A Hybrid CNN and ACT-R Model for Handwriting

Dinc Basar, Ulani Qi

85-412 Cognitive Modeling, Anderson

I. Motivation

Visual-motor integration is a frequently researched topic in neural and cognitive psychology, often studied through tasks such as handwriting, catching and throwing a ball, or playing a video game through a game controller. These tasks typically require the visual cortex to encode featural information and spatial location from a visual stimuli, and the motor cortex to be able to produce output in response to that information. Subsequently, much coordination occurs between the two cortices for each iteration of the task. For our final project, we are interested in using ACT-R to develop a computational model of human performance on a visual-motor integration task that will require control over fine motor skills, and will involve sophisticated, less-repetitive levels of variation in motor output. We have decided to devise a handwriting task for our model, because it would involve fine motor coordination between the eyes and the hand, visual classification of a set of unique characters, and retrieval of unique procedural steps from motor memory. For the motor portion of the task, we will utilize ACT-R's pre-existing manual module, which allows the model to perform motor actions through the computer's input devices. The visual portion of the task becomes slightly more involved. Since children who are learning to write letters typically do so by following visual examples from a teacher, we want to integrate a feature for letter-recognition with our model, so we as the "teacher" can write a letter on a canvas screen on the computer, and have the ACT-R model be able to reproduce it. The capabilities of ACT-R's visual module do not currently include a method that supports feature recognition, so we will be pairing the model with a convolutional neural network that would be able to recognize and classify the example letter that the "teacher" writes, and present the classification to ACT-R as input. We additionally want our model to be able to demonstrate learning for the association between a visual input and a motor output, by using ACT-R's memory modules to simulate the storage and retrieval of semantic and procedural information associated with each letter in memory.

A. Handwriting: The Biological Perspective

Our design of the model task assumes a feature-matching model of visual recognition. The feature-matching model searches for simple, characteristic features of an object; the presence of such features implies a match, and that recognition occurred (Smith & Kosslyn, 2007). More features matched on an input would then imply a stronger accuracy of recognition. A "feature" in this situation refers to certain attributes of a visual stimulus that help distinguish it from others. The distinguishing factor can be more general or specific than just shape, color or edges; it could also be an agglutination of certain sub-features. One of the pros of the feature-matching model is its flexibility: it is not necessary to have a complete visual specification of the input or to have the features arranged in the same way for accurate

recognition, as long as the features are present. Another area in which feature-matching surpasses alternative models is its space efficiency; relatively few features are enough to help recognize many unique objects from the same category. It turns out that for visually simplistic stimuli such as the alphabet in our case, it is sufficient to engage only the initial stages of visual analysis that detect edges and colors. This is because the English alphabet is comprised of limited features that are line segments varying in length, orientation, and curvature. In contrast, complex stimuli such as faces would have the eyes, nose, and mouth as identifying features, requiring that further analysis be done on those features (and perhaps on their sub-features) before getting to the face itself.

Before delving deeper into how features contribute to recognition, it is important to first understand how these features are obtained from the input. According to Smith and Kosslyn (2007), spots and edges, colors and shapes, movements and textures are the fundamental building blocks of perception. Individually they are not considered objects, but together they define how we see all the objects in our surroundings. These features first enter the visual system as light reflected from the objects around us, and are converted into electrochemical signals by an array of photoreceptor cells in the retina. Signals from the photoreceptor cells pass through a network of interneurons in the second layer of the retina to reach ganglion cells in the third layer. The neurons in these two retinal layers exhibit complex receptive fields that enable them to detect contrast changes within an image; these changes might indicate edges or shadows. Ganglion cells gather this information along with other information about color, and send their output into the brain through the optic nerve.

The role of the optic nerve is primarily to relay information to the cerebral cortex, where visual perception occurs. Most projections end up in a part of the thalamus called the lateral geniculate nucleus (LGN), deep in the center of the brain. The LGN separates retinal inputs into parallel streams, one containing color and fine structure, and the other containing contrast and motion. Cells that process color and fine structure make up the top four of the six layers of the LGN, and the cells processing contrast and motion make up the bottom two layers of the LGN. From the LGN, the cells lead all the way to the back of the brain, where the primary visual cortex (V1) is. Cells in V1 are arranged in several ways that allow the visual system to calculate where objects are in space. First, V1 cells are organized retinotopically, which means that a point-to-point map exists between the retina and primary visual cortex, and neighboring areas in the retina correspond to neighboring areas in V1. This allows V1 to position objects in two dimensions of the visual world, horizontal and vertical. The third dimension, depth, is mapped in V1 by comparing the signals from the two eyes. Those signals are processed in stacks of cells called ocular dominance columns, a checkerboard pattern of connections alternating between the left and right

eye. A slight discrepancy in the position of an object relative to each eye allows depth to be calculated by triangulation. Finally, V1 is organized into orientation columns, stacks of cells that are strongly activated by lines of a given orientation (Hubel & Wiesel, 1959). Orientation columns allow V1 to detect the edges of objects in the visual world, and so they begin the complex task of visual recognition. With each following layer of the visual cortex, the level of processing becomes increasingly complex; V2 is responsible in maintaining color constancy (how we continue to perceive a leaf as green even under varied lighting), V3 is responsible for color detection, V4 is responsible for form detection, the inferior temporal lobe performs face and object recognition, and the parietal lobe takes control of motion and spatial awareness.

The other half of the equation, motor cognition, involves three main motor regions, which we will describe in order from low-level to high-level. The primary motor cortex (or M1) is in charge of providing low level motor functionality. It is located in the frontal lobe of the brain, and its role is to generate neural impulses that control the execution of movement. Signals from M1 cross the body's midline to activate skeletal muscles on the opposite side of the body, meaning that the left hemisphere of the brain controls the right side of the body, and the right hemisphere controls the left side of the body . The organization of M1 is also quite unique: the amount of brain matter devoted to any particular body part represents the amount of control that the primary motor cortex has over that body part (Smith & Kosslyn, 2007). For example, a lot of cortical space is required to control the complex movements of the hand and fingers, and these body parts have larger representations in M1 than the trunk or legs, whose muscle patterns are relatively simple.

The premotor cortex is involved in preparing and executing limb movements and coordinates with other regions to select appropriate movements. It arranges for sequences of actions to be carried out, and sends directions to M1. At the topmost level is the supplementary motor area, which sets up and carries out overarching plans for action. Unlike in the visual cortex where the level of abstraction on an input increases, the level of abstraction of a motor output in the motor cortex tends to decrease. But without information, there is no way for the supplementary motor area to plan any action. This is where basal ganglia come in useful. Basal ganglia are involved in a complex loop that connects them to various areas of the cortex. The information from the frontal, prefrontal, and parietal areas of the cortex passes through the basal ganglia, then returns to the supplementary motor area via the thalamus. The basal ganglia are thus thought to facilitate movement by channelling information from various regions of the cortex to the SMA. The basal ganglia may also act as a filter, blocking the execution of movements that are unsuited to the situation. Also crucial to motor function is the cerebellum, which actually plays several

roles. It stores learned sequences of movements, it participates in fine tuning and coordination of movements produced elsewhere in the brain, and it integrates all of these things to produce movements so fluid and harmonious that we are not even aware of them. As an analogy, consider this: for one to perform even so simple a gesture as touching the tip of their nose, it is not enough for the brain to simply command the hand and arm muscles to contract. To make the various segments of the hand and arm deploy smoothly, you need an internal "clock" that can precisely regulate the sequence and duration of the elementary movements of each of these segments. That clock would be the cerebellum.

