Uncovering the latent dynamics of whole-brain fMRI tasks with a sequential variational autoencoder

Eloy Geenjaar
TReNDS center
Georgia Institute of Technology
egeenjaar@gatech.edu

Donghyun Kim
TReNDS center
Georgia Institute of Technology

Riyasat Ohib
TReNDS center
Georgia Institute of Technology

Marlena Duda
TReNDS center
Georgia State University

Amrit Kashyap
Berlin Institute of Health
Charite University Hospital

Sergey Plis
TReNDS center
Georgia State University

Vince Calhoun
TReNDS center
Georgia Institute of Technology

Abstract

The neural dynamics underlying brain activity are critical to understanding cognitive processes and mental disorders. However, current voxel-based whole-brain dimensionality reduction techniques fail to capture these dynamics, producing latent timeseries that inadequately relate to behavioral tasks. To address this issue, we introduce a novel approach to learning low-dimensional approximations of neural dynamics using a sequential variational autoencoder (SVAE) that learns the latent dynamical system. Importantly, our method finds smooth dynamics that can predict cognitive processes with accuracy higher than classical methods, with improved spatial localization to task-relevant brain regions, and we find fixed points for the dynamics that are stable across random initialization of the model.

1 Introduction

Functional magnetic resonance imaging (fMRI) is a highly informative non-invasive whole-brain modality used to study oxygen-based changes in the brain, which has been essential in understanding cognitive processes [1]. The analysis of fMRI data is also challenging due to its low signal-to-noise ratio and relatively few training samples compared to its high spatial dimensionality. Researchers have attempted to overcome these challenges using powerful dimensionality reduction techniques. The most prominent dimensionality reduction techniques currently used are averaging/grouping voxels based on a neuroanatomical atlas parcellation [2, 3, 4], independent component analysis (ICA) [5, 6, 7], and principal component analysis (PCA) [8, 9]. All three map the functional signal to a temporal trajectory in a low-dimensional subspace without explicitly taking the dynamics of the signal into account.

While not yet utilized in whole-brain imaging data, learning low-dimensional dynamics from neural data is rather commonplace for neural spiking data [10, 11, 12]. Furthermore, evidence of a low-dimensional manifold is emerging for fMRI data [13]. To learn both the projection as well as the dynamics in this latent space, we propose to use a neural network that parameterizes both the projection into the latent space and its autonomous dynamical system. Autonomous dynamical systems are dynamical systems that do not require any inputs, and given an initial state can completely

1st Workshop on Deep Generative Models for Health at NeurIPS 2023.

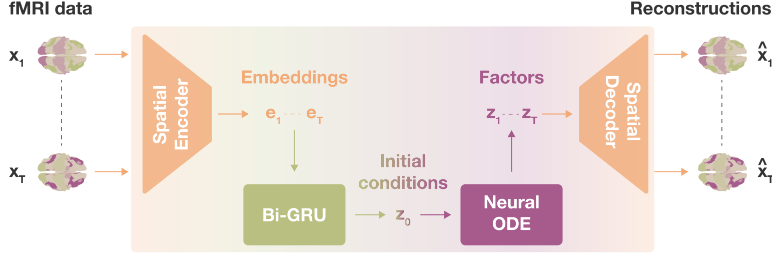

Figure 1: Our model architecture.

unroll the temporal dimension of the data. We demonstrate that our proposed method both more accurately relates to the task-related cognitive processes and directly models spatially localized task-based variance in the data. Finding underlying dynamical models can highlight potential mechanisms for further study through intervention and can inform potentially fruitful future directions for causal inference research in the brain, such as with transcranial magnetic stimulation (TMS) [14].

For task fMRI data, the dynamical system we are modeling is the excitation and relaxation of the hemodynamic response. If we split the dataset into separate windows that correspond to different tasks, we can model each task as an autonomous dynamical system because we assume the only input is given at time $t = 0$, at the start of the task. Thus, the only vector required to model the autonomous dynamics of the fMRI task is the initial condition in the latent space $\mathbf{z}_0$. These initial conditions, together with $\mathrm{F}_{\boldsymbol{\theta}}$ (the temporal decoder in Figure 1) should learn to completely unroll the low-dimensional dynamics of each task. The dynamical system is described in Supplement 4.1. Due to the high dimensionality of the data (90k spatial dimensions), we first embed the original data ($\mathbf{x}_t \in \mathbb{R}^N$) to an embedding vector ($\mathbf{e}_t \in \mathbb{R}^d$) with the same dimensionality as the final latent vectors (the spatial encoder in Figure 1). A bi-directional GRU then learns a final embedding that parameterizes the initial condition, which is used by the temporal decoder to unroll the full latent timeseries. To map the latent vectors produced by the temporal decoder/dynamical system back to the original space, we use a spatial decoder, see Figure 1. Both to regularize the network [15, 16, 17] and to impose structure onto the initial conditions, we sample them from a variational distribution. Similar to a normal variational autoencoder (VAE), we train our sequential variational autoencoder with a reconstruction loss and a KL-divergence loss. Given the noise and high dimensionality of the signal, we vary the complexity of both the spatial encoder and the decoder by training models with both linear and non-linear (models with suffix -NL) spatial encoders and decoders.

2 Experiments & Discussion

Training We use task fMRI data surface data, with $N = 91282$ voxels, from the Human Connectome Project [18]. The tasks used are the motor, working memory, and relational processing tasks from the HCP dataset because the motor task has well-defined ground truth spatial localization, and the other two are complex cognitive processes, thus making for a harder classification task. To train the model, we split the dataset into training (70%), validation (10%), and test set (20%) that was held out until the final evaluation. Further, we separate the timeseries into non-overlapping windows (23, 41, and 27 timesteps for the motor, working memory, and relational task, respectively) because our model assumption is that the brain acts as an autonomous system without any inputs in those sub-task windows. We train separate models in an unsupervised manner for each task.
Each model was trained with 2, 4, 8, 16, 32, and 64 latent dimensions (the size of $\mathbf{e}_t$ and $\mathbf{z}_t$) and across four seeds. A stability analysis performed across folds is provided in Supplement 4.6 and more general training and dataset details are provided in Supplement 4.2. We compare our model against equivalent versions of our model without the addition of the dynamical system: PCA and a variational autoencoder (VAE). We compare two dynamical systems for our model: a recurrent neural network (RNN) and a neural ODE (NODE) [19], both are novel models in the context of voxelwise fMRI. All code is provided in Supplement 4.7.

