Phonetic category emergence and adaptation in a bidirectional phonological and phonetic neural network

Martin Rorick Terlou
Student number: 12140686

Universiteit van Amsterdam

Abstract

This exploratory paper uses three topics to investigate the phonetic categorization behavior of the Neural Network model of bidirectional phonetics and phonology 'BiPhon-NN' by Paul Boersma (2021). The first topic looks at the long-term progress of the Perceptual Magnet Effect. Using the PME as a tool to define phonetic categorization, the model shows a robust initial categorization of auditory inputs. The magnetism toward the mean of the auditory input decreases until it reaches an equilibrium at a distance slightly closer to the prototype than the input value. Introducing a second language triggers the model to adjust existing categories to the new inputs until the model reaches a new equilibrium. Testing the network at this point with its native vowel inventory shows that the model has lost the original categories and will have to re-train to assign the native inputs correctly. The model thus shows that phonetic categories are flexible and adjust to recent inputs.

BA Thesis - Linguistics
Supervisor: dr. Paul Boersma
Date: 21-06-2021

Contents

1 Introduction

Modeling natural events can help increase understanding and give insight into important aspects, thus help further develop tools to improve our interaction with the event. Take an example of the well-known modeling technique of cellular automata (first developed in the 1940s by Stanislaw Ulam), which uses a 2-dimensional grid to represent natural events. One of its early applications was to portray a forest with all its complexities and many unknown facets. The cellular automata made it possible to better understand the way fire spreads, and as a result, the model contributed to saving forests and lives. This model has continuously developed as more influences on the spread of fires became known, becoming increasingly realistic in its representation of actuality (Bendicenti et al. 2002; Ghisu et al. 2015; Alexandridis et al. 2011).

Although many linguistic phenomena have been observed and researched, numerous factors remain unknown. While we have much knowledge of superficial linguistic behavior, many uncertainties in human linguistic capabilities stem from a relatively small understanding of the human brain, the central processor that allows us to learn, teach, process, and produce. However, it is unclear how it is capable of that.

In an attempt to model human language production and perception using Optimality Theory, Boersma (2011) made a bidirectional model of phonology. This model was meant to handle 'all' phonological (and related) phenomena while staying minimalistic. Starting from the Phonetic representation (the sound) in figure 1, a listener walks up through different steps and ends up with a meaning; the speaker travels down the figure, from an intended meaning to the articulation (phonetic representation). The two intermediate levels influence the potential change from one end to the other.

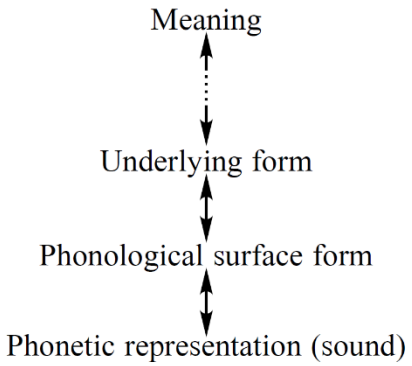

Figure 1. Bidirectional phonology and phonetics (Boersma 2019)

The above model makes use of existing phonological categories. Boersma sought to further develop the bidirectional phonetics and phonology model (BiPhon) using neural networks. In doing so, he could possibly show how the phonological categories emerge via distributional learning, using an 'Inoutstar' algorithm (Boersma, Benders & Seinhorst 2020) or Hebbian learning (Boersma 2019). Neural networks can be preferred over often better performing Artificial Intelligence techniques, as neural nets grant us the ability to observe and measure all the inner workings and thus increase our understanding.

Like the original BiPhon model, the neural network version was made under the assumption that phonological systems are emergent. This means that phonological features gradually develop by bottom-up and top-down processing of phonetics and morphology. The features thus emerge through repeated production and perception tasks affecting the category edges. A neural network, if functioning realistically, could supply more tools for the discussion of whether phonological categories are emergent or innate.

2 Neural networks and phonetic categories

In a subsequent version of the neural network adaptation of BiPhon (henceforth: BiPhon-NN), Boersma, Chládková & Benders (2021) simulate the perception and production of a limited number of sound–meaning pairs using a Hebbian learning algorithm in a deep Restricted Boltzmann Machine (dRBM).

It uses a layer of 49 nodes representing the basilar membrane on the same layer as five nodes that represent the meaning. There are two so-called hidden layers between the two surface layers instead of the standard single layer. Hence the need for the added term "deep" to the RBM. The word 'Restricted' refers to the lack of connections between nodes within the same layer.

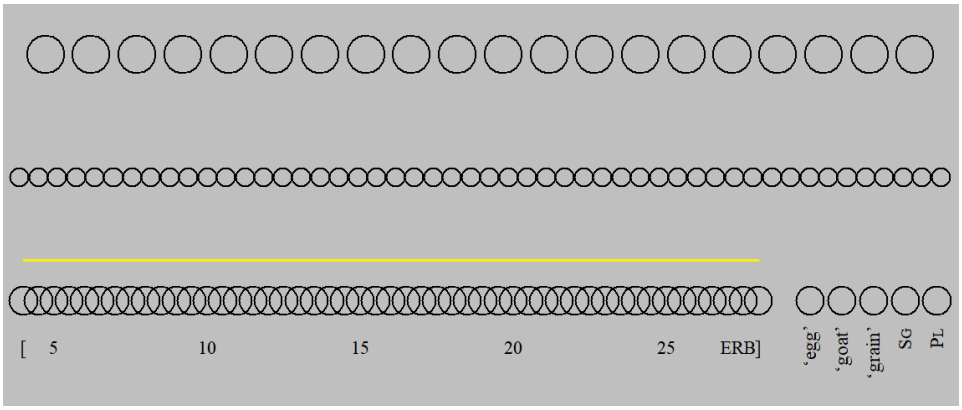

Figure 2. A graphical representation of the BiPhon-NN model (Boersma, Chládková & Benders 2021)

The left 'slab' of the bottom layer in figure 2 represents the basilar membrane and shows the Equivalent Rectangular Bandwidth (ERB) values, ranging from 4.0 to 28.0 ERB. The bottom right nodes represent the semantic correlates. The model can simulate perception when nodes on the basilar membrane *x* are activated; the activation spreads to the 'hidden' middle layer *l*, then to the hidden top layer *m*, and finally back down via the middle layer to the meaning layer *x* (as well as back to the left slab). In a trained network, this activation spreading will end up activating the correct meaning layer node depending on the input value on the basilar membrane. The activation direction is reversed to simulate production, starting at the meaning layer going to the basilar membrane layer.

Training the network is done in multiple steps, and those are carefully explained in section 3. For now, it is important to explain that the network is trained by either inputting auditory values or inputting a meaning or a sound–meaning pair, all befitting a toy language (consisting of five vowels). Then the input activations spread through the network and after which the network adjusts the weights (the connection between nodes) and biases (regulating the sensitivity of the node to activate). For example, when training the network with the vowel '*a'*, the values for '*a*' are normally distributed with a standard deviation of 1 ERB. The mean values for the perceptual vowel height (F1) and perceptual vowel backness (F2) are 13 and 19 ERB, respectively. Thus an '*a*' with values 14 and 20 ERB is inserted into the network less often than one with 13 and 19 ERB. This distribution is done to simulate real-life auditory input as a perceived sound can be influenced by, e.g., adjacent sounds, speaker-specific variation, or even environmental noise.

2.1 Categorical perception in BiPhon-NN

When training the system using only sound inputs, Boersma et al. (2019; 2021) found a difference in the activations of the nodes on the ERB layer after letting the network spread the activation via the hidden layers and echoing back to the ERB layer. The bidirectionality of the network causes any activation to return to the original layer after an input. When the activation spreads back to the layer where the sample input originated, the activated nodes, or their level of activation, can differ from the original input. This 'echo' appears to have different values from the initial input, especially when the input values were peripheral regarding the trained vowel distribution. In their paper, Boersma et al. show this by an original input '*a*' (14 and 18 ERB), returning an echo with an activation on 13 and 19 ERB, i.e., the mean of the training distribution of '*a*'.

The authors identified this as the Perceptual Magnet Effect (Kuhl 1991). This effect is where perceptual space is 'warped' around phonetic categories. Acoustic patterns are perceived closer together when they are near phonetic category centers (native language prototypes) than when equally spaced patterns are further away from the prototypes. This allows the listener to group a large degree of variation in incoming sounds under one umbrella. Language-specific behavior in the Perceptual Magnet Effect (PME) has been shown by its absence in newborn infants (Polka 1995). However, the PME arises from six months of age (Kuhl 1991). Exploring the existence of a perceptual magnet effect in adults can be problematic and deserves its own section. Several approaches are explored in section 2.2.

Some argue that phonological categories are formed due to innate categorical knowledge (Kuhl 1995). But Guenther and Gjaja's (1996) proposed an emergent nature in which continuous auditory representations play a significant role. They used a neural map model to show the perceptual magnet effect. The inputs for their network did not include any featural information or labeling on the distributed sounds during training. Their model showed a skewed perception of peripheral inputs toward native language prototypes, and they concluded that categorization is caused by a statistical distribution of native language prototypes during infancy. Similar to Guenther and Gjaja, Boersma attributed the activation difference of the echo in his BiPhon-NN model to the distributional learning methodology of the auditory samples. His network was initialized as a blank state without any linguistic labels on the auditory inputs. Assuming phonological categories are emergent, the artificial network is then comparable to a newborn human.

2.2 Categorical perception in adults

The BiPhon-NN might thus accurately simulate the emergence of phonological categories from a newborn to an infant. However, Boersma et al. (2021) observed a change in the network after 10.000 training steps where the echo no longer differed from the category-peripheral inputs. The question then arises, "Is the disappearance of the PME reflective of adult human behavior?". If adults can more easily distinguish diverse auditory inputs that are category-peripheral than more central inputs, then PME is not necessarily caused by a perceptual disability. Assuming phonological categories are emergent, adults will already have formed such categories and could now be (subconsciously) classifying auditory inputs into these categories. Furthermore, if one argues that a listener is physically no longer able to perceive such variations, this posits some challenges in second language acquisition. If auditory information is perceived in an altered

manner due to the listener's native language, how do they learn to perceive or even develop new phonetic contrasts? For this reason, the present paper focuses on phonetic categorization by only looking at auditory processes removed from any semantic information.

Studying second language learners' ability to attain phonetic contrasts absent in their native language(s) is a way to investigate this problem. One option is to explore the ability to split a native category into two separate non-native categories. Ylinen et al. (2010) wanted to know whether non-native contrasts could be acquired through phonetic training. They looked at both speech production behavior and electrophysiological data (specifically MMN responses). The participants' MMN responses showed that before training, they mostly used duration changes to distinguish the non-native categories, namely the difference between /i/ and /ɪ/. After training, the physiological data showed that the participants had enhanced preattention to the spectral differences. Therefore, adult learners are able to shift existing categories.

Boersma (2021) was able to see the PME in his network while not using semantic data. Similar to Guenther and Gjaja, the network can be continuously trained using only auditory data. Thus it is possible to constantly identify the level of the PME without any potential semantic interference, hence measure the level of categorization in the network even at an adult-like state.

2.3 Native Category Retention

Categorization differences can thus be seen during infancy, where initial categories emerge depending on the native linguistic environment. At a later stage, when humans learn a second language, it is possible to see the native categories adapt to suit the new language. A third and final stage that can be used to measure the efficacy of BiPhon-NN to mirror human behavior is native category retention or attrition.

Category retention refers to the ability to recognize auditory differences of the native language by speakers that have been fully submerged in a second language. First language attrition appears clearly in production and comprehension (Nicoladis & Grabois 2002). While researching phonological L1 attrition of adopted Korean children, Ventureyra, Pallier & Yoo (2004) found that native language phonology was severely attritted. Within their study, the participants had been entirely severed from their native languages after their adoption. If phonological categories emerge from phonetic information, this finding would imply a degradation of the perceptibility of the native language. Other studies show some retention of phonetic perception, potentially due to phonetic overlap in the native and second language or an incomplete separation from the native language (Oh et al. 2003; Au et al. 2002; Werker et al. 1981).

2.4 Goal of the study

This paper compares the behavior of BiPhon-NN to human behavior as found by previous studies regarding categorical perception. The paper shows how large the distance is between the echo and the original values, thus visualizing the Perceptual Magnet Effect at various stages of the network's training. If we equate the initialized blank slate of the network to a newborn infant, we can make the first comparison. The network is expected to initially show little shifting of input values. The distance between the values of the echo and the input is likely to increase at subsequent steps in the network's training until it reaches an equilibrium state.

The second comparison is then the reaction to being introduced to a second language. The network will be trained with inputs belonging to a new language. The difference between the echo distance and the input distance is expected to be large while the network is just starting its training. The new inputs will be assigned to native categories that lie further away and will therefore not be returned closer to the L2 prototypical values. As the training continues, it is expected the distance will decrease until it reaches a new equilibrium.

The level of retention of the native categories is the final comparison made. Since the network employs an algorithm that causes unlearning of unused connections (see chapter 3.5), it is expected to show greater first language attrition when there is less overlap between the first and second languages. These three topics will help further determine if the network can mirror human behavior of phonetic categorization accurately.

In the upcoming chapter (chapter 3), the algorithms that make up the network's training are explained. Chapter 4 focuses on implementing the measuring tools on the basilar membrane layer to show the perceptual magnet over time. Chapter 5 Inspects the flexibility of categorization in the network by looking at the PME changes when switching the language the network is trained with. In chapter 6, a similar investigation will be done but focusing on native category retention or attrition. Finally, in section 7, the findings of sections 4, 5, and 6 are discussed and compared to observational data of human behavior. This section will also state the model's current deficiencies and potential improvements in future research.

3 Training the network

Opposed to the network introduced in chapter 2, the network utilized in this paper will not use a meaning layer. Similar to Boersma (2019), the Perceptual Magnet Effect is explored only via auditory information.

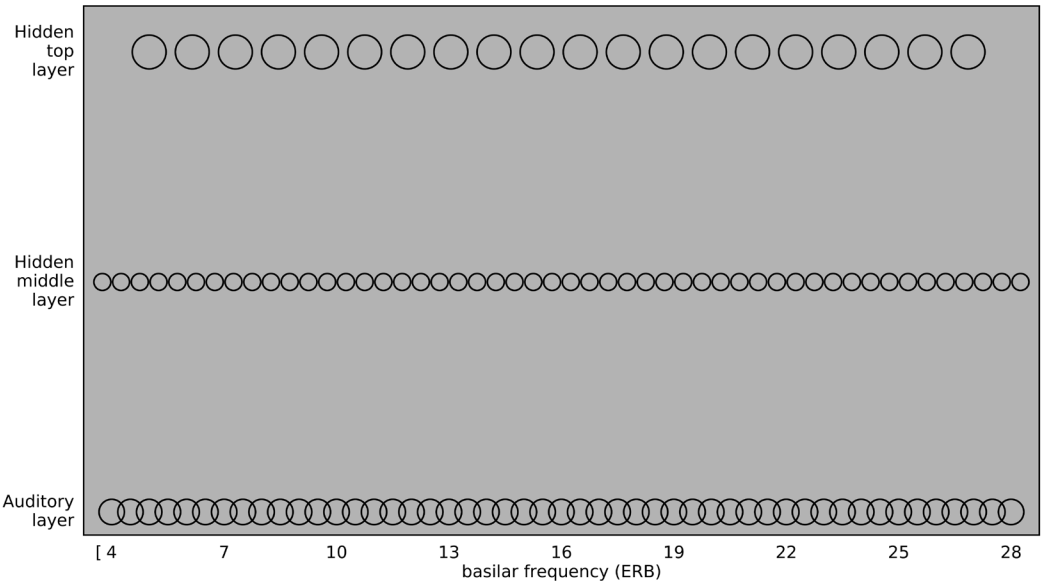

Figure 3. BiPhon-NN (no meaning layer)

As seen in figure 3, the network has an auditory layer consisting of 49 nodes representing a continuum of 4 to 28 ERB (in steps of 0.5 ERB). The middle and top hidden layers consist of 50 and 20 nodes, respectively. All nodes are initialized with a bias of 0. Every node on a layer is connected to each node on the adjacent layer(s), however not connected to nodes on the same layer. All connections are initialized with a weight of 0.

From this initial state, the network is trained. One training step consists of four phases and starts by setting the activations of all nodes to 0. Then, one piece of data is applied to the network by choosing a random vowel and picking the corresponding F1 and F2 values from its distribution.

3.1 Auditory input

As explained in chapter 2, the F1 and F2 values are randomly taken from a normal distribution with a standard deviation of 1.0 ERB. Table 1 gives an overview of the mean values for each input vowel. These values correspond to the toy language in Boersma (2021).

Table 1. Input vowels mean F1 and F2 ERB values

Input (=utterance)	*Mean F1 (ERB)*	*Mean F2 (ERB)*
a	13	19
e	10	22
i	7	25
o	10	16

u	7	13

After the F1 and F2 values are determined, the activation on the basilar membrane layer (k) is calculated for every node. The activation is not limited to one node, but neighboring nodes are activated to a lesser degree following a gaussian distribution with a standard deviation of 0.68 ERB. This distributed activation causes the basilar membrane to show two bumps of activation. Figure 4 shows an example of an input picked from the potential values of 'a'.

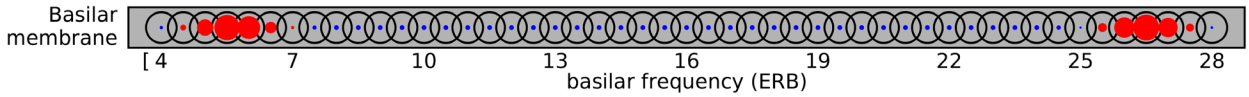

Figure 4. Example auditory '*i*' input

Figure 4 displays the two formants of the input by the larger red dots representing positive excitation. The blue dots signify nodes that are inhibited, i.e., have a negative activation. The size of the dots corresponds to the degree of excitation or inhibition.

