High-performance brain-to-text communication via handwriting

Francis R. Willett, Donald T. Avansino, Leigh R. Hochberg, Jaimie M. Henderson*, Krishna V. Shenoy*

Supplementary Information

I. Methods

1. Experimental procedures

1.1 Study participant

This study includes data from one participant (identified as T5) who gave informed consent and was enrolled in the BrainGate2 Neural Interface System clinical trial (ClinicalTrials.gov Identifier: NCT00912041, registered June 3, 2009). This pilot clinical trial was approved under an Investigational Device Exemption (IDE) by the US Food and Drug Administration (Investigational Device Exemption #G090003). Permission was also granted by the Institutional Review Boards of Stanford University (protocol #20804). T5 gave consent to publish photographs and videos containing his likeness. All research was performed in accordance with relevant guidelines/regulations.

T5 is a right-handed man, 65 years old at the time of data collection, with a C4 AIS C spinal cord injury that occurred approximately 9 years prior to study enrollment. In August 2016, two 96 electrode intracortical arrays (Neuroport arrays with 1.5-mm electrode length, Blackrock Microsystems, Salt Lake City, UT) were placed in the hand "knob" area of T5's left-hemisphere (dominant) precentral gyrus. Data are reported from post-implant days 994 to 1246 (see Table M1 below for a list of all sessions). Array placement locations registered to MRI-derived brain anatomy are shown in Extended Data Fig. 7. Note that both arrays still recorded high-quality spiking activity from many electrodes; on average, 81.9 ± 5.6 (mn ± sd) out of 192 electrodes recorded spike waveforms each day at a rate of at least 2 Hz when using a spike-detection threshold of -4.5 RMS (see Extended Data Fig. 7 for example waveforms).

T5 retained full movement of the head and face and the ability to shrug his shoulders. Below the injury, T5 retained some very limited voluntary motion of the arms and legs that was largely restricted to the left elbow; however, some micromotions of the right hand were visible during attempted handwriting (see (Willett et al., 2020) for full neurologic exam results and Supplementary Video 2 for hand micromotions). T5's neurologic exam findings were as follows for muscle groups controlling the motion of his right hand: Wrist Flexion=0, Wrist Extension=2, Finger Flexion=0, Finger Extension=2 (MRC Scale: 0=Nothing, 1=Muscle Twitch but no Joint Movement, 2=Some Joint Movement, 3=Overcomes Gravity, 4=Overcomes Some Resistance, 5=Overcomes Full Resistance). In a recent study from our group which included data from participant T5, we found that body parts which T5 still had control over (e.g. head, shoulder) did *not* have a stronger neural representation than body parts which were fully or almost fully paralyzed (Willett et al., 2020); thus, T5's limited hand motion likely did not have a large effect on the neural activity, which seems to be generated primarily by the intention to move and not overt motion itself.

Interestingly, although T5's hand was almost completely still during attempted handwriting (and he did not hold a pen), T5 reported *feeling* as though an imaginary pen in his hand was physically moving and tracing out the letter shapes as he attempted to handwrite. This

subjective feeling of motion appeared to obey some physical constraints, as T5 reported being able to "write" more quickly if he attempted to write smaller letters.

1.2 Neural signal processing

Neural signals were recorded from the microelectrode arrays using the NeuroPort™ system (Blackrock Microsystems). More details are described in (Hochberg et al., 2006; Jarosiewicz et al., 2015; Pandarinath et al., 2017). Neural signals were analog filtered from 0.3 Hz to 7.5 kHz and digitized at 30 kHz (250 nV resolution). Next, a common average reference filter was applied that subtracted the average signal across the array from every electrode to reduce common mode noise. Finally, a digital bandpass filter from 250 to 3000 Hz was applied to each electrode before threshold crossing detection. This filter was applied non-causally (using a 4 ms delay) in order to improve spike detection (Masse et al., 2014).

We used multiunit threshold crossing rates as neural features for analysis and neural decoding (as opposed to spike-sorted single units). We made this choice to simplify the methods, not because spike waveforms could not be recorded (see Extended Data Fig. 7 for example waveforms). Recent results suggest that neural population structure can be accurately estimated from threshold crossing rates alone (Trautmann et al., 2019), and that neural decoding performance is comparable (within 5%) to using sorted units (Chestek et al., 2011; Christie et al., 2014) – although see also (Todorova et al., 2014). For threshold crossing detection, we used a -3.5 x RMS threshold applied to each electrode, where RMS is the electrode-specific root mean square of the voltage time series recorded on that electrode. Threshold crossing times were "binned" into 10 ms bins (for analysis) or 20 ms bins (for decoding) to estimate the threshold crossing rate in each bin. For each bin, the estimated rate was equal to the number of threshold crossings in that bin divided by the bin width.

1.3 Overview of data collection sessions

Neural data were recorded in 3-5 hour "sessions" on scheduled days, which typically occurred 2-3 times per week. During the sessions, T5 sat upright in a wheelchair with his hand resting on his lap. A computer monitor placed in front of T5 indicated which sentence (or single character) to write and when. Data were collected in a series of 5-10 minute "blocks" consisting of an uninterrupted series of trials. In between these blocks, T5 was encouraged to rest as needed. The software for running the experimental tasks, recording data, and implementing the real-time decoding system was developed using MATLAB and Simulink Real-Time (MathWorks, Natick, MA). Table M1 below is an exhaustive list of all 11 data collection sessions reported in this work.

Session Number	Date (Post-Implant Day)	Description	Data
1	2019.05.08 (994)	Sentence-writing and character-writing pilot day (no real-time decoding)	• Sentence writing collected as training data (102 sentences, no decoding) • Single character writing (27 repetitions per character)
2	2019.06.03 (1020)	Attempted handwriting of straight lines	• Instructed delay straight-line writing (24 repetitions each for 48 straight-line conditions)
3	2019.11.25 (1195)	Real-time decoding pilot day	• Sentence writing for training data (50 sentences, no decoding) • Sentence writing for performance evaluation (34 sentences, with real-time decoding) • Single character writing (10 repetitions per character)
4	2019.12.09 (1209)	Real-time decoding pilot day	
5	2019.12.11 (1211)	Copy-typing evaluation	
6	2019.12.18 (1218)	Copy-typing evaluation	
7	2019.12.20 (1220)	Copy-typing evaluation	
8	2020.01.06 (1237)	Copy-typing evaluation	
9	2020.01.08 (1239)	Copy-typing evaluation	
10	2020.01.13 (1244)	Free-answer evaluation	• Sentence writing with artificial pauses (30 sentences, no decoding) • Phrase writing from memory (100 phrases, no decoding) • Sentence writing for performance evaluation (25 free-answer questions) • Single character writing (10 repetitions per character)
11	2020.01.15 (1246)	Free-answer evaluation	

Table M1. List of all data collection sessions included in this study.

1.4 Instructed delay paradigm

All tasks employed an instructed delay paradigm. For the single character writing task shown in Fig. 1a, the delay period duration was drawn from an exponential distribution (mean of 2.5 s); values that fell outside of the range of 2.0 – 3.0 s were re-drawn. After the delay period, the text prompt changed to "Go" and the red square (stop cue) turned green for 1 second, cueing T5 to begin attempting to write.

During sentence writing blocks, the delay period always lasted 5 seconds. During this delay period, the upcoming sentence was displayed on the screen, providing T5 time to read through it and prepare to write it. After the delay period, the red stop cue then turned green, and the sentence remained displayed on the screen while T5 attempted to handwrite it letter by letter. When T5 finished writing the sentence, he turned his head to the right, which our system detected and automatically triggered the next sentence. Head position was tracked optically with the OptiTrack V120:Trio bar (Corvallis, OR) containing three infrared cameras that tracked the position of optical markers worn on a head band.

1.5 Decoder evaluation sessions

Sentence-writing days where real-time decoding was tested (sessions 3-11) had the following structure (illustrated in Extended Data Fig. 2a). First, we collected interleaved blocks of single character writing (2 blocks, 5 repetitions of each character per block) and sentence writing (5-8 blocks with 10 sentences or 20 phrases per block); no decoder was active during these blocks. Then, we retrained the decoder using these blocks of data (combined with data from all past sessions). Finally, we collected evaluation blocks where T5 used the decoder to copy sentences (sessions 3-9, 4 blocks per session) or freely answer questions (sessions 10-11, 3 blocks per session). Note that the reported data in Fig. 2 are from sessions 5-9, since sessions 3-4 were pilot sessions devoted to exploring different decoding approaches.

Since no 'backspace' was implemented and T5 had no ability to correct errors, T5 reported spending most of his time looking at the prompt he was copying instead of watching the decoded letters appear on the screen (eye tracking data confirmed that T5 spent 93% of the time looking at the prompt during the copy typing task; Tobii 4C eye tracker).

Note that since the handwriting BCI is entirely self-paced, T5 determines the speed of the BCI (characters per minute) by choosing how quickly to attempt to write each character. We instructed T5 to proceed as quickly as possible. T5 reported to us that he increased his writing speed over time as he gained confidence that the BCI could maintain its accuracy at high speeds.

1.6 Sentence selection

<u>Copy typing sessions</u>

In each copy typing session (sessions 3-11), 5 blocks of training data (with 10 sentences each) were always collected before decoder evaluation for the purposes of decoder retraining (see Extended Data Fig. 2a for a session flow diagram). Sentences were chosen from the British

National Corpus (BNC) using the Sketch Engine tool. First, we randomly selected words from a list of the top 2,000 most common words in the BNC. Then, for each randomly chosen word, the BNC was searched for example sentences containing that word. From these examples, we hand-selected sentences of reasonable length (no more than 120 characters) and whose meaning was not too confusing out of context, so as not to be distracting to T5. The end result was a diverse sample of sentences from many different contexts (spoken English, fiction, non-fiction, news, etc.). Finally, we also included 5 pangrams (sentences containing all 26 letters) in each session's training data that did not appear in the BNC, in order to increase the frequency of rare letters.

After collecting training data, the RNN decoder was retrained and then evaluated on four evaluation blocks. Two of the four evaluation blocks always used the 7 sentences employed in (Pandarinath et al., 2017) for a direct comparison to this prior state-of-the-art point-and-click typing BCI (Supplementary Video 3). The other two evaluation blocks contained 10 unique sentences selected from the BNC (according to the same selection process described above).

Importantly, the RNN decoder was never evaluated on a sentence that it had been trained on, and every sentence was unique (except for the "direct comparison" blocks that always used the same 7 sentences from (Pandarinath et al., 2017)). When we retrained the decoder each day before performance evaluation, we retrained it using all previously collected data (from all prior days) *except* for these direct comparison blocks, in order to prevent the RNN from overfitting to these repeated sentences.

Free typing sessions

The two free-typing sessions (sessions 10-11) used 8 blocks of sentence-writing data for decoder training (instead of 5) and used a different set of sentences to add more realistic variability to the training data. For 3 of the 8 sentence-writing blocks, we randomly added hash mark characters (#) to sentences selected from the BNC, which signaled T5 to take a short, artificial pause from writing at different points in the sentence. For the other 5 blocks, we used 2-4 word phrases instead of complete sentences from the BNC, and asked T5 to write them from memory (instead of copying what was on the screen). To enforce writing from memory, we removed the phrase from the screen during the "Go" period. These features were designed to train the RNN to be robust to irregular writing speeds and unpredictable pauses that might occur more often during free typing.

2. Neural representation of attempted handwriting (Fig. 1)

2.1 PCA visualization and time-warping

To create the data visualization shown in Fig. 1b-c, threshold crossing rates were first binned into 10 ms bins and smoothed by convolving with a gaussian kernel (sd = 30 ms) to remove high frequency noise. To find the top 3 PCs used to visualize the neural activity, the smoothed rates were then compiled into a matrix of dimension N x TC, where N is the number of microelectrodes (192), T is the number of 10 ms time bins (200), and C is the number of characters (31). Each row contains the trial-averaged response of a single electrode to each character, in a time window from -500 ms to 1500 ms around the go cue (the 31 trial-averaged responses were concatenated together into a single vector). Principal components analysis was then applied to the columns of this matrix to find the top 3 PCs used for data visualization.