Sub-task classification The first experiment evaluates how well the latent timeseries relates to the cognitive process evoked by the tasks being performed in the scanner. To do this, we train a logistic regression classifier on each timestep independently and calculate the average classification accuracy

Classification	Hand vs foot	Motor	Working memory	Relational	Memory types	Motor from visual
2 Dimensions	$0.60 \pm 2.8E-3$	$0.29 \pm 1.6E-3$	$0.59 \pm 4.8E-3$	$0.60 \pm 2.8E-3$	$0.36 \pm 8.7E-3$	$0.26 \pm 3.1E-3$
4 Dimensions	$\mathbf{0.94 \pm 5.3E-2}$***	$0.44 \pm 6.3E-2$	$\mathbf{0.83 \pm 6.4E-3}$***	$0.71 \pm 5.2E-3$	$\mathbf{0.55 \pm 6.2E-3}$***	$0.32 \pm 3.1E-3$
8 Dimensions	$\mathbf{0.99 \pm 1.3E-3}$***	$\mathbf{0.85 \pm 3.7E-3}$***	$\mathbf{0.87 \pm 5.5E-3}$***	$\mathbf{0.84 \pm 2.6E-2}$***	$\mathbf{0.83 \pm 2.0E-2}$***	$0.46 \pm 5.6E-3$
16 Dimensions	$\mathbf{1.00 \pm 9.7E-4}$***	$\mathbf{0.97 \pm 3.6E-3}$***	$\mathbf{0.90 \pm 5.2E-3}$***	$\mathbf{0.87 \pm 4.5E-3}$***	$\mathbf{0.87 \pm 1.7E-2}$***	$\mathbf{0.65 \pm 3.5E-3}$***
32 Dimensions	$\mathbf{1.00 \pm 2.4E-4}$***	$\mathbf{0.98 \pm 2.2E-3}$***	$\mathbf{0.96 \pm 1.9E-3}$***	$\mathbf{0.89 \pm 3.4E-3}$***	$\mathbf{0.93 \pm 4.0E-3}$***	$\mathbf{0.71 \pm 2.3E-3}$***
64 Dimensions	$\mathbf{1.00 \pm 1.5E-3}$***	$\mathbf{0.99 \pm 6.9E-4}$***	$\mathbf{0.96 \pm 2.1E-3}$***	$\mathbf{0.94 \pm 4.2E-3}$***	$\mathbf{0.96 \pm 2.6E-3}$***	$\mathbf{0.77 \pm 3.1E-3}$***

Table 1: NODE results, 'Dimensions' in the table refers to the number of latent dimensions in the model. Significance is calculated with respect to the PCA results, using an independent t-test over the test set. Standard deviations are calculated over seeds (different model initializations). Conditions under which our model is significantly better than PCA are made bold and are indicated by stars. To calculate the significance of the variance explained results, we independently correlate our model and PCA's spatial maps to the group average task map. We then compare the two test statistics using Fisher's z transform and use the normal distribution's survival function to obtain a p-value. $^{***} = p < 0.0005$.

across time. Given that our model encodes the full timeseries into initial conditions, we also assess whether the initial condition alone is a good predictor of the cognitive process. We split the motor task into three classification results: first, we simplify the problem to include only left-hand or left-foot tapping sub-task blocks and test the model discriminability between the two (Figure 2a). We also compare the classification of all five sub-tasks: left hand, left foot, right hand, right foot, and tongue tapping (Figure 2b). Lastly, we use voxels only from the visual area and predict which of the five motor sub-tasks is being performed (Figure 2f). Since subjects receive a visual cue in the scanner indicating the upcoming sub-task, we hypothesize that dynamics stemming from the visual region alone may sufficiently encode the motor sub-task they will perform. More in-depth results for this classification task are provided in Suplement 4.3 The working memory task is also evaluated with two different classification tasks. We classify if a timeseries is a 0-Back or 2-Back block (Figure 2c), and we classify what visual element subjects need to remember; places, bodies, faces, or tools (Figure 2e). Lastly, for the relational processing task, we only evaluate whether a timeseries is a relational or control block (Figure 2d).

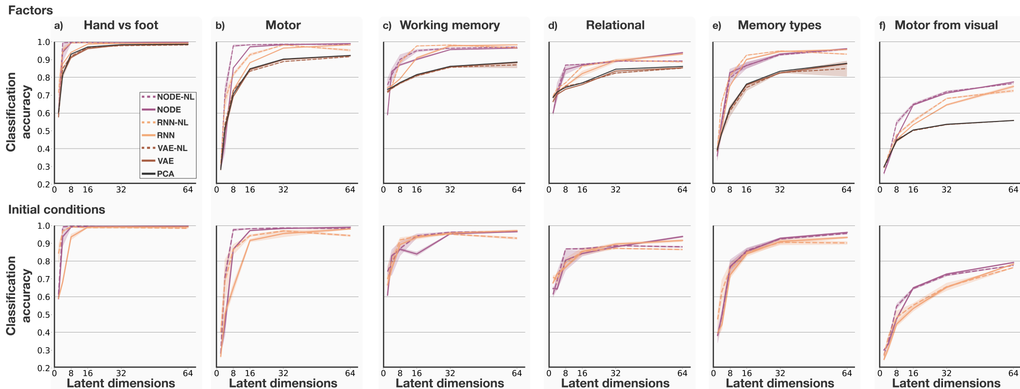

Figure 2: Sub-task classification accuracy from easy (left) to hard (right). Both our models, and even their initial conditions, outperform more common dimensionality reduction techniques, especially with increasing difficulty.

The results in Figure 2 and their statistical significance are also summarized in Table 1.

Based on Figure 2 and Table 1, we show that our method significantly outperforms PCA at higher latent dimensionalities across all sub-tasks, and visually outperforms the VAE, and the non-linear VAE across the classification tasks. Furthermore, the NODE-based dynamical system outperforms the RNN-based dynamical system on the motor, 'hand vs. foot', and 'motor from visual' tasks, even at low latent dimensionalities for the relational task. Additionally, the non-linear projection generally improves performance for the RNN-based model more clearly than the NODE-based model, although both benefit from it. Lastly, the performance of the initial conditions is often extremely close, or even higher than the factor's average performance across time.

Variance explained	Visual	Left foot	Left hand	Right foot	Right hand	Tongue
8 Dimensions	$\mathbf{0.91 \pm 9.3E-3}$***	$0.64 \pm 1.9E-2$	$0.52 \pm 2.5E-2$	$\mathbf{0.68 \pm 1.5E-2}$***	$\mathbf{0.62 \pm 3.1E-2}$***	$\mathbf{0.78 \pm 5.5E-3}$***
16 Dimensions	$\mathbf{0.94 \pm 1.7E-3}$***	$\mathbf{0.80 \pm 1.5E-2}$***	$\mathbf{0.68 \pm 6.7E-3}$***	$\mathbf{0.76 \pm 1.1E-2}$***	$0.70 \pm 1.7E-2$	$\mathbf{0.83 \pm 5.6E-3}$***
32 Dimensions	$0.95 \pm 8.6E-4$	$\mathbf{0.88 \pm 1.8E-3}$***	$\mathbf{0.84 \pm 3.8E-3}$***	$\mathbf{0.87 \pm 4.9E-3}$***	$\mathbf{0.82 \pm 3.7E-3}$***	$\mathbf{0.87 \pm 4.0E-3}$***
64 Dimensions	$0.96 \pm 1.4E-4$	$\mathbf{0.90 \pm 1.8E-3}$***	$\mathbf{0.88 \pm 1.5E-2}$***	$\mathbf{0.90 \pm 1.7E-3}$***	$\mathbf{0.86 \pm 1.9E-3}$***	$\mathbf{0.90 \pm 1.0E-3}$***

Table 2: NODE results, 'Dimensions' in the table refers to the number of latent dimensions in the model, and significance results are calculated the same way as in Table 1. $^{***} = p < 0.0005$.