3.2 Settling phase

The first phase of the learning step starts when the input has been transformed into basilar membrane activities x_k, where k runs from 1 to K and K is the maximum number of input nodes (49). The hidden middle and top layers excitations are set to 0, which corresponds to activations of 0.5. The bottom-layer activations do not change (remain clamped) during this phase. The activation is then spread to the middle layer from both the bottom and the top layer. The activation of a node on the middle layer y_l, where l goes from 1 to L where L is the maximum number of nodes of the middle layer (50), is calculated via the following formula (1):

$$y_l \rightarrow \sigma\left(b_1 + \sum_{k=1}^{K} x_k u_{kl} + \sum_{m=1}^{M} z_m v_{lm}\right) \quad (1)$$

The section between parentheses calculates the excitation of a middle layer node. This contains three parts separated by addition markers. The first part is only b_1, which refers to the bias of the current node. The second part starts with the summation symbol that shows that the multiplication of x_k by u_{kl} is done for each node and connection. Here, x_k refers to the activation of the bottom layer node, and u_{kl} refers to the weight of the connection between the node on the bottom layer and the current node on the middle layer. The weight and activation are thus

multiplied and summed for every possible connected node from the bottom layer. The final part in the formula is identical to the second part but referring to the top layer nodes (z_m) and connections (v_{lm}) using $_m$ for nodes on the top layer from 1 to M (20).

The excitation calculated above is then transformed using a sigmoid function shown by σ, into the nodes' activations. This σ-function takes the excitation (a limitless positive or negative number) and converts it to fit a number between 0 and 1. A negative excitation is transformed to a value between 0 and 0.5, and a positive excitation takes a value between 0.5 and 1. Formula (2) shows the formula for the sigmoid:

$$\sigma(x) := \frac{1}{1 + \exp(-x)} \tag{2}$$

The formula (1) spreads the activations on the bottom and top layers to the middle layer. Following this step, the activations will be spread from the middle layer to the top layer using the formula in (3):

$$z_m \rightarrow \sigma\left(b_m + \sum_{l=1}^{L} y_l v_{lm}\right) \tag{3}$$

Steps (1) and (3) are repeated ten times to let the network reach a near-balanced state in the middle and top layers. The initial settling phase is complete after these ten repetitions, and the system will continue with the second phase.

3.3 Hebbian learning phase

During the Hebbian learning phase, the network will adjust the biases of all nodes and the weights of all connections. The current activation of a node controls the change in the node's bias. The bias is strengthened if the node is active and is modified via the following formulae:

$$a_k \leftarrow a_k + \eta x_k \tag{4}$$

$$b_l \leftarrow b_l + \eta y_l \tag{5}$$

$$c_m \leftarrow c_m + \eta z_m \tag{6}$$

These formulae, one for each layer of nodes, mean that a node's bias is increased by the learning rate (η), a fixed scalar of 0.001, multiplied by the activation of the same node. For example: if a node's current bias is 0.2 and its current activation is 0.6, then its new bias will be

$0.2 + 0.001 \cdot 0.6 = 0.2006$, increasing the node's future activity when receiving an identical input.

Connections are strengthened when both attached nodes are active, following Hebb's law: "neurons that fire together, wire together.". The connections are changed as follows:

$$u_{kl} \leftarrow u_{kl} + \eta x_k y_l \tag{7}$$

$$v_{lm} \leftarrow v_{lm} + \eta y_l z_m \tag{8}$$

Here, the weight of a current connection is increased by adding to it the learning rate multiplied by the activity of each of the two connected nodes. The higher the activity of each node, the larger the increase in the connection's weight. If the connections weight starts at a value of 1 and is connected to nodes with an activation of 0.6 and 0.7, the new weight would become $1 + (0.001 \cdot 0.6 \cdot 0.7) = 1.00042$.

If an unlearning phase does not counteract this learning phase, the weights and biases can increase indefinitely. The subsequent two phases are introduced to remedy this, where some of what the system has learned is stochastically unlearned.

3.4 Dreaming Phase

Via the dreaming phase, the network will introduce a level of stochasticity. The activity of the network during this step is no longer directly based on a predetermined input. First, the activity in the previous step on the middle layer will be spread to the bottom layer. Following a similar formula as in (1) and (3) but specified for the bottom layer and not using a sigmoid function:

$$x_k \leftarrow a_k + \sum_{l=1}^{L} u_{kl} y_l \tag{9}$$

a_k represents the bias on the basilar membrane node $_k$. The connections u_{kl} are thus used to spread the activity from the middle to the bottom layers and back up as in (1). This shows the bidirectionality in the system by using the same connections for bottom-up and top-down processing.

The next step in this phase shows the stochasticity of the network. Similar to (3) and (1), the activity is spread to the top layer and middle layer respectively, ten times in a row, with one adjustment:

$$z_m \sim \beta \left(\sigma \left(b_m + \sum_{l=1}^{L} y_l v_{lm} \right) \right) \tag{10}$$

$$y_l \sim \beta \left(\sigma \left(b_1 + \sum_{k=1}^{K} x_k u_{kl} + \sum_{m=1}^{M} z_m v_{lm} \right) \right) \tag{11}$$

In (10) and (11), a Bernoulli distribution β modifies the result given by the original formula explained in (1) and (3). This equation sets the node's activation randomly to 1 or zero, with a higher chance to be 1 if the outcome of the sigmoid is above 0.5. Remember, the sigmoid function forces the activation to be between 0 and 1. If the sigmoid produces a value of 0.7, there is a 70% chance of the node's activation to be 1.

3.5 Anti-Hebbian learning

After stochastically controlling the activation of the middle and top layer nodes, the network unlearns some of what it has learned. This unlearning follows the same methodology as the learning in the Hebbian learning phase. The greatest unlearning takes place on (or between) active nodes. As the current activity of the nodes is randomly assigned, the unlearning has a level of stochasticity as well.

$$a_k \leftarrow a_k - \eta x_k \tag{12}$$

$$b_l \leftarrow b_l - \eta y_l \tag{13}$$

$$c_m \leftarrow c_m - \eta z_m \tag{14}$$

$$u_{kl} \leftarrow u_{kl} - \eta x_k y_l \tag{15}$$

$$v_{lm} \leftarrow v_{lm} - \eta y_l z_m \tag{16}$$

To sum up, one learning step follows a spreading of the activation on the basilar membrane to the middle and top layers via (1) and (3) ten times, then applies a learning algorithm by computing the new weights and biases using (4) through (8). In the dreaming phase, the network first unclamps the bottom level activation and spreads the middle layer activation to it (9). Then it randomly spreads activation via (10) and (11). Steps (9), (10), and (11) are repeated ten times. Finally, the system unlearns a bit of what it has learned using anti-Hebbian learning by applying (12) through (16).

4 Network behavior

Now that the learning methodology of the network is explained, we can look at the network's behavior at different points in its training. When the network receives an input, we can let the network process the signal and set activations on the middle and top layers. Finally, the

network returns a new signal based on the activations in those hidden layers to the basilar membrane layer. The new activations of the echo might be different from the original input and, according to Boersma et al. (2021), show the perceptual magnet effect.

In the following paragraphs, it is first explained what inputs will be used for all tests. Then it is described how the perceptual magnet effect can be seen in the current network. Following that, before showing the perceptual magnet effect at increasing training steps, a singular value needs to be found that can accurately describe the perceptual magnet effect.

4.1 Experiment input selection

As described in chapter 3.1, the network is trained with inputs taken from a normal distribution with a mean corresponding to the vowel prototype. The frequency distribution of the vowel data creates zones in the network, in which highly frequent inputs define the center of the zone.

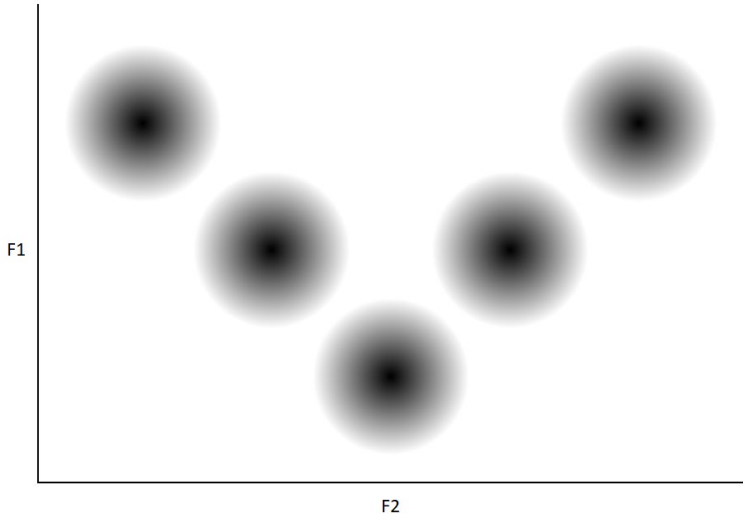

Figure 5. Category creation in the network

When plotting the first and second formants in a graph, this results in the zones shown in figure 5, where black areas signify frequent inputs and grey areas infrequent. The further away from the center, the less frequent a value is inserted (inputted into the network) during the network's training. Giving the network a test input far from the center can cause the network to assign these values to any of the vowel categories generated or to no categories and, in turn, give inconclusive results.

For this reason, a fixed distance from the zone's center was chosen for the input vowel formants, where the likelihood of the network having trained with this input is high. The F1 and F2 values were selected by choosing a random point on a circle around the center of the category.

The radius of this circle was set at 2 Standard deviations of the distribution of values the system was trained with (see chapter 3.1). Thus all inputs are 2 ERB removed from the prototype values.

4.2 The difference in echo and input activations

At 2000 training steps, the network shows a change in activity on the basilar membrane layer between the input and the echo. Figures 6 shows the input activations (upper layer) and echo activations (lower layer) for each of the five input vowels. For this visual, the input values were chosen following the explanations in the previous paragraph.

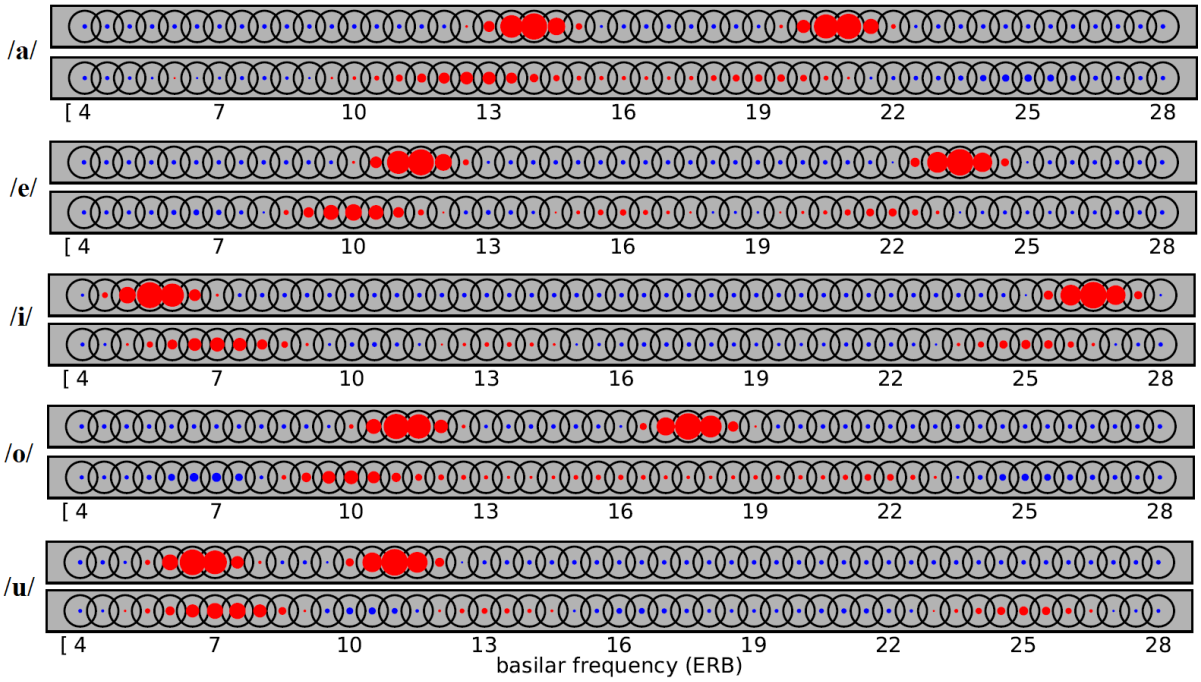

Figure 6. Input and echo activation on the basilar membrane

At 2000 training steps, we can see that the activation bumps of the echo are shifted slightly toward the closest vowel prototypes. This is especially clear in the vowels /a e i/, yet much less so with /o/ and /u/. The difference could be caused by these vowels sharing formant values with the other vowels. The network then might 'think' a different vowel was inserted, like with [u] being partially returned as [i] by having a third activation bump on a value corresponding to the F2 of /i/.

It is possible to visualize the activation difference over many more inputs by plotting the two highest (most pronounced) bumps on a graph[1]. The first bump on the basilar membrane would be the first formant, and the second bump the second formant. These two formants together form

[1] The returned formants were estimated by finding the two highest local maxima in a list of activation values, smoothed via ‘cubic spline’ interpolation, on the basilar membrane nodes.

coordinates to a two-dimensional plane. Between the two points, one for the input coordinates and one for the echo coordinates, it is then possible to draw a line that shows the drift of the echo activation from the input.

In figure 7, we can see such lines for a total of 200 inputs for each vowel. The inputs are all picked from the outer edge of the circular shape (see 4.1). Thus the circle's edge corresponds to the input activation values, and on the other end of the line are the echo's activations.

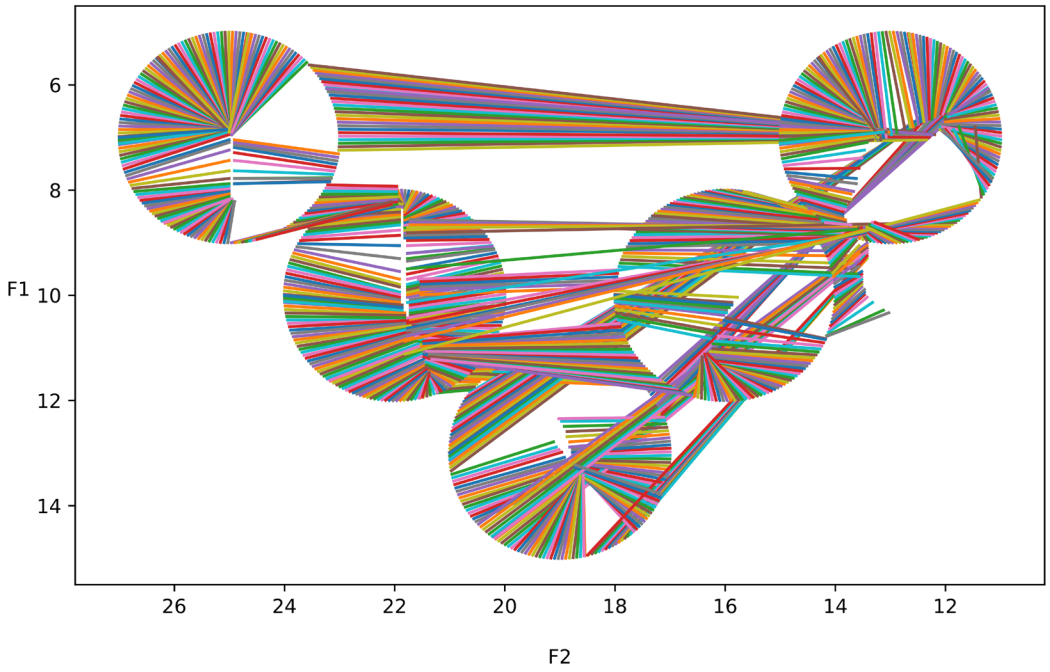

Figure 7. Drift lines, at 2000 training steps

There are two main characteristics in figure 7. First, most drift lines go from the circle periphery to its center, meaning the network perceives the vowel as more central than the input was. However, the second characteristic is that some lines travel to the center of a circle that it did not start on. These inputs get 'confused', potentially due to a lack of training, and jump to a different category. This was also seen in figure 6, where [u] is returned as [i]. The other category jumps present are mainly between /o/ and /e/, which share F1 values, and between /u/ and /a/. These latter jumps might seem random at first glance, but /u/ and /a/ both have an activation bump close to 14 ERB. For /u/, this is the value corresponding to the first formant, while for /a/, it is the second formant.

4.3 Input/echo distance

It is impractical to use many figures in a row like the one in figure 5 to show the network's behavior over an increasing number of training steps. Therefore, it is necessary to come to a single

number to represent drift. This number can then be calculated at different training steps and visualized.

When plotting the formants for both the input and the echo in a similar way to figure 5, it is possible to draw a line between those coordinates and those of the prototype mean instead of to each other. The length of those lines reflects the distance the peaks are removed from the prototype formants. For example, if an input with an F1 of 14.79 ERB and an F2 of 19.87 ERB is inserted, the Euclidean distance from this input to the prototype center of /a/ is approximately 2 ERB. By doing the same for the formant values of the echo, we have two scalar numbers that can be visualized. These values will be referred to as simply: 'input distance' and 'echo distance'.

4.4 Network behavior

Having all the needed definitions and data, it is now possible to show the network's behavior over an increasing number of training steps. Figure 8 shows the median[2] of the distances of 1000 random inputs. The blue line shows these input distances, which are all 2 ERB removed from the prototype center. The distance calculation is done every 500 training steps until the network has been trained for 250000 steps.

