challenges by exploring an effective framework for Japanese vowel classification. Japanese vowels, characterized by their well-defined and distinct phonetic properties, provide a valuable basis for analyzing speaker-specific features [5]. Our objective is to achieve robust speaker recognition through efficient feature extraction and classification techniques tailored to Japanese speech data.

To this end, we leverage two-dimensional singular value decomposition (2dSVD) for feature extraction, a method that preserves the intrinsic matrix structure of the speech data while reducing computational complexity [6]. Unlike traditional methods that require flattening the data into vectors, 2dSVD maintains spatial relationships and minimizes information loss. Following feature extraction, we employ a k-nearest neighbors (kNN) classifier due to its simplicity and adaptability in handling non-linear decision boundaries [1]. Moreover, a systematic grid search is conducted to fine-tune the model parameters, ensuring an optimal balance between classification accuracy and computational efficiency.

This project contributes to the field by introducing a streamlined pipeline that synergizes advanced feature extraction and classification techniques. The integration of 2dSVD and kNN not only simplifies the computational process but also enhances the accuracy of speaker recognition systems. Our approach, validated on the Japanese Vowels dataset, provides insights that can be extended to other languages and speech recognition tasks, thereby fostering future advancements in the domain.

I. Data

The experiments are carried out on a real-world datasets Japanese Vowels dataset. It contains 640 time series of 12 LPC cepstrum coefficients (i.e. MTS sample with 12 variables) taken from nine male speakers. The length of MTS sample is between 7 and 29. The task is to distinguish nine male speakers by their utterances of two Japanese Vowels/ae/. Speakers 1–9 have the corresponding number of samples: 61, 65, 118, 74, 59, 54, 70, 80, 59. Thus, speakers 1 has 61 samples (61 utterances of/ae/), speaker 2 has 65 samples (65 utterances of/ae/), and so on. Samples per male can be regarded as one class. Random 30 samples per speaker are taken with labels to form the training samples, the remaining samples per speaker are considered to be the testing samples. As depicted in Figure 1, the dataset visualization illustrates the variability in the time series lengths and the distribution of 12 LPC cepstrum coefficients across four selected speakers. This figure underscores the diversity in sample characteristics and the uneven distribution of utterances per speaker, which further motivates the need for robust feature extraction and classification techniques.

II. Methodology

Below, we explain the preprocessing steps applied to the data before classifying it with a K-nearest neighbors (KNN)

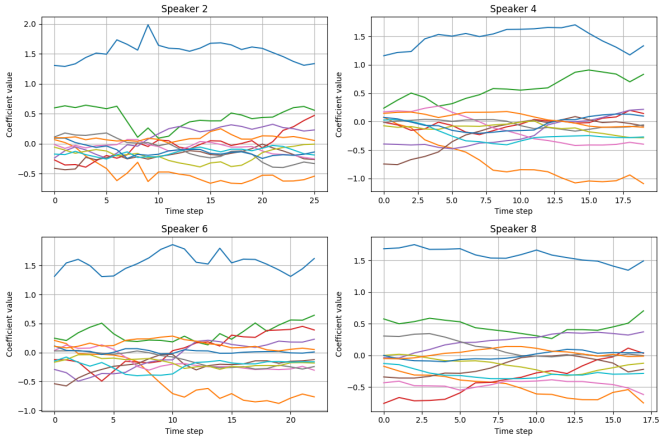

Fig. 1: Data representation of various speakers across the 12 channels

Speaker	1	2	3	4	5	6	7	8	9
No. of samples	61	65	118	74	59	54	70	80	50

TABLE I: Number of recordings for each speaker

classifier or a Convolutional Neural Network (CNN).

A. Preprocessing

We follow the approach described in [5]. The data matrices are transformed using 2DSVD to extract lower-dimensional meaningful features from the data.

For every data point, there are recordings from 12 channels but these recordings can have different lengths for each person. In the test dataset, the longest recording is 26 timesteps but the shortest is only 7. To unify recording lengths, we duplicate some of the recordings in short sequences to reach a length of 26. For example, if we have a sequence length of 15: [1, 2, ..., 15] and we want to extend it to have a length of 20, we can duplicate some of the recordings (in roughly equal intervals) so the sequence becomes: [1, 1, 2, 3, 4, 4, 5, 6, 7, 8, 8, 9, 10, 11, 12, 12, 13, 14, 15, 15]. If in the test set, we encounter a sequence longer than the longest sequence in the test set, we truncate it. This approach worked better for [5] than truncating all the sequences to 7 (the shortest in the training set).