With an understanding of how both systems work separately, we can now consider what happens when the two systems interact. Figure 1 below shows a graphical view of the steps involved in handwriting. First, we have the visual input, which is detected by the photoreceptors and passed through the layers of visual cortices, where feature detection occurs. After obtaining the all the feature information, the next step (orthographic analysis) is to classify the input based on its features, and place the classification (along with any other information necessary) in the graphemic buffer, a type of working memory buffer. Using the classification stored in the graphemic buffer, we can attempt to retrieve from the allographic store some procedural knowledge on how the letter should be written. If no allographic information was retrieved, the posterior parietal cortex will try to translate spatial coordinates into motor coordinates, and the supplementary motor area will plan the sequence of actions that would be taken to draw the character. If allographic information was retrieved, the information would be directed to the next layer of motor neurons. In both situations, once procedural information is obtained, it can be sent to the M1 neurons, where motor output is produced in the form of writing (Wing, 2000).

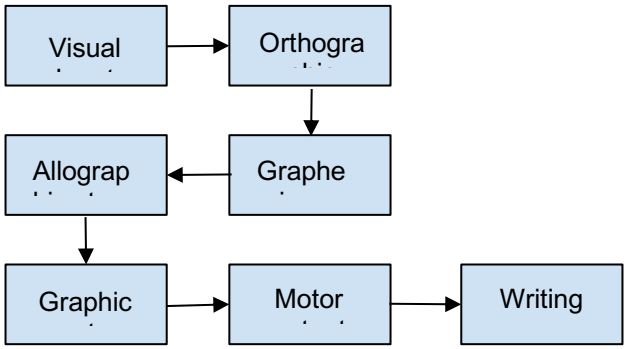


Figure 1: Processes in the production of handwriting

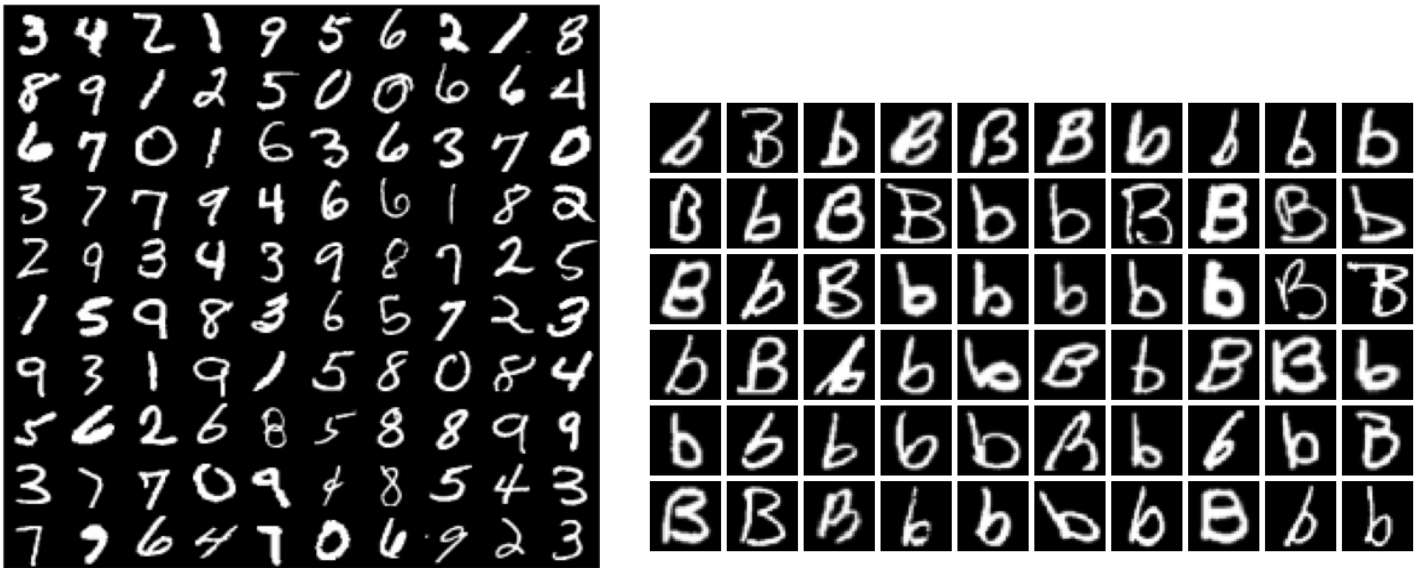
B. A Computational Model: Connecting the Components

We want to be able to draw analogies between the processes in our model of the task and the actual neurological processes of the task. From the start, it was evident to us that we would not be able to perform the visual classification part of the task using just ACT-R. We decided to pair it with a convolutional neural network, which would serve as ACT-R's visual cortex and would handle the feature detection and orthographic analysis in the task. CNN's are biologically inspired by the visual cortex and how its ganglion cells are sensitive to specific regions in a photoreceptor field. In 1962, an experiment by Hubel and Wiesel showed that some individual neuronal cells in the brain responded (or fired) only in the presence of edges of a certain orientation, i.e. some fired for horizontal edges, some for vertical edges; others fired when lines were oriented a certain angle. It was discovered that all of these neurons were organized in a columnar architecture and that together, they were able to produce visual perception (Hubel & Wiesel, 1959). This idea of specialized components inside of a system having specific tasks (the neuronal cells in the visual cortex looking for specific characteristics) is one that machines use as well, and is the basis behind CNNs. Convolutional neural networks are very similar to ordinary neural networks: both consist of neurons with learnable weights and biases, each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. Both networks start from the raw image pixels on one end and result in classification scores (probabilities for each classification) at the other. And both still use a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer. The difference is that convolutional network architectures explicitly assume that the inputs are images, which allows us to encode certain properties into the architecture that make the implement efficient and vastly reduce the amount of parameters involved.

Once the classification is achieved, the CNN's role will be complete and it will pass the information on to ACT-R, whose memory buffers are crucial in the next step, involving the retrieval of procedural knowledge. In this case, the graphemic buffer can be analogous to ACT-R's imaginary buffer, and to retrieve procedural knowledge from the allographic store, we can just make a retrieval request in ACT-R. To produce the motor output, we would direct ACT-R to "draw" the strokes that would make up a letter, by moving the mouse then clicking on a point to signify the start of the stroke, and moving the mouse then clicking again to signify the end of a stroke. In the backend, we will draw a straight line starting at the first point and ending at the second point to visualize the strokes the model produces.

II. Convolutional Neural Network: Implementation

A. The EMNIST dataset



Images: the full dataset, some portion of which was not used as the degrees of freedom in our drawing matrix would not support it. This will be discussed in more detail in the following pages.