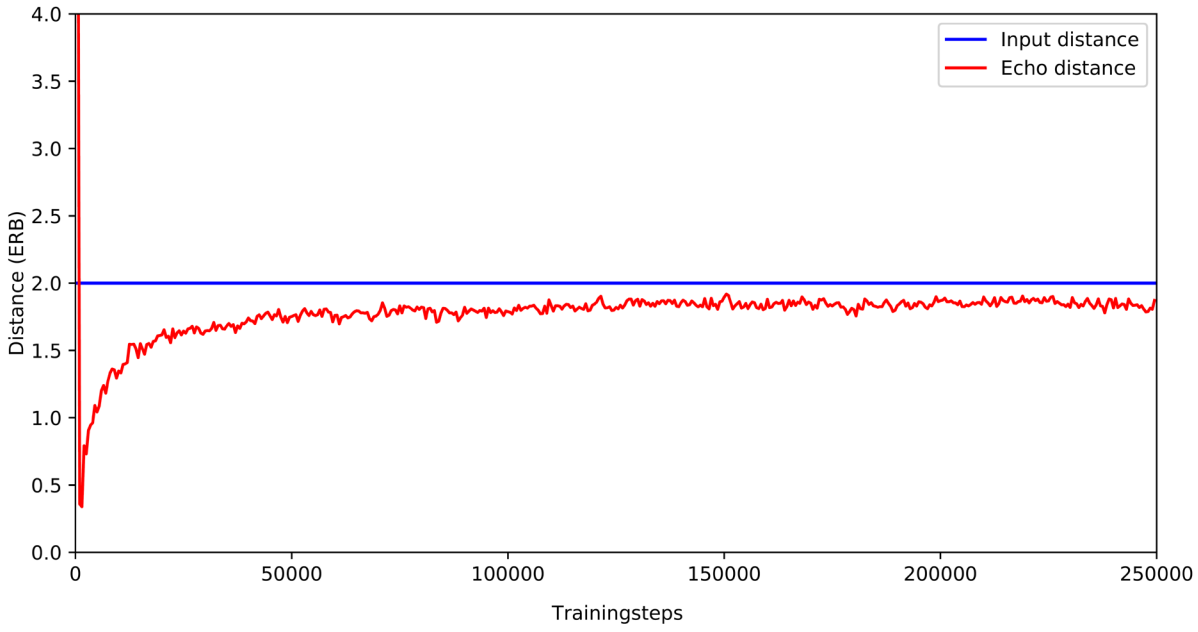

Figure 8. Echo and input distance values

[2] Comparison between results using the median and mean distance of 1000 inputs, showed that the network settles at the same value. The median is a better representation of the drift lines in fig 7 and is thus preferable. A downside is that slight changes get overshadowed, more on this in chapter 5.

The orange line represents the *echo distance* and starts out large. This is likely caused by the untrained state of the network, causing it to return an almost random activation. The random activations are often interpreted as category jumps that end up at a large distance from the original input prototype, severely affecting the overall result.[3] The network sees steep progress where the distance decreases until it reaches a minimum of approximately 0.4 ERB. The *echo distance* then starts increasing again, however, never ending up larger than the input.

The network seems to reach an equilibrium state at 50000 training steps. At that point, the average *echo distance* is only slightly less than the *input distance*, varying between 1.70 to 1.80 ERB. To ensure this small value could still be attributed to the Perceptual Magnet Effect, we have to inspect the directions in which the echoes drift. Recreating figure 7, but now using the system at 50000 training steps, will help clarify the network's behavior.

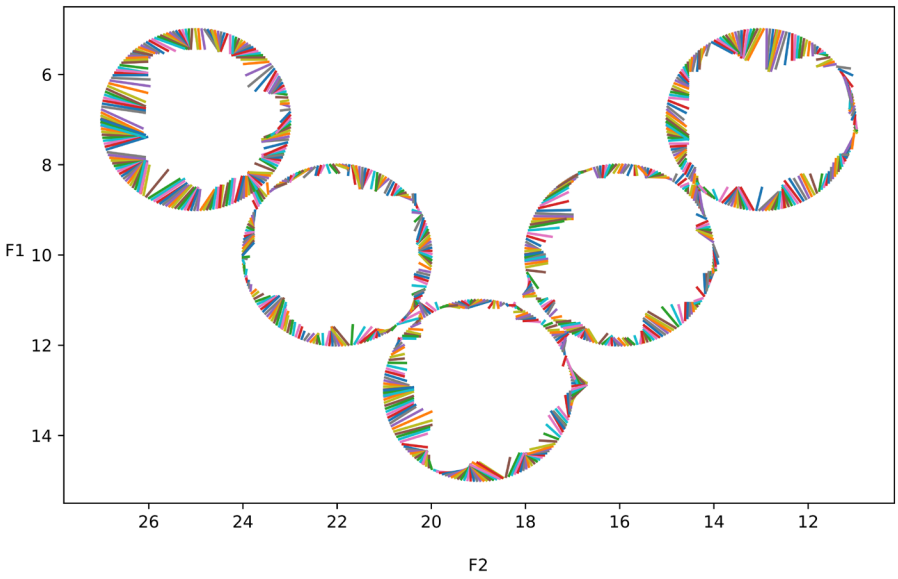

Figure 9. Drift lines at 50000 steps

Figure 9 shows that the slightly lower distance in the network's equilibrium state is an accurate depiction of the behavior of the network. A clear difference between this figure and the one from figure 7, where the network was trained only 2000 steps, is that the drift lines are a lot shorter. This is because the network has returned the input with a similar activation or has 'accepted'

[3] Category jumps are taken into account, as ignoring them could cause in distance values falsely suggesting a very strong magnet effect at early stages in the network's training

the input as it was. This could indicate that the network perceives values on one side of the category edge as different from those on the opposite side.

Albeit slightly, the echo is generally closer to the prototype center than the input was. An exception to this is the lines that move toward a different category center than to which the input was assigned. These deviants only appear when their F1 or F2 values are close to a neighboring category prototype. The artifacts rarely happen on the outer edges of /i/, /a/, and /u/.

On the outer edges of the two vowels with the most remote formant values, /i/ and /u/, the test inputs drift more inward than inputs on the inner edges. Further indications for a more balanced dispersion of vowel categories can be found where the vowel prototype categories are close. Most notably, inputs on the left edge of /a/, the bottom left edge of /o/, and the bottom edge of /e/ drift more inwards than the other edges. This spreads apart the returned activations from one another. The returned categories might thus be pushed further apart, which adheres to the 'Dispersion Theory' (Liljencrants & Lindblom 1972). According to this theory, sounds with larger contrastive features are preferred over more sounds that are very similar. The network seems to fit the theory by having a larger variety of formant values make up the center of the category. On the other hand, the network can distinguish different inputs from the same category, which does not follow previous human studies where humans were less able to distinguish sounds that belong to one category.

5 Categorical flexibility of Biphon-NN

Further steps need to be taken to explore and compare the artificial network to humans, and this can be done by studying how the network reacts to a merging or splitting of categories. As explained in chapter 2.2, studies have shown that humans can adapt to new auditory inputs that do not follow their existing categorical structure. Humans can adjust their current categories to fit the layout of the new inputs of a new language.

By giving the network a new set of vowels and retraining the network with that set, we can examine if and how the network adapts to the new inputs. Keep in mind, as this network does not incorporate semantic data, the results might not perfectly represent real-world data. In reality, learning a second language cannot be done purely auditorily; therefore, categories made or altered when learning a second language could be influenced by semantic data.

This chapter will examine the network's reaction to a category split and a category merger. Before delving into the split, we will need to train the network with a four-vowel system detailed

in chapter 5.1. The performance of the network with a four-vowel input is then compared to the original five-vowel input. Following that comparison, the reaction to split will be analyzed. Finally, the chapter will finish by comparing the reaction to a merger.

5.1 New input vowels - Four vowel language

The first step is to create a language with four input vowels, where two of the five-vowel language vowels are merged. The network will be trained exactly the same as explained in chapter 3.

Input (=utterance)	*Mean F1 (ERB)*	*Mean F2 (ERB)*
a	13	19
ɪ	8.5	23.5
o	10	16
u	7	13

Table 2. Input vowels mean F1 and F2 ERB values

The vowel /ɪ/ is now added to the set in place of /i/ and /e/. Table two shows the formant values paired with this new vowel which lie between the original /e/ and /i/. Note that the IPA symbol, /ɪ/, is chosen somewhat arbitrarily; the formant values are leading. The network will again be trained by picking a random vowel from the above table and then picking an F1 and F2 from a normal distribution with a standard deviation of 1 ERB.

5.2 Four vowel system network

Analyzing the behavior with a four-vowel system is done similarly to chapter 4.1, namely by looking at the drift lines at 2000 steps and the distance to the prototype center up to 250000 steps.

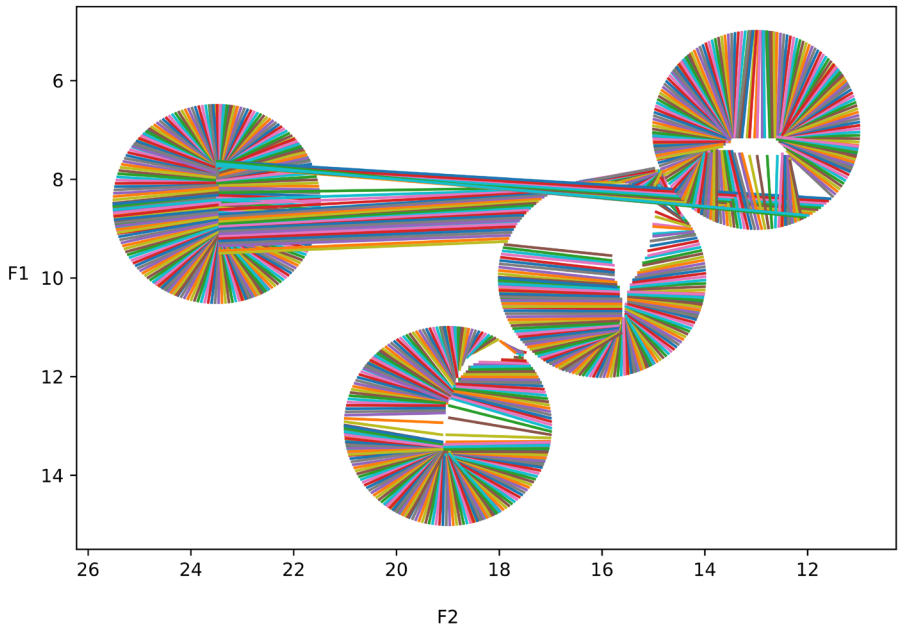

Figure 10. Drift lines, four-vowel system, 2000 steps

The network is performing similarly to when it was trained with five separate vowels. Overall, the drift lines are directed from the circle's edge toward the prototype center. However, the network seems to show a slightly more substantial perceptual magnet effect than when the network is trained with five vowels. This could be caused by a larger distance between the different categories, causing the neighboring inputs to interfere less with the categorization.

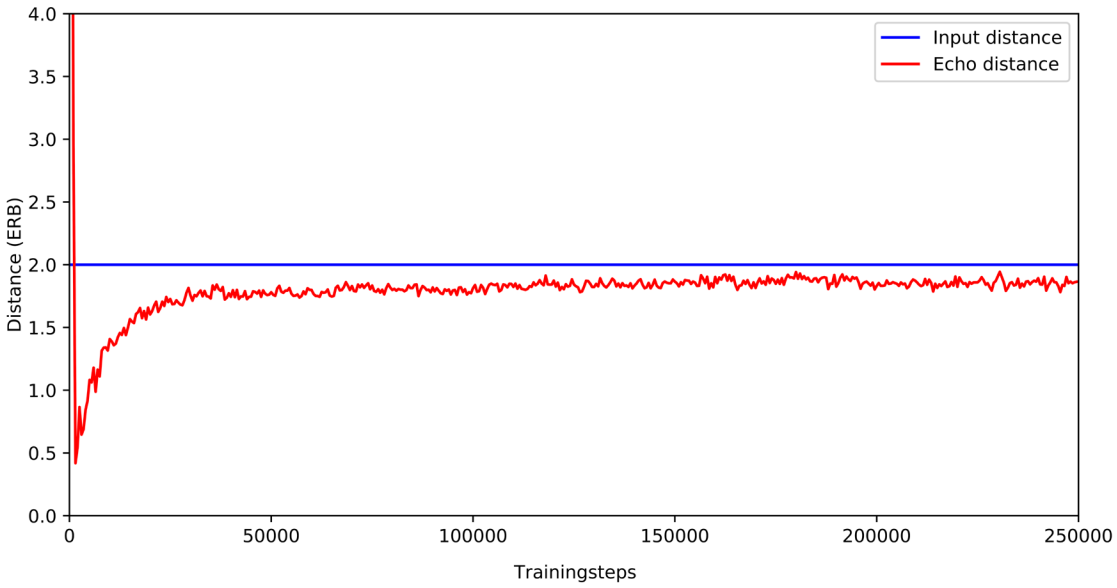

Figure 11. Input and echo distance, four-vowel input

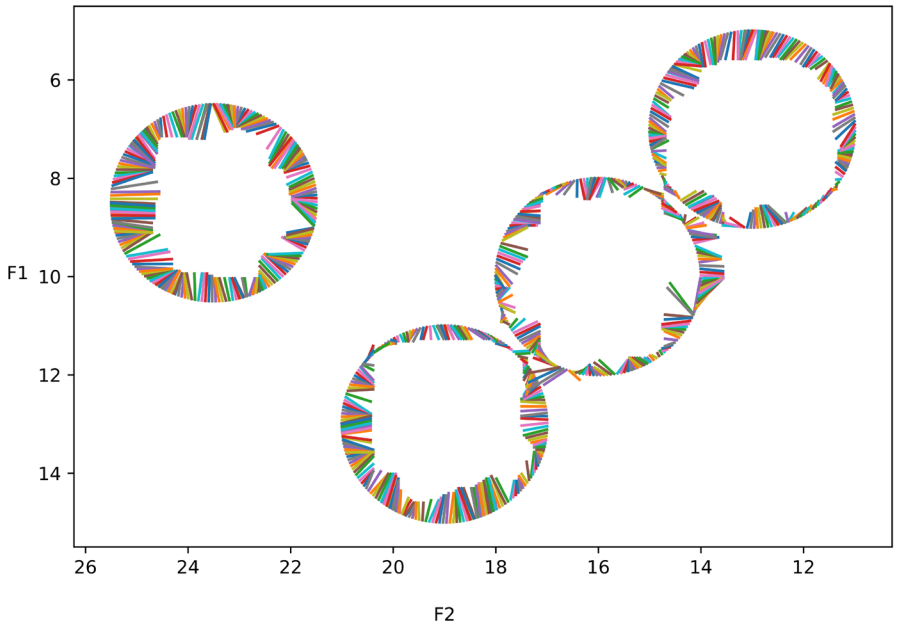

Figure 12. Drift lines, four-vowel system, 50000 steps

Figures 11 and 12 also illustrate similar results to when the network is trained with five vowels. In addition, the figures show how the inputs are returned closer to the prototype. Even over many training steps, the network shows comparable results for the distance values.

Since the network is behaving as expected with a four-vowel input, it is possible to look at the effect of introducing a split after training the system with four vowels up to an equilibrium state (at 50000 steps).

5.3 Network behavior – category split

The system is trained initially with only four vowels and then needs to learn five vowels by splitting one category into two. In this setting, the network has not yet trained with inputs that belong to the new vowel values, /i/ and /e/ but did not belong to /ɪ/. Such inputs would be expected to be assigned either to other existing nearby categories or drift randomly. We would expect a spike in echo distance when the network switches its training data due to the random drifts and assignments.

Figures 13, 14, and 15 give the drift lines at 50000, 52000, and 100000 steps (in total). In figure 13, the network has not yet been trained with the new vowel system, but its behavior is tested when the new vowels are inserted. We can see that the unaffected vowels (/a/, /o/, and /u/) are treated identically as in figure 12.

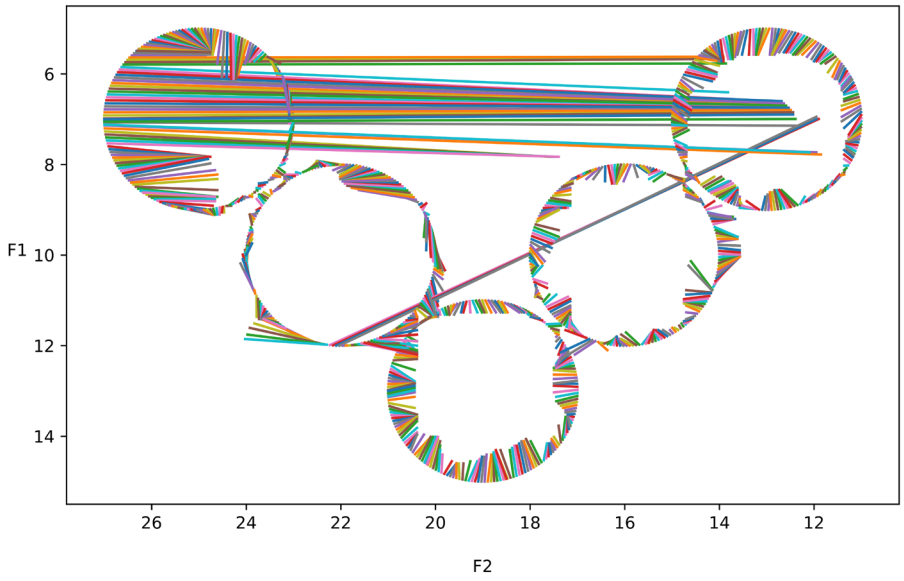

Figure 13. Drift lines, 4-vowel to 5-vowel split, 0 (+50000) steps

In figure 13, where the system has not yet trained with the new data, the inputs belonging to the new vowels are returned as closer to the originally trained set. Many of the /i/-inputs are even returned as if belonging to a completely different category. This follows expectations, as the network has not yet had a chance to adjust its weights and biases to these new inputs.