Once we have the new training data $\{D_i\}_{i=1}^{I}$ where each datapoint D_i is a $m \times n$ matrix where m is the number of timesteps (26), n ís the number of channels (12) and I is the number of datapoints, we can calculate the row-row and column-column covariance matrices $\mathbf{R}$ and $\mathbf{C}$. These are calculated in the following way:

$$\bar{D} = \frac{1}{I}\sum_{i=1}^{I} D_i$$

$$\mathbf{R} = \frac{1}{I}\sum_{i=1}^{I}(D_i - \bar{D})(D_i - \bar{D})^T$$

$$\mathbf{C} = \frac{1}{I}\sum_{i=1}^{I}(D_i - \bar{D})^T(D_i - \bar{D})$$

After this, we need to select the first r principal eigenvectors of $\mathbf{R}$ and the first c principal eigenvectors of $\mathbf{C}$. The principal eigenvectors of a matrix are the eigenvectors corresponding to its largest eigenvalues in magnitude. They represent the directions of maximum variance of the data. For a square matrix $\mathbf{A}$, an eigenvector $\mathbf{v}$ satisfies the equation:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v},$$

where λ is the eigenvalue associated with $\mathbf{v}$. The principal eigenvector is the eigenvector corresponding to the largest eigenvalue, λ_{max}.

Now we van define the matrices $\mathbf{U}_r$ containing the first r principal eigenvectors of $\mathbf{R}$ and $\mathbf{V}_c$ containing the first c principal eigenvectors of $\mathbf{C}$. Since $\mathbf{R} \in \mathbb{R}^{m\times m}$ and $\mathbf{C} \in \mathbb{R}^{n\times n}$, $\mathbf{U}_r \in \mathbb{R}^{m\times r}$ and $\mathbf{V}_c \in \mathbb{R}^{n\times c}$.

Once we have $\mathbf{U}_r$ and $\mathbf{V}_c$, for every training data point D_i we can get a $r \times c$ feature matrix M_i by:

$$M_i = \mathbf{U}_r D_i \mathbf{V}_c,$$

and we can transform every data point in the test set the same way. It should be noted that the parameters r and c are hyperparameters of the preprocessing step. We find values for these by cross-validation. Another important point is that since we have matrices as data points instead of vectors, calculating the distance between matrices in a KNN classification problem is not trivial. The distance function proposed by [5] is described in the next section.

1) Baseline: For the baseline, we implemented a simple preprocessing step where we extracted the mean, standard deviation and the range (maximum-minimum) from each recording from the 12 channels. This means that we end up with 36-dimensional feature vectors for every data point. During the baseline KNN training, we only tuned the value of k with cross-validation.

B. kNN classifier

The k-Nearest Neighbors (KNN) classifier is a method used for classification tasks. The algorithm classifies inputs based on their distances to their k-nearest neighbors in the feature space using a distance function.

Given a feature matrix M_i, the KNN algorithm classifies M_i as follows:

1) Compute the distances between M_i and all the other training data points in the dataset
2) Identify the set of k nearest neighbors of M_i with the smallest distances
3) Determine the class label $\hat{y}$ for M_i using majority voting

The distance function proposed by [5] for two matrices $\mathbf{M}_1$ and $\mathbf{M}_2$ is:

$$d(\mathbf{M}_1, \mathbf{M}_2) = \sum_{k=1}^{c} \|Y_k^1 - Y_k^2\|,$$

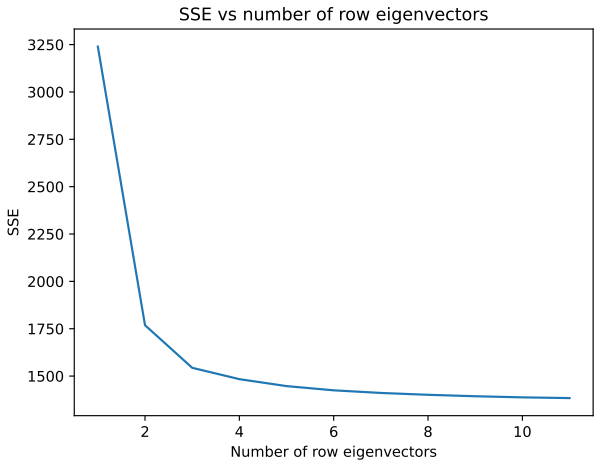

Fig. 2: SSE vs the number of row eigenvectors. The elbow point of the graph is at 3

where $[Y_1^i, Y_2^i, \cdots, Y_c^i]$ are the columns of a feature matrix M_i and $\|x\|$ is the L_2 norm of x.