The EMNIST dataset which we trained our neural network on is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset (Cohen, Afshar, Tapson & van Schaik, 2017). EMNIST categorizes its data by two main categories: ByClass and ByMerge. Class, short for classification, represents the most useful organization from a classification perspective as it contains the segmented digits and characters arranged by class. ByClass contains 62 classes comprising of [0-9], [a-z] and [A-Z]. The data is also split into suggested training and testing sets. The ByMerge categorization addresses an interesting problem in the classification of handwritten digits, which is the similarity between certain uppercase and lowercase letters. The similarities were obtained after examining the confusion matrix resulting from the full classification task on the By Class dataset. This variant on the dataset merges certain letter classifications, to produce a 47-class classification task. The classes were merged for the letters C, I, J, K, L, M, O, P, S, U, V, W, X, Y and Z. Four additional subsets also exist in the latest version, one of them being EMNIST Letters, a complete set of training data for all of the 26 capital letters. In total, the EMNIST Letters consists of 145,600 characters, in 26 balanced classes. According to the official EMNIST documentation, the EMNIST Letters dataset was created to fix misclassification issues regarding due to letter case in the By Class and By Merge datasets. The solution was accomplished by combining the uppercase and lowercase versions of each letter into a single class and removing the digit classes entirely. The resulting dataset therefore provides a different classification task from the other datasets and represents a letter only classification task, requiring classifiers to associate two different representations of a letter with a single class label. To demonstrate an increase in accuracy, the same classification network was applied to all the letter data (uppercase and lowercase) in the EMNIST

ByMerge dataset as well as the Letters dataset. Ten trials of each experiment were performed. The classification performance for the classifiers trained on the two digit datasets are shown below in Figure 2. Clearly, the variance caused by mixing upper and lowercase letters in the same dataset are greatly reduced in EMNIST Letters, and the accuracy is also significantly higher for EMNIST Letters. Additionally, they produced a confusion matrix (Figure 3) for a single trial of their classifier, which contained 10,000 hidden layer neurons and was trained and tested using the full EMNIST Letters dataset. They reported that most of the confusions occur between classes that inherently contain ambiguity, primarily between the I and L classes. The G and Q classes also appear to suffer from ambiguity resulting from the similarity in their lowercase representations.

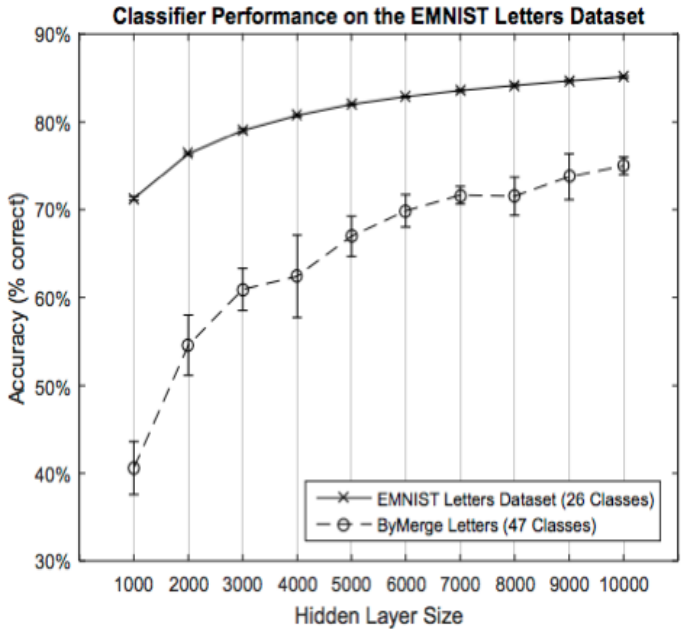


Figure 2.
Classifier performance on the EMNIST Digits and EMNIST MNIST

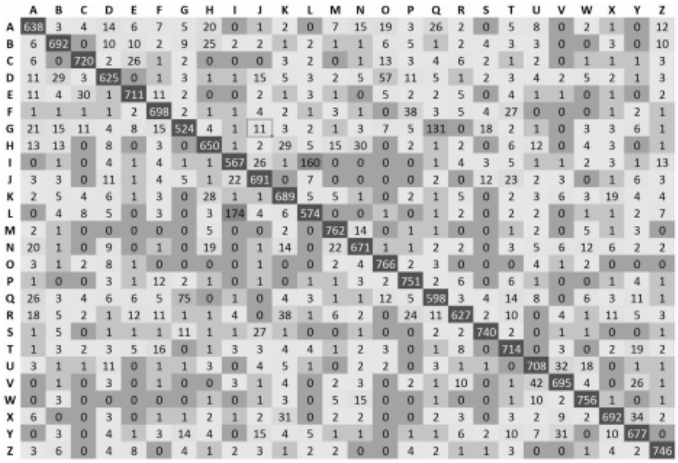


Figure 3. Confusion matrix for the EMNIST Letters Dataset
for a network with 10,000 hidden layer neurons.
(Source: Cohen, Afshar, Tapson & van Schaik, 2017)

B. Neural Network Architecture

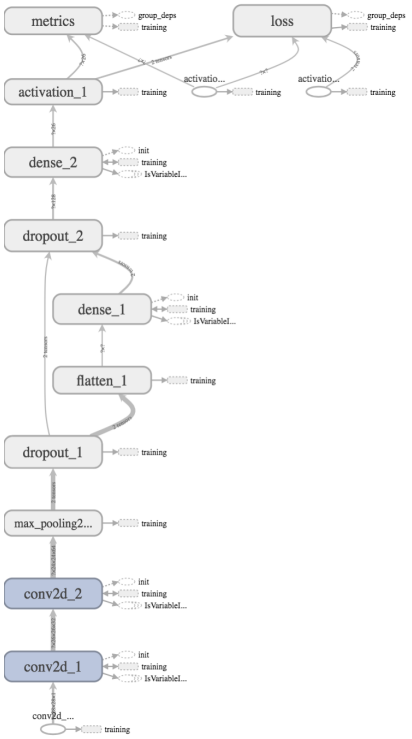


Figure 4. Final convolutional neural network architecture.

In training our model, we decided to use Keras, an open source neural network API written in Python, because it would allow us to experiment with model parameters and model architecture without spending too much time on low level implementation and debugging. Originally, our intention was to implement the neural network using the same classifier as the one from the paper on EMNIST by Cohen, Afshar, Tapson & von Schaik. However, although they specified that they used an Online Pseudo Inverse Update Method (OPIUM) classifier, there was no further information provided about the implementation. Therefore, we decided to start with a baseline network architecture and build upon it from there. One of the features of Keras is its ability to save the model state after training, and then reload it later to either train it more or make predictions with it. The model architecture would be saved in a .yaml file, and the model weights would be saved in an .h5 file. The final form of the architecture we implemented can be seen in Figure 4.