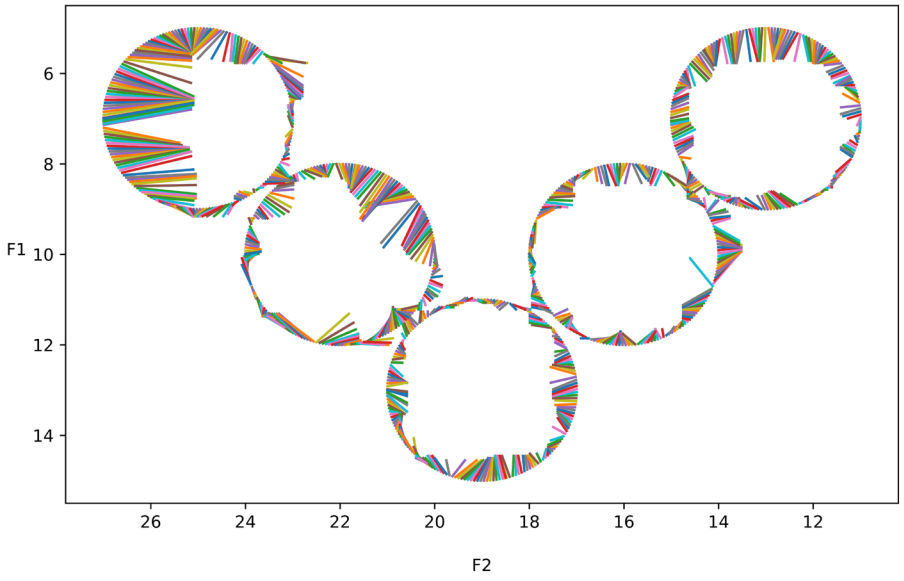

Figure 14. Drift lines, 4-vowel to 5-vowel split, 2000 (+50000) steps

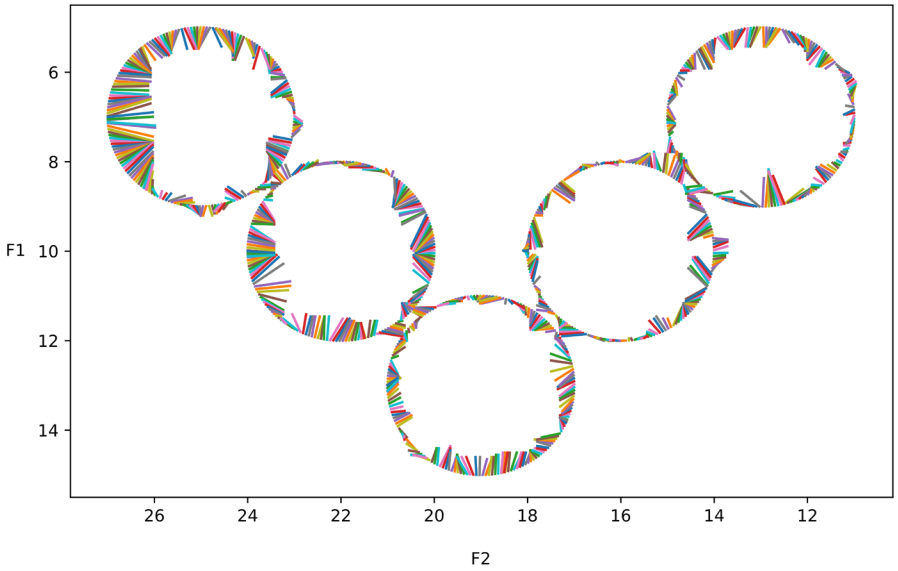

Figure 15. Drift lines, 4-vowel to 5-vowel split, 50000 (+50000) steps

The effect of training becomes evident in figure 13, where after even just 2000 training steps, these 'misinterpretations' no longer appear. In figures 13 and 14, the drift lines again show a preference for a smaller yet dispersed inventory, yet not identical to the native five-vowel network at 50000 steps (figure 9). The main difference between the two figures is the lines that drift away from the prototype center on vowel categories that remained the same after the split. An example is the inputs at the upper edge of /o/. These inputs are returned further away from the center, which is not the case in figure 9.

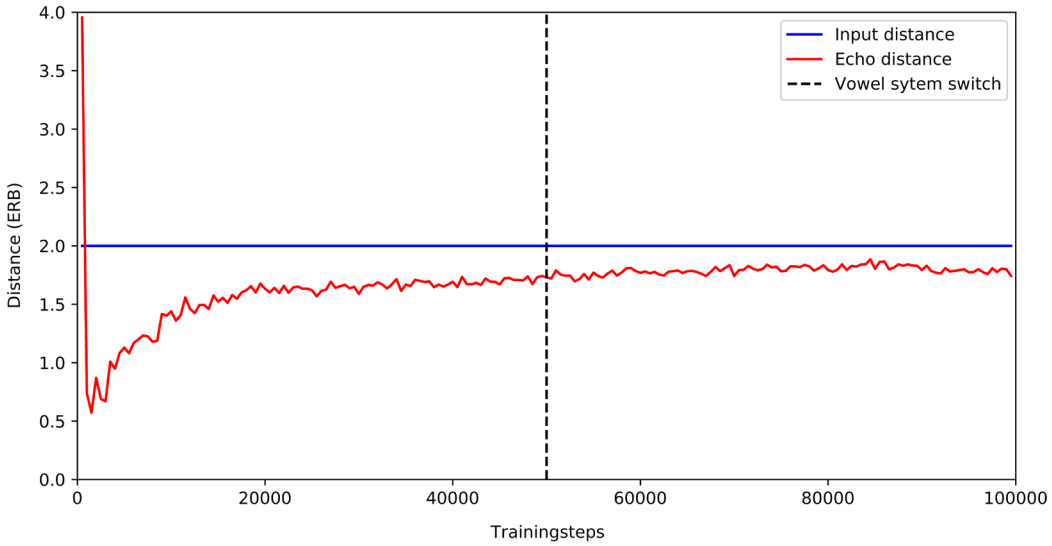

Figure 16. Input and echo distance, 4-vowel to 5-vowel split

Figure 15 shows that the distance to the prototype center remains stable even over many training steps. Interestingly, the figure does not show a spike in distance after switching the input

system. The category jumps in figure 12, where /i/ inputs are returned as belonging to /u/ or the larger lines pointing toward the old /ɪ/, do not seem to affect the average distance.[4] This could mean these large drifts are infrequent. It could also be that the unaffected vowels (/u/, /o/, and /a/) hide any changes by forming a stable average. By ignoring these vowels, it is possible to inspect the distance values for only the affected vowels. Figure 17 first shows the distances for only /ɪ/ inputs and, after the switch, distances for both /i/ and /e/.

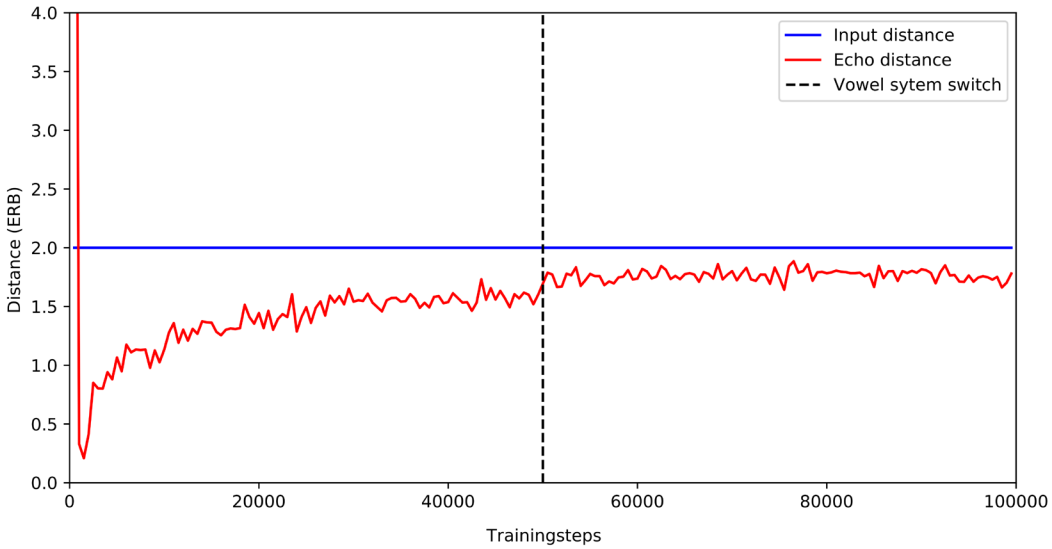

Figure 17. Input and echo distance, 4-vowel to 5-vowel split, relevant vowels

The figure shows a very slight change in distance when looking at only the affected vowels. As this change may be incidental or insignificant, it is necessary to repeat the calculation numerous times and compare the results to the same calculations done with a five-vowel trained network.

Thus, the average echo distance for /i/ and /e/ inputs were tested with a solely five-vowel trained network and a four-vowel trained network at 50000 steps. The significance was tested by repeating the test 200 times, simulating 200 learners for each vowel system (totaling 400 learners). The echo distance of vowels /i/ and /e/ in a network trained with five vowels [$M = 1.910$, $SD = 0.048$] was higher than the distance when training the network with four vowels [$M = 1.896$, $SD = 0.043$], as the independent-sample t-test showed ($p = 0.002$).

[4] The distance calculations were also done while taking the mean difference instead of the median distance since a mean is more easily affected by such extreme values. However, a similar lack in change was observed and thus the paper continues with using the median for reasons explained in footnote 2

This slight difference can be explained by looking at figure 12. There we see the drift lines when a four-vowel trained network is tested with five vowels. In addition, we see large lines on the left side of /i/ that drift toward the original /ɪ/ prototype center, causing the average distance to the new categories to be smaller.

5.4 Network behavior – category merger

We have now only seen how the network deals when we introduce a split, separating one existing category into two. This created inputs that the network had not yet been trained with. When we initiate a merger, melding two existing categories into one, we ask the network to adjust the categorization of inputs it is already familiar with. By reversing the order of vowel system inputs, we can simulate a merger. In this section, the same steps are taken to investigate the network's behavior. Figures 16, 17, and 18 give the drift lines at 50000, 52000, and 100000 steps (in total).

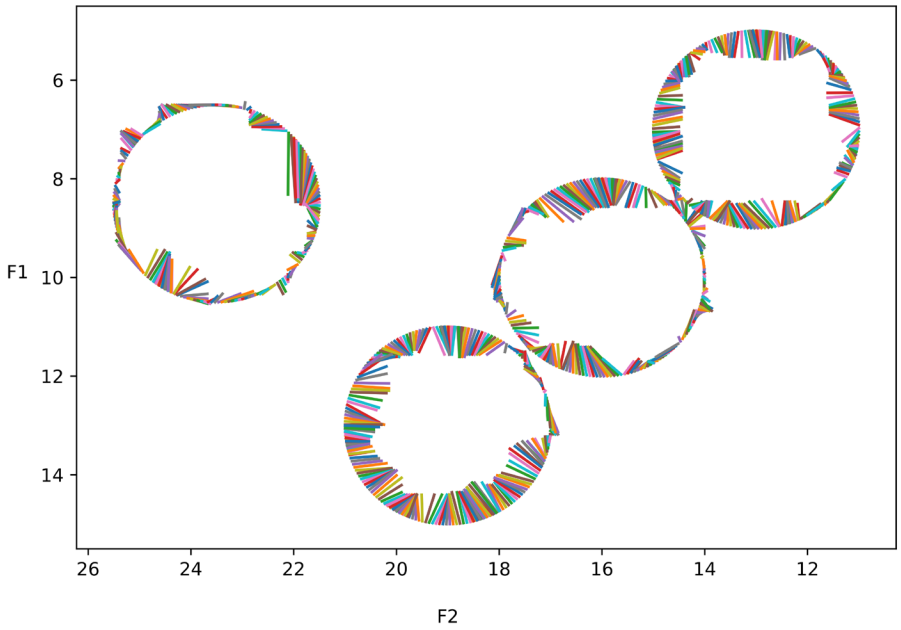

Figure 18. Drift lines, 5-vowel to 4-vowel merger – 50000 steps

In figure 18, we can immediately see a substantial difference compared to the split in the previous section. Unlike figure 12, no category jumps from the newly created category inputs /ɪ/ are present, and the overall drift is shorter. A straightforward explanation is that many new inputs would not need to drift much to fit into the original categories. Inputs that show the most drift in the new category are the ones on the left-bottom and the right-top of /ɪ/. These drift diagonally toward the original prototype centers, as they would have been very peripheral of the original distribution and would thus have the need to drift to fit into the categories.

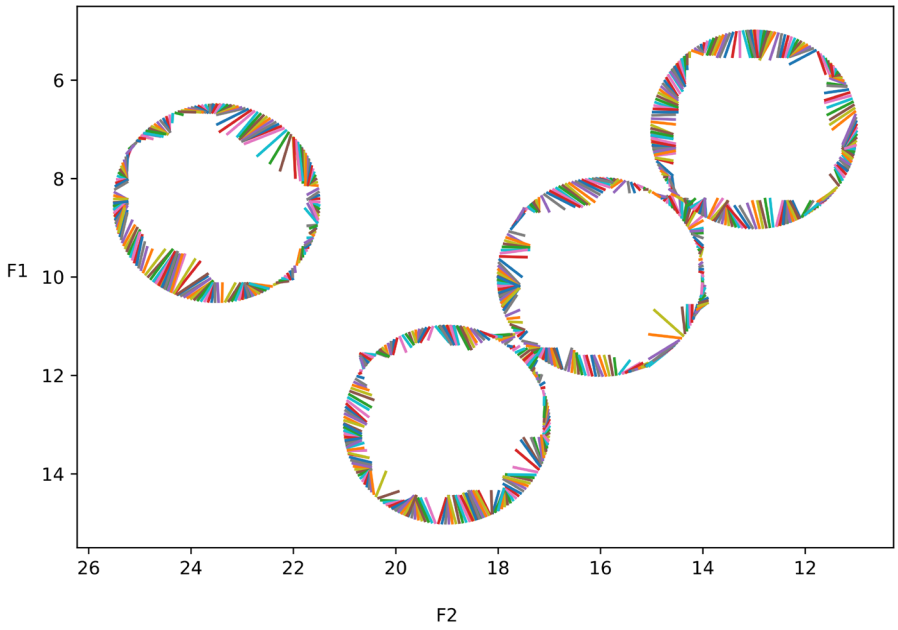

Figure 19. Drift lines, 5-vowel to 4-vowel merger – 52000 steps

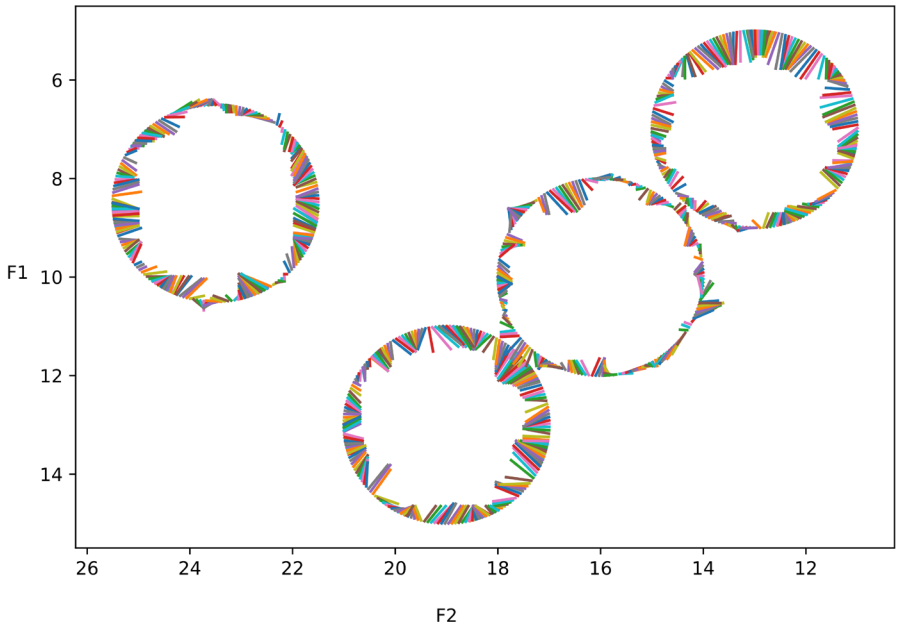

Figure 20. Drift lines, 5-vowel to 4-vowel merger, 100000 steps

Figure 19 starts showing slight categorization for /ɪ/, compared to 16, as the lines now all point more toward the new merged category center. Finally, in figure 20, the drift lines round /ɪ/ are slightly shorter, showing the same trend as at the beginning of the network's training.