C. Hyperparamter search

For the kNN classifier, we need to decide on k, the number of closest neighbors that the algorithm should consider and r, c, the parameters of the preprocessing step. We follow the approach of [5] to find reasonable starting values for r and c and later carry out cross-validation around these values and different values of k.

First, c is chosen so that it explains 98% of the column-column variations in the dataset. We need to find the smallest k such that:

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{n} \lambda_i}$$

is at least 0.98. Where λ_i are the eigenvectors of the column-column covariance matrix and n is the total number of eigenvectors.

Once we have found c, in our case $c = 9$ we need to find r. [5] defines the reconstruction error SSE, between a datapoint D and its reconstruction $\hat{D} = \mathbf{U}_r M_i \mathbf{V}_c^T$ where the covariance matrices have r, and c eigenvectors as:

$$SSE(r) = \sum_{D \in D_{\text{TRAIN}}} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(D(i,j) - \hat{D}(i,j) \right)^2 .$$

We can calculate this error for all $r = 1, 2 \ldots m$. This error will always decrease as we increase r but we can find a spot where it stops decreasing rapidly which means that we capture a reasonable amount of information from the training data. The "elbow point" of this graph is found at $r = 3$ as illustrated in Figure 2.

As we have found reasonable starting values for c, and r, we can design a cross-validation scheme to find even better values for them and a value for k. We base our hyperparameter grid search around these values and choose the ones that have the highest weighted F1 score on the validation set in 5-fold cross-validation on the training data.

The weighted F1 score is calculated as the weighted average of the F1 scores for each class, where the weight for each class is determined by the number of true instances in that class. The formula is:

$$F1_{\text{weighted}} = \frac{\sum_{i=1}^{C} w_i \cdot F1_i}{\sum_{i=1}^{C} w_i}$$

where C is the number of classes, w_i is the number of instances of class i and $F1_i$ is the F1 score for class i. The F1 score for a class is computed as:

$$F1_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

where:

- $\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}$,
- $\text{Recall}_i = \frac{TP_i}{TP_i + FN_i}$,
- TP_i, FP_i, and FN_i are the true positives, false positives, and false negatives for class i, respectively.

The tested hyperparameters and their best value are summarized in Table II. The best average performance on the validation set was with $k = 3$, $r = 5$, $c = 10$ with a weighted F1 score of 0.9663.

Parameter	Values tested	Best value
k	$[1, 3, 5, 7]$	3
r	$[1, 2, 3, 4, 5]$	5
c	$[7, 8, 9, 10, 11]$	10

TABLE II: Hyperparameters tested through gridsearch

D. Convolutional Neural Network

In the following section we will dive into a custom implementation of a convolutional neural network tasked with classification of the 9 speakers of the dataset.

We begin by describing the architecture of the employed CNN in section II-D1, follow by traversing the typical data flow of the model in section II-D2, then dive into the design decisions involved into the model's development in section II-D3 and finish by addressing measures of protection against risks of overfitting in section II-D5.

1) Architecture: We will now describe the architecture utilized for creating the CNN. An illustration of the said architecture can be observed in figure 3.

The network is initiated by a convolution layer that transforms an input example $x \in \mathbb{R}^{12 \times 30}$. The layer has a number of 12 input channels and 32 output channels, a kernel size of 3 and a padding of 1. The used activation function is ReLu.

Assume $W_k^{(1)}$ be the filter for the k^{th} output channel of the first convolution. It also presents a bias term $b_k^{(1)} \in \mathbb{R}, k = 1, 2, .., 32$. Convolution is applied for 30 time steps, value to which every input sequence was padded to initially. Therefore,