Next, we used time-warped PCA (https://github.com/ganguli-lab/twpca) (Poole et al., 2017; Williams et al., 2020) to find continuous, regularized time-warping functions that align all trials belonging to the same character together (Fig. 1c). We verified that these warping functions appeared close to the identity line, smoothly bending away from it after the go cue in order to account for variations in writing speed from trial to trial (as can be seen in the example shown in Fig. 1b-c). We used the following time-warping parameters: 5 components, 0.001 scale warping regularization (L1), and 1.0 scale time regularization (L2).

2.2 Pen trajectory visualization

Overview

To construct the character traces shown in Fig. 1d, we trained a linear decoder to readout pen tip velocity from the neural activity as follows:

$$v_t = Dx_t + b$$

Here, v_t is a 2 x 1 pen tip velocity vector containing X and Y velocity at time t, D is a 2 x 192 decoding matrix, x_t is a 192 x 1 vector of binned threshold crossing rates, and b is a 2 x 1 offset term. Importantly, decoding was cross-validated by holding out each character in a leave-one-out fashion. That is, the pen tip velocities for any given character were obtained using a decoder that was trained on all *other* characters, preventing the decoder from trivially overfitting to high-dimensional neural data.

To train the decoder, we used hand-made templates that describe each character's pen trajectory. The character templates were made by drawing each character with a computer mouse in the same way as T5 described writing the character. As each character was drawn, the X and Y velocity trajectories of the mouse pointer were recorded. These templates then defined the target velocity vector for the decoder on each time step of each trial, much like prior work has trained decoders to predict the user's "intended" velocity for continuous movement tasks (Collinger et al., 2013; Gilja et al., 2015). These templates were only intended to be a rough approximation of T5's intended pen tip velocities, based on the assumption that another person

drawing the same character shape with a computer mouse would naturally follow a similar velocity trajectory (up to some time-scaling factor, to account for differences in overall writing speed). Nevertheless, the reconstructed pen tip velocities that were decoded in Fig. 1d were well correlated with the mouse templates (r = 0.74 across all characters).

Since drawing the characters with a computer mouse may not yield the same overall writing speed and reaction times as T5, the reaction time and time-scaling factor for each velocity template must also be optimized along with the decoder. We therefore trained the decoder in an iterative process as follows:

(1) Train the linear decoder using the currently-identified reaction time and time-scaling factors, using ordinary least squares regression to minimize the error between the template velocities and the decoded velocities.

(2) Apply the newly-trained decoder to the neural data to generate decoded velocities.

(3) Optimize the reaction times (template start times) and time-scaling factors (linear time stretching/shrinking) via a grid search to best match the current decoded velocities.

(4) Return to Step 1 until 3 iterations have been performed.

Finally, to visualize the pen trajectories for each character, the decoder was applied to time-warped and trial-averaged neural activity. Trial-averaging denoises the decoder output, while time-warping aligns all the trials together in time so that they can be averaged without losing detail. The decoded velocity was then integrated to yield a pen tip position trajectory.

Implementation details

Each trial of neural activity was represented by a matrix of binned and smoothed threshold crossing rates with dimensions 200 x 192 (200 time steps and 192 electrodes) taken from a -0.5 to 1.5 second window around the go cue. Then, these matrices were concatenated vertically to form a predictor matrix for the linear regression of size 200*N x 192, where N is the total number of trials (N=864). Finally, a column of ones was added to the predictor matrix to create a constant offset term, yielding a design matrix *X* of dimension 200*N x 193. We also constructed a response matrix *Y* of dimension 200*N x 2 containing the template velocity vectors at each time step that the decoder should predict.

For the first iteration, *Y* was constructed by setting the entries of each trial equal to the velocity vectors in that character's template. The decoder coefficient matrix *B* was then computed with ordinary least squares to minimize the following mean squared error cost function:

$$\|Y - XB\|_F^2$$

For the subsequent two iterations, we used the decoded velocity vectors *XB* to time-scale and shift the character templates before constructing *Y*. We performed a grid search for each character, searching for possible template shift times between -0.4 and 0.4 seconds and

possible template time-scaling factors between 0.5 and 2.0. Templates were time-scaled by resampling with linear interpolation to stretch/shrink them in time. The shift-time and time-scale factors that maximized the correlation (Pearson's *r*) between the template and the previously decoded velocity vectors was chosen.

2.3 Fraction of variance accounted for by pen velocity

To estimate how much of the neural activity can be explained by pen tip velocity (30%), we computed the fraction of variance accounted for in the *trial-averaged* (and time-warped) neural activity by a linear encoding model that describes neural activity as a linear function of the decoded pen tip velocity trajectories.

To denoise the result, we only analyzed neural activity in the top 10 neural dimensions found by principal components analysis (see above section 2.1 for PCA details). Otherwise, noise in higher dimensions can dominate the result.

The encoding model was fit with ordinary least squares regression and can be described as follows:

$$f_t = Ev_t + b$$

Here, f_t is a 10 x 1 vector of trial-averaged neural activity in the top 10 neural dimensions, E is a 10 x 2 encoding matrix (of preferred directions), v_t is a 2 x 1 pen tip velocity vector containing reconstructed X and Y velocity at time t, and b is a 10 x 1 offset term. We included all neural activity within a time window from 100 ms after the "Go" cue up until when each letter was finished being written (as determined by visual inspection of the pen tip trajectories).

The fraction of variance accounted for (FVAF) by the pen tip velocity was then computed as follows:

$$FVAF = 1 - \frac{SS_{err}}{SS_{tot}} = 1 - \frac{\sum_{t=1}^{T}(Ev_t - f_t)^2}{\sum_{t=1}^{T} f_t^2}$$

Here, T is the total number of time steps, SS_{err} is the sum of squared errors, and SS_{tot} is the total variance.

Because only 2D pen tip velocity was modeled (i.e., lifting-off-the-page motion was not considered here), we only included characters that required no pen-lifting in this analysis (a, b, c, d, e, g, h, l, m, n, o, p, q, r, s, u, v, w, z, >, ~).

2.4 t-SNE and k-NN classifier

For Fig. 1e, we used t-distributed stochastic neighbor embedding (t-SNE) (Maaten and Hinton, 2008) to nonlinearly reduce the dimensionality of trials of neural activity for visualization (perplexity=40). Before applying t-SNE, we smoothed the neural activity and reduced its

dimensionality to 15 with PCA (using the methods described above). Each trial of neural activity was thus represented by a 140 x 15 matrix (140 time bins by 15 dimensions, with the time window spanning 0.1 to 1.5 seconds after the go cue). We applied t-SNE to these matrices using the following "time-warp" distance function:

$$d(X,Y) = \underset{\alpha \in [0.7, 1.42]}{\operatorname{argmin}} \|X - f(Y,\alpha)\|_F^2$$

Here, α is a time-warp factor, *f* is a warping function, and *X* and *Y* are trials of neural activity. The function *f* time-warps a trial of neural activity by resampling with linear interpolation by a factor of α. After warping, only the first N shared time bins between *X* and *Y* are used to compute the distance. The warping function serves to account for differences in writing speed across trials, so that the same pattern of neural activity occurring at a different speed is considered nearby.

We also used the time-warp distance function to perform k-nearest neighbor classification (k=10), which resulted in 94.1% accuracy (compared to 88.8% accuracy with a Euclidean distance function). The 95% confidence interval reported was a binomial proportion confidence interval (Clopper-Pearson).

3. Decoder performance metrics

3.1 Error rate and characters per minute

Character error rate was defined as the edit distance between the decoded sentence and the prompt (i.e., the number of insertions, deletions or substitutions required to make the strings of characters match exactly). Similarly, word error rate was the edit distance defined over sequences of "words" (strings of characters separated by spaces; punctuation was included as part of the word it appeared next to). For the free typing sessions, the intended sentence was determined by discussing with the participant his intended meaning.

Note that the reported error rates are the combined result of many independent sentences. To combine data across multiple sentences, we summed the number of errors across all sentences and divided this by the total number of characters/words across all sentences (as opposed to computing error rate percentages first for each sentence and then averaging the percentages). This helps prevent very short sentences from overly influencing the result.

Characters per minute was defined as 60N/(E-S), where N was the number of characters in the target sentence, E was the end time and S was the start time (in seconds). For copy typing, S was the time of the go cue and E was the time of the last decoded character. Rarely, T5 had lapses of attention and did not respond to the go cue until many seconds later; we thus capped his reaction time (the time between the go cue and the first decoded character) to no more than 2 seconds. For the free typing sessions, we defined S as the time of the first decoded character (instead of the go cue), since T5 often took substantial time after the go cue to formulate his response to the prompt.

3.2 Data exclusion

We removed a small number of trials with incomplete data when reporting decoder performance (9 out of 220 = 4%). Three copy typing trials were removed because T5 accidentally triggered the next sentence by moving his head too far to the right before he had finished typing the sentence (our system triggered the next sentence upon detection of a rightward head turn). During free typing, we removed one sentence where T5 could not think of a response and wanted to skip the question, and one sentence where we were unable to determine T5's intended spelling of a restaurant name. Finally, in one copy typing session, a software bug incorrectly initialized the RNN decoder at the beginning of each block; for this session, we excluded the first trial of each block (4 trials total).

3.3 Able-bodied smartphone typing rate

To estimate the able-bodied smartphone typing rate of people in T5's age group (115 characters per minute, as mentioned in the Summary), we used the publicly available data from (Palin et al., 2019). We took the median over all participants greater than 60 years old (T5 was 65 at the time of data collection).

4. RNN architecture

We used a two layer, gated recurrent unit RNN (Cho et al., 2014) to convert T5's neural activity into a time series of character probabilities (see Extended Data Fig. 1 for a diagram). We found that a recurrent neural network decoder outperformed a simple hidden Markov model (HMM) decoder (see section "7. Comparison to an HMM decoder" for more details).

As a pre-processing step, threshold crossing rates were binned in 20 ms time steps, z-scored, causally smoothed by convolving with a Gaussian kernel (sd = 40 ms) that was delayed by 100 ms, and concatenated into a 192 x 1 vector x_t. We used the following variant of the gated recurrent unit RNN that is implemented by the cuDNN library(Chetlur et al., 2014):

$$r_t = \sigma(W_r x_t + R_r h_{t-1} + b_{Wr} + b_{Rr})$$
$$u_t = \sigma(W_u x_t + R_u h_{t-1} + b_{Wu} + b_{Ru})$$
$$c_t = \sigma_h(W_h x_t + r_t * (R_h h_{t-1} + b_{Rh}) + b_{Wh})$$
$$h_t = (1 - u_t) * c_t + u_t * h_{t-1}$$

Here, σ is the logistic sigmoid function, σ_h is the hyperbolic tangent, x_t is the input vector at time step t, h_t is the hidden state vector, r_t is the reset gate vector, u_t is the update gate vector, c_t is the candidate hidden state vector, W, R and b are parameter matrices and vectors, and * denotes the element-wise multiplication.

We used a two-layer RNN architecture (where the hidden state of the first layer was fed as input to the second layer). Importantly, the RNN was trained with an output delay. That is, the RNN was trained to predict the character probabilities from 1 second in the past; this was necessary to ensure that the RNN had enough time to process the entire character before deciding on its identity. The output probabilities were computed from the hidden state of the second layer as follows:

$$y_t = \mathrm{softmax}(W_y h_t + b_y)$$
$$z_t = \sigma(W_z h_t + b_z)$$

Here, σ is the logistic sigmoid function, h_t is the hidden state of the second layer, W and b are parameter matrices and vectors, y_t is a vector of character probabilities (one entry for each character), and z_t is a scalar probability that represents the probability of any new character beginning at that time step. During real-time operation, we thresholded z_t (threshold = 0.3) to decide when to emit a new character. Whenever z_t crossed the threshold, we emitted the most probable character in y_t 300 ms later.

We updated the second layer of our RNN decoder at a slower frequency than the first layer (every five 20 ms time steps instead of every single time step). We found that this increased the speed and reliability of training, making it easier to hold information in memory for the length of the output delay (e.g., for a 1 second delay, the slower frequency means that the top layer must hold information in memory for only 10 steps as opposed to 50).