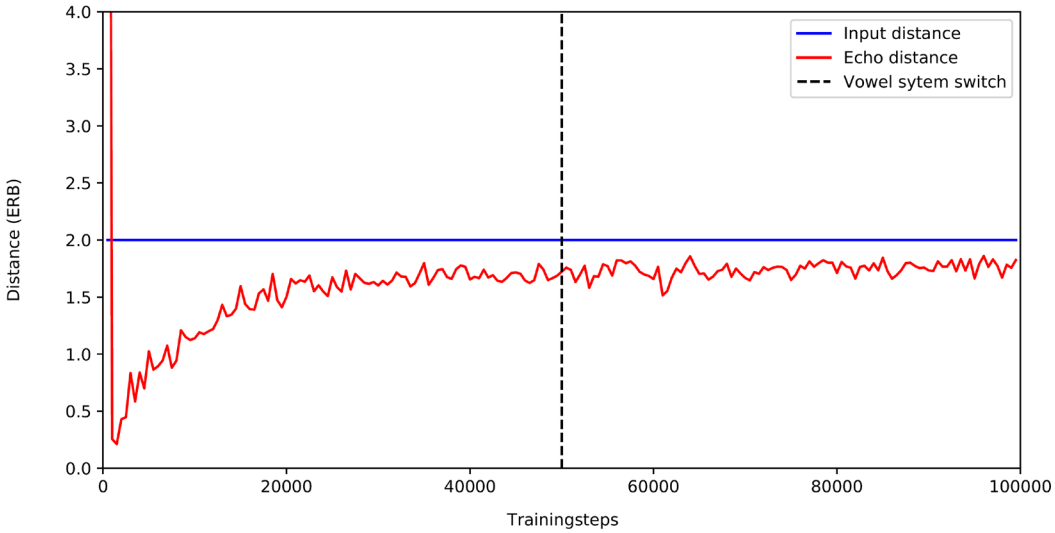

Figure 21. Echo and input distance, 5-vowel to 4-vowel merger

Figure 21 investigates whether the network shows changes after a longer training period after training with the new vowel system. Similar to figure 17, it only shows the distances for the vowels that change (/i/ and /e/, and /ɪ/). However, the network seems to have immediately reached an equilibrium state. Furthermore, it shows no spikes in drift like it does at the start of the network's learning (when all weights and biases are still 0). Statistics show that the echo distance of vowel /ɪ/ in a network trained with four vowels [$M = 1.843$, $SD = 0.063$] was lower than the distance when training the network with five vowels [$M = 1.984$, $SD = 0.048$], as the independent-sample t-test showed ($p < 0.001$).

When looking back at figure 18, the top left inputs of /ɪ/ drift toward the old categories centers of /i/ and /e/. This outward drift causes the distance of a five-vowel trained network tested with four vowels to be higher than when the network is trained with four vowels. Then the drift lines point toward the /ɪ/ prototype.

6 Native Category Retention in BiPhon-NN

The third level at which the network can be compared to human behavior is the retention of original categories. This chapter investigates to what degree the native categories are maintained after full submersion into the second language.

6.1 Retention after a split

For the first scenario, the network is trained with the four-vowel inventory. After 50000 steps, the split occurs (see chapter 5.3), and after another 50000 steps, the network switches back to the initial inventory.

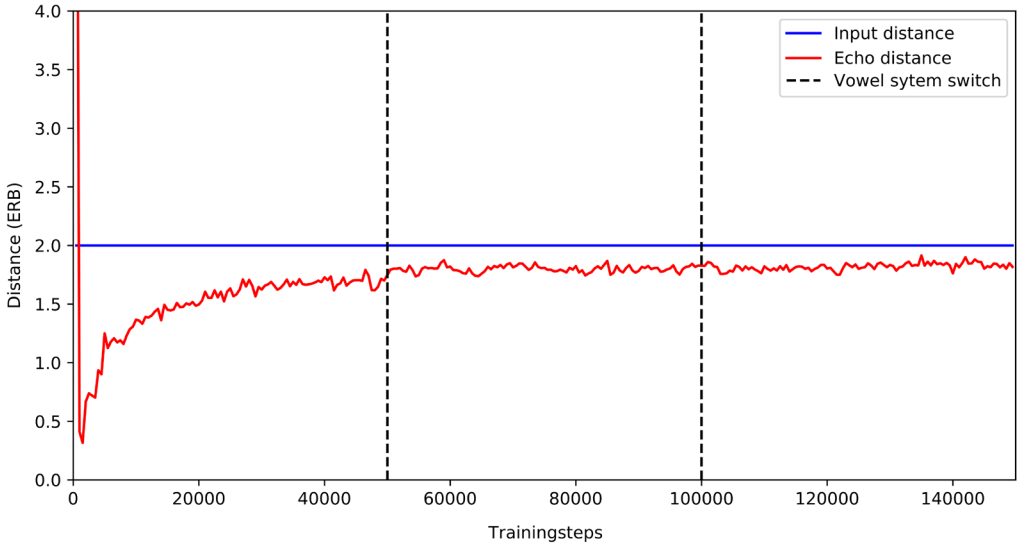

Figure 22. Echo and input distance, 4-vowel to 5-vowel to 4-vowel

Figure 22 shows little to no change in overall distance values. This indicates that the network has no problem with the switch back to the original input. Further details cannot be taken from this figure; hence we have to inspect the drift lines. Figure 23 shows the drift lines of the original 4-vowel inputs. At this time, the network has started its training with the four vowels, then split to the 5-vowel system and trained that for 0 steps.

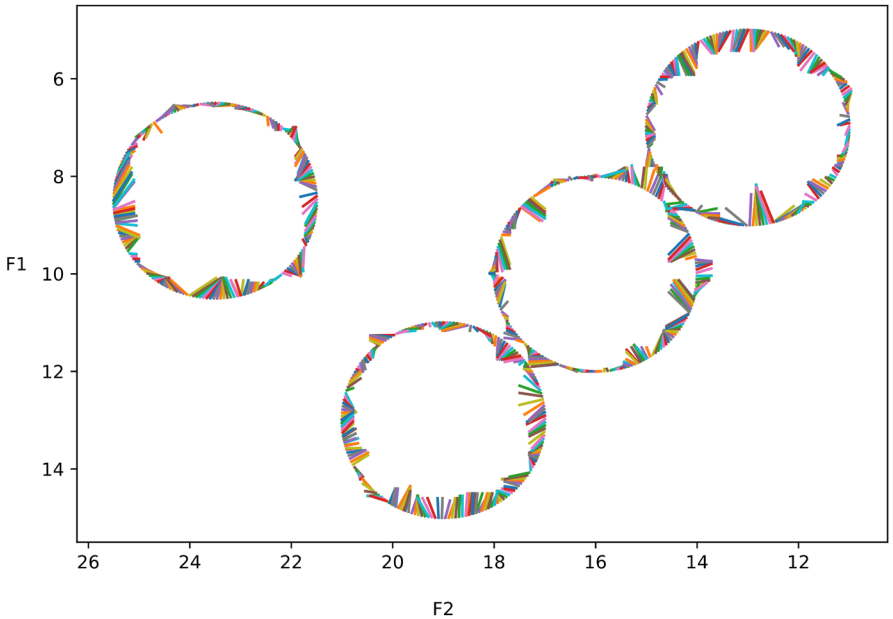

Figure 23. Drift lines, return to 4-vowel after split, 0 steps

The drift lines around the native /ɪ/ do not generally drift toward from the prototype center of /ɪ/, but either toward the second language categories /i/ and /e/ or barely drift at all. This suggests that the network has 'unlearned' to assign these values to the native /ɪ/ category. Continuing to

train the network with the native inventory results in it re-categorizing the inputs reasonably quickly, as shown in figures 24 and 25.

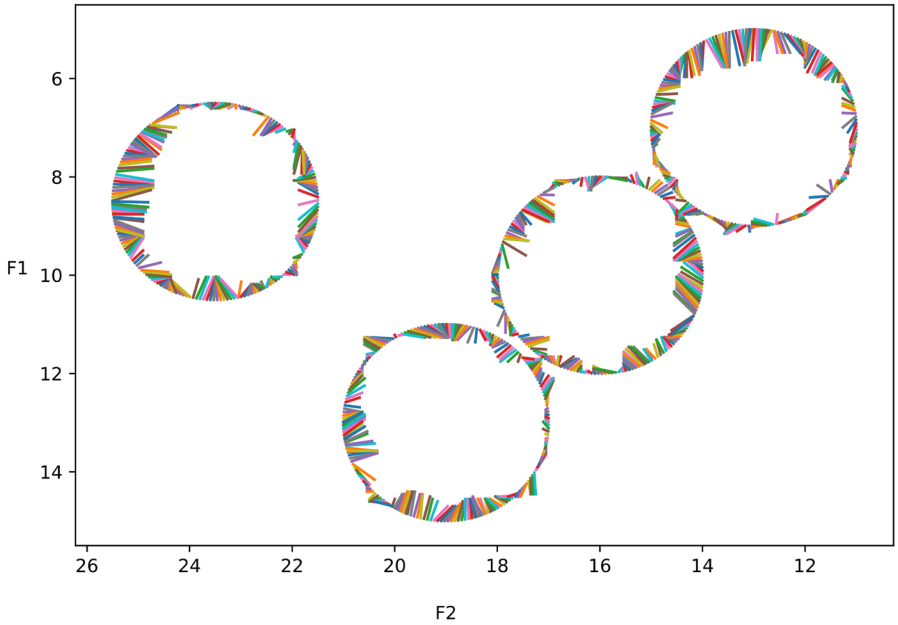

Figure 24. Drift lines, return to 4-vowel after split, 2000 steps

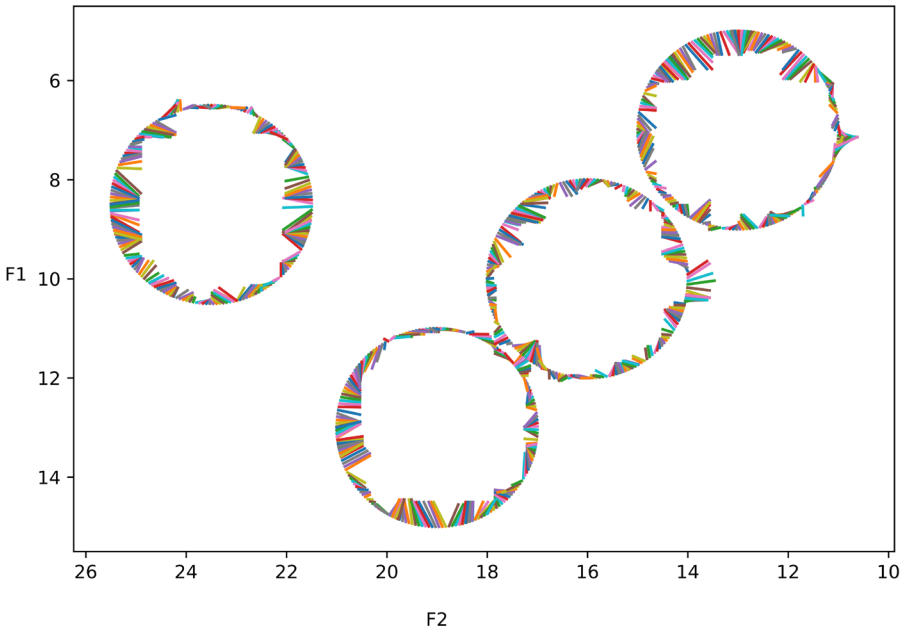

Figure 25. Drift lines, return to 4-vowel after split, 50000 steps

In the two figures, we see the network initially reverting to assign inputs from the native language to the native language categories as its training continues. To test whether the original categories are retained, the distance to /ɪ/ was measured at 100000 steps with a network that was only trained with four vowels (like a monolingual speaker). This was then compared to the distance

in a network initially trained with four vowels, then continued to train with five vowels (similar to the adopted children in Ventureyra, Pallier & Yoo 2004).

The echo distance of vowel /ɪ/ in a fully four-vowel trained network [M = 1.825 , SD = 0.062] was lower than the distance when training the network with four and then five vowels [M = 1.908 , SD = 0.064], as the independent-sample t-test showed ($p < 0.001$). This result can be interpreted as the same 'unlearning' of the native categories, as seen in figure 22.

6.2 Retention after a merger

In the second scenario, we look at the retention of native categories after a merger. This can be meaningful since there might occur more unlearning than with a split, as the more peripheral native /i/ inputs are not inserted during the second language training. The relevant connection for /i/ inputs will likely be weakened more than other connections.

Figure 26 shows the distances for the vowels undergoing changes throughout the network's learning.

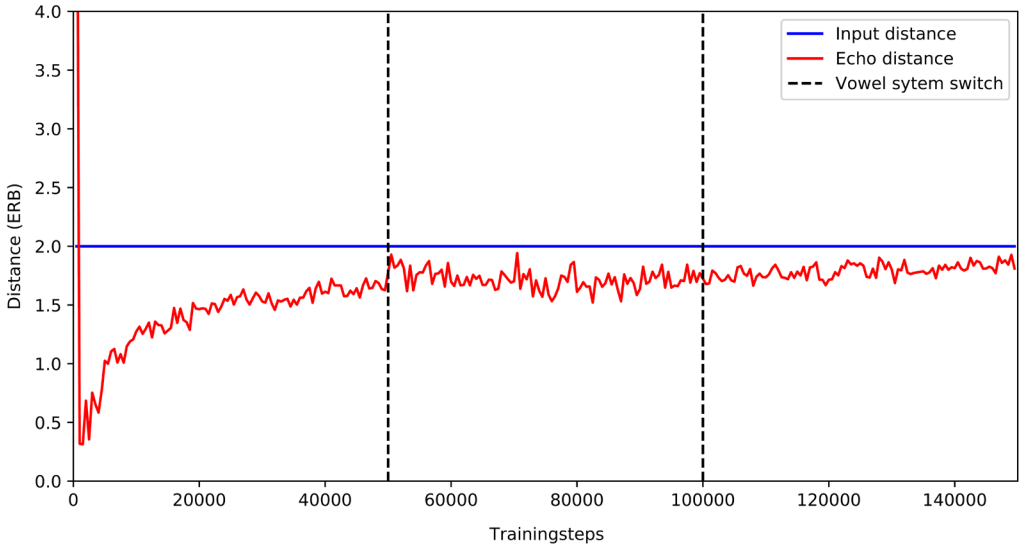

Figure 26. Echo and input distance, 5-vowel to 4-vowel to 5-vowel

Once again, any change in distance caused by the vowel system switch is not apparent via this graph. Since the critical details are likely lost, we turn to the visuals showing the drift lines. The following three figures show the network's behavior after it has trained with its native system for 50000 steps, following 50000 steps in the second vowel system. Figure 27 shows the behavior immediately after, figure 28 after 2000 steps of training in the native system, and finally, figure 29 after another 50000 steps.

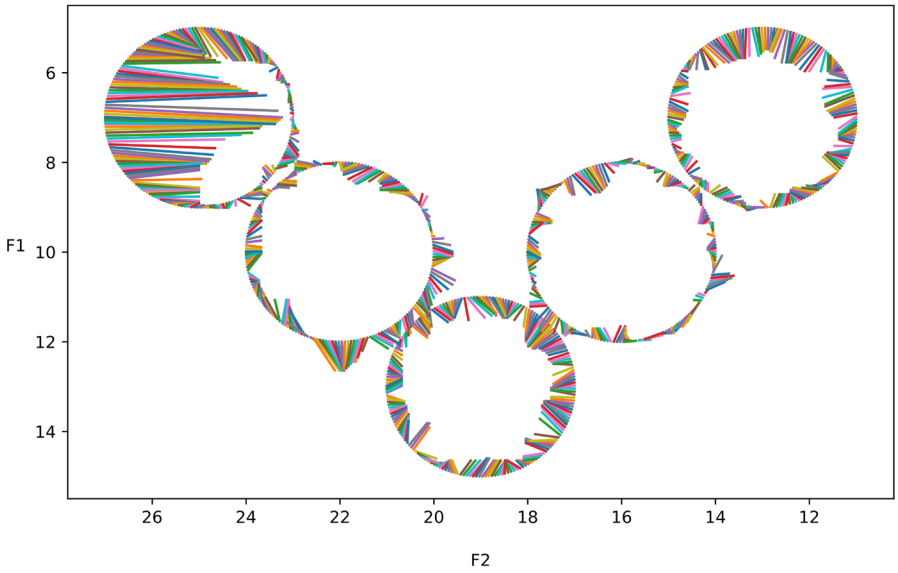

Figure 27. Drift lines, return to 5-vowel after merger, 0 steps

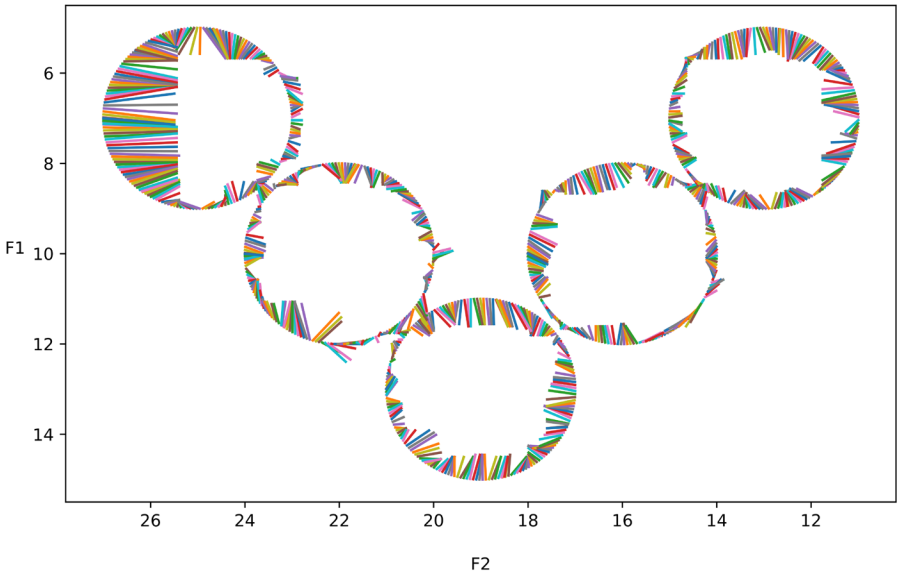

Figure 28. Drift lines, return to 5-vowel after merger, 2000 steps

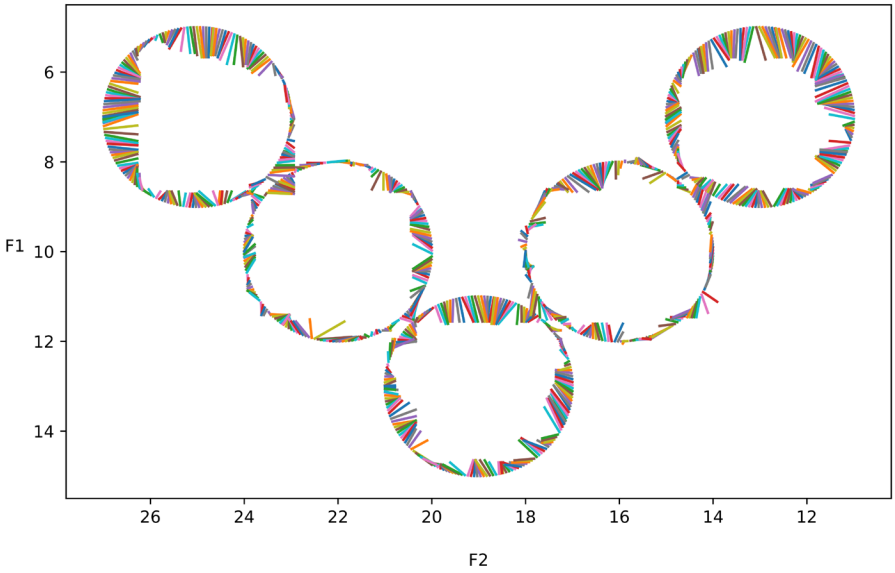

Figure 29. Drift lines, return to 5-vowel after merger, 50000 steps

The figures all look very similar to those made when the network undergoes a split (without prior training). From these figures alone, it seems like the network does indeed unlearn the native categories that are then re-learned. This is supported by comparing the distance values of the native categories of a solely native inventory-trained network to one that has switched to the second language. The echo distance of vowels /i/ and /e/ in a fully five-vowel trained network [$M = 1.966$, $SD = 0.064$] was greater than the distance when training the network with five and then four vowels [$M = 1.925$, $SD = 0.060$], as the independent-sample t-test showed ($p < 0.001$). Although the echo distance here is greater, it still shows a category loss as explained in chapter 5.3.[5]