Spatial specificity Our previous result raises the question of whether the transformation from the latent space to the voxel space (temporal decoder) itself is more task-specific as well. To understand if this is the case, we compare the version of our model with a linear projection to the baseline models. To understand how well the linear transformation from the latent space to the voxel space captures specific areas of the brain, we linearly regress each voxel to a brain map representing the motor homunculus. We only show results for 8 latent dimensions and higher because the variance explained for 2 and 4 latent dimensions is extremely low, the full plots are provided in Supplement 4.4. We also include an example of interpolation between the mean initial conditions for two sub-tasks in Figure 3a, to demonstrate the high reconstruction quality of our model and interpolation as a way to perform interpretability analyses on the model.

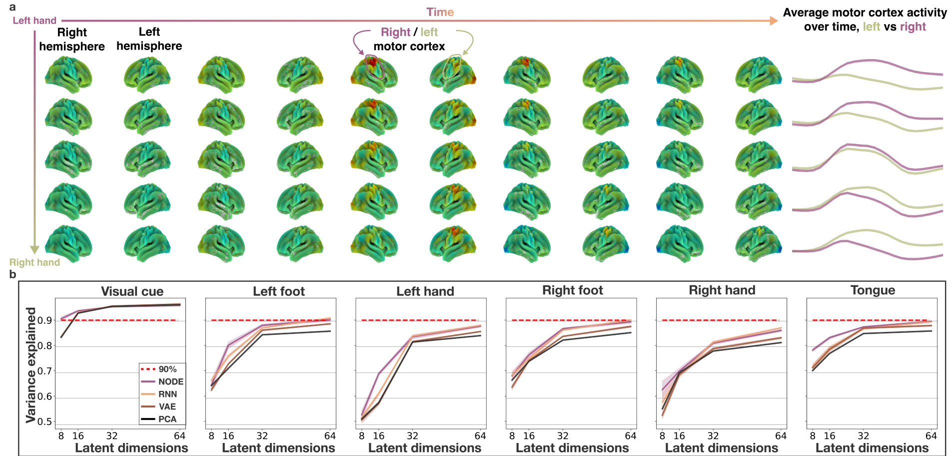

Figure 3: An interpolation between initial conditions in subfigure a, exhibiting interesting dynamic interpolations (right). Variance explained with respect to the motor group maps in subfigure b.

A summary of the results in Figure 3b with statistical analyses is provided in Table 2.

In Figure 3a we show an interpolation between reconstructions of left-hand to right-hand dynamics. The only input we vary is the initial condition we provide the NODE. Since the initial conditions are trained with a variational loss, the manifold they exist on is fairly smooth, enabling realistic interpolations. Note that the left part of the body is represented in the right hemisphere, and vice versa. The spatial specificity results in Figure 3b and Table 2 show that the NODE is better than the RNN and VAE at low dimensionalities, and significantly better than PCA for most tasks and latent dimensions.

Experiment & Evaluation An especially unique aspect of our method is the ability to analyze the behavior of the learned dynamical system. The most common approach to understanding a dynamical system is to look at points where the derivative is zero, the so-called fixed points. These fixed points are learned in our model during training (See Figure 4a), and by linearizing around them to calculate the Jacobian at the fixed point, we can analyze the behavior of the system through the eigenvalues of the Jacobian (See Figure 4). Since neuroimaging data is very noisy, the goal of this experiment is to establish the possibility of finding robust fixed points from fMRI data. Specifically, we want to stress how non-trivial it is to obtain the same eigenvalues across models trained with a different seed given the underspecification of neural networks [20]. In Supplement 4.5, we provide more information about this experiment. All analyses use the NODE-based system because it produces more accurate eigenvalues on simulation data [21].

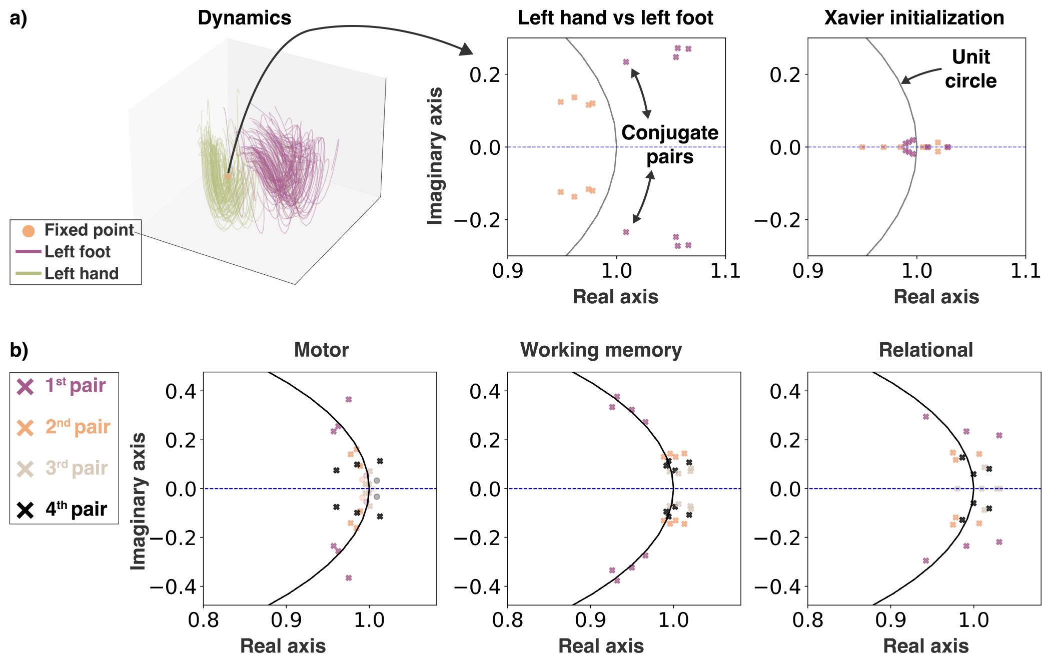

Figure 4: Subfigure a visualizes how we go from a fixed point in the latent space (left) to eigenvalues (middle) by linearizing around the fixed point and what the eigenvalues look like at initialization (right). We repeat each run for four different seeds. Subfigure b shows the eigenvalues for the fixed point we find for motor, working memory, and relational. The goal of subfigure b is to assess the robustness of the fixed points.