4.1 Estimated decoding latency

We estimate that characters were emitted between 0.4-0.7 seconds after T5 completed them. For a 90 characters per minute typing rate, characters take on average 60/90 = 0.66 seconds to complete. After a new character begins, our decoder takes 1 second to recognize this (as it is trained with 1 second delay). Thus, it will emit a new character start signal (z_t) ~1 second after the character begins; adding an additional 0.1 seconds for the causal gaussian smoothing delay and 0.3 seconds to emit the character after z_t crosses threshold yields a 1.4 second delay. This means the character will appear on the screen ~ 1.4 – 0.66 = 0.74 seconds after T5 completes it. For slower typing rates (e.g., 60 characters per minute), the delay is shorter (~0.4 seconds).

5. RNN training overview

Here, we give an overview of the main steps and algorithms used to train the RNN (see Extended Data Fig. 2b for a diagram of the main training steps). For more details, the next section "6. RNN Training Details" provides a complete protocol, and our publicly released code also implements these training methods. Our methods are an adaptation of neural network methods used in automatic speech recognition (Hinton et al., 2012; Graves et al., 2013; Zeyer et al., 2017; Xiong et al., 2017), with key changes to achieve high performance on neural activity in a highly data-limited regime (1-10 hours of data, compared to 1-10k hours, e.g. (Cieri et al., 2004; Panayotov et al., 2015; He et al., 2019)).

5.1 Data labeling

A major challenge that had to be overcome for training decoders with our data is that we don't know what character T5 was writing at any given moment in time in the training data. There are two major approaches used to solve this problem in automatic speech recognition: forced-alignment labeling with hidden Markov Models (HMMs) (Young et al., 2006; Hinton et al., 2012; Xiong et al., 2017), or unsupervised inference with connectionist temporal classification (Graves et al., 2006) (and other similar cost functions, e.g. (Collobert et al., 2016)). We found that forced-alignment worked better with our data, potentially because of the relatively small dataset size; it also enabled data augmentation via synthetic sentence generation (see below). In the forced-alignment method, HMMs are used to infer what character is being written at each time step, fusing knowledge of the sequence of characters that were supposed to be written with the neural activity recorded. These character labels can then be used to construct target probabilities that the RNN is trained to reproduce in a supervised manner.

To construct the data-labeling HMMs, we first processed the single character data to convert it into trial-averaged spatiotemporal "templates" of the neural activity patterns associated with each character. Next, these templates were used to define the emission probabilities of the HMMs, and the states and transition probabilities were set to express an orderly march through the sequence of characters in each sentence (Extended Data Fig. 2c). We then used the Viterbi algorithm to find the most probable start time of each character given the observed neural activity. The start times of each character could then be used to construct target time series of character probabilities for the RNN to reproduce.

The vector of target character probabilities (denoted as y_t above) was constructed by setting the probability values at each time step to be a one-hot representation of the most recently started character (i.e., the most recently started character's entry in the vector is equal to 1 while all other entries are 0). The scalar character start probability (denoted as z_t above) was set to be equal to 1 for a 200 ms window after each character began, and was otherwise equal to 0. The character start probability allows the decoder to distinguish repeated characters from single characters (e.g., "oo" vs. "o").

One advantage of this strategy for representing the RNN output is that uncertainty about whether pauses are occurring between characters should not degrade performance, since the

labeling routine only needs to identify when each character begins (not when it ends). Note that this representation causes the RNN to output a "sample-and-hold"-type signal, where it will continue to output the most recently started character until the next one begins.

5.2 Supervised training

Once the data were labeled, we used those labels to cut out snippets of each character from the data. These snippets were then re-assembled into artificial sentences, which were added to the training data to augment it and prevent overfitting (Extended Data Fig. 2e). Although this method is simplistic in assuming that the neural representation of a character is independent of past and future characters, it was nevertheless important for achieving high performance (including augmented data decreased the error rate percentage by 12.9 when training on single days and 2.7 when training on all days; Extended Data Fig. 3a). Finally, with the labeled and augmented dataset in hand, we used TensorFlow v1.15(Abadi et al., 2016) to train the RNN with gradient descent (using Adam (Kingma and Ba, 2017)) following standard supervised training approaches. To train the RNN to account for changes in the *means* of neural features (i.e. their baseline firing rates) which naturally accrue over time (Downey et al., 2018; Jarosiewicz et al., 2015), we added artificial perturbations to the feature means (similar to (Sussillo et al., 2016)). This step was also essential to achieving high-performance (it decreased the error rate percentage by 5.7; Extended Data Fig. 3b).

On each new day, we re-trained the RNN to incorporate that new day's data before doing real-time performance evaluation. The new data were combined with all previous days' data into one large dataset while training. To account for differences in neural activity across days (Degenhart et al., 2020; Jarosiewicz et al., 2015), we separately transformed each days' neural activity with a linear transformation that was simultaneously optimized with the other RNN parameters (see "Combining data across days" in section 6.5 for more detail). Including multiple days of data, and fitting separate input layers for each day, significantly improved performance (decreased the error rate percentage by 4.7 and 1.6, respectively; Extended Data Fig. 3c-d).

Hyperparameter values were largely hand-tuned; for later sessions, some parameters were tuned via small random searches over possible parameter values. Ultimately, automated parameter tuning may be required (and would certainly be useful) when applying these techniques to new participants in future clinical applications.

5.3 Unsupervised training

For the offline analysis shown in Fig. 3b, we used an un unsupervised method to train the RNN using all 50 sentences of training data collected at the beginning of each day before real-time performance evaluation. This was meant to simulate the effect of running an "in-the-background" unsupervised training routine that updates the decoder as the user writes sentences normally (and does not rely on any prior knowledge of the characters in each sentence). First, we applied the previous sessions' RNN to the calibration data without any retraining, which simulates what would be decoded if the user begins the day with an old RNN. The output of that RNN was then fed through our language model (see below) which infers the

most likely character that was written at each time step given the statistics of the English language as well as the RNN output. These language model inferences were then used to construct character probability targets to retrain the RNN, using the same supervised process as described above in “Supervised Training”. This procedure can be expected to improve the RNN by teaching it not to make any errors that the language model was able to properly detect and fix.

6. RNN training details

The RNN training procedure consists of four stages, as diagrammed in Extended Data Fig. 2b. Note that we have publicly released code that implements the RNN training procedure (https://github.com/fwillett/handwritingBCI).

The training procedure was designed to address two main challenges: (1) inferring when each character was written in the training data, so that supervised learning techniques could be used to train the RNN, and (2) augmenting the training data with synthetic sentences, to prevent the RNN from overfitting to limited data.

6.1 Data preprocessing

The single character data were pre-processed by binning the recorded threshold crossings into 10 ms bins. The binned rates were then z-scored (by subtracting the mean of each electrode and dividing by its bin-by-bin standard deviation) and smoothed by convolving with a gaussian kernel (sd = 30 ms).

The sentence data were also binned (10 ms bins), smoothed by convolving with a gaussian kernel (sd = 40 ms), and z-scored. For the sentence data, z-scoring was performed by applying the means and standard deviations from all trials of the single character data. During real-time decoding, these same means and standard deviations were also used to z-score the data.

6.2 Stage 1: Single character time warping & averaging

For each session, at least two blocks of single character data were collected as part of the training data (10 repetitions of each character total). We then used time-warped PCA (https://github.com/ganguli-lab/twpca)(Poole et al., 2017; Williams et al., 2020) to find continuous, regularized time-warping functions to align all trials corresponding to a single character together. We used the following time-warping parameters: 5 components (temporal factors), 0.001 scale warping regularization (L1), and 1.0 scale time regularization (L2).

After time-warping, the neural activity was averaged across trials for each character, yielding an N x T mean neural activity matrix for each character. N is the number of microelectrodes (192) and T is the number of time steps. The time window to use for each character was chosen by visual inspection of the character shapes, using the data from session 1 (i.e., the shapes shown in Fig. 1). After the character durations were chosen, they were held fixed for all subsequent sessions. The first 100 ms of reaction time after the go cue was excluded, yielding a time window for each character that spanned 100 ms after the go cue until the end time chosen by inspection. Finally, the neural activity was then down-sampled to 50 ms time steps by averaging every 5 bins together, yielding single character “neural templates” that were used to build HMMs for data labeling (see below).

6.3 Stage 2: Sentence labeling with hidden Markov models

Data labeling was accomplished in 6 steps as follows:

(1) Construct an HMM for each sentence, using the neural templates from the single character data.
(2) Use the Viterbi algorithm to infer when each character began and ended in each sentence.
(3) Refine the character start times and durations using a local grid search.
(4) Update the HMM emission probabilities (but not the state transition probabilities) using the inferred character start times and durations.
(5) Repeat Steps 2-3.
(6) Construct the final targets for supervised RNN training (y_t and z_t), using the letter start times found in (5).

Step 1: HMM construction

We used hidden Markov models (HMMs) to label our data, similar to how HMMs have been used in speech recognition to determine when phonemes begin and end in an utterance where the transcription is known (this is called "forced alignment") (Young et al., 2006). For each sentence of training data, we constructed an HMM whose states and state transitions defined an orderly march through the characters of that sentence (Extended Data Fig. 2c). Each individual character was represented with a sequence of HMM states, whose multivariate Gaussian emission probabilities were determined by the single character neural templates. Specifically, for each character, the number of HMM states was equal to the number of 50 ms bins in the neural template for that character (plus an additional "blank" state), and the mean vector for each state's emission distribution was equal to the corresponding firing rate vector from the neural template. The covariance matrix was set equal to the identity matrix.

Let us denote the states of the HMM model as $s_{i,j}$, where j denotes the character number in the sentence, and i iterates through the states within each character. Table M2 below lists the state transition probabilities for states $s_{1,j}$ to $s_{N,j}$ (and the optional blank state B_j) for each character, which were hand-tuned to reasonable values. In the expressions below, N is the number of states in character j, and M is the total number of characters in the sentence.

States	Description	Transition Probabilities
$\boldsymbol{s_{x,j}}$ **for x<N-1**	All states before the second to last, for character j	$P(s_{x,j} \rightarrow s_{x,j}) = 0.2$ $P(s_{x,j} \rightarrow s_{x+1,j}) = 0.6$ $P(s_{x,j} \rightarrow s_{x+2,j}) = 0.2$
$\boldsymbol{s_{N-1,j}}$	Second to last state for character j	$P(s_{N-1,j} \rightarrow s_{N-1,j}) = 0.2$ $P(s_{N-1,j} \rightarrow s_{N,j}) = 0.8$
$\boldsymbol{s_{N,j}}$	Last state for character j	$P(s_{N,j} \rightarrow s_{N,j}) = 0.2$ $P(s_{N,j} \rightarrow B_j) = 0.1$ $P(s_{N,j} \rightarrow s_{1,j+1}) = 0.7$
$\boldsymbol{B_j}$	Blank state for character j	$P(B_j \rightarrow B_j) = 0.5$ $P(B_j \rightarrow s_{1,j+1}) = 0.5$

$s_{N,M}$	Last character state in the sentence	$P(s_{N,M} \rightarrow s_{N,M}) = 0.7$ $P(s_{N,M} \rightarrow B_M) = 0.3$
B_M	Last blank state in the sentence	$P(B_M \rightarrow B_M) = 1.0$

Table M2. Hidden Markov model parameters.

Note that B_j is a "blank" state that can be entered into at the end of the jth character (or skipped) and can model pauses in between characters. The emission vector for all blank states was set to the average neural activity vector across all character templates.

Step 2: Viterbi algorithm

We used the Viterbi algorithm (Rabiner, 1989) to find the most likely sequence of HMM states given the observed neural activity, with the constraint that the last state of the sequence was the last character's final state ($s_{N,M}$) or the last blank state (B_M) (this enforces that the sentence is completely finished). This constraint was implemented by setting the observation probability to zero for all other states at the final time step. We added an additional constraint that helped prevent pathological solutions: each state had to occur within a certain window of time centered on its character's location in the sentence. Let us denote the duration of the entire sentence as T. Each character j's time window was centered on the time $(j/M)T$ and extended $0.3T$ in either direction. For example, if the character "m" occurred in the middle of the sentence, then the states for this "m" had to occur between times $0.2T$ and $0.8T$. Similarly, if the character "t" occurred in the beginning of the sentence, it had to occur between times 0 and $0.3T$. This constraint was implemented by setting a states' observation probability to zero if it lied outside of this window.