7 Discussion

This paper attempted to deliver further insights into the validity of BiPhon-NN as a model of human phonological behavior. Boersma et al. (2021) showed that in BiPhon-NN, the emergence of phonological categories is based on phonetic and semantic similarities between utterances. The present paper further explored the validity of the network by comparing it to multiple human studies that focus on phonological behavior via the perceptual magnet effect, phonetic distinction in second language learners, and native category retention. This chapter will discuss the findings in each of these comparisons and suggest further considerations and research.

[5] It is counterintuitive that between a merger or a split the difference in distance infer different conclusions. This is another example why the present's paper's representation of PME might not be ideal.

7.1 Perceptual magnet effect

Chapter 4 was built upon the foundation laid by Boersma et al. (2021), where they showed that the activation on the auditory layer returned by the network varied from the input activation. In a trained network, the activation peaks (interpreted as formants) of a returned activation were closer to the mean prototype value the network was trained with than peripheral inputs. Similar results were found in the present paper, from where further analysis was done over a larger number of training steps. What was seen here is that the average distance of 0.5 ERB from the prototype at approximately 2000 training steps[6] increases to a stable distance between 1.75 and 1.9 ERB. Though slightly, the network shows that the inputs are returned more category inward.

This result could be interpreted as the network showing a slight perceptual magnet effect, where peripheral inputs are perceived as more category-central than the actual input. The perceptual magnet effect also states that category-central sounds are perceived as being the same vowel. However, the network does not return identical activations for all inputs within one vowel category. This means the network does not process these inputs as if being the same.

To further analyze the categorical behavior, the present paper turned to phonetic distinction by human second language learners. However, future research might need to focus on this problem. By comparing testing inputs that lie within trained distributions to inputs that lie outside of the training distributions, the results could better reflect the original definition by Kuhl (1991).

Whether a small echo distance in the stable state of the BiPhon-NN represents the PME remains unclear. As the adult perceptual magnet effect cannot easily be attributed to a pure perceptual cause, it is hard to identify whether the relatively low distance values are comparable to the adult PME. Future research might want to explore the effect of a semantic layer in BiPhon-NN on distance values over many training steps. Suppose the categorization is affected by the presence of semantic information. In that case, this could support the view that adult phonological categorization is not purely phonetic but that the perception of speech sounds depends on inferring category membership using exemplars stored in memory (or in this network possibly the hidden layers) (Lacerda 1995).

---

[6] This change might be explained that the network adequately models the perceptual magnet effect in children. However, potentially unlike children, it continues to categorize purely based on auditory perception, whereas adult might start using semantic data or the emerged phonological categories.

7.2 Categorical flexibility

By introducing a new vowel system to a network that has already been trained, the network could be compared to a human second language learner. Humans can adjust existing categories created by their native language to adhere to the contrasts found in the new language. The network shows a similar category inward movement of auditory layer activations after training with the new vowel system. Thus, the network can adjust its weights and biases to ensure the new inputs are returned as if belonging to new categories.

When recreating a split, creating two categories from auditory values that initially represented the same vowel, we can see that the network first assigns new inputs to the old category. This allocation follows existing human studies where participants have trouble perceiving a difference between two vowels in the new language when these represent only one vowel in their native language (Ylinen, 2010; McAllister, 2007;). In BiPhon-NN, this 'confusion' is short-lived, as, after only 2000 steps, the largest deviants had disappeared, and the network had started to assign the inputs to the new categories. Future research might be wise to inspect if the learning speed of the second language categorization is compatible with human findings.

When simulating a merger of categories, the network shows some initial confusion where the inputs still drift towards the native vowel categories. The confusion is less apparent than with a split, as the native vowel category enveloped most of the inputs of the new vowel system. The network could adequately mirror human experience as the loss of contrast is empirically easier to deal with than creation.

Following the Speech Learning Model (Flege, 1995), second language learners will have increasingly more difficulty perceiving phonetic contrasts the more similar the involved sounds are to their native language. Learning completely novel contrasts would then be easier. In the present paper, only the first scenario was modeled using Biphon-NN. The two 'languages' the network was trained with shared three vowels, and the remaining vowel(s) shared formant values. The network had to learn the new contrast but did not show any difficulty in doing so. The difference in difficulty as proposed by SLM could be investigated by letting the network learn a completely new vowel that is separated from all existing vowels. Caution should be exerted when deciding on such a new vowel. If the new vowel shared formant values with other vowels, this might affect the learnability. Situations as those explained in chapter 5.3, where /u/ and /a/ were confused due to overlapping first and second formant values, could distort results if not taken into account.

The difficulty of learning new phonetic or phonological contrasts does not only depend on the native and the target language. It also depends on the type of contrast needed to be learned. Sensitivity to perceptual differences can also be found within different native or non-native sounds (Best & Tyler 2007). For example, McAllister (2007) saw a relative inability of Swedish learners to attain the English /s/-/z/ contrast. Instead, they started to utilize several strategies to distinguish the sounds. Compare this to Ylinen's study explained in chapter 2.2, where the participants achieved spectral distinction between the non-native sounds /i/ and /ɪ/. The voicing contrast between /s/ and /z/ might be more challenging to attain than a difference in formant value between /i/ and /ɪ/. To further explore the network's validity, it might be necessary first to expand the network to process more sounds (or more different features like duration or pitch). This would make it possible to explore how different types of contrast can be attained at varying levels of success.

Up to this point in the paper, the distance between the echo activation and the prototype was used to portray the categorization of the network over time (see figure 8). This chapter makes it apparent that the distance calculation might not show small yet essential details of the network's behavior. Drift lines that indicate a level of confusion do not show up in the echo distance graphs. The values plotted in such graphs represent the average distance between the prototype mean and the echo. Even echoes that were closer to or further from the prototype were lost in this calculation. Going forward, a better representation of 'drift over time' would need to be found. Overall, the current calculation gives a decent initial intuition; however, for detailed inspection, one needs to refer to the visual drift lines (like figure 7).

7.3 Native Category Retention

The human capability for maintaining native language phonetic contrasts was found in studies investigating the performance of adopted children. Two main distinctions were made in these studies. First, they looked at children whose native language had similarities with their adopted language or continued to be moderately exposed to their native language. Second, children who had to learn a vastly different language or were entirely separated from their native language.

The first situation was simulated in BiPhon-NN, and the results were comparable though not identical. The network was initially trained in the vowel system with four vowels. Then it was introduced to a five vowel system where one of the native vowels was split into two new categories. At the end of two sets of 50000 training steps, the network was inserted with the values from the native vowel distributions. The echoes the network returned for these vowels did not immediately

drift toward the native prototype values. The inability to instantly assign the native sounds to their native categories follows the assumption that the native categories have degraded (Werker & Tees 1981). However, after little training in the native system, the echoes showed activations closer to the native language prototypes. Furthermore, the speed at which the network seems to stabilize when returning to the native system suggests the original categories were not fully unlearned. The network's visual state at 2000 steps more resembled a stable solely 4-vowel trained network (at 50000 steps) than one at 2000 steps (figures 7, 9, and 28).

The network's seeming incapability of fully retaining older categories seems compatible with Best's Perceptual Assimilation Model (Best 1994; Best 1995). This model proposes that second language sound contrasts are classified into new native or non-native categories when the new sounds are perceived as speech sounds. Whether the new sound can be assimilated into the existing language depends on whether they are categorized as native or non-native. As the present paper uses second language sounds that can be assimilated, the network might not create a new category. This could explain why the network needs training in the native language after being trained in a second language before assigning native sounds correctly.

8 Conclusion

By looking at various human studies on first and second language acquisition and comparing those to the behavior of the neural network model BiPhon-NN (Boersma, 2021), this paper attempts to assess the validity of the network. The findings are three-part. First, the bidirectional network shows a discrepancy between perceived inputs and returned echoes. Whether this discrepancy can be seen as an apparent perceptual magnet effect remains unclear. However, it is a sound measurement of phonetic categorization when one assumes that categories emerge through distributional learning. By looking at second language training, it seems as though the network adheres to human behavior by adjusting existing categories to assign a second language's vowel inventory correctly. Native category retention is then used to show the network does not create a new set of categories for each language but shifts existing ones to fit the non-native sounds.

References

Alexandridis, A., L. Russo, D. Vakalis, G. V. Bafas & C. I. Siettos (2011): Wildland fire spread modelling using cellular automata: evolution in large-scale spatially heterogeneous environments under fire suppression tactics. *International Journal of Wildland Fire* 20(5). 633. doi:10.1071/WF09119.

Au, Terry Kit-fong, Leah M. Knightly, Sun-Ah Jun & Janet S. Oh (2002): Overhearing a Language During Childhood. *Psychological Science* 13(3). 238–243. doi:10.1111/1467-9280.00444.

Bendicenti, Erminia, Salvatore Di Gregorio, Francesco M. Falbo & Angela Iezzi (2002): Simulations of Forest Fires by Cellular Automata Modelling. In Gianfranco Minati & Eliano Pessa (Hrsg.), *Emergence in Complex, Cognitive, Social, and Biological Systems*, 31–40. Boston, MA: Springer US. doi:10.1007/978-1-4615-0753-6_3.

Best, C. T (1995): A direct realist view of cross-language speech perception. *Speech Perception and Linguistic Experience*. York Press. 171–206.

Best, Catherine T. (1994): The emergence of native-language phonological influences in infants: A perceptual assimilation model. *The development of speech perception: The transition from speech sounds to spoken words* 167(224). 233–277.

Best, Catherine T. & Michael D. Tyler (2007): Nonnative and second-language speech perception: Commonalities and complementarities. In Ocke-Schwen Bohn & Murray J. Munro (Hrsg.), *Language Learning & Language Teaching*, vol. 17, 13–34. Amsterdam: John Benjamins Publishing Company. doi:10.1075/lllt.17.07bes.

Boersma, Paul (2011): A programme for bidirectional phonology and phonetics and their acquisition and evolution. In Anton Benz & Jason Mattausch (Hrsg.), *Linguistik Aktuell/Linguistics Today*, vol. 180, 33–72. Amsterdam: John Benjamins Publishing Company. doi:10.1075/la.180.02boe.

Boersma, Paul (2019): Simulated distributional learning in deep Boltzmann machines leads to the emergence of discrete categories. 5.

Boersma, Paul, Titia Benders & Klaas Seinhorst (2020): Neural network models for phonology and phonetics. *Journal of Language Modelling* 8(1) doi:10.15398/jlm.v8i1.224.

Boersma, Paul, Kateřina Chládková & Titia Benders (2021): Phonological features emerge substance-freely from the phonetics and the morphology. 50.

Ghisu, Tiziano, Bachisio Arca, Grazia Pellizzaro & Pierpaolo Duce (2015): An Improved Cellular Automata for Wildfire Spread. *Procedia Computer Science* 51. 2287–2296. doi:10.1016/j.procs.2015.05.388.

Guenther, Frank H. & Marin N. Gjaja (1996): The perceptual magnet effect as an emergent property of neural map formation. *The Journal of the Acoustical Society of America* 100(2). 1111–1121. doi:10.1121/1.416296.

Kuhl, Patricia K. (1991): Human adults and human infants show a "perceptual magnet effect" for the prototypes of speech categories, monkeys do not. *Perception & Psychophysics* 50(2). 93–107. doi:10.3758/BF03212211.

Kuhl, Patricia K (1995): Mechanisms of developmental change in speech and language. *Proceedings of the XIIIth international congress of phonetic sciences*, vol. 2, 132–139.

Lacerda, Francisco (1995): The perceptual-magnet effect: An emergent consequence of exemplar-based phonetic memory. *Proceedings of the XIIIth international congress of phonetic sciences*, vol. 2, 140–147. Stockholm University Stockholm.

Liljencrants, Johan & Björn Lindblom (1972): Numerical Simulation of Vowel Quality Systems: The Role of Perceptual Contrast. *Language* 48(4). 839. doi:10.2307/411991.

McAllister, Robert (2007): Strategies for Realization of L2-Categories: English /s/ — /z/. In Ocke-Schwen Bohn & Murray J. Munro (Hrsg.), *Language Learning & Language Teaching*, vol. 17, 153–166. Amsterdam: John Benjamins Publishing Company. doi:10.1075/lllt.17.16mca.

Nicoladis, Elena & Howard Grabois (2002): Learning English and losing Chinese: A case study of a child adopted from China. *International Journal of Bilingualism* 6(4). 441–454. doi:10.1177/13670069020060040401.

Oh, Janet S, Sun-Ah Jun, Leah M Knightly & Terry Kit-fong Au (2003): Holding on to childhood language memory. 12.

Polka, Linda (1995): Developmental patterns in infant speech perception. *Proceedings of the XIIIth International Congress of Phonetic Sciences*, vol. 2, 148–155.

Ventureyra, Valérie A.G, Christophe Pallier & Hi-Yon Yoo (2004): The loss of first language phonetic perception in adopted Koreans. *Journal of Neurolinguistics* 17(1). 79–91. doi:10.1016/S0911-6044(03)00053-8.

Werker, Janet F., John H. V. Gilbert, Keith Humphrey & Richard C. Tees (1981): Developmental Aspects of Cross-Language Speech Perception. *Child Development* 52(1). 349. doi:10.2307/1129249.

Ylinen, Sari, Maria Uther, Antti Latvala, Sara Vepsäläinen, Paul Iverson, Reiko Akahane-Yamada & Risto Näätänen (2010): Training the Brain to Weight Speech Cues Differently: A Study of Finnish Second-language Users of English. *Journal of Cognitive Neuroscience* 22(6). 1319–1332. doi:10.1162/jocn.2009.21272.