Findings For each seed, the model dimensions are sorted by the magnitude of the imaginary part of the eigenvalue (higher magnitude = higher rank), depicted by the color of each point in Figure 4. Thus, clustering of the same color in the figure corresponds to more robust fixed points across the 4 seeds. For the motor task, one model did not converge well, its eigenvalues are shown as circles in 4b, and the last fixed point did not converge for the relational task, so we did not obtain any eigenvalues. The eigenvalues of the working memory task seem to be tightly clustered and quite robust. For the relational task, the first conjugate pair always seems to capture roughly the same frequency (lie on the same y-axis), and the fourth pair for the motor task exhibits similar behavior. Overall, the eigenvalues are notably robust.

Discussion Our method consistently outperforms equivalent dimensionality reduction techniques that do not learn a dynamical system reduction techniques both in representing the dynamics of cognitive processes (Figure 1) and in the mapping between the latent and voxel space (Figure 2). We believe our model outperforms the other methods because it directly learns smooth dynamics for the fMRI data, and is thus constrained less likely to learn noise. Lastly, we show that the fixed points our model learns are robust across random initializations. Finding robust fixed points and their eigenvalues together with our other main results provides a foundation for future work where dynamics can be studied in patient populations.

3 Acknowledgments

Data were provided by the Human Connectome Project, WU-Minn Consortium (Principal Investigators: David Van Essen and Kamil Ugurbil; 1U54MH091657) funded by the 16 NIH Institutes and Centers that support the NIH Blueprint for Neuroscience Research; and by the McDonnell Center for Systems Neuroscience at Washington University. This material is supported by the National Science Foundation under Grant No. 2112455 and the National Institutes of Health grant #R01EB006841. Eloy Geenjaar was additionally supported by the Georgia Tech/Emory NIH/NIBIB Training Program in Computational Neural-engineering (T32EB025816).

References

[1] Russell A Poldrack. The future of fmri in cognitive neuroscience. *Neuroimage*, 62(2):1216–1220, 2012.

[2] Nathalie Tzourio-Mazoyer, Brigitte Landeau, Dimitri Papathanassiou, Fabrice Crivello, Octave Etard, Nicolas Delcroix, Bernard Mazoyer, and Marc Joliot. Automated anatomical labeling of activations in spm using a macroscopic anatomical parcellation of the mni mri single-subject brain. *Neuroimage*, 15(1):273–289, 2002.

[3] Matthew F Glasser, Timothy S Coalson, Emma C Robinson, Carl D Hacker, John Harwell, Essa Yacoub, Kamil Ugurbil, Jesper Andersson, Christian F Beckmann, Mark Jenkinson, et al. A multi-modal parcellation of human cerebral cortex. *Nature*, 536(7615):171–178, 2016.

[4] BT Thomas Yeo, Fenna M Krienen, Jorge Sepulcre, Mert R Sabuncu, Danial Lashkari, Marisa Hollinshead, Joshua L Roffman, Jordan W Smoller, Lilla Zöllei, Jonathan R Polimeni, et al. The organization of the human cerebral cortex estimated by intrinsic functional connectivity. *Journal of neurophysiology*, 2011.

[5] Erikki Oja and A Hyvarinen. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5):411–430, 2000.

[6] Martin J McKeown and Terrence J Sejnowski. Independent component analysis of fmri data: examining the assumptions. *Human brain mapping*, 6(5-6):368–372, 1998.

[7] Vince D Calhoun and Tülay Adali. Unmixing fmri with independent component analysis. *IEEE Engineering in Medicine and Biology Magazine*, 25(2):79–90, 2006.

[8] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[9] Stephen M Smith, Aapo Hyvärinen, Gaël Varoquaux, Karla L Miller, and Christian F Beckmann. Group-pca for very large fmri datasets. *Neuroimage*, 101:738–749, 2014.

[10] David Sussillo, Rafal Jozefowicz, LF Abbott, and Chethan Pandarinath. Lfads-latent factor analysis via dynamical systems. *arXiv preprint arXiv:1608.06315*, 2016.

[11] Chethan Pandarinath, Daniel J O'Shea, Jasmine Collins, Rafal Jozefowicz, Sergey D Stavisky, Jonathan C Kao, Eric M Trautmann, Matthew T Kaufman, Stephen I Ryu, Leigh R Hochberg, et al. Inferring single-trial neural population dynamics using sequential auto-encoders. *Nature methods*, 15(10):805–815, 2018.

[12] Mohammad Reza Keshtkaran, Andrew R Sedler, Raeed H Chowdhury, Raghav Tandon, Diya Basrai, Sarah L Nguyen, Hansem Sohn, Mehrdad Jazayeri, Lee E Miller, and Chethan Pandarinath. A large-scale neural network training framework for generalized estimation of single-trial population dynamics. *Nature Methods*, pages 1–6, 2022.

[13] Stephen J Gotts, Adrian W Gilmore, and Alex Martin. Brain networks, dimensionality, and global signal averaging in resting-state fmri: Hierarchical network structure results in low-dimensional spatiotemporal dynamics. *NeuroImage*, 205:116289, 2020.

[14] Sven Bestmann, Christian C Ruff, Felix Blankenburg, Nikolaus Weiskopf, Jon Driver, and John C Rothwell. Mapping causal interregional influences with concurrent tms–fmri. *Experimental brain research*, 191:383–402, 2008.

[15] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[16] Xi Chen, Diederik P Kingma, Tim Salimans, Yan Duan, Prafulla Dhariwal, John Schulman, Ilya Sutskever, and Pieter Abbeel. Variational lossy autoencoder. *arXiv preprint arXiv:1611.02731*, 2016.

[17] Jinsong Su, Shan Wu, Deyi Xiong, Yaojie Lu, Xianpei Han, and Biao Zhang. Variational recurrent neural machine translation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[18] David C Van Essen, Stephen M Smith, Deanna M Barch, Timothy EJ Behrens, Essa Yacoub, Kamil Ugurbil, Wu-Minn HCP Consortium, et al. The wu-minn human connectome project: an overview. *Neuroimage*, 80:62–79, 2013.

[19] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[20] Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. Underspecification presents challenges for credibility in modern machine learning. *The Journal of Machine Learning Research*, 23(1):10237–10297, 2022.

[21] Andrew Sedler, Christopher Versteeg, and Chethan Pandarinath. Expressive architectures enhance interpretability of dynamics-based neural population models. *Neurons, Behavior, Data analysis, and Theory*, 2023.

[22] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[23] David Sussillo and Omri Barak. Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, 25(3):626–649, 2013.

[24] Matthew D Golub and David Sussillo. Fixedpointfinder: A tensorflow toolbox for identifying and characterizing fixed points in recurrent neural networks. *Journal of Open Source Software*, 3(31):1003, 2018.