Step 3: Local refinement of character start times

The sequence of states found by the Viterbi algorithm define the start time and duration of each character. The quality of fit can be roughly assessed with correlation heatmaps that show the correlation (Pearson's r) between the neural template for a character and the observed neural activity, as a function of character start time and character "stretch factor" (linear time dilation factor) The identified start time and duration should lie on a hotspot (Extended Data Fig. 2d). For these heatmaps, the correlation coefficient was computed for each microelectrode channel separately; the resulting 192 correlation coefficients were then averaged together to produce a final value.

We implemented a refinement step after the Viterbi search which maximized the correlation of each character with the observed activity via a grid search of adjusted start times and template stretch factors. The grid search varied the possible start times from 0.5 seconds before to 0.5 seconds after the HMM-identified time (in steps of 0.05 seconds). The stretch factor varied from 0.4 to 1.5 in steps of 0.0786. The stretch factor determines how the character template is contracted or dilated in time (using linear interpolation) to be longer or shorter than its average duration. Values that caused the adjusted character template to intersect adjacent character

templates were not considered. This refinement procedure effectively placed each character on a nearby maximum in the heatmaps shown in Extended Data Fig. 2d.

Step 4: Updating HMM emissions

We updated the HMM emission probabilities based on the newly labeled sentence data. The emission probabilities were updated in the following way. First, for each character class, all example snippets of that character were gathered together based on the character start times and durations found above. Then, each example was time-normalized by resampling to N time steps (using linear interpolation, where N was the original number of time steps in the template). Then, the time-normalized examples were averaged together to compute a new neural template for that character. The emission probabilities were not updated for characters with less than 18 examples (e.g., rare characters such as “q” or “x”).

In principle, the state transition probabilities of the HMM could also be updated (e.g. by using the Baum-Welch algorithm). However, we did not explore that here, as we found that updating the emission probabilities alone seemed sufficient to yield high quality labels.

Step 5: Repeat with new HMM emissions

With the updated emissions, we performed one additional iteration of HMM labeling and subsequent refinement (further iterations did not seem to improve label quality).

Step 6: Construct RNN targets

Finally, target variables for supervised RNN training were generated using the letter start times found above. Two target time series were created: a series of one-hot character vectors (y_t), where each vector is a one-hot representation of the most recently started character, and a scalar time series (z_t) that indicates whether *any* new character has recently been started. The z_t signal allows repeated characters to be distinguished (these would otherwise appear identical to a longer, single character as seen through y_t).

Intuitively, y_t is a “sample and hold” signal that stores whatever the most recently started character was indefinitely. For example, even if T5 pauses for several seconds after writing the character “a”, y_t will still continue to reflect “a” indefinitely until a new character is started. The z_t signal is a complementary binary signal that goes high for a brief time whenever *any* new character begins. z_t can be thresholded to detect the presence of new letters and type them on the screen, which we did online. More formally, y_t and z_t were defined as follows:

$$y_{t,i} = \begin{cases} 0, & \text{the most recently started character was not i} \\ 1, & \text{the most recently started character was i} \end{cases}$$

$$z_t = \begin{cases} 0, & \text{the most recent character was started} > 200 \text{ ms ago} \\ 1, & \text{the most recent character was started} \leq 200 \text{ ms ago} \end{cases}$$

One potential advantage of this nontraditional representation is that only the character start times are required; thus, any uncertainty about when each character ends shouldn't degrade performance (i.e., uncertainty about the length of time spent transitioning between letters, either with long pauses or short bouts of pen repositioning). Additionally, by not including multiple sub-states per character (which could be an alternative way to distinguish repeated letters), this method gives the RNN freedom to decide how to break apart each character into sub-states.

6.4 Stage 3: Synthetic data generation

Once the character start times were inferred for each sentence, we generated new synthetic sentences by rearranging the characters into different sequences (Extended Data Fig. 2e). This data augmentation step improved decoding performance (Extended Data Fig. 3a), as it helped to prevent the RNN from overfitting to a limited training dataset. Here, we describe the synthesis process in detail.

Making the snippet library

First, we created a snippet library of neural activity snippets for each character. Entries were taken by extracting time windows of activity from the sentences data, beginning at the letter start time identified by the HMM procedure and ending at the start time of the next letter. In this way, any pauses and transition-related activity are included at the end of the snippets.

Generating random sentences

Next, we used the snippet library to generate synthetic sentences (24 seconds long, which was the length of data used in each minibatch during RNN training). First, the character sequence for each sentence was chosen by selecting words at random from a list of 10,000 common words (the 10,000 most frequent words appearing in the Google Web 1T 5-gram database) (Brants and Franz, 2006). Words were selected one at a time, with no dependence on the prior word, according to the following simple heuristic:

- 64% chance: a word was chosen uniformly at random from the entire list
- 20% chance: one of the twenty most frequent words was chosen uniformly at random
- 16% chance: a word with rare letters ("q", "x", "j", or "z") was chosen uniformly at random from the set of all such words (this helped prevent the RNN from neglecting rare characters)

To make sure punctuation characters were represented, apostrophes were randomly added in between the last and second-to-last letter of the word (3% chance) and commas were randomly added at the end of the word (7% chance). All words either ended in a period (5% chance), question mark (5% chance), or space (90% chance).

We used the above heuristic to generate sentences instead of using real sentences in order to discourage the RNN from "baking-in" a model of the English language that extends beyond single words.

Synthesizing the neural activity

Once the synthetic character sequence was determined, the corresponding neural activity was synthesized one character at a time. For each character, a snippet was chosen from the library at random in a way that attempted to respect pen transition movements between letters. For example, when transitioning from “e” to “t”, the pen must traverse upwards before beginning the downstroke for “t”. However, when transitioning from “d” to “t”, no such pen re-positioning is needed (when written in the way shown in Fig. 1). To do this, we discretized the starting heights for each character to the following values: 0, 0.25, 0.5, 1. The assignment of each letter to each category is depicted in Table M3 below.

Start Height	**0**	**0.25**	**0.5**	**1.0**
Character	comma	a, o, e, g, q	c, d, m, j, i, n, p, r, s, u, v, w, x, y, z, space (>), period (~)	b, t, f, h, k, l, apostrophe, question mark

Table M3. Assignment of characters to starting-height categories.

When choosing a snippet from the library, we selected at random from all snippets whose next character in the training data began at the same height as the next character in the synthetic sentence. When this wasn’t possible, we selected uniformly at random from all snippets.

After a snippet was chosen, we randomly time-warped the snippet by resampling it to a different length of time (chosen uniformly from 0.7 to 1.3 times its original length). This helps the RNN to be more robust to changes in letter timing. Finally, we also sometimes added a long pause at the end of the snippet at random, to train the RNN to be robust to unpredictable pauses made by the user. The probability of adding a pause was 3%; if a pause was added, its duration was drawn from an exponential distribution (mean of 1 second). The synthetic neural activity during the pause was white noise with a standard deviation of 1.

6.5 Stage 4: Supervised RNN training

The RNN architecture is described in the Methods and illustrated in Extended Data Fig. 1. Here, we describe in detail how the RNN weights were optimized using supervised learning.

Implementation

Optimization of the RNN weights was implemented with TensorFlow v1.15 (Abadi et al., 2016). We used a desktop machine with 4 NVIDIA GeForce GTX 1080 Ti GPUs and a 32-core AMD RYZEN Threadripper 2990WX CPU to train the RNNs. To train a single RNN, only a single GPU was used, allowing parallel training of up to 4 RNNs. A single minibatch took ~0.25 seconds to complete, resulting in training times ranging from 4 minutes (1k minibatches, when updating the RNN to a new day of data) to 3.5 hours (50k minibatches, when training from scratch). Note that the number of sentences used for training was small (572 by the last day of copy typing), so only a few minibatches were required to cycle through all sentences.

Mini-batches & gradient descent

Gradients were computed on mini-batches of 64 data snippets using backpropagation through time (Goodfellow et al., 2016). We used the gradient descent method "Adam" (Kingma and Ba, 2017) (beta1 = 0.9, beta2 = 0.999, epsilon = 0.01). The learning rate was decreased linearly from 0.01 to 0 over a pre-specified number of minibatches (1k when updating a pre-trained RNN with a new day of data, 50k when training from scratch). To prevent gradient explosion, gradient magnitudes were clipped at 10.

Each data snippet used in a mini-batch was 24 seconds long and was selected at random from the training sentences by uniformly drawing a random start time from 0 to τ-24, where τ is the duration of the sentence. Each minibatch mixed together synthetic sentences and real sentences, in a proportion that was tuned to optimize performance (beginning at 75% synthetic and ending at ~40%). Each mini-batch selected data snippets from a single day only, which was chosen at random amongst all available days of training data. We weighted the most recent day more highly.

Cost function

We used the following cost function for a single snippet of data, which expresses the sum of an L2 weight regularization, a cross-entropy loss, and a squared error loss:

$$\lambda \sum_i \|W_i\|_F^2 - \frac{1}{T}\sum_{t=50}^{T}\sum_{c=1}^{C} y_{t,c}\log \hat{y}_{t+d,c} + \frac{1}{T}\sum_{t=50}^{T}(z_t - \hat{z}_{t+d})^2$$

Here, λ scales the L2 regularization of the RNN weight matrices W_i (penalizing large weights), *T* is the number of time steps in the data snippet (1200, 20 ms time steps), *C* is the number of characters (31), $y_{t,c}$ is a one-hot representation of the most recently started character at time step *t*, $\hat{y}_{t+d,c}$ is the RNN's prediction of $y_{t,c}$ (*d* time steps in the future, d=50), z_t is a scalar representation of whether a character was started within the last 200 ms, and $\hat{z}_{t+d}$ is the RNN's prediction of z_t.

The one second delay (*d*) between the RNN output ($\hat{y}_{t+d,c}$, $\hat{z}_{t+d}$) and the target signals ($y_{t,c}$, z_t) was added to give the RNN enough time to observe all of the neural activity corresponding to a character before deciding on its identity. Note also that there is a one second "burn-in" time before the error is counted, to ensure that the RNN is not penalized for incorrectly identifying characters at the very beginning of the snippet that may begin somewhere in the middle of the character (since snippets start at random times).

Noise perturbations

We added two types of artificial noise to the neural features in order to regularize the RNN. First, we added white noise directly to the input feature vectors, which greatly improved performance (Extended Data Fig. 3a, middle panel). Adding white noise to the inputs asks the RNN to map clouds of similar inputs to the same output, improving generalization. The standard

deviation of the white noise was an important hyperparameter that we tuned to optimize performance (see parameter values below).

We also added artificial changes to the means of the neural features, to make the RNN robust to non-stationarities in the neural data (which has been an important problem for intracortical BCIs (Jarosiewicz et al., 2015; Sussillo et al., 2016; Degenhart et al., 2020)). These artificial mean changes greatly improved the RNN's ability to generalize to held-out blocks of data occurring later in a session (Extended Data Fig. 3b). We added two types of perturbations to the neural features to simulate non-stationarities: constant offset noise and random walk noise.

The above-mentioned types of noise (white noise, constant offset noise, and random walk noise) were all combined together to transform the input vector in the following way:

$$\tilde{x}_t = x_t + \varepsilon_t + \varphi + \sum_{i=1}^{t} \upsilon_i$$

Here, x_t are the original neural features, ε_t is a white noise vector unique to each time step, φ is a constant offset vector, and υ_t are white noise vectors that are cumulatively summed to simulate a random walk.

Combining data across days

Combining multiple days of data together greatly improved performance relative to using just a single day (Extended Data Fig. 3c). To combine data across days, we optimized day-specific affine transforms of the input that could account for changes in the neural features across days (as observed previously in e.g. (Jarosiewicz et al., 2015; Downey et al., 2018; Degenhart et al., 2020)):

$$\acute{x}_t = A_i \tilde{x}_t + b_i$$

Here, $\tilde{x}_t$ is a vector of neural features (after artificial noise has been added, see above), A_i is a 192 x 192 matrix, and b_i is a 192 x 1 vector. The transformed features were then fed as input to the RNN. The A_i and b_i parameters are optimized simultaneously along with all other RNN parameters. Using separate input layers for each day improved performance relative to using a single shared input layer (Extended Data Fig. 3d).