C. Image Processing

Before we could feed input data to the neural network for training or prediction, we had to process the input so that it would be compatible with the neural net. The image processing workflow of our training and testing input (provided by EMNIST) and our prediction input (saved from the Tkinter canvas) were similar, except that input EMNIST images were 128x128 pixels, whereas the prediction images that we got were 256x256 pixels. The EMNIST Letters data that we downloaded from their official website was originally stored inside one .mat file, where each row of the matrix contained the byte data for one image. Thus part of our training process was to extract and transform the image data. First, we first had to turn the image into grayscale, and crop it around a region of interest sot he the character can be centered in the image. We also needed to reshape the dimensions of both our training and

testing images to be 28x28 pixels, convert the type of the image byte array to float32, and normalized the image byte values. The steps we took were similar to how Cohen, Afshar, Tapson & von Shaik processed their images for their classifier in the EMNIST paper (Figure 5).

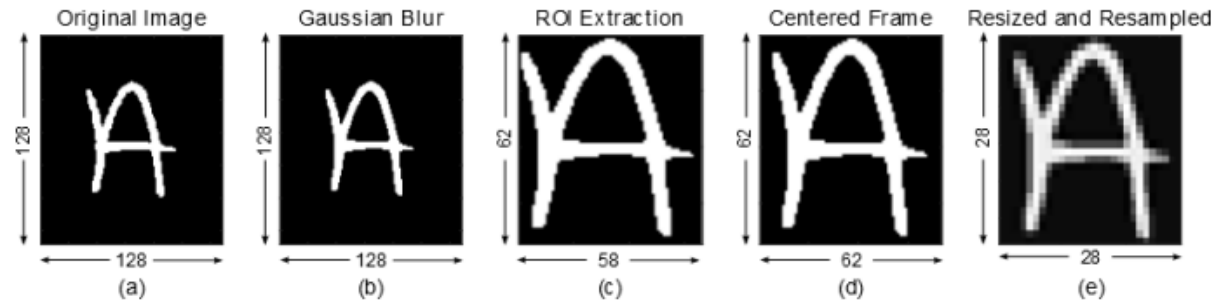


Figure 5. Diagram of the conversion process used to convert the NIST dataset. The original images are stored as 128 × 128 pixel binary images as shown in (a). A Gaussian filter with σ = 1 is applied to the image to soften the edges as shown in (b). As the characters do not fill the entire image, the region around the actual digit is extracted (c). The digit is then placed and centered into a square image (d) with the aspect ratio preserved. The region of interest is padded with a 2 pixel border when placed into the square image, matching the clear border around all the digits in the MNIST dataset. Finally, the image is down-sampled to 28 × 28 pixels using bi-cubic interpolation. The range of intensity values are then scaled to [0, 255], resulting in the 28 × 28 pixel gray-scale images shown in (e)

D. TK Integration

One of the last features we decided to incorporate into our model was another canvas on the screen, implemented using Python's Tkinter, which would allow one to use the mouse and draw a letter for ACT-R to "copy". The purpose is to simulate how children learn by imitating examples from a teacher. The code implementing the teacher's canvas served as a bridge connecting the ACT-R model with the neural network. First, the "teacher" draws a letter on the canvas. Once they press the save button, the code in the backend captures the drawing on the canvas and saves it as a .png file named *output.png*. When they press the "Predict" button, the backend code takes the generated *output.png*, processes it to be compatible with the model, and saves the processed image in a new file, *resized.png*. This file is then fed as an input argument to the neural network, which returns a prediction for the letter. The canvas backend then loads the ACT-R model into the environment, and gives ACT-R the neural network's prediction as input.

III. ACT-R Model

A. Initial version:

For the first version of our ACT-R model, our aim was mainly to test out that the model was able to write all 26 characters on the canvas using our method of representing letters. Thus, the chunk mechanism was set up in a double layer, in the sense that a single chunk, along with an always-successful increment retrieval layer, contained all the necessary information for the model to draw the letter. However, we soon realized that this mechanism allowed the model to learn much more quickly than a realistic curve, and it would be able to either fully retrieve the letter or fail completely - there was no partial success. This, in our view, does not represent a real world scenario, because learning such a multi-component chunk cannot be completed in one step. In fact, when we ran the initial model, we saw that it was able to learn much more quickly than the updated model, requiring significantly less training examples. Figure 6 depicts that the old version of the model is able to reach almost 100% accuracy by the 500th trial, while the new version requires 2500 trials to reach 75% accuracy.

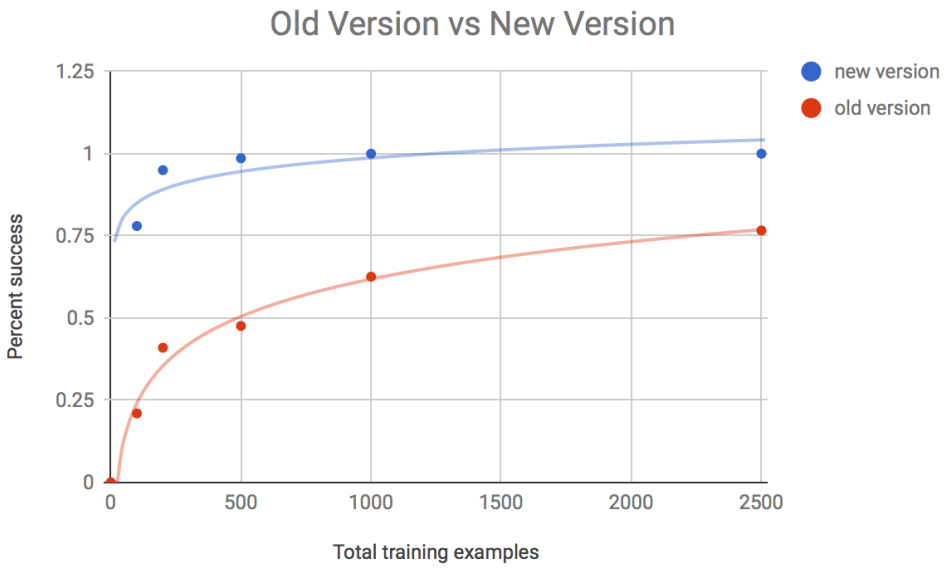


Figure 6: Learning curves of the old and new model

An example chunk of the letter A is as follows: (A ISA outline s0 c01 d0 c10 s1 c10 d1 c20 s2 c01 d2 c12 s3 c12 d3 c22 s4 c10 d4 c12). In this format, s# slots define the starting point for a mouse movement, and d# slots define the destinations. The c## chunks are the respective coordinates in the grid matrix, c00 being top left, c11 middle, c12 middle right, and so on. One c## chunk example is as follows, for c12: (c12 ISA visual-location screen-x 60 screen-y 50). Screen-x increases as the first coordinate in c## is increased (for example, from c01 to c11), and screen-y increases as the second coordinate is increased.

A single retrieval request was made by the model to retrieve the entire outline by a production, and this chunk was then encoded by another production into the imaginal buffer once it was retrieved. If there was a retrieval failure, the model would not draw anything, thus ending the trial and resulting in the relevant feedback set according to the tuneable hyperparameters. If the retrieval and encoding was successful, an increment production would start from slot s0 and feed its value (c01 in the case of A) to a production that would move the mouse to the desired location, and click. Then, the increment production would retrieve d0, and the process would repeat again, until whichever s# slot of the chunk was set to nil (s5 in the case of A). Once this state was reached, the model would reset its goal buffer and get ready for a new presented trial.