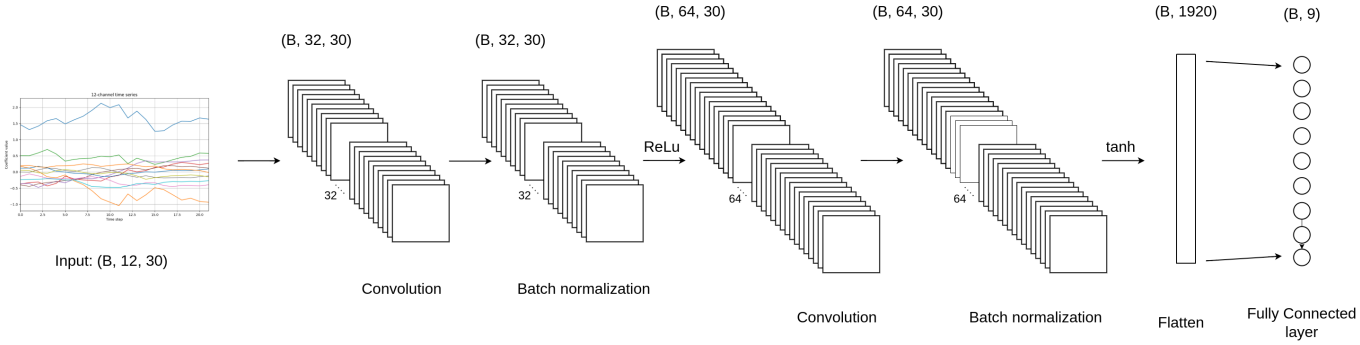

Fig. 3: Architecture of the convolutional neural network

the result outputted at each time step can be computed by the resolution of the following formula:

$$Z_{k,t}^{(1)} = \sum_{c=1}^{12} \sum_{\tau=-1}^{1} W_k^{(1)}(c, \tau+1) x_{c,t+\tau} + b_k^{(1)}$$

Each convolution layer is followed by a batch normalization layer in order to stabilize the network. Let $\gamma_k^{(1)}$ be the normalization scale and $\beta_k^{(1)}$ the shift for the k^{th} channel. Then we denote μ_k be the running mean and σ_k the standard deviation, both computed during training. Then, the result of the first batch normalization operation is given by:

$$\tilde{Z}_{k,t}^{(1)} = \gamma_k^{(1)} \hat{Z}_{k,t}^{(1)} + \beta_k^{(1)}$$

The notation $\hat{Z}_{k,t}^{(1)}$ refers to the computation:

$$\hat{Z}_{k,t}^{(1)} = \frac{Z_{k,t}^{(1)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

All of these operations are then activated by the ReLu function, yielding a partial result of the first block of operations $H_{k,t}^{(1)}$:

$$H_{k,t}^{(1)} = max(0, \tilde{Z}_{k,t}^{(1)})$$

The block is followed by a suite of similar operations, preceded by a second convolutional layer. This layer has a number of 32 input channels and 64 output channels, a kernel size of 3 and a padding of 1. We define $W_m^{(2)}$ as the m^{th} output channel of the second convolution, while $b_m^{(2)} \in \mathbb{R}$ represents the bias term, where $m = 1, 2, .., 64$. This convolutional layer yields a comparable result:

$$Z_{m,t}^{(2)} = \sum_{k=1}^{32} \sum_{\tau=-1}^{1} W_m^{(2)}(k, \tau+1) H_{k,t+\tau}^{(1)} + b_m^{(2)}$$

Analogously, the network then applies another batch normalization layer, for which we define familiar variables $\gamma_m^{(2)}, \beta_m^{(2)}, \mu_m, \sigma_m$, as previously described. Therefore, the output of the second batch normalization becomes:

$$\tilde{Z}_{m,t}^{(2)} = \gamma_m^{(2)} \hat{Z}_{m,t}^{(2)} + \beta_m^{(2)}$$

The notation $\hat{Z}_{m,t}^{(2)}$ refers to the computation:

$$\hat{Z}_{m,t}^{(2)} = \frac{Z_{m,t}^{(2)} - \mu_m}{\sqrt{\sigma_m^2 + \epsilon}}$$

The functionality of this second block is then activated by the $tanh$ function:

$$H_{m,t}^{(2)} = \tanh \tilde{Z}_{m,t}^{(2)}$$

The obtained features maps are collected and transformed into a single dimensional vector through a flattening layer. We denote this operation as $h \in \mathbb{R}^{64 \times 30} = \mathbb{R}^{1920}$.This is followed by the last step present in the neural network architecture, mapping results to a final category through a fully connected layer. We define the weight matrix obtained from the previous step as $W^{(fc)} \in \mathbb{R}^{9 \times 1920}$ and bias $b^{(fc)} \in \mathbb{R}^9$. Then, the result of the final layer, represented as logits is given by the formula:

$$logits = W^{(fc)} h + b^{(fc)}$$

2) Data flow: We will now examine the data flow traversed through our custom CNN model.