Hyperparameters

We summarize the hyperparameters used for RNN training in the table below. Most parameters were hand-tuned; in later days, we performed small hyperparameter optimization routines where we trained 100 RNNs with parameters drawn at random (from pre-specified lists of reasonable values). Parameters for the next session were then taken from the top performing RNN. Thus, sometimes different hyperparameter values were used on different sessions; in particular, the amount of regularization decreased as more data were accumulated (the fraction

of synthetic sentences and the white noise became smaller). Since parameters varied, we summarize their typical ranges and values in Table M4 below (rather than exhaustively list each combination for each session).

Parameter	Description	Typical Value(s)
λ	L2 Weight Regularization	1e-5
H	Hidden state size	512
σ	Standard deviation of white noise	1.0 – 1.6 (Extended Data Fig. 3a)
γ	Standard deviation of constant offset noise	0.6 (Extended Data Fig. 3b)
ζ	Standard deviation of random walk noise	0.02
χ	Fraction of synthetic trials used in each mini-batch	0.375 – 0.75 (Extended Data Fig. 3a)
ω	Probability of a min-batch drawing sentences from the most recent day	0.25 – 0.5
α	Learning rate	0.01
M	Number of min-batches to train	1k (updating for new day), 50k (training from scratch across all days)
N	Min-batch size	64
T	Duration of data snippets in each mini-batch	24 seconds

Table M4. RNN hyperparameters.

6.6 RNN parameter sweeps and variants

Retrospectively, we tested the effect of five important parameters on RNN performance (Extended Data Fig. 3): (1) the amount of synthetic data used during training, (2) the amount of artificial white noise added to the inputs during training, (3) the amount of artificial feature *mean* noise ("drift") added during training (which simulates slow changes in baseline firing rates that accrue over time), (4) whether the RNN was trained with all days of data or just a single day, and (5) whether multiple days were combined with separate input layers or the same input layer. The results confirm that synthetic data, input white noise, feature mean noise, and combining data across days with separate input layers were all essential for high performance

When training multi-day RNNs for this analysis, we used all 10 sessions where open-loop data were collected (i.e., blocks where T5 was handwriting sentences but no real-time decoding was performed). We then evaluated performance on the 7 sessions that had both open-loop data and closed-loop copy typing data. We evaluated the character error rate on either held-out open-loop sentences ("held-out trials") or held-out closed-loop blocks ("held-out blocks") which occurred later in the session (~1 hour later). For the held-out trials, we held out 10% of the open-loop trials at random; for the held-out blocks, all data were used. When training single-day RNNs that used only a single session of data, we trained separate RNNs for all 7 evaluation sessions.

For testing the effect of synthetic data and input white noise (Extended Data Fig. 3a), we performed a joint grid search over both the synthetic data fraction and amount of white noise. Since both parameters have a regularizing effect, we searched over both at the same time so we could make more definitive statements about whether both were really needed (or whether just one tuned to the correct value provided enough regularization to reach peak performance). We searched over four possible values of the synthetic data fraction (0, 0.25, 0.5, and 0.75) and five possible values of white noise standard deviation (0, 0.6, 1.2, 1.8, and 2.4), making for a 4 x 5 grid. The character error rates shown in Extended Data Fig. 3a were averaged over the 7 evaluation days.

For testing the effect of feature mean noise (Extended Data Fig. 3b), results are shown for each of four pairs of constant offset noise (γ) and random walk noise (ζ): (γ=0, ζ=0), (γ=0.3, ζ=0.01), (γ=0.6, ζ=0.02), (γ=1.2, ζ=0.04). For each pair, 3 separate RNNs were trained.

For testing the effect of training on all days of data vs. just a single day (Extended Data Fig. 3c), results are shown from one multi-day RNN and seven single-day RNNs (one for each evaluation session).

For testing the effect of using separate input layers for each day vs. a single layer when training across multiple days (Extended Data Fig. 3d), results are shown from one separate-layer RNN and one shared-layer RNN.

6.7 Bidirectional RNN

Bidirectional RNNs are used commonly whenever the computation at hand is not required to be causal (e.g., for non-real-time speech recognition, or for machine translation). To implement bidirectionality, we made each of the two layers (depicted in Extended Data Fig. 1) bidirectional by adding an identical GRU component that runs in the opposite direction (i.e., begins with the last neural feature vector x_T and ends with the first neural feature vector x_1). The first RNN layer thus has a total of 1024 hidden units (512 units in the forward direction, 512 units in the backwards direction); these 1024 hidden units were concatenated together in a vector and fed as input to both the forward and backward components of the second layer. Likewise, the hidden state of both the forward and backward components in the second layer were concatenated together at each time step in order to compute the output probabilities.

Since the RNN was bidirectional, no output delay was added during training. To train the RNN, data from all available sessions were used. Instead of training and testing on separate blocks of data, as was done for the real-time performance evaluation, we used all blocks of data (both the “open-loop” blocks where no real-time decoding occurred, and the “closed-loop” blocks with real-time decoding). We randomly selected 10% of the sentences from each day as held-out test sentences for evaluation. We excluded all blocks of the 7 repeated sentences that we collected for comparison with (Pandarinath et al., 2017), since we didn’t want the RNN to overfit to these sentences.

We used the following training parameters: λ=1e-5, H=512, σ=1.2, γ=0, ζ=0, χ=0.375, ω=0.1, α=0.01, M=100k, N=64, T=24.

7. Comparison to an HMM decoder

To test whether an RNN was necessary for achieving high-performance compared to a simpler decoding approach, we tested the performance of a straightforward hidden Markov model decoder (see Table M5 below). The results confirm that our RNN outperforms a simple HMM, especially for held-out blocks where the feature means are likely to have changed substantially due to neural non-stationarities (Jarosiewicz et al., 2015; Downey et al., 2018). Nevertheless, it is noteworthy that even an HMM decoder could perform reasonably well when feature mean drift was accounted for, suggesting that the neural activity itself is highly discriminable.

We designed the HMM decoder in the same way as we designed the forced-alignment HMMs used to label the sentence data, except instead of containing character states that marched forward through a fixed sequence of characters, each character could transition to any other character with equal probability. We also tweaked the state transition probabilities within each character to the following values (to improve decoding performance):

$$P(s_{x,j} \rightarrow s_{x,j}) = 0.4$$
$$P(s_{x,j} \rightarrow s_{x+1,j}) = 0.2$$
$$P(s_{x,j} \rightarrow s_{x+2,j}) = 0.4$$

When evaluating decoder performance, we applied the language model to both the RNN and the HMM. Using a language model makes the comparison fairer by compensating for the fact that the RNN itself could learn character transition probabilities that better model the English language than the simple uniform transition model we used for the HMM.

	Held-Out Trials + Mean Subtraction	**Held-Out Trials**	**Held-Out Blocks**
RNN	0.23 %	0.23 %	0.70 %
HMM	2.96 %	6.70 %	80.08 %

Table M5. RNN vs. HMM Performance. Offline performance (character error rate) of the RNN decoder was compared to a hidden Markov model (HMM) decoder, both with a language model applied. Results show that the RNN strongly outperforms the HMM, especially in situations where the feature means are likely to have changed. The HMM has no built-in mechanism for adapting to changes in the feature means (drifts in the baseline firing rates that accrue over time), leading to very poor performance when generalizing to held-out blocks, and best performance when subtracting within-block feature means to account for any changes that may have occurred over time.

8. Decoder retraining analysis (Fig. 3)

8.1 Decoder performance as a function of calibration data

To estimate how performance would have changed if we had used less calibration data to retrain the decoder each day (50 calibration sentences were originally used), we "re-ran" the copy typing sessions offline using RNNs that were trained in the same way as originally done (*except* that less calibration data were used).

First, we began with an RNN trained on our "base" dataset of 242 sentences collected on three preliminary days. Then, for each copy typing day *Y*, we retrained the RNN using *X* calibration sentences from that day (plus *X* calibration sentences from each copy typing day prior to *Y*). After retraining, we then applied the RNN to all online performance evaluation blocks. The RNN output on these blocks was used to evaluate character error rate in the same way as was done with the online data. Essentially, this procedure simulates what would have happened if the copy typing experiment were re-run in the exact same way except with a different amount of calibration data.

When reducing the amount of calibration data, we subsampled from the original 50 sentences at even intervals (thus ensuring that the subsampled data contained sentences spaced evenly in time). Note that results are similar when choosing sentences uniformly at random. To test this, we re-ran the analysis 10 more times using 10 sentences chosen randomly instead of evenly. The reported error rate in Fig. 3a was 8.5% for 10 sentences; the mean of these 10 random runs was 9.2% with a standard deviation of 0.6%.

To reduce the amount of training data even further, we also used less "single letters" data in addition to using fewer sentences. Originally, ten repetitions of each single character were collected each day as part of the decoder retraining process (this data was used to initialize the forced alignment HMM). Here, we used only 10 * (X/50) trials (i.e. 2, 4, 6, 8 and 10 trials for the 10, 20, 30, 40 and 50 sentences conditions, respectively).

Finally, for the "no retrain" condition shown in Fig. 3a-b, the inputs to the RNN were re-scaled by multiplying all input features by 1.5; this counteracts the effect of shrinkage in the original neural subspace as neural features change over time (Extended Data Fig. 4c shows the shrinkage effect).

8.2 Decoder performance as a function of days since last retrain

In Fig. 3b, we showed that less decoder retraining is needed when transferring to new days that occur closer in time (best performance was seen for 7 days or less). Eight days of copy typing data were included in this analysis, listed in the table below (see Table M1 for a more complete list of all sessions). This set of sessions were chosen so as to include as many session pairs as possible while also allowing for enough "baseline" training data for the RNN (so that it was trained on *at least* all 242 baseline training sentences before transferring to a new day).

Date (Trial Day)	Description
2019.12.09 (1209)	Real-time decoding pilot day
2019.12.11 (1211)	Copy-typing evaluation
2019.12.18 (1218)	Copy-typing evaluation
2019.12.20 (1220)	Copy-typing evaluation
2020.01.06 (1237)	Copy-typing evaluation
2020.01.08 (1239)	Copy-typing evaluation
2020.01.13 (1244)	Free-answer evaluation
2020.01.15 (1246)	Free-answer evaluation

Table M6. List of sessions included in the decoder retraining analysis (Fig. 3b).

Trial days 1209 to 1239 contained copy typing calibration blocks as well as online evaluation blocks. In this analysis, decoders were trained using some subset of the calibration data and then tested on the evaluation data. Free typing days (1244 and 1246) included copy typing data only for decoder calibration; we split this data into training and test sets so that copy typing performance could be evaluated on these days as well.

With this set of 8 sessions, we performed the following analysis for all session pairs (X, Y): train an RNN on all data from session X plus all sessions prior to X, then evaluate the RNN on session Y under different retraining conditions: no retraining, retraining with a small number of calibration sentences from session Y, or unsupervised retraining using all available calibration data from session Y.

9. Estimating neural nonstationarity (Extended Data Fig. 4)

In Extended Data Fig. 4 we showed how the recorded neural activity changed across days. Here, we explain in detail how we quantified the correlations in neural patterns across days (Extended Data Fig. 4b) and made the visualization of how neural activity contracts in the original subspace but otherwise retains similar structure (Extended Data Fig. 4c).

9.1 Neural correlations

To quantify the similarity of neural activity across days, we used correlation (Pearson's *r*) to summarize the degree of similarity across the entire neural population and all 31 characters. We used the "single letters" data collected on each day for decoder retraining.

First, threshold crossing rates were binned into 10 ms bins and smoothed by convolving with a gaussian kernel (sd = 30 ms) to remove high frequency noise. Neural activity was then time-warped to align all repetitions of a single character together (as described in the main Methods). Neural activity was then "centered" within each day by subtracting the mean firing rate observed on each electrode. This step prevents changes in baseline firing rate from affecting the similarity measure. We wanted to exclude these, since these changes are relatively straightforward to remove/account for (as we did in this work by training the RNN to be robust to changes in mean firing rate).

Next, "pseudo-trials" were created by concatenating together a single trial from each character, resulting in pseudo-trial vectors of length NTC. Here, N is the number of electrodes (192), T is the number of time steps included from each trial (140), and C is the number of characters (31). The result of this step is a set of ten vectors $\{v_1, v_2, \ldots, v_{10}\}$, one vector for each of the ten repetitions of all 31 characters. When assessing the similarity between any two days, we then have two sets of vectors to consider: $\{v_1, v_2, \ldots, v_{10}\}$ and $\{u_1, u_2, \ldots, u_{10}\}$. Consider each of these vectors as a random draw from a day-specific distribution (let us denote the two distributions as *V* and *U*).