[25] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

4 Supplements

4.1 Supplement I: Dynamical and GRU equations

Dynamical equations To learn the dynamics in a low-dimensional latent space, we parameterize a latent autonomous dynamical system of the following form. Autonomous dynamical systems are dynamical systems that do not require any inputs, and given an initial state can completely unroll the temporal dimension of the data.

$$\dot{\mathbf{z}}_t = \mathrm{F}_{\boldsymbol{\theta}}(\mathbf{z}_t)$$

Where $\mathbf{z}_t$ is the latent vector at timestep t and $\mathrm{F}_{\boldsymbol{\theta}}$ parameterizes the time evolution of the latent vectors. We discretize the latent vectors based on what time each fMRI volume is acquired. For $\mathrm{F}_{\boldsymbol{\theta}}$, we use a gated recurrent unit (GRU) [22] and a neural ordinary differential equation (NODE) [19]. Specifically, we can rewrite a discretized version of the equation as a mapping from $\mathbf{z}_t$ to $\mathbf{z}_{t+1}$ as follows.

$$\begin{aligned}\mathbf{z}_{t+1} &= \mathbf{z}_t + \dot{\mathbf{z}}_t \\ &= \mathbf{z}_t + \mathrm{F}_{\boldsymbol{\theta}}(\mathbf{z}_t)\end{aligned}$$

For the GRU, we can write this based on the hidden state of the GRU, the GRU equations are provided below. The GRU's hidden state dimensionality, however, needs to be bigger than the latent dimension to effectively learn the dynamics [21]. Low-dimensional dynamics often emerge from high-dimensional RNNs [23], so we use a linear mapping to obtain the latent vector at each timestep from the hidden state, as follows.

$$\begin{aligned}\mathbf{h}_{t+1} &= \mathrm{GRUCell}(\mathbf{h}_t) \\ \mathbf{z}_{t+1} &= \mathbf{h}_{t+1}\mathbf{W}_z + \mathbf{b}_z \\ &= \mathbf{z}_t + \mathrm{F}_{\boldsymbol{\theta}}(\mathbf{z}_t) \\ \mathrm{F}_{\theta}(\mathbf{h}_t) &= \mathrm{GRUCell}(\mathbf{h}_t)\mathbf{W}_z + \mathbf{b}_z - \mathbf{z}_t\end{aligned}$$

For the NODE, we parameterize $F_\theta(\cdot)$ as a multi-layer perceptron (MLP), and obtain the following equation.

$$\mathbf{z}_{t+1} = \mathbf{z}_t + \int_t^{t+1} \mathrm{MLP}(\mathbf{z}_t)\mathrm{dt} \tag{1}$$

We point out two important differences between the GRU and NODE. First, the GRU requires a larger hidden dimensionality for its state updates and does not directly update the latent vector. Second, because the NODE parameterizes the derivative instead of the next time step, it uses numerical solvers to calculate the integral in Equation (1), leading to smoother dynamics that can interpolate time points with a higher sampling rate than the data.

GRU equations The following equations describe the equations for a gated recurrent unit (GRU). The GRU consists of three gates, a reset gate $\mathbf{r}_t$, a new gate $\mathbf{n}_t$, and an update gate $\mathbf{u}_t$. Given that we assume an autonomous dynamical system, the equations do not contain an input $\mathbf{x}_t$, but only a hidden state $\mathbf{h}_t$. This leads to the following equations.

$$\begin{aligned}\mathbf{r}_{t+1} &= \sigma\left(\mathbf{W}_r\mathbf{h}_t + \mathbf{b}_r\right) \\ \mathbf{u}_{t+1} &= \sigma\left(\mathbf{W}_u\mathbf{h}_t + \mathbf{b}_u\right) \\ \mathbf{n}_{t+1} &= \mathrm{Tanh}\left(\mathbf{r}_{t+1} \odot \left(\mathbf{W}_{\mathbf{n}}\mathbf{h}_t + \mathbf{n}\right)\right) \\ \mathbf{h}_{t+1} &= \left(1 - \mathbf{u}_{t+1}\right) \odot \mathbf{n}_{t+1} + \mathbf{u}_{t+1} \odot \mathbf{h}_t\end{aligned}$$

4.2 Supplement II: Training and dataset details

Training setting Training is performed for 500 epochs, except for the fixed point results, those models are trained for up to 1000 epochs. We use a reduce-on-plateau learning rate scheduler, with a patience of 10 epochs, a reduction factor of 0.95, and a minimum learning rate of $1E-5$. We use gradient clipping, with a norm of 50, and also clip the variational standard deviations between $1E-9$ and 5 for the VAE. Furthermore, we use an annealing strategy to linearly increase the importance KL-divergence term between the first and 50th epoch. Furthermore, to balance out the mean squared error (MSE) (reconstruction error) term and the KL-divergence term, we multiply the MSE by 1000.

Dataset To ensure we use minimally pre-processed whole-brain data, we use task fMRI data from the Human Connectome Project [18]. The dataset consists of 1080, 1083, and 1040 subjects for the motor, relational, and working memory tasks, respectively. We use surface data, with $N = 91282$ voxels for each task, except for the task where we only use visual data, which has $N = 8788$ voxels. For the motor task, subjects are tasked with tapping either their left or right fingers, squeezing their left or right toes, or moving their tongue. These motor blocks are preceded by a visual cue that tells the subject what body part they should move. Each motor block is 12 seconds, and each visual cue is 3 seconds. For the working memory task, the subjects receive a 2.5-second visual cue informing them of the task type, and for the 0-Back memory condition, this cue also shows the target. Then, subjects are tasked with either remembering the target (0-Back) or whether the picture they see is the same picture from the 2-Back condition (i.e., 2 images prior). These two sub-tasks are done in independent blocks of 25 seconds, each block has 10 2.5-second sub-blocks. In total, there are 8 larger blocks, four for 0-Back and four for 2-Back. These blocks can be subdivided by target type: a tool, body, face, or place. Lastly, for the relational task, the subjects see two pairs of objects, one at the top and one at the bottom of the screen. They first need to decide how the top pair differs (either in shape or texture). Then, subjects should determine whether the bottom pair also differs similarly. This block is called the relational trial. For the control block, the subjects are shown "shape" or "texture" on the screen, and only one object is at the bottom of the screen. The subjects should determine whether the bottom object matches any of the top two objects in terms of the word. Each block lasts 18 seconds, with 4 3.5-second sub-blocks, with 500ms between them, for the relational blocks, and 5 2.8-second sub-blocks, with 400ms between them, for the control blocks. We chose these tasks because the motor task has well-defined ground truth spatial localization, and the other two are complex cognitive processes, thus making for a harder classification task. To train the model, we split the dataset into a training (70%), validation (10%), and test set (20%) that was held out until the final evaluation. To train the models, we separate the timeseries into non-overlapping windows (23, 41, and 27 timesteps for the motor, working memory, and relational task, respectively) that are as long as the minimum time between two sub-tasks. We perform the windowing because our model assumption is that it is an autonomous system without any inputs, within the windows there are no inputs, but the visual cue itself during the full timeseries is an input to the system. We train separate models for each task and evaluate the models based on the sub-task label belonging to each window. The model is not privy to these labels during training.