The increment production would use sequence chunks of the form (one ISA sequence identity s0 next d0), and all of these chunks had high base level activations set to make sure there was no error in their retrieval. This, in our opinion, was realistic, since we did not want to model a child that was learning to increment numbers, but rather a child that was learning to retrieve information regarding a letter.

Since we thought that this was not a realistic model of learning, we decided to update our chunk mechanism, which will be discussed next.

B. Final version

Updating the chunk mechanism was crucial in achieving more gradual and realistic learning in our model. We decided to increase the depth of these chunks, so that there would be three layers of retrieval instead of two as the old version. We also switched from defining the chunks in the model code to defining them in the python code, as this would make providing feedback to the model much easier. Thus, now the chunks would be in the form ['A-first', 'isa', 'outline', 'letter', "'a'", 'count', 'first', 'start-coor', 'c01', 'dest-coor', 'c10'] followed by ['A-second', 'isa', 'outline', 'letter', "'a'", 'count', 'second', 'start-coor',

'c10', 'dest-coor', 'c20'], and so on until enough chunks are reached to complete the letter outline. This meant that the model would not be able to retrieve a single chunk and be either entirely successful or entirely fail anymore, it would have to retrieve a succession of chunks to complete a letter. Our goal chunk-type was defined as follows, with its relevant slots being used by different productions throughout the run-time of the model: (chunk-type goal letter state number). The letter is a single character string, for example, “A”, the state can take various different values such as start-retrieval, click-start, move-end, and so on, and the number is the count of which stroke is being processed at the moment, “first”, or “third”, as examples. We will now go over the productions of the model in detail, explaining how they use the goal and the imaginal buffer while processing the information and attempting to retrieve and draw a letter.

1. find-goal & encode-goal-letter

The model starts with a production named “find-goal”, which finds a displayed letter on the screen and commands the visual module to move its attention to it. Then, a production named “encode-goal-letter” finds the displayed letter and places it in the goal buffer. These two productions symbolize the transfer of information from the visual stream into the executive control mechanism in the brain, where a course of action will be planned and executed. It is also possible to bypass these two productions and set the goal chunk to the relevant example while training the ACT-R model, but the screen-instruction system remains useful to display what the model is getting as input and how it is executing moves step-by-step.

2. start-retrieve

Subsequently, if there is a letter in the goal buffer, a production named “start-retrieve” attempts to retrieve the *first* chunk relating to that letter, A-first, for example. This retrieval symbolizes the attempt to reach the initial outline for a letter in the brain, and if it fails, the productions stop firing as well, just like a person would stop in case they had no memory on how to proceed with a task.

3. fail-retrieve

In our model, the production mentioned in the previous subsection is named “fail-retrieve”, and it is used modularly in any case of retrieval failure, either at the beginning of drawing or throughout any of the steps if necessary. It resets the goal buffer to prepare the model for a new trial.

4. encode-letter-outline

In case the retrieval was successful, this production fires, and it is used by all the steps throughout the drawing, once again, in a modular fashion. It first gets the value of the "number" slot in the goal buffer and checks that the count in the retrieved chunk matches that. It also gets the start-coor and dest-coor values from the retrieved chunk and encodes them into the imaginal buffer, where they will be used throughout the motor movements. As the retrieval buffer is now free, this production also makes a retrieval request to find what value the "number" slot in the goal buffer should be incremented to. As mentioned earlier, these sequence type chunks have high base levels set to make sure that this retrieval does not fail, as it is not what we are intending to study.

5. move-start-coordinate

Now, the actual drawing action is initiated by the model with this production. There will be four steps for a single stroke to be completed, which are: move-start-coordinate, click-mouse-start, move-end-coordinate, and click-mouse-end, all of which are used modularly by all of the strokes in all of the letters, so that they are never repeated. These four steps signal the python code to record the actions of the model, so that they can be checked against correct answers and relevant feedback can be provided to the model. The move-start-coordinate production takes the start-coor value from the imaginal buffer, making sure that the chunk's count and goal's number also match, and commands the motor module to move the cursor to that start-coor. It also prints out the start-coor value to the user's terminal, in order to show where the model clicked as it steps through the productions. In order to keep the imaginal chunk from being harvested, it also has an empty =imaginal> command on the RHS of the production. To signal that the model is ready to start drawing a stroke, it sets the state of the goal chunk to click-start. This chain of actions can be seen as the simulation of a person moving their pen to the top of the letter "A", for example, before they start to make their first stroke.

6. click-mouse-start

At this point, as the model is ready to make the stroke, this production checks that the goal chunk's state is set to click-start and makes a command to the motor module to click-mouse. This will be watched and recorded by the python code, in order to start keeping the list of coordinates that the mouse was clicked on to be used by the correctness and feedback functions. Apart from that, the production also outputs a single "Click" to the user's terminal, in order to track the movements of the model, once again.

The click-mouse-start production in general can be seen as the person placing their pen on a piece of paper, ready to move it and draw the stroke on paper. The production also sets the state slot of the goal chunk to “move-end”, which will be a signal to the next production to be discussed.

7. move-dest-coordinate

Signaled by the previous production, this production takes the dest-coor value from the encoded imaginal buffer chunk and makes a motor module command to move the cursor to that position. It also outputs a string in the format “Move [%dest-coor]” to the user’s terminal, making it easier to track the model’s movements and the exact coordinates once again. The state of the goal chunk is set to click-end, signaling the next production that the model is ready to finish a stroke and start a new one. This chain of actions can be seen as a person moving a pen from the start of a stroke to the end, from the top middle of “A” to the bottom-left for the first stroke, for example. However, the stroke is yet to be finished by the next production to be discussed.

8. click-mouse-end

Now it is time for the model to finish the retrieved stroke, so this production is the final link in the chain of four productions described in the subsection 5. This production will also be required to initiate the next stroke’s retrieval, so it checks that the count of the goal buffer is the same as the retrieved sequence chunk (for increment purposes) are the same. The next slot of this retrieved chunk represents the count of the next stroke to be drawn, so if “first” stroke is being finished currently, then “second” should be retrieved in the next slot of this chunk. The production sets the state of the goal buffer to outline-retrieval, which will signal that it is ready to encode a new outline to be used in the encode-letter-outline production, and sets the number slot of the goal chunk to next. In other words, it is now ready to retrieve “A-second”, and encode that chunk into the imaginal buffer and start the cycle over again. In order to do that, it makes a retrieval request to retrieve the chunk “A-second”, which may or may not fail due to the three level depth mechanism, and commands the motor module to click the mouse, signaling to the python code that the model has finished drawing a stroke. Together with the start-coor that was recorded in the click-mouse-start production, the current coordinate of the mouse will be recorded and appended to the click list, to be used by the correctness function once the model is completely done with the letter drawing attempt. There are two possible outcomes after this production: the model can retrieve the next stroke, or it might fail at the step. If it succeeds, the model will proceed as discussed, but if it fails, the fail-retrieve production will be fired as discussed earlier.