To quantify similarity, we estimated the correlation between the *means* of *V* and *U* (note that the means themselves are also vectors). The quantity of interest here is the *mean* because this represents the average firing rates observed for each character (i.e. the neural encoding of each character). To estimate the correlation between the means of *V* and *U*, we used a cross-validated measure of correlation that reduces the impact of noise (see our prior work (Willett et al., 2020) and accompanying code repository https://github.com/fwillett/cvVectorStats). Importantly, this method is different from simply correlating $\frac{1}{10}\sum_i v_i$ and $\frac{1}{10}\sum_i u_i$, which would underestimate the true correlation due to noise that causes the estimated means to appear more dissimilar than they really are. For example, even if *V* and *U* have identical means, noise in v_i and u_i would always cause the estimated correlation to be less than 1 when correlating $\frac{1}{10}\sum_i v_i$ and $\frac{1}{10}\sum_i u_i$.

To account for changes in writing speed across days, we computed the above correlation measure using varying amounts of time dilation applied uniformly to all trials from one of the days. Time dilation was implemented simply by linearly interpolating the neural activity to stretch or contract it in time. Ten dilation factors were considered that ranged from 0.7 to 1.42 in even increments. We used the time dilation value that maximized the correlation. This step effectively removes the potential confound of a change in writing speed causing the neural representations to appear more different than they really are.

Finally, to compute the correlation in the "anchor" space, we first projected the neural representations from each day onto the top ten principal components that describe the *earlier* day's data. This measures how much the neural activity has changed in the original neural subspace. The principal components were found by first concatenating all trial-averaged spatiotemporal patterns into an N x TC matrix, where N is the number of electrodes (192), T is the number of time steps (140) and C is the number of characters (31). Principal components analysis was then applied to the columns of this matrix.

9.2 Contraction in original space

Extended Data Fig. 4c shows a low-dimensional representation of each character's neural representation (using axes found on Day -2). To do this, each character's trial-averaged neural representation was first converted into a vector of length NT, where N is the number of electrodes (192) and T is the number of 10 ms time steps per character (140). These vectors were concatenated into a matrix of dimension NT x C, where C is the number of characters (31). Finally, principal components analysis was applied to the columns of this matrix. The top 2 PCs where then used to visualize each character's neural representation.

10. Temporal variety improves decoding (Fig. 4)

10.1 Pairwise neural distances

The characters dataset analyzed in Fig. 4 is the same as that shown in Fig. 1 (session 1). The straight-lines dataset was collected on a separate session (session 2 in Table M1), where T5 attempted to handwrite straight-line strokes in an instructed delay paradigm identical to what was used for writing single characters (except instead of a text cue, a line appeared on the screen to indicate the direction of the stroke).

To compute the pairwise distances reported in Fig. 4c, the threshold crossing rates were first binned into 10 ms bins and smoothed by convolving with a Gaussian kernel (sd = 30 ms). Then, neural activity within a 0 to 1000 ms window after the go cue (for characters) or 0 to 600 ms (for lines) was time-aligned across trials using the time-warping methods described above (for Fig. 1). These time windows were chosen by visual inspection of when the neural activity stopped modulating. The time-aligned data were then trial-averaged and re-sampled to 100 time points (using linear interpolation) to generate a set of mean spatiotemporal neural activity matrices (of dimension 192 electrodes x 100 time steps).

Pairwise distances were defined as the Euclidean norm (square root of the sum of squared entries) of the difference matrix obtained by subtracting one spatiotemporal neural matrix from another. Pairwise distances were estimated using cross-validation, according to the methods in (Willett et al., 2019) (https://github.com/fwillett/cvVectorStats); without cross-validation, noise would inflate the distances and make them all appear larger than they are. Pairwise distances for simulated data (Fig. 4f-g and Extended Data Fig. 6) were computed without cross-validation (because there was no estimation noise).

We normalized the pairwise distances reported in Fig. 4c by the number of time steps included in the analysis and the number of electrodes by dividing by $\sqrt{NT}$, where N is the number of electrodes (192) and T is the number of time steps (100). This makes the distances roughly invariant to the number of time steps and electrodes; for example, if each electrode fires at 150 Hz for condition A and 50 Hz for condition B, then the distance between B and A is $\sqrt{150NT - 50NT} = 10\sqrt{NT}$.

10.2 Neural and temporal dimensionality

Dimensionality, as computed in Fig. 4e, was estimated using the “participation ratio” (Gao et al., 2017), which is a continuous metric that quantifies how evenly-sized the eigenvalues of the covariance matrix are. It is roughly equivalent to the number of dimensions needed to explain 80% of the variance in the data.

To compute the neural dimensionality, the smoothed, time-warped, and trial-averaged neural activity was arranged into a matrix X of dimensionality N x TC, where N is the number of electrodes (192), T is the number of time steps (100), and C is the number of movement conditions (16). Each row is the trial-averaged response of a single electrode to each movement

condition concatenated together. The eigenvalues u_i of the covariance matrix XX^T were then used to compute the participation ratio:

$$PR = \frac{(\sum_i u_i)^2}{\sum_i u_i^2}$$

Similarly, to compute the temporal dimensionality, the neural activity was arranged into a matrix X of dimensionality T x NC, where each row contains the trial-averaged response of all neurons across all conditions concatenated together for a single time step (and each column is a neural response for a single condition). Roughly, the temporal dimensionality quantifies how many T-dimensional neural response basis vectors are needed to explain 80% of the variance of all of the neural responses (PSTHs).

To reduce bias, we used cross-validation to estimate the covariance matrix (otherwise, the presence of estimation noise would artificially inflate the dimensionality). Specifically, we split the trials into two folds, and computed the X matrix separately for each fold (yielding X_1 and X_2). The covariance matrix was then estimated as $X_1X_2^T$. To compute confidence intervals for dimensionality (and dimensionality ratios), we used the jackknife method (see our prior work (Willett et al., 2019) https://github.com/fwillett/cvVectorStats).

10.3 Simulated classification accuracy

To simulate classification accuracy for the lines and characters as a function of neural noise (Fig. 4d), we used the cross-validated pairwise distances between all conditions. This ensures that accuracy is not inflated by overestimating the distances between conditions. We used classic multidimensional scaling to find a set of low-dimensional points that have the same pairwise distances; these are low-dimensional representations of the neural activity patterns associated with each movement class. Then, we simulated a trial of classification with these points by first picking a point at random (the true class), and then adding Gaussian white noise to this point in the low-dimensional space (to generate the observation). Classification was correct if the observation lay closest to the true point. This simulates a simple classifier that chooses which class an observation belongs to by choosing the class with the nearest mean (this corresponds to a maximum likelihood classifier in the case of spherical Gaussian noise).

When dealing with simulated data (Fig. 4f-g or Extended Data Fig. 6), no multidimensional scaling was needed since no estimation noise was present. Thus, we performed the simulated classification using the true neural patterns themselves (but still in the presence of observation noise). The simulated trajectories were discretized into 100 time steps and white noise was added to each time step independently.

11. Optimized alphabet (Extended Data Fig. 6)

We considered the following optimization problem to find a new set of 26 letters that maximizes the distance between the closest pair of letters (Extended Data Fig. 6A):

$$\underset{X_1,X_2,\ldots,X_{26}}{\operatorname{argmax}} \min_{i,j} \left\| S(Q(X_i)) - S(Q(X_j)) \right\|_F^2$$

Here, X_i is a 2 x 100 matrix that represents the pen tip velocity trajectory for letter *i* (each column is a velocity vector for one time step), *Q* is a "squashing" function that constrains each column of X_i to lie within the unit circle, *S* is a smoothing function that constrains the trajectories to be smooth by convolving each row with a Gaussian kernel (sd = 8), and $\|X\|_F$ is the Frobenius norm of X (i.e., the square root of the sum of squared entries in the matrix X).

Q was defined as follows (where x_i is the *i*th column of *X*):

$$Q(X) = [q(x_1) \quad q(x_2) \quad \ldots \quad q(x_{100})]$$
$$q(x_i) = \frac{x_i}{\|x_i\|}\left(1 - e^{-\|x_i\|}\right)$$

We found local minima of the above cost function by using gradient descent (implemented with TensorFlow v1.15 to compute the gradients, using the "Adam" optimization method (Abadi et al., 2016; Kingma and Ba, 2017)).

We varied the width of the Gaussian kernel used for smoothing until the temporal dimensionality of the optimized letters was equal to the dimensionality of the Latin alphabet (~4D), resulting in a set of letters that has a visually similar level of complexity and curviness to the Latin letters (Extended Data Fig. 6a). To define the pen trajectories of the Latin alphabet, we used the trajectories reconstructed in Fig. 1, which capture how T5 wrote the letters. These trajectories were resampled (linearly time-warped) to 100 time steps (as was done in Fig. 3).

As a comparison, we also optimized for a set of 26 straight lines (Extended Data Fig. 6b). To do so, we used the above cost function except that each column of X_i was constrained to be equal, enforcing a straight-line trajectory.

Note that no neural encoding model is explicitly considered in the optimization. However, if we assume linear neural tuning to pen tip velocity (i.e., "cosine tuning"), then the cost functions are equivalent under reasonable assumptions of uniformity in the neural tuning. That is, maximizing the nearest neighbor distance of pen trajectories is the same as maximizing the nearest neighbor distance of evoked neural features under this assumption. The result follows from the fact that distances are preserved under orthogonal transformations.

To show this, let *E* be a matrix of linear tuning coefficients (of size 192 x 2) for 192 hypothetical neural features. Assume the neural features are tuned to pen tip velocity in the following way:

$$f_t = Ev_t + b$$

Here, f_t is a 192 x 1 vector of neural features for time step t, v_t is a 2 x 1 pen tip velocity vector, and b is a 192 x 1 offset vector. The squared distance between the neural feature matrices associated with two different letters can then be expressed in the following way (where v_t is the pen tip velocity for letter A and u_t is the pen tip velocity for letter B):

$$\sum_{t=1}^{N} (Ev_t + b - [Eu_t + b])^T (Ev_t + b - [Eu_t + b])$$
$$= \sum_{t=1}^{N} (E(v_t - u_t))^T (E(v_t - u_t))$$
$$= \sum_{t=1}^{N} (v_t - u_t)^T E^T E (v_t - u_t)$$

Let us assume that the columns of E are roughly orthogonal to each other, which would be the case for "uniform" neural tuning where the rows of E (i.e., the "preferred directions") are uniformly distributed about the origin. Then, E^TE is a diagonal matrix with approximately equal entries along the diagonal (we can denote this entry as α). Then, we have:

$$\sum_{t=1}^{N} (v_t - u_t)^T E^T E (v_t - u_t)$$
$$\approx \sum_{t=1}^{N} (v_t - u_t)^T \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} (v_t - u_t)$$
$$= \alpha \sum_{t=1}^{N} (v_t - u_t)^T (v_t - u_t)$$

Thus, linear neural tuning to pen tip velocity implies that the neural distances are directly proportional to the distances between the velocity trajectories themselves.

More generally, the neural encoding of a pen trajectory is not solely linear. However, we conjecture that as long as the neural encoding function is a reasonably smooth function of the pen tip velocity, then far-apart pen trajectories are likely to evoke far-apart neural activity, making it a reasonable approach to optimize over pen trajectories directly.

12. Language model

12.1 Overview

In a retrospective offline analysis, we used a custom, large vocabulary language model to autocorrect errors made by the decoder. Here, we give an overview of the major steps involved (note that our code release also contains the language model and associated scripts for applying it - https://github.com/fwillett/handwritingBCI). The language model had two stages: (1) a 50,000-word bigram model that first processes the neural decoder's output to generate a set of candidate sentences, and (2) a neural network to rescore these candidate sentences (OpenAI's GPT-2, 1558M parameter version; https://github.com/openai/gpt-2) (Radford et al., 2018). This two-step strategy is typical in speech recognition (Xiong et al., 2017) and plays to the strengths of both types of models. Although the rescoring step improved performance, we found that performance was strong with the bigram model alone (1.48% character error rate with the bigram model alone, 0.89% with rescoring, using the copy typing data).