4.3 Supplement III: Motor from visual classification over time

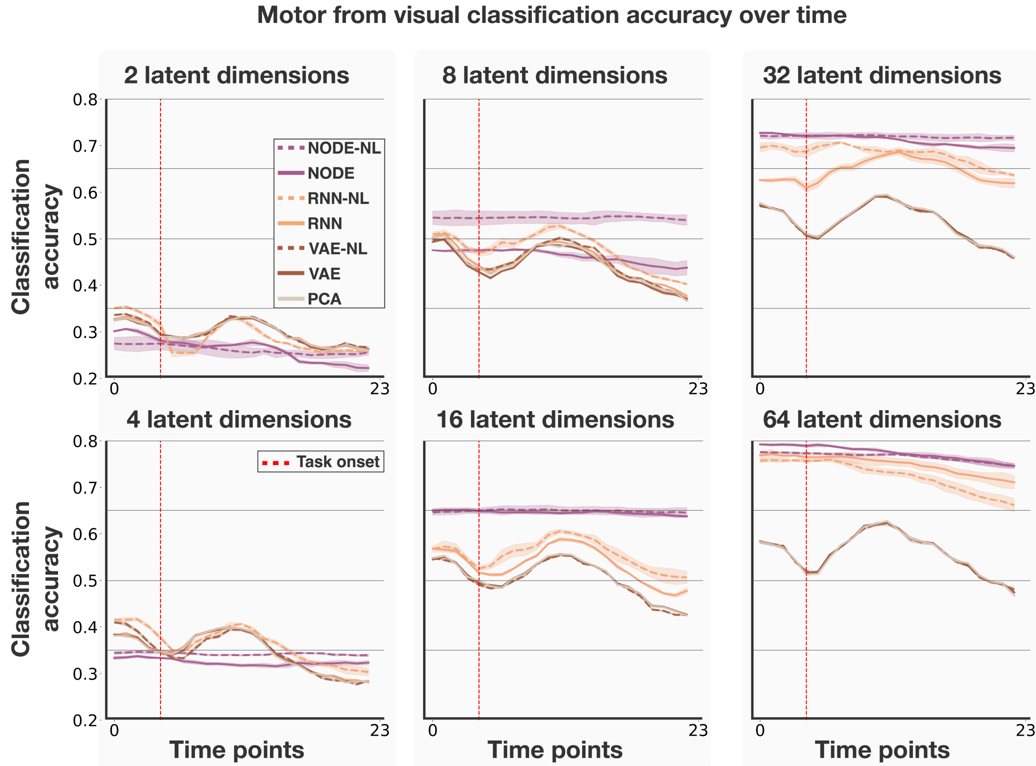

Figure 5: The classification accuracy for varying numbers of latent dimensions over time. A separate logistic regression model is fit for each timestep. The RNN, VAE, and PCA have a shape resembling a hemodynamic response after the motor task starts.

Figure 5 shows the in-depth results for the 'motor from visual' task, where we classify what motor task is being performed using only the voxels in the visual area. Our hypotheses for why this should work are twofold: one is that the information is encoded in the visual region from the visual cue that starts at the beginning of the timeseries in Figure 5, or because of feedback connections from the motor area to the visual cortex. Our results, presented in Figure 5, provide support for both of these hypotheses. Peaks in accuracy are seen immediately after the presentation of the visual cue (very start of the timeseries), as well as shortly after the start of the task block (dashed red line). The latter peak closely resembles the hemodynamic response curve. Thus, these resultant peaks could be attributable to the visual cue and task-relevant motor-visual feedback connections, respectively. However, this is only true for the PCA, VAE, and RNN models. For the NODE, which performs the best, especially for more than 8 latent dimensions, the classification accuracy is relatively stable over time and slightly higher at the beginning. This is likely because the model has learned a different trajectory for each motor sub-task that is separable over time.

4.4 Supplement IV: Full spatial specificity plots

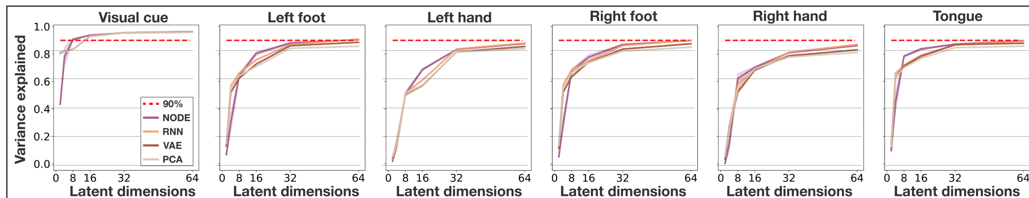

Figure 6: The full version of the spatial specificity figure (Figure 3). Due to the low variance explained for low numbers of latent dimensions (2 and 4), the differences between models are hard to distinguish. For completeness, we include this figure.

Figure 6 shows the full spatial specificity results, including 2 and 4 latent dimensions. The plot was shortened because the variance explained values for those two latent dimensions were so low that they made it hard to distinguish methods.

4.5 Supplement V: Fixed point search

Our method for obtaining fixed points is largely based on previous work [23, 24, 21]. Given that NODEs can model potentially highly nonlinear flows, it can be hard to find the location of the fixed point, especially for higher-dimensional spaces. To ensure the search for a fixed point is not exhaustive, previous works have proposed a minimization approach to find the location of fixed points with gradient descent-based approaches [23, 24]. To do this, we first generate latent timeseries for multiple subjects and use each latent location as an initial starting point to find fixed points. We turn off gradients for all parameters within the model and enable them for each of the latent points. We then optimize over the latent points such that their location in the latent space minimizes the norm of the derivative, as defined by the flow in the NODE, with the Adam optimizer [25]. We optimize until a maximum number of iterations (10000) are reached, and latent points whose derivative is within a certain tolerance ($1E-10$) are used as fixed points. We use 0.01 as a learning rate and reduce the learning rate by 90% after every 2000 iterations.