Fail-retrieve might fire for two reasons at this point: the outline might be actually finished, so the letter is completed successfully, or the model is not able to remember the next step, in which case the failed chunk's retrieval needs to be strengthened. In either case, since there is a separate python function that checks for the correctness of the model's output, it does not make a difference how the model finishes drawing a letter, as long as it does finish and get ready to start a new trial.

IV. Modularity of ACT-R Model Code

The human brain is known to use same neural circuitry for a number of tasks, so it is important that a cognitive model is also able to do the same. For example, an image of a cat and an airplane are both first received by the retinal ganglion cells in the eye and transferred to V1 from the same pathways, only to start being separated as the visual streams start to classify them into different categories. We believe that the same type of decision making and memory retrieval processes also use common circuitry, so we attempted to make our ACT-R productions as modular as possible, not repeating them over and over again. This, as a side effect, makes it much easier to debug code and fix issues once they are traced to their roots, as there is a much smaller number of productions to be modified. In order to achieve this, we wrote both our encode-letter-outline and the four productions to draw a stroke in a modular fashion, where they can be used for any stroke of any letter. This allows our ACT-R code to be relatively concise, and thus, we were able to spot any mistakes very quickly and fix them. If we had productions separated for each stroke count or each letter, for example, we might not have spotted mistakes in the process, and a certain letter or a certain stroke could be incorrectly completed no matter how many training examples the model was given. However, with the current set up, any bug in our code showed up with any letter we tried, so we were able to spot them immediately and look for fixes.

V. Integration of Neural Net and ACT-R

Once the implementation stages of the NN and the ACT-R model were complete, we wrote the required python code to take the recognized letter from the NN and initiate the python code for the model with that letter. This means that if the user draws an "A" on the TK screen, then saves and runs the model, this will be saved by the NN code. Once the user clicks the Send to ACT-R button, the model code is initiated, and a screen is opened with the recognized "A" letter displayed. This signals the model that it should retrieve the letter outline information from its memory and start drawing. Since this integration is only for demonstration purposes, this model simulates a perfectly trained human that is never able to fail

drawing a letter. However, it is still possible to run it with imperfect models in order to see how and when it fails.

VI. Learning of Neural Net

In total, the neural net was trained over 12 epochs of 124,800 training examples (letters) and tested on 20,800 testing examples. We were able to achieve a final test score of around 0.1963, and a final test accuracy of about 0.9383. Figures 7a and 7b show the final model's improvement in accuracy while training over the 12 epochs.

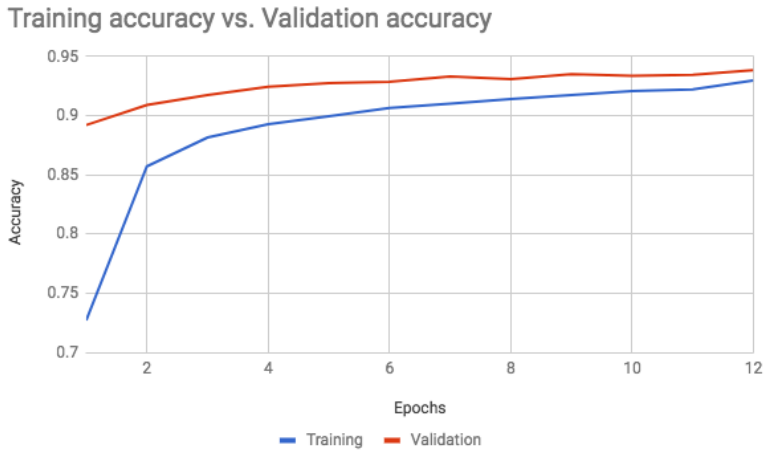


Figure 7a. Training vs validation accuracy over 12 epochs of NN training

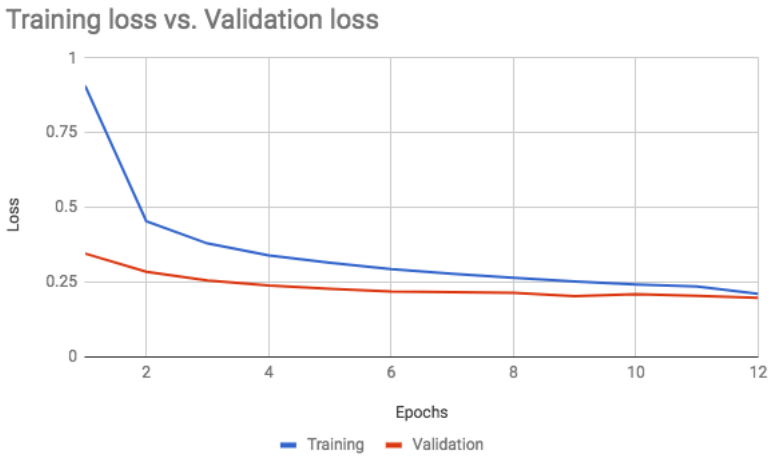


Figure 7b. Training vs validation loss over 12 epochs of NN training

From Figures 7a and 7b we can also note that the loss curves for training and validation appears to converge nicely at the final epoch, which is evidence that this model is a good fit to the data. If the curves did not converge, the model would have been underfitting the data, meaning that it was insufficiently trained and therefore accuracy would not be very high if it was made to predict the classification of an arbitrary input. If the curves converged too soon and then parted again, it would have been a sign that the model was overfitting the data, implying that it was trained too much, and that although performance on the train set is good and continues to improve, performance on the validation set will only improve up to a point and then begin to degrade.

VII. Learning of ACT-R model

Since studying the memory retrieval was the main goal of our project, we decided to take a look at the effects of three different hyperparameters on the learning curve of the model: the retrieval threshold, set by the :rt parameter in ACT-R; the number of training examples provided to the model in the case of a failure to draw the letter completely (keeping a constant one training example for correct answers); and the effect of how much the model waits after being trained on a letter. We will now discuss the effects of changing these parameters, as we train the model from scratch each time of 100, 200, 500, 1000, and 2500 trials. We were able to tune these parameters to find what we thought was the most realistic learning curve, and we will discuss this at the end of this section.

It is important to note that seeing the learning curve change while we tuned these parameters was also very important in making sure that our model was functioning properly. If we had seen that any of these changes did not result in a realistic outcome, we would have known that our memory mechanisms are not functioning as we expected them to, but that fortunately did not happen and we were able to observe what we expected.