The bigram model was created with Kaldi (Povey et al., 2011) using samples of text provided by OpenAI (250k samples from WebText, https://github.com/openai/gpt-2-output-dataset). These samples were first processed to make all text lower case and to remove all punctuation that was not part of our limited character set (consisting only of periods, question marks, commas, apostrophes, and spaces). Then, we used the Kaldi toolkit to construct a bigram language model, using the 50,000 most common words appearing in the WebText sample. The language model was represented in the form of a finite-state transducer which could be used to translate the RNN probabilities into candidate sentences (Mohri et al., 2008).

12.2 WebText preprocessing

Our bigram language model was created using 250k samples of text from WebText (https://github.com/openai/gpt-2-output-dataset), provided by OpenAI (San Francisco, CA). We pre-processed the WebText samples to convert them to lowercase, remove symbols that were not in our character set, and split into sentences (yielding a total of 5.1M sentences). We applied the following step-by-step recipe to pre-process the sentences:

(1) Replace newlines and hyphens with spaces
(2) Convert all letters into lower-case
(3) Delete all characters not in the character set (which consisted only of the letters a-z, periods, spaces, commas, question marks and apostrophes)
(4) Replace repeated spaces with single spaces
(5) Remove spaces in front of periods and commas
(6) Replace repeated periods with single periods
(7) Strip surrounding whitespace from the sample
(8) Split the sample into sentences by splitting at periods and question marks

12.3 Constructing the bigram language model

We used publicly available scripts ((Puigcerver, 2017); https://github.com/jpuigcerver/Laia/tree/master/egs/iam) as a starting point for constructing our bigram language model. These scripts tokenize the list of sentences created above into discrete words, and then call Kaldi (Povey et al., 2011) and SRILM (Stolcke et al., 2011) programs to count the frequency with which these words (and pairs of words) appear in the sentences. Word frequencies are then encoded into a weighted finite state transducer (Mohri et al., 2008) that can be used to infer the most likely sequence of characters given the word frequency counts combined with the character probabilities output by the neural decoder.

To ensure that our language model was sufficiently general to allow the expression of a wide variety of sentences, we used a large vocabulary (consisting of the 50,000 most common words in the WebText samples). A space character was included at the beginning of each word to enforce that words were always separated by spaces (except for words containing punctuation, such as contractions).

In big picture, the language model is essentially a large hidden Markov model where each state corresponds to a character, and where the state transition probabilities encode the statistics of which words are likely to follow other words. Inference is done with the language model by using an approximate Viterbi search (beam search) to find likely sequences of characters. The beam search combines information from the language priors (state transition probabilities) and information from the neural decoder about which characters are likely occurring at each moment in time (observations).

The language model was represented as the composition of weighted finite state transducers (Mohri et al., 2008) that encode information about different parts of the model:

$$H \circ C \circ L \circ G$$

Here, $\circ$ denotes composition, G is the grammar that encodes legal sequences of words and their probabilities (based on the unigram and bigram probabilities), L is the lexicon that encodes what characters are contained in each legal word, and H and C encode information about the character sub-states used in decoding (see Kaldi documentation for more detail). Each character had two sub-states s_1 and s_2. s_1 emits a CTC blank (since (Puigcerver, 2017) used a neural network trained with the CTC loss function, see below) and s_2 emits the corresponding character. We used the following transition probabilities: $P(s_1 \rightarrow s_1)$=0.6, $P(s_1 \rightarrow s_2)$=0.2, $P(s_1 \rightarrow s_{next})$=0.2, $P(s_2 \rightarrow s_2)$=0.6, $P(s_2 \rightarrow s_{next})$=0.4 , where s_{next} is the first sub-state of the next character.

12.4 Inference with the bigram language model

The scripts we used from Puigcerver and colleagues (Puigcerver, 2017) configured the language model to work with outputs generated by a neural network trained with the connectionist temporal classification (CTC) loss function (Graves et al., 2006). We therefore transformed our neurally decoded probabilities to make them look more like CTC outputs before using the language model. To generate the CTC blank probability $blank_t$, we transformed z_t (the character start signal):

$$blank_t = 1 - \sigma(4 + 4\sigma^{-1}(z_{t+20}))$$

This hand-tuned function inverts z_t and makes it sharper, so that it stays mostly at 1 and dips to 0 only briefly whenever a new character is written. It also shifts the signal forward by 20 time steps (400 ms), so that it dips to 0 at times when the character probabilities y_t have already finished transitioning from the previous character to the next. Finally, we also modified y_t so that all entries of y_t plus the blank signal $blank_t$ sum to 1:

$$y'_t = y_t(1 - blank_t)$$

To perform inference with the language model and the above probabilities, we used Kaldi (and custom decoding functions by Puigcerver https://github.com/jpuigcerver/kaldi-decoders) to perform a beam search to generate lattices of candidate word sequences with high likelihood (beam=65, max active=5000, acoustic score=1.0, lattice beam=10). On average, the decoding process completed 3.74 times faster than real-time (averaged over the last two sessions of copy typing evaluation, where T5 wrote the fastest).

12.5 Rescoring with GPT-2

We used OpenAI's neural network language model "GPT-2" to rescore the candidate sentences inferred by the bigram language model ((Radford et al., 2018), 1558M parameter version, https://github.com/openai/gpt-2). Rescoring using a neural network model is motivated by the fact that neural networks are powerful language models that can model long-range semantic dependencies, but may be too slow to use to efficiently search through many different possibilities. Thus, a simpler N-gram model can propose a list of plausible candidate sentences which a neural network model rescores (e.g., (Xiong et al., 2017)).

When run on a sequence on characters, GPT-2 returns the conditional probability of each character given the previous characters. These probabilities can be used to compute the log probability of any candidate sentence in the following way:

$$\log P(c_1, c_2, \ldots, c_N) = \log P(c_1) + \log P(c_2 \mid c_1) + \ldots + \log P(c_N \mid c_{N-1}, \ldots, c_1)$$

Here, $P(c_1, c_2, \ldots, c_N)$ is the probability of observing the character sequence $c_1, c_2, \ldots, c_N$ and $P(c_N \mid c_{N-1}, \ldots, c_1)$ is the conditional probability of observing character c_N given previous characters $c_{N-1}, \ldots, c_1$.

When generating candidate sentences for rescoring, an "acoustic score" and a "language model score" was returned for each candidate sentence. The acoustic score contains the cumulative log probability of observing the character sequence given the sentence, and the language model score contains the log probability of observing the sentence given the language model. When rescoring, we replaced the language model score with the probability returned by GPT-2, scaled the acoustic score by 0.5, and summed them together to generate a final score. The minimum score across all candidate sentences was then chosen.

12.6 Performance without rescoring

We compared the performance of the language model with rescoring to the bigram model alone. When decoding with the bigram model alone, we also used an acoustic score of 0.5. The character error rate with the bigram model alone was 1.48% [1.11, 1.85] (95% CI), and the word error rate was 4.72% [3.67, 5.87].

13. Statistics

The following table lists statistical details for each hypothesis test and confidence interval reported in this work. In this study, uncertainty was quantified mainly with 95% confidence intervals computed with nonparametric methods (bootstrap or jackknife). Only two hypothesis tests were performed (Fig. 4c).

Result	Statistical Details
K-NN Classifier Accuracy	The 95% confidence interval for classification accuracy reported in the main text (95% CI = [92.6, 95.8]) was a binomial proportion confidence interval (Clopper-Pearson). Classification accuracy was computed over 864 trials.
Table 1	Table 1 reports error rates derived from 163 independent trials (sentences). 95% confidence intervals were computed via bootstrap resampling over the trials (10,000 resamplings).
Fig. 3	For panel A, each data point represents the mean of 163 independent trials (sentences). 95% confidence intervals of the mean were computed via bootstrap resampling over the trials (10,000 resamples). For panel B, each data point shows the character error rate averaged across all session pairs belonging to that category. There were 8 session pairs for the 2-7 days category, 4 pairs for the 8-14 days category, and 16 pairs for the 15-37 days category. 95% confidence intervals were computed via bootstrap resampling over individual trials (10,000 resamples; resampling was performed within each session separately).
Fig. 4	For Fig. 4c, p-values were generated using two-sided, independent two-sample t-tests (with the statistical unit being a single movement condition, N=16). The assumption of normality was not quantitatively assessed. Nearest Neighbor Distance t-test: t(30)=6.86, p=1.2e-07 Mean Distance t-test: t(30)=10.83, p=6.8e-12 For Fig. 4e, the confidence intervals for spatial and temporal dimensionality were computed using the jackknife method. We used the jackknife here because the dimensionality metric was cross-validated. When dealing with cross-validated metrics, the bootstrap method can be inaccurate (see https://github.com/fwillett/cvVectorStats).

	Each jackknife subsample left out one trial from each movement condition. There were 27 trials per character condition and 24 trials per straight line condition. See (Severiano et al., 2011) for the jackknife formula, and see our prior work (Willett et al., 2019) for more examples on using the jackknife method with cross-validated metrics. Several confidence intervals were also reported in the main text (not in the figure). The confidence interval for the nearest-neighbor distance ratio (95% CI = [60%, 86%]) was computed using bootstrap resampling (10,000 resamplings of the 16 movement conditions). The confidence intervals for the dimensionality ratios (95% CI = [1.19, 1.30] and [2.58, 2.72]) were computed using the jackknife method, again because the dimensionality metric was cross-validated. Each jackknife subsample left out one trial from each movement condition.
Extended Data Fig. 3	In panel E, 95% confidence intervals for the differences in error rate between conditions were computed with bootstrap resampling of single trials (sentences). 10,000 resamplings were performed for each confidence interval. The number of trials contributing to each confidence interval varied depending on the analysis. • Synthetic data fraction – 228 trials • Input white noise – 228 trials • Feature mean noise, held-out blocks – 228 trials • Feature mean noise, held-out trials – 33 trials • Single-day vs. multi-day, held-out blocks – 228 trials • Single-day vs. multi-day, held-out trials – 33 trials • Day-specific layers vs. same layer, held-out blocks – 228 trials • Day-specific layers vs. same layer, held-out blocks – 33 trials
Extended Data Fig. 4	In panel D, each data point shows the character error rate averaged across all session pairs belonging to that category. There were 8 session pairs for the 2-7 days category, 4 pairs for the 8-14 days category, and 16 pairs for the 15-37 days category. 95% confidence intervals were computed via bootstrap resampling over individual trials (10,000 resamples; resampling was performed within each session separately).

Table M7. Statistical details for all reported confidence intervals and hypothesis tests.

II. Supplemental Note 1

In the toy example presented in Fig. 4f-h, we showed that additional temporal dimensions can be used to improve the classifiability of a set of neural patterns in the presence of Gaussian *white noise* that is uncorrelated across time points and neurons. Under these assumptions, the Euclidean distance between each pair of neural patterns is the relevant factor determining classification accuracy, and it therefore follows that greater temporal dimensionality will improve classification performance if it helps to spread out those patterns more evenly. Here, we examine how *correlated noise* might affect this result.

First, it is helpful to define some terms. Let f_x be a vector that describes the underlying neural trajectory for movement *x* (i.e., the mean neural firing rates across time for movement *x*). Each entry in the vector f_x is the mean firing rate for a single time step. To describe multiple neurons, the activity profile of each neuron can be stacked one on top of the other in the vector. Let ε be a neural noise vector of the same length that has a multivariate normal distribution with zero mean and covariance matrix Σ. If Σ is non-diagonal, the noise is said to be correlated.

Given a vector of noisy observed firing rates $r = f_x + \varepsilon$, a maximum likelihood classifier will choose to classify *r* into the class that has the minimum Mahalanobis distance to *r* (assuming uniform class priors). In other words:

$$\underset{x}{\operatorname{argmin}}(r - f_x)^T \Sigma^{-1} (r - f_x)$$

In the case of white noise, Σ is a diagonal matrix with all diagonal entries equal to σ. In this case, the classifier will simply choose the class whose mean has the smallest Euclidean distance to *r*. This justifies the idea that nearest neighbor distances should be increased to reduce classifier confusions (potentially via spreading the neural patterns out into additional temporal dimensions).