4.6 Supplement VI: Training folds

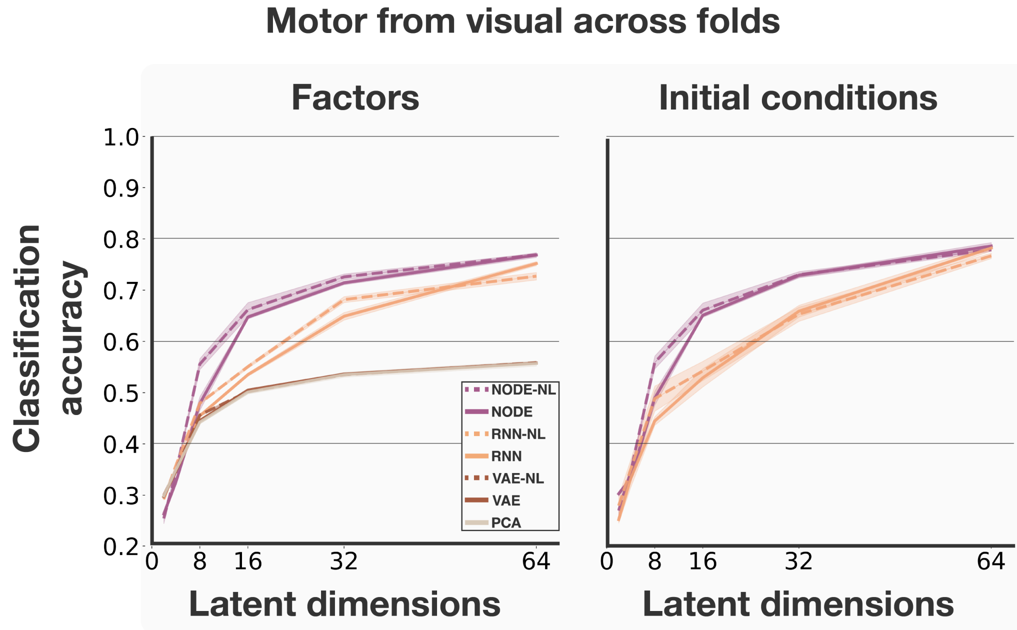

Figure 7: Results for the hardest classification task, the prediction of what motor task is being performed from voxels in the visual area, but averaged across folds instead of seeds. The results are the same as in the main text, and the standard deviation is small, indicating our methods' robustness across training folds.

To ensure that our model is not only robust across different initialization seeds but also across different training folds, we performed a 10-fold split for the 'motor from visual' task and assessed both the robustness and whether the results prevailed. We chose to hold out the test set until all other experiments were finished and only then updated the figures with results on the test set instead of the validation set. We chose to reduce any bias while developing the methods and performing experiments. However, making this choice, we could not assess the robustness across the whole dataset because the test fold was held out. Hence, we performed this experiment to ensure that the test set itself was not biased, and from the results shown in Figure 7, we conclude that there is little variation between results from different training and test folds.

4.7 Code

In alphabetical order

4.7.1 Criterion.py

```python
from torch import nn
from torch import distributions as D


class ELBO(nn.Module):
    def __init__(self, beta, mse_mult=1000):
        super().__init__()
        self.mse_mult = mse_mult
        self.mse_loss = nn.MSELoss(reduction='none')
        self.beta = beta
        # This is used for linear annealing
        self.kl_weight = 0.0

    def forward(self, model_output, x, validation=False):
        batch, timesteps = x.size()[:2]
        x = x.view(batch, timesteps, -1)
        # Calculate mean-squared error loss
        mse_loss = self.mse_loss(
            model_output['x_hat'],
            model_output['x']).mean(dim=(0, -1)).view(-1)
        # Calculate the KL-divergence for z0
        kl_loss = D.kl.kl_divergence(
            model_output['dist'], D.Normal(0., 1.)).mean(-1)
        if len(kl_loss.size()) > 1:
            # Average over timesteps for VAE
            # For the VAE all timesteps are distributions
            # not just z0
            kl_loss = kl_loss.mean(0)
        # Ensure that the annealing does not affect the early stopping
        if validation:
            # MSE mult is a way to balance the two losses
            loss = (self.mse_mult * mse_loss
                    + self.beta * kl_loss).mean(0)
        else:
            loss = (self.mse_mult * mse_loss
                    + self.beta * self.kl_weight * kl_loss).mean(0)
            # Anneal for roughly the first 50 epochs
            self.kl_weight = min(1, self.kl_weight + 1/(50 * 190))
        # Return the mse and kl-divergence for recording purposes
        crit_out = {
            'mse': mse_loss.detach().mean(0),
            'kl': kl_loss.detach().mean(0),
            }
        return loss, crit_out
```