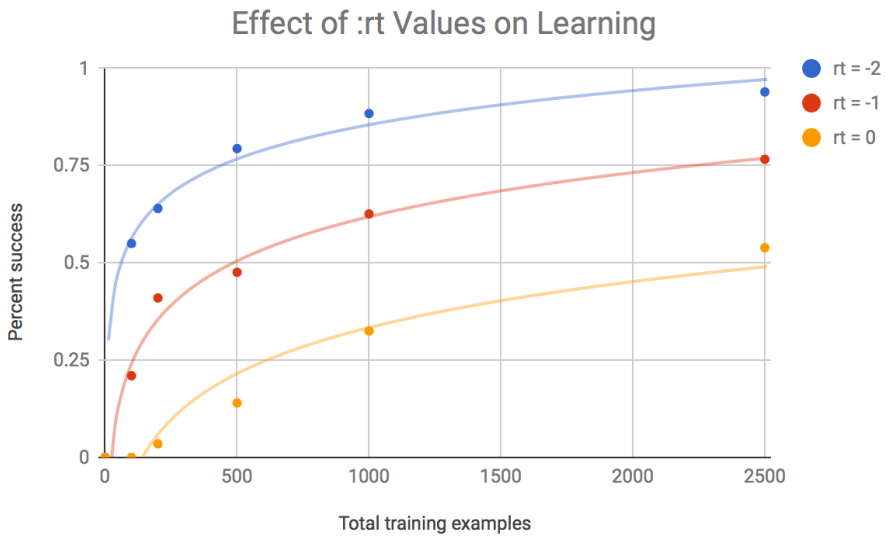
The training procedure involves a random letter from the alphabet being provided for the desired number of trials, where the python code will place the relevant chunk in the imaginal buffer of the model and then clear it, so that the chunk is reinforced. This mechanism was designed modularly as well, so that it could be used both while simply training the model, and while giving it feedback with correct and incorrect trials.

The model always starts with no knowledge of the letter; thus, it always fails at the 0th training example, and its success increases over the training examples it completes. The following graphs display

the average success over a number of training examples, so it depicts how much the model was able to learn by the end of the training set - as all models start with the same success rate of 0.

A. The Retrieval Threshold, :rt

We studied the effects of three different retrieval threshold values: -2, -1, and 0. We expected that a lower threshold would result in quicker learning, as a chunk only needs to be reinforced to a lower extent before it is retrieved. However, this also raises questions of how realistic a low threshold would be in a real world scenario, where similarities can be very high between a variety of chunks. If information can be learned very quickly, then these similar chunks can also be confused very quickly too, as they are not learned with as much detail as required. Once we ran our model with the three values, we saw a clear pattern of an inverse relation between retrieval threshold and the level of success after a set number of examples. At 1000 training examples, an rt value of 0 was only able to reach 32.5% successful drawings, while -1 was doing much better at 62.6% and -2 at 88.4%. This clear gap shows the dramatic effect, and it supports our hypothesis that increasing the rt value would make it harder for the model to learn the letters. Figure 8 below includes a graph of all of our 15 points of data, along with best fit curves to display how the relative learning rates changed over time. The best fit curve shows that a value of -2 is somewhat unrealistic, as the model is able to average almost 100% success after 2500 trials, meaning that it got a vast majority of the trials completely correct without any failure, since it started with 0% success in the beginning. However, on the other end of the spectrum, a threshold of 0 is also depicting that the model is losing its rate of learning rapidly - it will reach an asymptote below 75% no matter how many training examples are provided and will never be able to reach 100% success. This is not how a human would



learn, as educated adults tend not to have problem remembering how to draw letters completely. Since the value of -1 seems to have the most realistic learning curve of all, we will be setting that as the dependent variable from now on as we tune other parameters.

B. Number of training examples after incorrect answer

With our modular training mechanism, we were able to change how many times the model would get feedback if it made a mistake, in other words, it was not able to complete a letter fully. This parameter symbolizes how many times a real human would repeat attempting to draw a letter after they fail to do so, while they only train once if they were successful. We tested the average success of our model with three different values for this parameter: 1, 3, and 5. Our initial hypothesis was that more trials would lead to more success at the same number of trials, and this was supported by the data we collected. At 1 training example after an incorrect attempt, the model was able to average only 21.8% success over 1000 attempts, at 3 training examples it was able to reach 62.6%, and at 5 examples it was able to reach a flat 71.0% success rate. The graph in figure 9 depicts the best fit curves for the 15 points of data that were collected over a different number of trials. Note that the retrieval threshold was set to -1 for all of these trials, since we want to only have one independent variable as with any study.

Figure 9: Graph of Effect of # training examples after incorrect trial on learning

It is possible to see that all three of the learning curves support our initial hypothesis, and the model is successfully able to display the expected trends of learning as the parameter is tuned. Once again, a brief analysis shows that only 1 example will not be able to reach 100% correctness ever, and thus is somewhat unrealistic, but it is not very clear which of the 3 and 5 would be a more realistic choice. If we had data collected on the matter that we could use to fit, we could make a more informed analysis and see which one is a more realistic choice. Thus, we will first accept the range of [3, 5] training examples as an acceptable range to select as we are not able to distinguish the results and rule one out directly, but we will assume that 3 trials after a mistake would be a more realistic learning method for a human to follow in real life, and will use that for the rest of our trials. That brings our set of optimal parameters as: {rt: -1, num_epochs: 3}.

C. Post practice delay time

One other important parameter we wanted to study was the time the model has to wait after being given the information to the time it can start the drawing. We think that this is a realistic choice, because it symbolizes that in a real world scenario, a person would see their teacher drawing a letter on the board and would have to wait until being given the instruction to try it on their own. We decided to tune this parameter to three different settings: 5 seconds, 10 seconds, and 20 seconds. Since this was a significantly finer range of tuning, we hypothesized that less time would lead to more learning, but the effect would not be as dramatic as changing either the retrieval threshold parameter or the number of examples.

In fact, the data we collected supported our hypothesis: a model that waited 5 seconds was able to

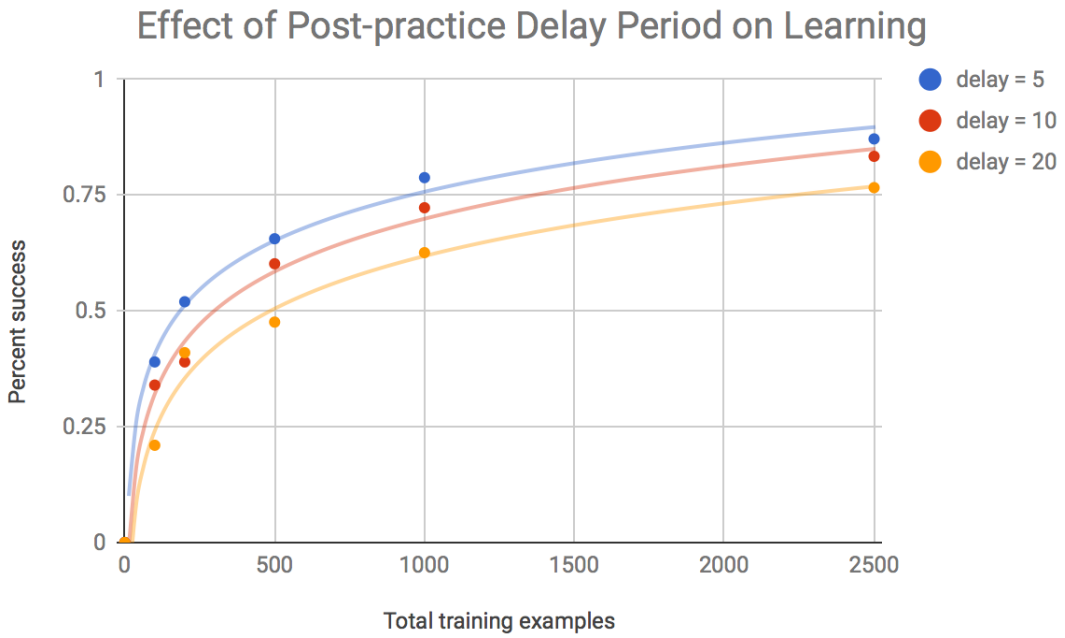


Figure 10: Graph of Effect of Post-practice Delay Period on Learning

reach 78.8% average success on 1000 trials, 10 seconds lead to 72.3%, and 20 seconds lead to 62.6%. Firstly, the range is only 12.2% between the best and the worst, while the range was 55.9% for the retrieval threshold parameter. This supports our initial thought that the effect would be not as dramatic, as changing the retrieval threshold from -2 to 0 is much more significant than changing the wait time from 5 seconds to 20 seconds. Figure 10 shows the best fit curves for all 15 points that we collected over these trials.