Assume an arbitrarily chosen batch size of size B. Each input example is of the form $(12, 30)$, which leads to final input tensor ξ to be forwarded to the network taking the shape of $(B, 12, 30)$.

$$\xi \in \mathbb{R}^{B \times 12 \times 30}$$

The tensor input ξ is then forwarded to the first element of the network, the first convolutional layer, denoted as $Z^{(1)}$. This layer has the effect of transforming the initial 12 channels into 32.

$$Z^{(1)} \in \mathbb{R}^{B \times 32 \times 30}$$

The first batch normalization does not affect the shape of the incoming data, as it is a stabilizing operation which re-centers and re-scales the input. Suppose we denote the result of this functionality as $\tilde{Z}^{(1)}$. The network proceeds to apply the ReLu activation function element-wise on the normalized result, which we symbolize with $H^{(1)}$.

$$H^{(1)} = ReLu(BN(Conv1d(\xi)))$$

After applying the first block of operations, the output retains the shape:

$$H^{(1)} \in \mathbb{R}^{B\times 32\times 30}$$

The workflow of the second block of operations, composed of a second convolutional layer, batch normalization and activation function performs analogously to the initial one. The first step, the convolutional layer, transforms the received 32-channel input into 64 channels, yielding an output of the form:

$$Z^{(2)} \in \mathbb{R}^{B\times 64\times 30}$$

Afterwards, a new batch normalization operation $\tilde{Z}^{(2)}$ is applied, which does not modify the output shape. This is followed by a $tanh$ activation function. The result of this block can be denoted as:

$$H^{(2)} = tanh(BN(Conv1d(H^{(1)})))$$

The resulting shape of $H^{(2)}$ retains the form:

$$H^{(2)} \in \mathbb{R}^{B\times 64\times 30}$$

As we approach the culminating point of the final classification output, each 2-dimensional feature map ought to be transformed into a simpler, 1-dimensional format. We realize this through employing a flattening operation. Therefore, we obtain a result of the shape:

$$flatten(H^{(2)}) \in \mathbb{R}^{B\times(65\cdot 30)} = \mathbb{R}^{B\times 1920}$$

The final step in the network is to corroborate the preceding results and yield a final output. We have achieved this through employing a fully connected layer.

$$logits = FC(flatten(H^{(2)})) \in \mathbb{R}^{B\times 9}$$

The output shape on this final operation is to be expected from a logical standpoint, as it is tasked with assigning each data point one of the 9 existent classes of speakers.

3) Model design: Construction of the CNN requirs careful planning and fine-tuning of the involved hyperparameters. This necessity was accentuated by the reduced dimension of the dataset, situation under which the model is highly prone to overfitting. In response to this, the model's architecture was designed with clear emphasis on simplicity and overfitting countermeasures.

The model was built incrementally, starting from simple, essential elements, such as convolutional layers and fully connected layers. Further elements have then been introduced (i.e. batch normalization, flattening layer). The model's performance was evaluated after each design iteration and the simplest model in terms of elements and hyperparameters was chosen. In the search for an appropriate model, various permutations of components have been assessed (i.e up to 3 convolutional layers, up to 3 batch normalization layers, up to 3 dropout layers, various activation functions: $ReLu, LeakyReLu, sigmoid, tanh$).

After settling on a definitive model, we proceed to hyperparameter fine-tuning in order to discover an optimal configuration. This process was conducted through a grid search across the algorithm. We have constructed hyperparameter pairs consisting of every possible permutation, against which the model was evaluated. The complete list of used values can be found in table III, where η represents the learning rate and λ signifies the weight decaying factor. While the first 2 parameters were selected through grid search, the number of epochs was chosen empirically, through independent trials.

Parameter	Values tested	Best value
η	$[1^{-5}, 1^{-4}, 1^{-3}, 1^{-2}]$	1^{-3}
λ	$[1^{-5}, 1^{-4}, 1^{-3}, 1^{-2}]$	1^{-3}
$epochs$	$[15, 30]$	30

TABLE III: Hyperparameters tested for optimal CNN configuration

Furthermore, for completeness considerations, cross-validation was used for maximizing the dataset's potential. This procedure entails splitting the dataset into 5 disjoint subsets D_1, D_2, D_3, D_4, D_5 of approximately equal size. For each fold $f \in \{1, 2, 3, 4, 5\}$, we define the training set as in equation 1 and the validation set as 2. Each fold will eventually play the role of the validation set.