If Σ is non-diagonal, this means that the noise cloud will extend more in some directions and less in others. The directions that are most harmful for classification are those that connect nearby class means (e.g., the direction f_x-f_y, as this would make noise more likely to 'corrupt' class *x* to look more like *y*). In the general case where Σ can take any arbitrary shape, it is not always true that classification accuracy can be improved by using extra temporal dimensions to increase Euclidean distances. For example, it could be the case that these extra temporal dimensions are particularly noisy, cancelling out the benefit of increased distance between the class means. Nevertheless, under reasonable constructions of Σ that we test below, we show that the toy model in Fig. 4 still holds in the presence of correlated noise.

Temporally Correlated Noise

First, we tested noise with *temporal* correlations (meaning that the noise associated with each neuron was positively correlated in time). This noise can describe slow (but random) fluctuations in neural firing rates over time, and in this sense is more realistic than white noise. Temporal correlations would generally cause the noise to be more concentrated along dimensions that span the class means, since the underlying neural patterns are also smooth

across time (as is the case in this toy example). Extended Data Fig. 5A shows examples of temporally correlated noise vectors and the covariance matrix used to generate them. The wide diagonal band in the covariance matrix causes nearby time steps to have correlated noise.

In Extended Data Fig. 5B, we compared the classification accuracy between time-varying trajectories and constant trajectories in the presence of temporally correlated noise, finding an even more pronounced improvement for time-varying trajectories. This is because neural patterns that vary more quickly in time are less aligned with slow-varying noise directions, enabling greater robustness to this type of noise. Here, classification was performed with a maximum likelihood classifier (under the assumption that the means of each class and the covariance matrix of the noise are known). However, results also hold using a simpler "Euclidean distance" classifier that assumes the noise is white by choosing the class whose mean has the smallest Euclidean distance to *r*.

Signal-Correlated Noise

Finally, we tested noise vectors that were directly correlated with the underlying neural signal (that is, noise vectors that contained variance *only* in signal-spanning dimensions that connect the class means, such as f_x - f_y). This type of noise is more realistic than white noise in the sense that neural variability is often larger in neural dimensions that carry the signal. To find these dimensions, PCA was applied separately to the constant and time-varying trajectories to find the one (constant) or two (time-varying) spatiotemporal axes containing the neural signal. The covariance matrix was then designed to place noise in these axes only (with equal variance for each axis):

$$\Sigma = \sigma \mathrm{A}\mathrm{A}^T$$

Here, A is a matrix whose columns are the PCA axes and σ scales the overall size of the noise. Extended Data Fig. 5c shows what these noise vectors look like for the time-varying trajectories. Because the time-varying trajectories have only two temporal dimensions, the noise vectors also have this structure (where the first 50 time points are highly correlated with each other and the last 50 time points are highly correlated with each other).

Again, even in the presence of noise that is correlated with the signal, we found that it is still easier to classify time-varying trajectories than constant trajectories (Extended Data Fig. 5d). This result can be explained by the fact that signal-spanning noise acts like white noise in dimensions that span the class means, but is zero elsewhere. Since noise in dimensions that *don't* align with the class means are not as relevant for classification performance, it makes sense that their absence does not change the main result.

III. References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. ArXiv:1603.04467 [Cs].

Brants, T., and Franz, A. (2006). Web 1T 5-gram Version 1.

Chestek, C.A., Gilja, V., Nuyujukian, P., Foster, J.D., Fan, J.M., Kaufman, M.T., Churchland, M.M., Rivera-Alvidrez, Z., Cunningham, J.P., Ryu, S.I., et al. (2011). Long-term stability of neural prosthetic control signals from silicon cortical arrays in rhesus macaque motor cortex. J. Neural Eng. *8*, 045005.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cuDNN: Efficient Primitives for Deep Learning. ArXiv:1410.0759 [Cs].

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. ArXiv:1406.1078 [Cs, Stat].

Christie, B.P., Tat, D.M., Irwin, Z.T., Gilja, V., Nuyujukian, P., Foster, J.D., Ryu, S.I., Shenoy, K.V., Thompson, D.E., and Chestek, C.A. (2014). Comparison of spike sorting and thresholding of voltage waveforms for intracortical brain–machine interface performance. J. Neural Eng. *12*, 016009.

Cieri, C., Miller, D., and Walker, K. (2004). The Fisher Corpus: a Resource for the Next Generations of Speech-to-Text. In Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04), (Lisbon, Portugal: European Language Resources Association (ELRA)), p.

Collinger, J.L., Wodlinger, B., Downey, J.E., Wang, W., Tyler-Kabara, E.C., Weber, D.J., McMorland, A.J., Velliste, M., Boninger, M.L., and Schwartz, A.B. (2013). High-performance neuroprosthetic control by an individual with tetraplegia. The Lancet *381*, 557–564.

Collobert, R., Puhrsch, C., and Synnaeve, G. (2016). Wav2Letter: an End-to-End ConvNet-based Speech Recognition System. ArXiv:1609.03193 [Cs].

Degenhart, A.D., Bishop, W.E., Oby, E.R., Tyler-Kabara, E.C., Chase, S.M., Batista, A.P., and Yu, B.M. (2020). Stabilization of a brain–computer interface via the alignment of low-dimensional spaces of neural activity. Nature Biomedical Engineering 1–14.

Downey, J.E., Schwed, N., Chase, S.M., Schwartz, A.B., and Collinger, J.L. (2018). Intracortical recording stability in human brain–computer interface users. J. Neural Eng. *15*, 046016.

Gao, P., Trautmann, E., Yu, B., Santhanam, G., Ryu, S., Shenoy, K., and Ganguli, S. (2017). A theory of multineuronal dimensionality, dynamics and measurement. BioRxiv 214262.

Gilja, V., Pandarinath, C., Blabe, C.H., Nuyujukian, P., Simeral, J.D., Sarma, A.A., Sorice, B.L., Perge, J.A., Jarosiewicz, B., Hochberg, L.R., et al. (2015). Clinical translation of a high-performance neural prosthesis. Nat Med *21*, 1142–1145.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep Learning (The MIT Press).

Graves, A., Fernández, S., Gomez, F., and Schmidhuber, J. (2006). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In Proceedings of the 23rd International Conference on Machine Learning, (Pittsburgh, Pennsylvania, USA: Association for Computing Machinery), pp. 369–376.

Graves, A., Mohamed, A., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 6645–6649.

He, Y., Sainath, T.N., Prabhavalkar, R., McGraw, I., Alvarez, R., Zhao, D., Rybach, D., Kannan, A., Wu, Y., Pang, R., et al. (2019). Streaming End-to-end Speech Recognition for Mobile Devices. In ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6381–6385.

Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al. (2012). Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. IEEE Signal Processing Magazine *29*, 82–97.

Hochberg, L.R., Serruya, M.D., Friehs, G.M., Mukand, J.A., Saleh, M., Caplan, A.H., Branner, A., Chen, D., Penn, R.D., and Donoghue, J.P. (2006). Neuronal ensemble control of prosthetic devices by a human with tetraplegia. Nature *442*, 164–171.

Jarosiewicz, B., Sarma, A.A., Bacher, D., Masse, N.Y., Simeral, J.D., Sorice, B., Oakley, E.M., Blabe, C., Pandarinath, C., Gilja, V., et al. (2015). Virtual typing by people with tetraplegia using a self-calibrating intracortical brain-computer interface. Science Translational Medicine *7*, 313ra179-313ra179.

Kingma, D.P., and Ba, J. (2017). Adam: A Method for Stochastic Optimization. ArXiv:1412.6980 [Cs].

Maaten, L. van der, and Hinton, G. (2008). Visualizing Data using t-SNE. Journal of Machine Learning Research *9*, 2579–2605.

Masse, N.Y., Jarosiewicz, B., Simeral, J.D., Bacher, D., Stavisky, S.D., Cash, S.S., Oakley, E.M., Berhanu, E., Eskandar, E., Friehs, G., et al. (2014). Non-causal spike filtering improves decoding of movement intention for intracortical BCIs. Journal of Neuroscience Methods *236*, 58–67.

Mohri, M., Pereira, F., and Riley, M. (2008). Speech Recognition with Weighted Finite-State Transducers. In Springer Handbook of Speech Processing, J. Benesty, M.M. Sondhi, and Y.A. Huang, eds. (Berlin, Heidelberg: Springer), pp. 559–584.

Palin, K., Feit, A.M., Kim, S., Kristensson, P.O., and Oulasvirta, A. (2019). How do People Type on Mobile Devices? Observations from a Study with 37,000 Volunteers. In Proceedings of the 21st International Conference on Human-Computer Interaction with Mobile Devices and Services, (Taipei, Taiwan: Association for Computing Machinery), pp. 1–12.

Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. (2015). Librispeech: An ASR corpus based on public domain audio books. In 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 5206–5210.

Pandarinath, C., Nuyujukian, P., Blabe, C.H., Sorice, B.L., Saab, J., Willett, F.R., Hochberg, L.R., Shenoy, K.V., and Henderson, J.M. (2017). High performance communication by people with paralysis using an intracortical brain-computer interface. ELife *6*, e18554.

Poole, B., Williams, A.H., Maheswaranathan, N., Yu, B., Santhanam, G., Ryu, S.I., Baccus, S., Shenoy, K.V., and Ganguli, S. (2017). Time-warped PCA: simultaneous alignment and dimensionality reduction of neural data. Cosyne Abstracts.

Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., Hannemann, M., Motlicek, P., Qian, Y., Schwarz, P., et al. (2011). The Kaldi Speech Recognition Toolkit. IEEE 2011 Workshop on Automatic Speech Recognition and Understanding.

Puigcerver, J. (2017). Are Multidimensional Recurrent Layers Really Necessary for Handwritten Text Recognition? In 2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR), pp. 67–72.

Rabiner, L.R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE *77*, 257–286.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2018). Language models are unsupervised multitask learners. OpenAI Technical Report.

Severiano, A., Carriço, J.A., Robinson, D.A., Ramirez, M., and Pinto, F.R. (2011). Evaluation of Jackknife and Bootstrap for Defining Confidence Intervals for Pairwise Agreement Measures. PLOS ONE *6*, e19539.

Stolcke, A., Zheng, J., Wang, W., and Abrash, V. (2011). SRILM at Sixteen: Update and Outlook.

Sussillo, D., Stavisky, S.D., Kao, J.C., Ryu, S.I., and Shenoy, K.V. (2016). Making brain–machine interfaces robust to future neural variability. Nat Commun *7*.

Todorova, S., Sadtler, P., Batista, A., Chase, S., and Ventura, V. (2014). To sort or not to sort: the impact of spike-sorting on neural decoding performance. J. Neural Eng. *11*, 056005.

Trautmann, E.M., Stavisky, S.D., Lahiri, S., Ames, K.C., Kaufman, M.T., O'Shea, D.J., Vyas, S., Sun, X., Ryu, S.I., Ganguli, S., et al. (2019). Accurate estimation of neural population dynamics without spike sorting. Neuron.

Willett, F.R., Deo, D.R., Avansino, D.T., Rezaii, P., Hochberg, L., Henderson, J., and Shenoy, K. (2019). Hand Knob Area of Motor Cortex in People with Tetraplegia Represents the Whole Body in a Modular Way. BioRxiv 659839.

Willett, F.R., Deo, D.R., Avansino, D.T., Rezaii, P., Hochberg, L.R., Henderson, J.M., and Shenoy, K.V. (2020). Hand Knob Area of Premotor Cortex Represents the Whole Body in a Compositional Way. Cell.

Williams, A.H., Poole, B., Maheswaranathan, N., Dhawale, A.K., Fisher, T., Wilson, C.D., Brann, D.H., Trautmann, E.M., Ryu, S., Shusterman, R., et al. (2020). Discovering Precise Temporal Patterns in Large-Scale Neural Recordings through Robust and Interpretable Time Warping. Neuron *105*, 246-259.e8.

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., and Stolcke, A. (2017). The Microsoft 2017 Conversational Speech Recognition System. ArXiv:1708.06073 [Cs].

Young, S.J., Evermann, G., Gales, M.J.F., Kershaw, D., Moore, G., Odell, J.J., Ollason, D.G., Povey, D., Valtchev, V., and Woodland, P.C. (2006). The HTK book version 3.4.

Zeyer, A., Doetsch, P., Voigtlaender, P., Schlüter, R., and Ney, H. (2017). A comprehensive study of deep bidirectional LSTM RNNS for acoustic modeling in speech recognition. In 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2462–2466.