It is clear how much more close the curves are than any of the two previous graphs, and the asymptotic behavior of these trials do not seem to change significantly. For that reason, once again since we do not have collected experimental data, we will not be able to rule out any of the three values directly, and must choose one depending on what we think a real world scenario would involve. Very simply, we believe that a teacher would take at least 20 seconds to finish drawing a letter and tell the student(s) to start drawing it on their own paper, so we will choose that value as the most realistic one. That brings our set of optimal parameters to {rt: -1; num_epochs: 3, delay: 20}, which we will describe one last time in the next section.

D. The optimal model

Despite not having access to data on the subject, we were able to pick the best of these three parameter choices to maximize the level of realisticity of our model, and we tested to make sure that it did as we expected. If we had access to previously collected data, it would have definitely been much more possible to represent a real world scenario and fine tune all of the parameters, but we are unfortunately

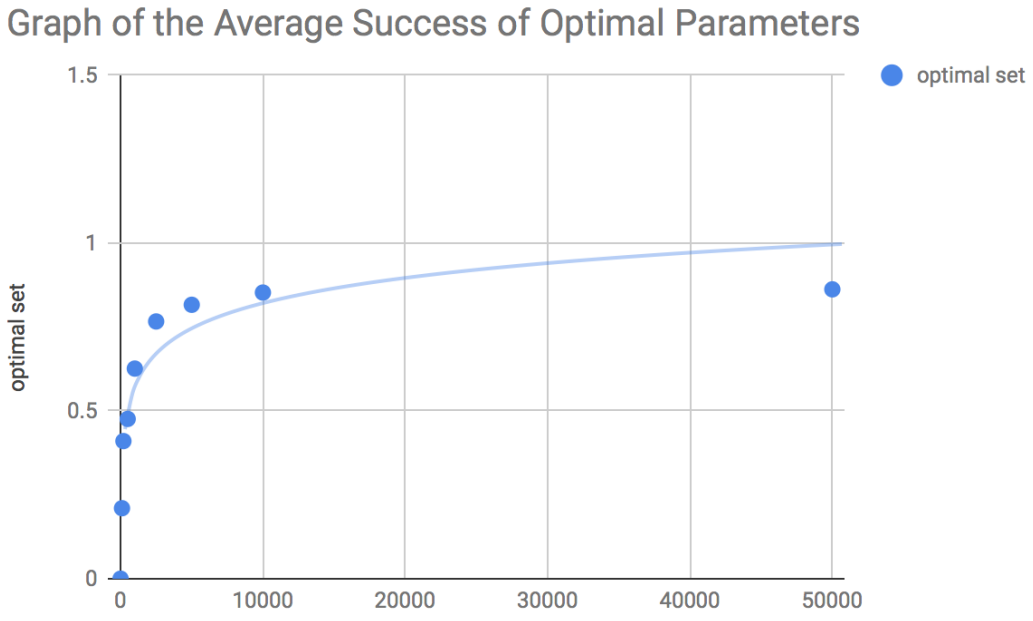


Figure 11: Graph of the Optimal Model

not able to access such data, so we had to reason through the tuning of some of the parameters. We will, nevertheless, test to make sure that our model still does as we expect, and Figure 11 shows how the model behaves over an extended number of trials, up to 50000, with a best fit curve showing how it would improve with these fine-tuned parameters.

We can see that the best fit curve reaches 100% success over 50000 trials, which would represent how a human learns to perfectly remember all letters. Once this many trials have been completed, it should realistically be impossible for a human to not remember any of the letters in the English alphabet, as we would expect the real world case to be. Thus, we conclude that with our optimal parameters, such an ACT-R model successfully simulates human learning of the alphabet letters, despite not being perfectly scaled, since there is no way to update and support them with scientific data.

VIII. Conclusion

Through our project, we were able to emulate the process of handwriting from visual recognition to transcription of a character, using our neural network in visual recognition and the ACT-R model for executive control and motor output. We were able to achieve nearly 94% final accuracy on the neural net, and the ACT-R model was able to learn all the characters with 100% accuracy, under the optimal parameters of rt: -1; num_epochs: 3, delay: 20. There are however certain areas in which we can improve upon for future reference. First, note that the ACT-R model in our project learns through retrieval, whereas in actuality, there is also a component of motor learning which is not reflected in ACT-R due to the limited capability of the motor module. Moreover, the errors that our ACT-R model makes as it learns the proper letter strokes are due to memory retrieval errors in which none or only part of the letter can be retrieved. But in fact, there are other sources of error in real-life learning of characters: for example, there might be interference where two highly similar letters interfere with each other, causing errors in visual recognition or in the motor output. To reflect these errors, we would have to change the neural network learning so that similar letters would be a source of recognition error. For the current neural network, it recognizes all characters with extremely high accuracy, except for the letter O, which it quite often mistakes for D, C or G, but on the contrary, it is able to recognize D, C, and G for what they are. To imitate this effect for other classes as well, we would probably have to train the model on the EMNIST ByMerge dataset, at the expense of accuracy, since according to the EMNIST paper, the creators of EMNIST attained only about 75% accuracy using their OPIUM-based classifier. The solution to that could be to increase the number of training and testing examples by creating our own handwritten data examples and training the model on those as well as EMNIST. Finally, since in real life, teachers give

more specific feedback than just whether or not a letter was correctly written, we would want to consider giving feedback regarding partial correctness of a letter. This could be another addition to make the model even more realistic. All in all, we were mostly limited by the fact that we did not have access to real world data which we could use as a primary guideline. If this issue was resolved, perhaps with a long term study on children's period of learning to write, it would be possible to claim our accuracy with stronger backgrounds and also expand the model in various different areas.

Sources

A.M. Wing, Motor control: Mechanisms of motor equivalence in handwriting. *Current Biology*, 10 (6) (2000), pp. 245-248, 10.1016/S0960-9822(00)00375-4

Cohen, G., Afshar, S., Tapson, J., & van Schaik, A. (2017). EMNIST: an extension of MNIST to handwritten letters. Retrieved from http://arxiv.org/abs/1702.05373

Hubel, D. H., & Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *The Journal Of Physiology*, 148574-591. doi:10.1113/jphysiol.1959.sp006308

Smith, E. E., & Kosslyn, S. M. (2007). Perception. *Cognitive psychology: Mind and brain*. (pp. 58-77) Upper Saddle River, N.J: Pearson/Prentice Hall.