$$D_{train}^{(f)} = \bigcup_{i=1, i\neq f}^{5} D_i \tag{1}$$

$$D_{val}^{(f)} = D_f \tag{2}$$

Suppose the parameter combination consisting of a pair (η, λ) is denoted as θ. For each pair in the complete grid $\Theta = \{\theta_1, \theta_2, ...\}$, we train the model on the training set $D_{train}^{(f)}$ with the specific hyperparameter combination and evaluate it on the validation set $D_{val}^{(f)}$. This process is repeated 5 times, for each fold and the results $s_f(\theta)$ are gathered. The final cross-validation score is obtained by computing the average of the fold scores:

$$S(\theta) = \frac{1}{5}\sum_{f=1}^{5} s_f(\theta)$$

Once all pairs $\theta = (\eta, \lambda) \in \Theta$ have been evaluated, a best-performing combination is identified:

$$\theta^* = (\eta^*, \lambda^*) = \arg\max_{\theta\in\Theta} S(\theta)$$

The optimal model, according to the obtained results, has the following configuration: learning rate $\eta = 0.001$, weight decay factor $\lambda = 0.001, epochs = 30$.

4) Optimizer: To train the CNNs we used the Adam optimizer introduced by [3]. Below, we describe how this algorithm updates the parameters of the network utilizing a moving exponential average of past gradients and scaling the learning rate for each parameter individually based on past gradients.

We have the first and second moment vectors m_0 and v_0 initialized to zero. We also have an objective function f parameteried by θ. In general, at timestep t, first calculate the gradient g_t:

$$g_t = \nabla_\theta f(\theta_{t-1}).$$

Then, we calculate the first and second moment vectors:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

Where β_1 and β_2 are the decay rates for the moment estimates so hyperparameters when using Adam for optimization. After that, we calculate the bias-corrected first and second moment estimates.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

This correction is needed because in the beginning m_0 and v_0 are initialized to 0 so the moving average is not accurate as it underestimates these values. Since in the beginning when t is small, the denominator will be small so m_0 and v_0 are scaled up accounting for the underestimation. When t is large enough then the denominator will be close to 1 so this step will not affect the estimates ($\beta_1, \beta_2 \in [0, 1)$).

After calculating the first and second moment estimates, we can update the parameter vector θ:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where α is the stepsize and ϵ is a small constant to prevent division by zero. This α parameter is also a hyperparameter of Adam.

5) Overfitting countermeasures: As previously stated, combating overfitting constituted a major point of focus in designing the convolutional network. In order to address this issue, we have employed a number of measures to ensure good generalization to novel data.

First of all, we have selected a small, simple architecture with only 2 convolutional layers and a single fully connected layer. The effect of this decision is a reduced number of learnable parameters, which is naturally less exposed to overfitting risks. This measure is especially efficient considering the small number of data points in the dataset (i.e. 270). Furthermore, we have eliminated additional complexity which might arise due to the variable sequence lengths by padding them to their maximum length of 30.

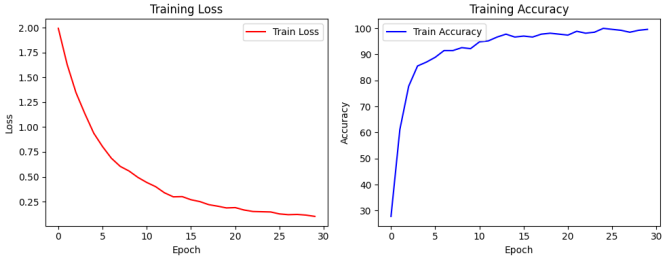

Fig. 4: Training loss and accuracy plot for the CNN model.

Furthermore, a notable source of standardization is illustrated by utilizing 2 batch normalization layers. This operation controls overfitting risks by stabilizing the training process and regulating the data. Batch normalization re-centers and re-scales the data, in other words, it smoothens the overall loss and maintains the actions of the activation functions well-conditioned.

Lastly, we make use of weight decaying in order to penalize large values. This measure constrains possible large weight vectors (in the optimizer), which helps control overfitting tendencies.

III. Results

During training the CNN showed ideal results with steadily decreasing training loss and increasing accuracy. Figure 4 shows the training loss and accuracy plot of the CNN.

To evaluate the effectiveness of the kNN and CNN for MTS classification on the Japanese vowels dataset and the 2dSVD for feature extraction. We analyzed the performance of the two models with and without the 2dSVD. The results are summarized in Table IV, showing the accuracy and F1 score of each method and the impact of the 2dSVD.