Appendices

1 BiPhon-NN - dRBM and learning codes

1.1 Main network code

```
# import my own codes
from learning_phases import one_learning_step
from setup_input import input_vowels, network_setup
from graphs import show_full_network
# import standard packages
import matplotlib.pyplot as plt

# setup the network and set the number of trainingsteps
new_nodes_connections = network_setup()
training_steps = 2000

# training steps
for _ in range(training_steps):
    new_nodes_connections[0] = input_vowels(new_nodes_connections[0], 'five')
    new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])

# show (and save) graph
graph = show_full_network(new_nodes_connections[0], new_nodes_connections[1])
plt.tight_layout()
graph.set_size_inches(9.2, 5)
graph.savefig('pdfs/network_figure.pdf', dpi=600)
plt.show()
```

1.2 Network setup

```
#########################################################
# Create all nodes and connections using numpy arrays #
#########################################################
# import standard packages
import numpy as np

def network_setup():
    """ This function returns an list containing all nodes and connections
            and their activation, bias, weights and coordinates

    Parameters:
    ----------


    Returns:
    --------
    nodes_connections: an list containing two lists for nodes and connections
            each with an array of data for each layer

    """

    # For each nodelayer, create four values per node
    inp_nodes = np.full((4,49), 0.0)
    mid_nodes = np.full((4,50), 0.0)
```

```
    top_nodes = np.full((4,20), 0.0)

    # for each node set the final two values as the coordinates (for graphing
purposes)
    for i in range(49):
        inp_nodes[2,i] = i+4    # x-axis
        inp_nodes[3,i] = 2      # y-axis
    for i in range(50):
        mid_nodes[2,i] = i*1+3.5
        mid_nodes[3,i] = 5
    for i in range(20):
        top_nodes[2,i] = i*2.3+6
        top_nodes[3,i] = 8

    # combine all node arrays into one list
    all_nodes = [inp_nodes, mid_nodes, top_nodes]

    # Setup the connections for all nodes in a matrix
    inp_mid_connections = np.zeros((49,50))
    mid_top_connections = np.zeros((50,20))

    # Also combine in a list
    all_connections = [inp_mid_connections, mid_top_connections]

    # Combine all nodes and connections in another list
    nodes_connections = [all_nodes, all_connections]

    return nodes_connections
```

1.3 Input vowels

```
# import standard packages
import random
import numpy as np

def input_vowels(new_nodes, vowel_system):
    """ This function chooses a random input sound and changes the
        activations on the input layer accordingly

        Parameters:
        ----------
        new_nodes: an array containing the node activations and biases
        vowel_system: a string stating the needed amount of vowels

        Returns:
        --------
        new_nodes: an array containing the node activations and biases

    """
    # Set up parameters
    erb = np.linspace(4,28,49)
    bump_width = 0.68

    # Choose vowel system and pick a random sound from the inventory
    if vowel_system == 'five':
        sounds = ['a', 'e', 'i', 'o', 'u']
    else:
```

```
        sounds = ['a', 'ei', 'o', 'u']
    step_input = random.choice(sounds)

    # Find the formant values dependent on the picked vowel
    if step_input == 'a':
        f1 = np.random.normal(13,1)
        f2 = np.random.normal(19,1)
    elif step_input == 'e':
        f1 = np.random.normal(10,1)
        f2 = np.random.normal(22,1)
    elif step_input == 'ei':
        f1 = np.random.normal(8.5,1)
        f2 = np.random.normal(23.5,1)
    elif step_input == 'i':
        f1 = np.random.normal(7,1)
        f2 = np.random.normal(25,1)
    elif step_input == 'o':
        f1 = np.random.normal(10,1)
        f2 = np.random.normal(16,1)
    elif step_input == 'u':
        f1 = np.random.normal(7,1)
        f2 = np.random.normal(13,1)

    # Calculate the activities of the node on the auditory layer
    activity = 5.0 * (np.exp(-(erb-f1)**2/(2*bump_width**2)) + np.exp(-(erb-
f2)**2/(2*bump_width**2))) - 0.5

    # Set the new activitions for all layers (input decides erb, rest is 0.5)
    new_nodes[0][0,:] = activity
    new_nodes[1][0,:] = 0.0
    new_nodes[2][0,:] = 0.0

    return new_nodes
```

1.4 One learning step

```
#### The full learning step
# import my own codes
from learning_phases.phases.dreaming import spread_to_inp
from .phases import settling_phase, hebbian_learning_nodes,
hebbian_learning_connections, dreaming_phase
from .phases import anti_hebbian_learning_nodes,
anti_hebbian_learning_connections

def one_learning_step(new_nodes, new_connections):
    """ This function goes through all learning steps and adjusts
        the network's values. finally it returns the entire network
        of nodes and connections new values

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases

    new_connections: an array containing all weights for all connections


    Returns:
```

```
    --------
    new_nodes_connections: a list containing the node array and
        connections array, all with the new values after one entire
        training step

    """
    # Settling phase
    new_nodes = settling_phase(new_nodes, new_connections)

    # Hebbian learning phase
    new_nodes = hebbian_learning_nodes(new_nodes, 0.001)
    new_connections = hebbian_learning_connections(new_nodes,
new_connections, 0.001)

    # Dreaming phase
    new_nodes = dreaming_phase(new_nodes, new_connections)

    # Anti-Hebbian learning
    new_nodes = anti_hebbian_learning_nodes(new_nodes)
    new_connections = anti_hebbian_learning_connections(new_nodes,
new_connections)

    # Combine new nodes and connections data and return
    new_nodes_connections = [new_nodes, new_connections]

    return new_nodes_connections
```

1.5 Settling phase

```
############################## Settling Phase ################################
# import standard packages
import numpy as np

def settling_phase(new_nodes, new_connections):
    """ This function calculates the activation on the top and middle layer
        after spreading from the other layers, 10 times

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases

    new_connections: an array containing all weights for all connections


    Returns:
    --------
    new_nodes_connections: a list containing the node array and, all with
        the new values after one spreading activations

    """
    # Spread the activation 10 times
    for _ in range(9):
        new_nodes = middle_spread(new_nodes, new_connections)
        new_nodes = top_spread(new_nodes, new_connections)

    return new_nodes
```

```
def middle_spread(new_nodes, new_connections):
    """ This function calculates the activation on the middle layer
        after spreading from the input and top layers

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases

    new_connections: an array containing all weights for all connections


    Returns:
    --------
    new_nodes_connections: a list containing the node array and, all with
        the new values after one spreading activations

    """
    # bottomup spreading
    inp_act_weight = np.matmul(new_nodes[0][0,:],new_connections[0])

    # top down spreading
    top_act_weight = np.matmul(new_connections[1],new_nodes[2][0,:])

    # store new activations
    new_nodes[1][0,:] = np.random.binomial(1, 1 / (1 + np.exp(-
(new_nodes[1][1,:] + inp_act_weight + top_act_weight))))

    return new_nodes


def top_spread(new_nodes, new_connections):
    """ This function calculates the activation on the top layer
        after spreading from the middle layers

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases

    new_connections: an array containing all weights for all connections


    Returns:
    --------
    new_nodes_connections: a list containing the node array and, all with
        the new values after one spreading activations

    """
    # bottomup spreading
    mid_act_weight = np.matmul(new_nodes[1][0,:],new_connections[1])

    # store new activations
    new_nodes[2][0,:] = np.random.binomial(1, 1 / (1 + np.exp(-
(new_nodes[2][1,:] + mid_act_weight))))

    return new_nodes
```

1.6 Hebbian Learning

```
########################## Hebbian Learning ##############################
# Calculate node training
def hebbian_learning_nodes(new_nodes, lr):
    """ This function calculates the bias changes of all nodes

    Parameters:
    ----------
    new_nodes: a dictionary containing all nodes in the network
        with their coordinates, activation and bias values
    lr: a float value for the learning rate

    Returns:
    --------
    nodes_connections: a list of two dictionaries of all nodes and
        connections with updates values biases and weights respectively

    """
    #bias + (lr * act)
    new_nodes[0][1,:] += lr * new_nodes[0][0,:]
    new_nodes[1][1,:] += lr * new_nodes[1][0,:]
    new_nodes[2][1,:] += lr * new_nodes[2][0,:]

    return new_nodes

# Calculate connection training
def hebbian_learning_connections(new_nodes, new_connections, lr):
    """ This function calculates the bias changes of all connections

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections
    lr: a float value for the learning rate

    Returns:
    --------
    new_connections: an array containing all weights for all connections

    """
    # weight + (lr * act_a * act_b)
    for i, row in enumerate(new_connections[0]):
        new_connections[0][i,:] = new_connections[0][i,:] + lr *
new_nodes[0][0,i] * new_nodes[1][0,:]
    for i, row in enumerate(new_connections[1]):
        new_connections[1][i,:] = new_connections[1][i,:] + lr *
new_nodes[1][0,i] * new_nodes[2][0,:]

    return new_connections
```

1.7 Dreaming Phase

```
######################## Dreaming Phase ##############################
# import standard packages
import numpy as np

def dreaming_phase(network_nodes, network_connections):
```

```
    """ This function calculates the new random binary activation on the
        middle and top layers after spreading from the other layers, 10 times

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections


    Returns:
    --------
    new_nodes: an array containing the node activations and biases

    """
    new_nodes = network_nodes

    for _ in range(9):
        new_nodes = spread_to_inp(new_nodes, network_connections)
        new_nodes = bernoulli_top_spread(new_nodes, network_connections)
        new_nodes = bernoulli_mid_spread(new_nodes, network_connections)

    return new_nodes

def spread_to_inp(new_nodes, new_connections):
    """ This function calculates the activation on the input layer nodes
        after spreading from the middle layer

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections

    Returns:
    --------
    new_nodes: an array containing the node activations and biases

    """
    mid_act_weight = np.matmul(new_connections[0],new_nodes[1][0,:])
    new_nodes[0][0,:] = new_nodes[0][1,:] + mid_act_weight



    return new_nodes

def bernoulli_top_spread(new_nodes, new_connections):
    """ This function calculates the new random binary activation on the top
layer
        nodes after spreading from the middle layer

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections

    Returns:
    --------
    new_nodes: an array containing the node activations and biases
```

```
    """
    # Loop through all nodes on middle layer
        # bottomup
    mid_act_weight = np.matmul(new_nodes[1][0,:],new_connections[1])

    # store new with bernoulli randomness
    sigmoided_exci = 1 / (1 + np.exp(-(new_nodes[2][1,:] + mid_act_weight)))
    new_nodes[2][0,:] = np.random.binomial(1, sigmoided_exci)

    return new_nodes

def bernoulli_mid_spread(new_nodes, new_connections):
    """ This function calculates the new random binary activation on the
        middle layer nodes after spreading from the bottom and top layer

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections

    Returns:
    --------
    new_nodes: an array containing the node activations and biases

    """
    # bottomup
    inp_act_weight = np.matmul(new_nodes[0][0,:],new_connections[0])

    # top down
    top_act_weight = np.matmul(new_connections[1],new_nodes[2][0,:])

    # store new
    sigmoided_exci = 1 / (1 + np.exp(-(new_nodes[1][1,:] + inp_act_weight +
top_act_weight)))
    new_nodes[1][0,:] = np.random.binomial(1, sigmoided_exci)

    return new_nodes
```

1.8 Anti-Hebbian Learning

```
###################### Anti-Hebbian Learning ############################
def anti_hebbian_learning_nodes(new_nodes):
    """ This function calculates the bias changes
        of all nodes by unlearning the dreaming phase

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases

    Returns:
    --------
    new_nodes: an array containing the node activations and biases

    """
    # Bias changes
    new_nodes[0][1,:] -= 0.001 * new_nodes[0][0,:]
    new_nodes[1][1,:] -= 0.001 * new_nodes[1][0,:]
```

```
    new_nodes[2][1,:] -= 0.001 * new_nodes[2][0,:]

    return new_nodes

def anti_hebbian_learning_connections(new_nodes, new_connections):
    """ This function calculates the weight changes
        of all  connections by unlearning the dreaming phase

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections

    Returns:
    --------
    new_connections: an array containing all weights for all connections

    """
    # Weight changes
    for i, row in enumerate(new_connections[0]):
        new_connections[0][i,:] = new_connections[0][i,:] - 0.001 *
new_nodes[0][0,i] * new_nodes[1][0,:]
    for i, row in enumerate(new_connections[1]):
        new_connections[1][i,:] = new_connections[1][i,:] - 0.001 *
new_nodes[1][0,i] * new_nodes[2][0,:]

    return new_connections
```

2 Creating the visual drift lines

2.1 Main code

This code contains parameters the user can alter to their wishes

```
# import my own codes
from learning_phases import one_learning_step
from setup_input import input_vowels, network_setup
from echoes import visual_echo
# import standard packages
import matplotlib.pyplot as plt
#########################################################################
##############  setup parameters to show the visual echoes  ##############
steps = 50000
vowels = 'four'           # initial vowel system
initial_intsteps = 2000   # steps for interval for visual echo display
change_vowels = 'five'    # if split/merge, set to opposite of vowels
split_merge = 'split'     # enter: 'split' or 'merge' or 'none4' or 'none5'
split_merge_steps= 50000     # if no split or merger, set to 0
split_merge_intsteps = 2000 # steps for interval for visual echo display
input_type = 'edge'       # state if you'd like random or cat-edge inputs
returner = True
#########################################################################
# setup the network
new_nodes_connections = network_setup()

# run first round of training
```

```
for i in range(steps):
    new_nodes_connections[0] = input_vowels(new_nodes_connections[0], vowels)
    new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])

    # save a figure of the drift lines at the interval step-count
    if i == initial_intsteps:
            echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], vowels, input_type)
            plt.tight_layout()
            echo_drift_nodes.set_size_inches(8, 5)
            echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge + '_' +
str(+i) +'.pdf', dpi=600)

# save a figure of the drift lines after first training round
echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], vowels, input_type)
plt.tight_layout()
echo_drift_nodes.set_size_inches(8, 5)
echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge + '_' + str(steps)
+'.pdf', dpi=600)

# run second training for split or merger test
if split_merge == 'split' or split_merge == 'merge':

    # save a figure of the drift lines with L2 before training L2
    echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], change_vowels, input_type)
    plt.tight_layout()
    echo_drift_nodes.set_size_inches(8, 5)
    echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge + '_immediate_'
+ str(steps) +'.pdf', dpi=600)

    # start training in L2
    for i in range(split_merge_steps):
        new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
change_vowels)
        new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])

        # save a figure of the drift lines at the interval step-count
        if i == split_merge_intsteps:
            echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], change_vowels, input_type)
            plt.tight_layout()
            echo_drift_nodes.set_size_inches(8, 5)
            echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge + '_' +
str(steps+i) +'.pdf', dpi=600)

    # save a figure of the drift values for the final state of L2
    echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], change_vowels, input_type)
    plt.tight_layout()
    echo_drift_nodes.set_size_inches(8, 5)
    echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge + '_' +
str(steps+split_merge_steps) +'.pdf', dpi=600)

# Do a third round of training by returning to L1
```

```
if returner == True:

    # save a figure of the drift lines with L1 before re-training L1
    echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], vowels, input_type)
    plt.tight_layout()
    echo_drift_nodes.set_size_inches(8, 5)
    echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge
+'_returner_immediate.pdf', dpi=600)

    # start L1 training
    for i in range(50000):
        new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
vowels)
        new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])

        # save a figure of the drift values at the interval step-count
        if i == split_merge_intsteps:
            echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], vowels, input_type)
            plt.tight_layout()
            echo_drift_nodes.set_size_inches(8, 5)
            echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge
+'_returner_2000.pdf', dpi=600)

    # save a figure of the drift values for the final state
    echo_drift_nodes = visual_echo(new_nodes_connections[0],
new_nodes_connections[1], vowels, input_type)
    plt.tight_layout()
    echo_drift_nodes.set_size_inches(8, 5)
    echo_drift_nodes.savefig('pdfs/visualecho_'+ split_merge
+'_returner_500000.pdf', dpi=600)
```

2.2 Visual echo (drift lines)

```
# import my own codes
from .echo_tools import echo_peaks, echo_spread, edge_input_echo,
random_input_echo
# import standard packages
import numpy as np
import matplotlib.pyplot as plt

def visual_echo(new_nodes, new_connections, vowel_system, input_method):
    """ This function creates a list of coordinates of the inputs and
    echoes of 1000 inputs (echo coordinates are found through local maxima)

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections
    vowel_system: a string stating the vowelsystem to be used
    input_method: a string stating a random input or an edge input


    Returns:
    --------
```

```
    [av_inp_dist, av_echo_dist]: a list containing median distance to the
        prototype values from the input coordinates and the echo coordinates
    """
    # set up coordinate lists and the erb values
    drifts_coord_x = []
    drifts_coord_y = []
    erb = np.linspace(4,28,49)

    # ensure correct activations on mid and top layer
    new_nodes[1][0,:] = 0.5
    new_nodes[2][0,:] = 0.5

    # gather 1000 echoes, 200 per prototype
    for i in range(1000):
        if input_method == 'edge':
            f1f2_act = edge_input_echo(vowel_system, i)
        else: # if input is required to be completely random
            f1f2_act = random_input_echo(vowel_system)
        new_nodes[0][0,:] = f1f2_act[2]
        f1 = f1f2_act[0]
        f2 = f1f2_act[1]

        # per input, let activation spread through system
        new_nodes = echo_spread(new_nodes, new_connections)

        # Interpolate the heights to get more refined peaks
        echo_f1f2 = echo_peaks(erb, new_nodes)
        echo_f1 = echo_f1f2[0]
        echo_f2 = echo_f1f2[1]
        # append x and y values to the drift lists
        drifts_coord_x.append((f2, echo_f2))
        drifts_coord_y.append((f1, echo_f1))

    # set new lists to return
    drift_coords = [drifts_coord_x, drifts_coord_y]

    # Set up the figure and plot data into graph
    fig, ax = plt.subplots()
    for i, coord in enumerate(drift_coords[0]):
        ax.plot(drift_coords[0][i], drift_coords[1][i], '-')

    # invert axes and add axis titles
    ax.invert_xaxis()
    ax.invert_yaxis()
    ax.text(29.5,10, 'F1', ha='center')
    ax.text(19,17, 'F2', ha='center')

    # Store and return the graph
    graph = fig
    return graph
```