The baseline kNN achieved an accuracy of 90.03% and an F1 score of 0.93, while the kNN with 2dSVD achieved 96.76% accuracy and 0.97 F1 score. This improvement in performance shows that the 2dSVD effectively helps the model extract meaningful features from the data.

The CNN achieved 97.03% and 97.53% accuracy without and with the 2dSVD and a 0.98 F1 score for both. The CNN outperforms the kNN on both metrics. However, the CNN gains only a minor improvement from using the 2dSVD.

Algorithm	Accuracy	F1-score
kNN (baseline)	90.37%	0.93
kNN + 2dSVD	96.76%	0.97
CNN	97.03%	0.98
CNN + 2dSVD	97.57%	0.98

TABLE IV: Algorithm results

In order to gain more detailed insight on how the models distinguish between the speakers' utterances the confusion matrices of the KNN and CNN are shown on figure 5 and figure 6 respectively. The two confusion matrices use the results of the models with the 2dSVD used as preprocessing,

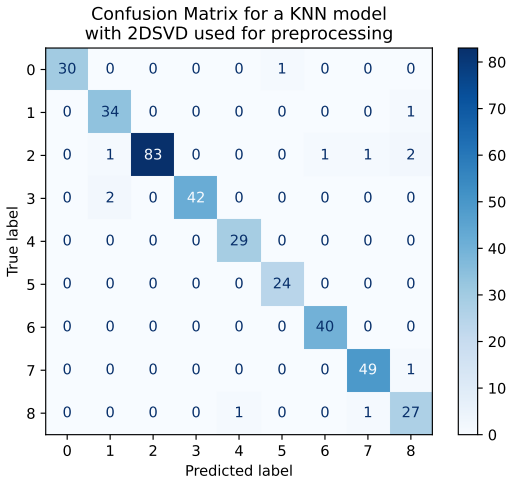

Fig. 5: Performance of the kNN on the test dataset with 2dSVD used as a preprocessing step ($k = 3$, $r = 5$, $c = 10$).

showing the number of occurrences of predicted speakers compared to the true speakers. The top-left to bottom-right diagonal values show the true positives, the number of correct predictions made for each speaker. Any other non-zero values are misclassifications.

The confusion matrices are almost identical, which is expected, because the two models have similar accuracy and F1 scores when using the 2dVSD. The vast majority of the predictions are on the diagonal indicating correct predictions. This shows that the models were able to learn features of the speakers' utterances in a meaningful way and are able to distinguish between all speakers with a high precision. No speaker was mistaken for the same other speaker more than 2 times.

IV. Discussion

The findings of our project show that both the kNN and the CNN are viable approaches for MTS vowel classification, achieving competitive results. The results of the kNN model highlight the effectiveness of using 2dSVD for feature extraction, highly improving its accuracy and F1 score. When using 2dSVD the kNN model achieved almost 97% accuracy close to the accuracy of the CNN model. On the other hand, the CNN outperformed the kNN, producing above 97% accuracy even without using 2dSVD for preprocessing. The 2dSVD has a better impact on the kNN because the model relies on additional feature extraction methods more than a CNN. The CNN is able to learn feature information through its convolutional layers.

While the automatic feature extraction in the CNN seems like an attractive option as we need to think very little about what should be done with the input data before feeding it to our learning algorithm, there are some disadvantages.

We can see from Figures 7 and 8 depicting the activations of the hidden layers that it is impossible to understand what features are learned by the CNN. If the training fails, we have

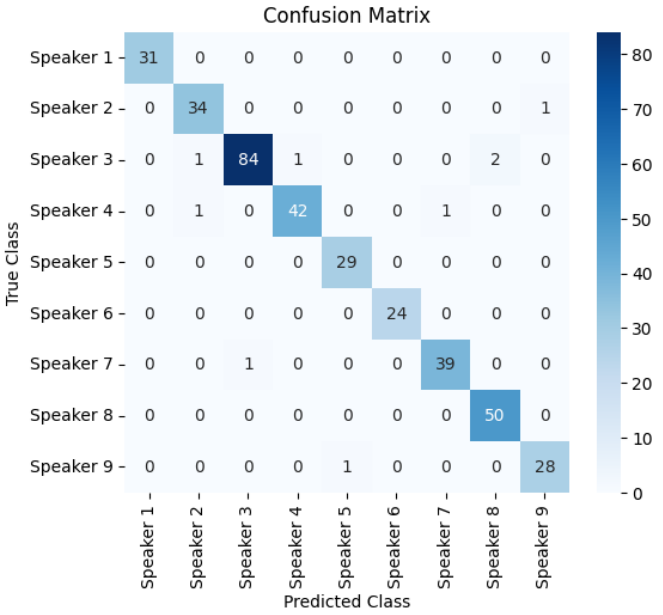

Fig. 6: Confusion matrix of the CNN with 2dSVD.