2.3 Input echo (edge or random)

```
# import standard package
import random
import numpy as np
from math import pi, cos, sin
```

```
# For inputs that are at a set distance from prototype
def edge_input_echo(vowel_system, step):
    """ This function chooses an input sound from the category edge and
changes
        the activations on the input layer accordingly

        Parameters:
        ----------
        vowel_system: a string of 'four' or 'five' (or +'_spec') that states
            the used vowel system
        step: a integer containing the number of the current input


        Returns:
        --------
        new_nodes: an array containing the node activations and biases

    """
    erb = np.linspace(4,28,49)
    bump_width = 0.68
    radius = 2

    if vowel_system == 'five':
        angle = 1.8 * (step % 200)

        if step <= 200: # 'a'
            f1 = sin(angle)*radius + 13
            f2 = cos(angle)*radius + 19
            proto_f1 = 13
            proto_f2 = 19
        if 200 < step <= 400: # 'e'
            f1 = sin(angle)*radius + 10
            f2 = cos(angle)*radius + 22
            proto_f1 = 10
            proto_f2 = 22
        if 400 < step <= 600: # 'i':
            f1 = sin(angle)*radius + 7
            f2 = cos(angle)*radius + 25
            proto_f1 = 7
            proto_f2 = 25
        if 600 < step <= 800: # 'o':
            f1 = sin(angle)*radius + 10
            f2 = cos(angle)*radius + 16
            proto_f1 = 10
            proto_f2 = 16
        if 800 < step <= 1000: # 'u':
            f1 = sin(angle)*radius + 7
            f2 = cos(angle)*radius + 13
            proto_f1 = 7
            proto_f2 = 13

    elif vowel_system == 'four':
        angle = 1.8 * (step % 250)
        if step <= 250: # 'a'
            f1 = sin(angle)*radius + 13
            f2 = cos(angle)*radius + 19
            proto_f1 = 13
```

```
            proto_f2 = 19
        if 250 < step <= 500: # 'ei'
            f1 = sin(angle)*radius + 8.5
            f2 = cos(angle)*radius + 23.5
            proto_f1 = 8.5
            proto_f2 = 23.5

        if 500 < step <= 750: # 'o':
            f1 = sin(angle)*radius + 10
            f2 = cos(angle)*radius + 16
            proto_f1 = 10
            proto_f2 = 16

        if 750 < step <= 1000: # 'u':
            f1 = sin(angle)*radius + 7
            f2 = cos(angle)*radius + 13
            proto_f1 = 7
            proto_f2 = 13

    # the following two inputs are only used in the distance tests and are
random
    elif vowel_system == 'four_spec':
        angle = random.random() * 2 * pi
        f1 = sin(angle)*radius + 8.5
        f2 = cos(angle)*radius + 23.5
        proto_f1 = 8.5
        proto_f2 = 23.5
    elif vowel_system == 'five_spec':
        sounds = ['a', 'i']
        angle = random.random() * 2 * pi
        step_input = random.choice(sounds)
        if step_input == 'e':
            f1 = sin(angle)*radius + 10
            f2 = cos(angle)*radius + 22
            proto_f1 = 10
            proto_f2 = 22
        elif step_input == 'i':
            f1 = sin(angle)*radius + 7
            f2 = cos(angle)*radius + 25
            proto_f1 = 7
            proto_f2 = 25

    # calculate the activation of all nodes on the auditory layer
    activity = 5.0 * (np.exp(-(erb-f1)**2/(2*bump_width**2)) + np.exp(-(erb-
f2)**2/(2*bump_width**2))) - 0.5

    # combine and return all needed variables
    f1f2_act = [f1,f2,activity, proto_f1, proto_f2]
    return f1f2_act


def random_input_echo(vowel_system):
    """ This function chooses an random input sound and changes the
activations
            on the input layer accordingly

        Parameters:
        ----------
```

```
        new_nodes: an array containing the node activations and biases
        vowel_system: a string of 'four' or 'five' that states the used vowel
            system

        Returns:
        --------
        new_nodes: an array containing the node activations and biases

    """
    erb = np.linspace(4,28,49)
    bump_width = 0.68

    if vowel_system == 'five':
        sounds = ['a', 'e', 'i', 'o', 'u']
        step_input = random.choice(sounds)
    elif vowel_system == 'four':
        sounds = ['a', 'ei', 'o', 'u']
        step_input = random.choice(sounds)
    elif vowel_system == 'four_spec':
        sounds = ['ei']
        step_input = 'ei'
    elif vowel_system == 'five_spec':
        sounds = ['a', 'i']
        step_input = random.choice(sounds)

    if step_input == 'a':
        f1 = np.random.normal(13,0.9)
        f2 = np.random.normal(19,0.9)
        proto_f1 = 13
        proto_f2 = 19

    elif step_input == 'e':
        f1 = np.random.normal(10,0.9)
        f2 = np.random.normal(22,0.9)
        proto_f1 = 10
        proto_f2 = 22

    elif step_input == 'i':
        f1 = np.random.normal(7,0.9)
        f2 = np.random.normal(25,0.9)
        proto_f1 = 7
        proto_f2 = 25

    elif step_input == 'o':
        f1 = np.random.normal(10,0.9)
        f2 = np.random.normal(16,0.9)
        proto_f1 = 10
        proto_f2 = 16

    elif step_input == 'u':
        f1 = np.random.normal(7,0.9)
        f2 = np.random.normal(13,0.9)
        proto_f1 = 7
        proto_f2 = 13

    activity = 5.0 * (np.exp(-(erb-f1)**2/(2*bump_width**2)) + np.exp(-(erb-
f2)**2/(2*bump_width**2))) - 0.5
    f1f2_act = [f1,f2,activity, proto_f1, proto_f2]
```

```
    return f1f2_act
```

2.4 Echo spread – spreading activations for echoes

```
# import standard packages
import numpy as np

def echo_spread(new_nodes, new_connections):
    """ This function spreads the activation given through the network
    from bottom to middle to the top, back to the middle (x10) and then back
    to the bottom. This activation final state it then returns

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections

    Returns:
    --------
    new_nodes: an array containing the node activations and biases
    """
    # spread to middle (while top = 0)
    inp_act_weight = np.matmul(new_nodes[0][0,:],new_connections[0])
    top_act_weight = np.matmul(new_connections[1],new_nodes[2][0,:])
    new_nodes[1][0,:] = 1 / (1 + np.exp(-(new_nodes[1][1,:] + inp_act_weight
+ top_act_weight)))

    # echo calculation (from middle to top and bottom, then back to middle)
    # 10 times
    for _ in range(9):
        # top layer
        mid_act_weight = np.matmul(new_nodes[1][0,:],new_connections[1])
        new_nodes[2][0,:] = 1 / (1 + np.exp(-(new_nodes[2][1,:] +
mid_act_weight)))
        # spread back to erb-layer
        mid_act_weight = np.matmul(new_connections[0],new_nodes[1][0,:])
        new_nodes[0][0,:] = new_nodes[0][1] + mid_act_weight
        # back to mid layer
        inp_act_weight = np.matmul(new_nodes[0][0,:],new_connections[0])
        top_act_weight = np.matmul(new_connections[1],new_nodes[2][0,:])
        new_nodes[1][0,:] = 1 / (1 + np.exp(-(new_nodes[1][1,:] +
inp_act_weight + top_act_weight)))

    return new_nodes
```

2.5 Echo peaks – find echo formants

```
# import standard packages
from scipy.signal import find_peaks
from scipy import interpolate
import numpy as np

def echo_peaks(erb, new_nodes):
    """ This function finds the two highest activations in the ERB layer.

    Parameters:
    ----------
```

```
    erb:
    new_nodes: an array containing the node activations and biases

    Returns:
    --------
    echo_f1f2: a list containing the two highest activaded erb values
    """
    # Interpolate the heights to get more refined peaks
    f = interpolate.interp1d(erb, new_nodes[0][0,:], kind='cubic')
    new_erb = np.linspace(4,28,490)
    new_heights = f(new_erb)

    # get all local maxima
    local_maxima_ind = find_peaks(new_heights, height=-4)
    # create lists for height values and erb index
    heights = local_maxima_ind[1]['peak_heights']
    indexes = local_maxima_ind[0]

    # sort both lists to get two maxima heights and erb at the end of the
list
    zipped = zip(heights, indexes)
    sorted_pairs = sorted(zipped)
    tuples = zip(*sorted_pairs)
    act_height, index_erb = [ list(tuple) for tuple in  tuples]

    # get two highest peaks and sort by index
    highest_acts = [index_erb[-1], index_erb[-2]]
    highest_acts = sorted(highest_acts)
    # get erb values for two highest maxima
    echo_f1 = new_erb[highest_acts[0]]
    echo_f2 = new_erb[highest_acts[1]]

    echo_f1f2 = [echo_f1, echo_f2]

    return echo_f1f2
```

3 Graphing Distance-to-prototype

3.1 Main code

This code contains parameters the user can alter to their wishes

```
# import my own codes
from setup_input import input_vowels, network_setup
from learning_phases import one_learning_step
from echoes import echo_drift
# import standard packages
import matplotlib.pyplot as plt
#########################################################################
##########  setup parameters to run the distance calculations  ###########
steps = 50000
original_vowels = 'five'
test_vowel = '_spec'    # if test for specific vowels add: '_spec'
split_merge = 'merge'  # enter: 'split' or 'merge' or 'none4' or 'none5'
split_merge_steps= 50000   # if no split or merger, set to 0
input_type = 'edge'     # state if you'd like random or cat-edge inputs
```

```
returner = True         # true if you want to test original vowel retention
#############################################################################

# setting up some variables
if split_merge == 'split':
    new_vowels = 'five'
elif split_merge == 'merge':
    new_vowels = 'four'

# setup the network
new_nodes_connections = network_setup()

# set up empty lists of drifts to plot
inp_drifts = []
echo_drifts = []
trainingsteps = []

# Train the network in L1 and collect median distances per 500 steps
for i in range(steps):
    new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
original_vowels)
    new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])
    if i > 1 and i%500 == 0:
        echo_drift_nodes = echo_drift(new_nodes_connections[0],
new_nodes_connections[1], 'five_spec', input_type)
        inp_drifts.append(echo_drift_nodes[0])
        echo_drifts.append(echo_drift_nodes[1])
        trainingsteps.append(i)

# Train the network in L2 and collect median distances per 500 steps
if split_merge == 'merge' or split_merge == 'split':
    for i in range(split_merge_steps):
        new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
'four')
        new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])
        if i > 1 and i%500 == 0:
            echo_drift_nodes = echo_drift(new_nodes_connections[0],
new_nodes_connections[1], 'four_spec', input_type)
            inp_drifts.append(echo_drift_nodes[0])
            echo_drifts.append(echo_drift_nodes[1])
            trainingsteps.append(i+split_merge_steps)

# Re-train the network in L1 and collect median distances per 500 steps
if returner == True:
    for i in range(50000):
        new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
original_vowels)
        new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])
        if i > 1 and i%500 == 0:
            echo_drift_nodes = echo_drift(new_nodes_connections[0],
new_nodes_connections[1], 'five_spec', input_type)
            inp_drifts.append(echo_drift_nodes[0])
            echo_drifts.append(echo_drift_nodes[1])
            trainingsteps.append(i+split_merge_steps+50000)
```

```
# Setup the figure and plot the distances
fig, ax = plt.subplots()
ax.plot(trainingsteps, inp_drifts, 'b-',label='Input distance')
ax.plot(trainingsteps,echo_drifts, 'r-',label='Echo distance')
ax.text(-10500,2, 'Distance (ERB)', va='center', rotation=90)

# add vertical dotted lines to show vowel system switch
if split_merge == 'merge' or split_merge == 'split':
    ax.axvline(steps, 0, 4, c='k', ls='--', label='Vowel sytem switch')

# set axis labels and sizes
if returner == True:
    ax.axvline((steps+split_merge_steps), 0, 4, c='k', ls='--')
    ax.text((steps+split_merge_steps+50000)/2,-0.5, 'Trainingsteps',
ha='center')
    plt.axis([0, (steps+split_merge_steps+50000), 0, 4])
else:
    plt.axis([0, (steps+split_merge_steps), 0, 4])
    ax.text((steps+split_merge_steps)/2,-0.5, 'Trainingsteps', ha='center')

# add a legend and save (show) the figure
ax.legend()
fig.set_size_inches(10, 5)
if returner == True:
    fig.savefig('new_pdfs/distance_returner_merge.pdf', dpi=600)
else:
    fig.savefig('new_pdfs/distance_'+ split_merge
+'_'+str(steps+split_merge_steps)+ str(test_vowel) +'.pdf', dpi=600)
plt.show()
```

3.2 Echo drift (median distance)

```
# import my own codes
from .echo_tools import random_input_echo, edge_input_echo, echo_peaks,
echo_spread
# import standard packages
import numpy as np
import math

def echo_drift(new_nodes, new_connections, vowel_system, input_method):
    """ This function calculates the median distance over X inputs from
    the input(and echo) coordinates to the vowel mean formants.

    Parameters:
    ----------
    new_nodes: an array containing the node activations and biases
    new_connections: an array containing all weights for all connections
    vowel_system: a string stating the vowel system to be used
    input_method: a string stating a random input or an edge input


    Returns:
    --------
    [av_inp_dist, av_echo_dist]: a list containing median distance to the
        prototype values from the input coordinates and the echo coordinates

    """
```

```
    # set steps depending on code goal
    if vowel_system == 'five':
        steps = 1000
    elif vowel_system == 'four':
        steps = 800
    elif vowel_system == 'five_spec':
        steps = 400
    elif vowel_system == 'four_spec':
        steps = 200

    # set up coordinate lists and the erb values
    inp_dists = []
    echo_dists = []
    erb = np.linspace(4,28,49)

    # ensure correct activations on mid and top layer
    new_nodes[1][0,:] = 0.5
    new_nodes[2][0,:] = 0.5

    # gather X echoes, 200 per prototype
    for i in range(steps):
        if input_method == 'edge':
            f1f2_act = edge_input_echo(vowel_system)
        else:
            f1f2_act = random_input_echo(vowel_system)

        new_nodes[0][0,:] = f1f2_act[2]
        f1 = f1f2_act[0]
        f2 = f1f2_act[1]
        proto_f1 = f1f2_act[3]
        proto_f2 = f1f2_act[4]

        # per input, let activation spread through system
        new_nodes = echo_spread(new_nodes, new_connections)

        # Find formants of the echo
        echo_f1f2 = echo_peaks(erb, new_nodes)
        echo_f1 = echo_f1f2[0]
        echo_f2 = echo_f1f2[1]

        # get distance from input to prototype, and append
        input_distance = math.sqrt( (f1 - proto_f1)**2 + (f2 - proto_f2)**2 )
        inp_dists.append(input_distance)

        # get distance from echo to prototype, and append
        echo_distance = math.sqrt( (echo_f1 - proto_f1)**2 + (echo_f2 -
proto_f2)**2 )
        echo_dists.append(echo_distance)

    # find the median of all distances
    av_inp_dist = np.median(inp_dists)
    av_echo_dist = np.median(echo_dists)

    return [av_inp_dist, av_echo_dist]
```

4 Codes for statistics

4.1 Testing “monolingual” to merger/split/retention

This code contains parameters the user can alter to their wishes

```
# import my own codes
from setup_input import input_vowels, network_setup
from learning_phases import one_learning_step
from echoes import echo_drift
# import standard packages
import csv
##########################################################
##########  setup parameters to run the test  ############
steps = 50000
original_vowels = 'five'  # L1 vowels
l2 = True           # if you want the network to learn an L2
new_vowels = 'four'      # L2 vowels
test_vowel = 'five'   # set L2(or L1) vowel
split_merge_steps= 50000
##########################################################################
# setup the network
new_nodes_connections = network_setup()

# set up empty lists of drifts to plot
echo_drifts = []

# repeat calculations for virtual 200 learners
for i in range(200):
    # train network in native vowel system
    for _ in range(steps):
        new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
original_vowels)
        new_nodes_connections = one_learning_step(new_nodes_connections[0],
new_nodes_connections[1])

    if l2 == True:
        # train network in L2
        for _ in range(split_merge_steps):
            new_nodes_connections[0] = input_vowels(new_nodes_connections[0],
new_vowels)
            new_nodes_connections =
one_learning_step(new_nodes_connections[0], new_nodes_connections[1])

    # Get only the median distance between input and echo
    echo_drift_nodes = echo_drift(new_nodes_connections[0],
new_nodes_connections[1], test_vowel+'_spec', 'edge')
    echo_drifts.append([str(echo_drift_nodes[1])])

# store data in a csv-file
with open('retention_five_merge.csv', 'w', encoding="ISO-8859-1", newline='')
as myfile:
    wr = csv.writer(myfile)
    wr.writerows(echo_drifts)
```

5 SPSS statistics outputs

5.1 Split test (chapter 5.3)

Group Statistics

	VowelSystem	N	Mean	Std. Deviation	Std. Error Mean
TESTFIVE	Native	200	1.9109	.04844	.00343
	New	200	1.8964	.04315	.00305

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
TESTFIVE	Equal variances assumed	.206	.650	3.170	398	.002	.01454	.00459	.00553	.02356
	Equal variances not assumed			3.170	392.795	.002	.01454	.00459	.00553	.02356

5.2 Merger test (chapter 5.4)

Group Statistics

	VowelSystem	N	Mean	Std. Deviation	Std. Error Mean
TESTFOUR	Native	200	1.8431	.06305	.00446
	New	200	1.9848	.04813	.00340

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
TESTFOUR	Equal variances assumed	12.938	.000	-25.271	398	.000	-.14174	.00561	-.15277	-.13071
	Equal variances not assumed			-25.271	372.161	.000	-.14174	.00561	-.15277	-.13071

5.3 Native retention – after split (chapter 6.1)

Group Statistics

	VOWELINVENTORY	N	Mean	Std. Deviation	Std. Error Mean
RETENTIONFOUR	Native	200	1.82506595	.061897478	.004376813
	NativeRetention	200	1.90796427	.064019094	.004526834

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
RETENTIONFOUR	Equal variances assumed	.997	.319	-13.165	398	.000	-.082898318	.006296722	-.095277311	-.070519325
	Equal variances not assumed			-13.165	397.549	.000	-.082898318	.006296722	-.095277354	-.070519283

5.4 Native retention – after merger (chapter 6.2)

Group Statistics

	VOWELINVENTORY	N	Mean	Std. Deviation	Std. Error Mean
RETENTIONFIVE	Native	200	1.96622397	.064190663	.004538965
	NativeRetention	200	1.92594766	.060786443	.004298251

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
RETENTIONFIVE	Equal variances assumed	4.622	.032	6.443	398	.000	.040276310	.006251173	.027986865	.052565756
	Equal variances not assumed			6.443	396.824	.000	.040276310	.006251173	.027986754	.052565867