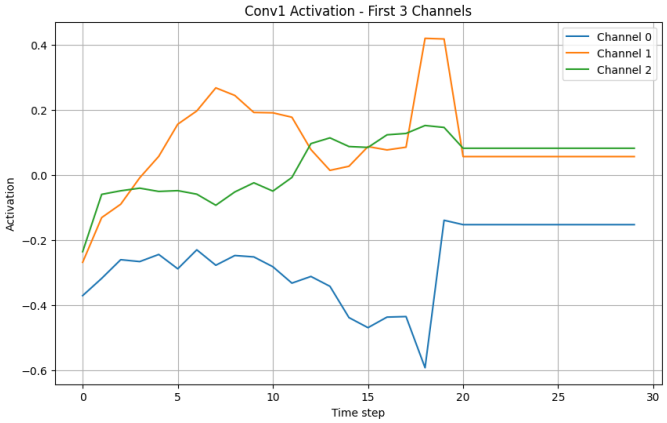

Fig. 7: Activations of the first 3 channels in the first input layer of the CNN

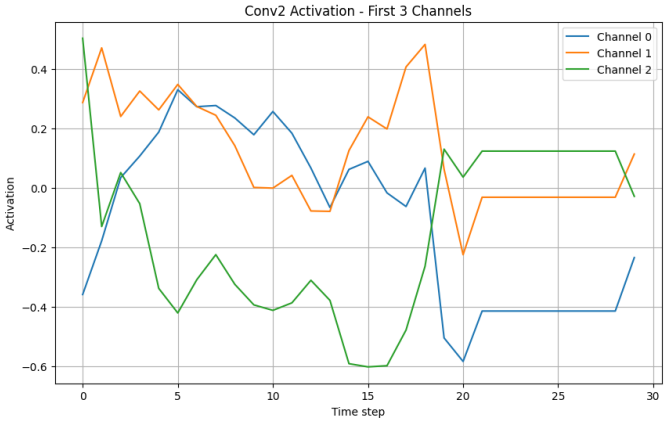

Fig. 8: Activations of the first 3 channels in the second input layer of the CNN

no idea what went wrong due to the 'black-box' nature of deep learning models. It is much easier to fix or improve a learning algorithm when all the steps in the pipeline are understood by the designer.

V. Conclusion

We designed multiple approaches to solve this MTS classification task. First, we used kNNs with crude feature extraction methods to get a baseline, then we followed the approach described in [5] for more advanced feature extraction. For comparison, we also proposed a deep learning approach. We trained two CNNs, one on the raw input data and the other one on the features selected by the method explained in [5]. While the careful preprocessing step proved to be quite beneficial for the kNN, there was little improvement for CNN.

Overall, we see that this elementary classification task demonstrates that a simple model like a vanilla kNN can produce comparable results to more computationally expensive deep learning approaches. This requires careful pre-processing so the model can learn on meaningful features. While the CNN produced marginally better results on the test dataset, it should be noted that the kNN requires a fraction of the computation of the CNN both during training (kNN does not even have a training phase) and when classifying new examples, the kNN only needs to calculate the pairwise distances while the CNN needs to perform a whole forward pass requiring many matrix multiplications. We conclude that it is often beneficial to put more effort into carefully designing a good feature extraction pipeline than to apply heavy machine learning frameworks and to effectively *"use a sledgehammer to crack a nut"*.

References

[1] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[2] Daniel Fogerty and Larry E Humes. The role of vowel and consonant fundamental frequency, envelope, and temporal fine structure cues to the intelligibility of words and sentences. *The Journal of the Acoustical Society of America*, 131(2):1490–1501, 2012.

[3] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn. Speaker verification using adapted gaussian mixture models. *Digital Signal Processing*, 10(1-3):19–41, 2000.

[5] Xiaoqing Weng and Junyi Shen. Classification of multivariate time series using two-dimensional singular value decomposition. *Knowledge-Based Systems*, 21(7):535–539, 2008.

[6] J. Yang, D. Zhang, A. F. Frangi, and J. Y. Yang. Two-dimensional principal component analysis: A new approach to appearance-based face representation and recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(1):131–137, 2004.