4.7.2 dataset.py

```python
import numpy as np
import pandas as pd
from torch.utils.data import Dataset


# Each dataset refers to a csv file generated using prep_*.py
# We represent important parameters as properties of the class
# The first dataset is the 'motor from visual' class
class HCPVisual(Dataset):
    def __init__(self, data_type, *args, **kwargs):
        self.data_type = data_type
        self.df = pd.read_csv('/path/to/visual.csv', index_col=0)
```

```
13         if 'fold' in kwargs.keys():
14             fold = kwargs['fold']
15             # Roll the dataframe for different folds
16             roll_value = int(0.1 * self.df.shape[0] * fold)
17             # By rolling the dataset we get different
18             # training, validation, and test splits for each fold
19             roller = lambda x: np.roll(x, roll_value)
20             rolled_df = self.df.apply(roller, axis=0)
21             rolled_df.index = roller(rolled_df.index.values)
22             self.df = rolled_df.copy()
23         if self.data_type == 'train':
24             self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy
    ()
25         elif self.data_type == 'valid':
26             self.df = self.df.iloc[
27                 int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
28         elif self.data_type == 'test':
29             self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
30         elif self.data_type == 'train_valid':
31             self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()
32
33     @property
34     def window_size(self):
35         return 23
36
37     @property
38     def num_tasks(self):
39         return 5
40
41     @property
42     def num_occurrences(self):
43         return 2
44
45     @property
46     def paths(self):
47         return self.df['fmri'].tolist(), self.df['targets'].tolist()
48
49     def __len__(self):
50         return self.df.shape[0]
51
52
53 # The left.csv file is generated in prep_motor.py
54 # Left foot vs left hand
55 class HCPLeft(Dataset):
56     def __init__(self, data_type, *args, **kwargs):
57         self.data_type = data_type
58         self.df = pd.read_csv('/path/to/left.csv', index_col=0)
59         if self.data_type == 'train':
60             self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy
    ()
61         elif self.data_type == 'valid':
62             self.df = self.df.iloc[
63                 int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
64         elif self.data_type == 'test':
65             self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
66         elif self.data_type == 'train_valid':
67             self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()
68
69     @property
```

```
    def window_size(self):
        return 23

    @property
    def num_tasks(self):
        return 2

    @property
    def num_occurrences(self):
        return 2

    @property
    def paths(self):
        return self.df['fmri'].tolist(), self.df['targets'].tolist()

    def __len__(self):
        return self.df.shape[0]


class HCPMotor(Dataset):
    def __init__(self, data_type, *args, **kwargs):
        self.data_type = data_type
        self.df = pd.read_csv('/path/to/motor.csv', index_col=0)
        # Creating the data splits
        if self.data_type == 'train':
            self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy()
        elif self.data_type == 'valid':
            self.df = self.df.iloc[
                int(self.df.shape[0] * 0.7):int(self.df.shape[0] * 0.8)].copy()
        elif self.data_type == 'test':
            self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy()
        elif self.data_type == 'train_valid':
            self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy()

    @property
    def window_size(self):
        return 23

    @property
    def num_tasks(self):
        return 5

    @property
    def num_occurrences(self):
        return 2

    @property
    def paths(self):
        return self.df['fmri'].tolist(), self.df['targets'].tolist()

    def __len__(self):
        return self.df.shape[0]


class HCPWM(Dataset):
    def __init__(self, data_type, *args, **kwargs):
        self.data_type = data_type
        self.df = pd.read_csv('/path/to/wm.csv', index_col=0)
        if self.data_type == 'train':
            self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy()
```

```
130         elif self.data_type == 'valid':
131             self.df = self.df.iloc[
132                 int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
133         elif self.data_type == 'test':
134             self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
135         elif self.data_type == 'train_valid':
136             self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()
137 
138     @property
139     def window_size(self):
140         return 41
141 
142     @property
143     def num_tasks(self):
144         return 2
145 
146     @property
147     def num_occurrences(self):
148         return 4
149 
150     @property
151     def paths(self):
152         return self.df['fmri'].tolist(), self.df['targets'].tolist()
153 
154     def __len__(self):
155         return self.df.shape[0]
156 
157 
158 class HCPRelational(Dataset):
159     def __init__(self, data_type, *args, **kwargs):
160         self.data_type = data_type
161         self.df = pd.read_csv('/path/to/relational.csv', index_col=0)
162         if self.data_type == 'train':
163             self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy
    ()
164         elif self.data_type == 'valid':
165             self.df = self.df.iloc[
166                 int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
167         elif self.data_type == 'test':
168             self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
169         elif self.data_type == 'train_valid':
170             self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()
171 
172     @property
173     def window_size(self):
174         return 27
175 
176     @property
177     def num_tasks(self):
178         return 2
179 
180     @property
181     def num_occurrences(self):
182         return 3
183 
184     @property
185     def paths(self):
186         return self.df['fmri'].tolist(), self.df['targets'].tolist()
187 
```

```
188     def __len__(self):
189         return self.df.shape[0]
190
191
192 # The long datasets are used for the fixed point finding
193 # (find_fixed_points.py)
194 class HCPMotorLong(Dataset):
195     def __init__(self, data_type, *args, **kwargs):
196         self.data_type = data_type
197         self.df = pd.read_csv('/path/to/motor_long.csv', index_col=0)
198         if self.data_type == 'train':
199             self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy
    ()
200         elif self.data_type == 'valid':
201             self.df = self.df.iloc[
202                 int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
203         elif self.data_type == 'test':
204             self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
205         elif self.data_type == 'train_valid':
206             self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()
207
208     @property
209     def window_size(self):
210         return 42
211
212     @property
213     def num_tasks(self):
214         return 3
215
216     @property
217     def num_occurrences(self):
218         return 1
219
220     @property
221     def paths(self):
222         return self.df['fmri'].tolist(), self.df['targets'].tolist()
223
224     def __len__(self):
225         return self.df.shape[0]
226
227
228 class HCPWMLong(Dataset):
229     def __init__(self, data_type, *args, **kwargs):
230         self.data_type = data_type
231         self.df = pd.read_csv('/path/to/wm_long.csv', index_col=0)
232         if self.data_type == 'train':
233             self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy
    ()
234         elif self.data_type == 'valid':
235             self.df = self.df.iloc[
236                 int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
237         elif self.data_type == 'test':
238             self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
239         elif self.data_type == 'train_valid':
240             self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()
241
242     @property
243     def window_size(self):
244         return 57
```

```
    @property
    def num_tasks(self):
        return 4

    @property
    def num_occurrences(self):
        return 1

    @property
    def paths(self):
        return self.df['fmri'].tolist(), self.df['targets'].tolist()

    def __len__(self):
        return self.df.shape[0]


class HCPRelationalLong(Dataset):
    def __init__(self, data_type, *args, **kwargs):
        self.data_type = data_type
        self.df = pd.read_csv('/path/to/relational_long.csv',
    index_col=0)
        if self.data_type == 'train':
            self.df = self.df.iloc[:int(self.df.shape[0] * 0.7)].copy
    ()
        elif self.data_type == 'valid':
            self.df = self.df.iloc[
                int(self.df.shape[0] * 0.7):int(self.df.shape[0] *
    0.8)].copy()
        elif self.data_type == 'test':
            self.df = self.df.iloc[int(self.df.shape[0] * 0.8):].copy
    ()
        elif self.data_type == 'train_valid':
            self.df = self.df.iloc[:int(self.df.shape[0] * 0.8)].copy
    ()

    @property
    def window_size(self):
        return 47

    @property
    def num_tasks(self):
        return 3

    @property
    def num_occurrences(self):
        return 1

    @property
    def paths(self):
        return self.df['fmri'].tolist(), self.df['targets'].tolist()

    def __len__(self):
        return self.df.shape[0]

    def __getitem__(self, ix):
        pass
```

4.7.3 embed_results_folds.py

```
import torch
import importlib
import numpy as np
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
```

```
6 from utils import (purge_logs, get_log_string, get_default_config,
7                    init_model, subset_configs, create_dataloaders,
8                    embed_data)
9
10
11 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
12
13 redo = False
14 datasets = ['HCPVisual']
15 for dataset_name in datasets:
16     # Obtain all experiments that correspond
17     # to any of these variables
18     experiment_config = {
19         'dataset': [dataset_name],
20         'hidden_sizes': [[], [128]],
21         'temporal_hidden_sizes': [[128]],
22         'beta': [1.0],
23         'latent_dim': [2, 4, 8, 16, 32, 64],
24         'model': ['NODE', 'RNN', 'VAE'],
25         'seed': [42],
26         'dropout': [0.1]
27     }
28     data_module = importlib.import_module('dataset')
29     dataset = getattr(data_module, dataset_name)
30
31     # Load the datasets
32     train_dataset = dataset('train')
33     valid_dataset = dataset('valid')
34     test_dataset = dataset('test')
35
36     # Training and validation splits
37     train_df = train_dataset.df.copy()
38     valid_df = valid_dataset.df.copy()
39     test_df = test_dataset.df.copy()
40
41     # Obtain number of subjects for each split
42     train_subjects = len(train_dataset)
43     valid_subjects = len(valid_dataset)
44     test_subjects = len(test_dataset)
45
46     # For these models we use the training and valid set to train
    classifiers
47     train_subjects = train_subjects + valid_subjects
48
49     # Get the base configuration, and change the input size
50     # for the 'motor from visual' task, since it only uses the
51     # visual cortex voxels
52     base_config = get_default_config([None])
53     if dataset_name == 'HCPVisual':
54         base_config['input_size'] = 8788
55
56     # Obtain all trained models that correspond to the experiment
    config
57     config_list = purge_logs(base_config)
58     config_list = subset_configs(experiment_config, config_list)
59
60     # Get the dataset and use the A40 GPU
61     dataset_module = importlib.import_module('dataset')
62     dataset = getattr(dataset_module, dataset_name)
63     base_config['batch_size'] = 1
64     base_config['gpu'] = 'A40'
65
66     # Obtain (and pre-load data into RAM) for each of the folds
67     (train_loader, valid_loader, test_loader), _ \
68         = create_dataloaders(dataset, base_config)
```

```
    # Get information from dataset
    num_tasks = train_dataset.num_tasks
    num_occurrences = train_dataset.num_occurrences
    window_size = train_dataset.window_size

    # Start performing inference for all pre-trained models
    for config in config_list:
        # Loop over the folds
        folds = list(range(10))
        model_module = importlib.import_module('model')
        model_type = getattr(model_module, config['model'])
        for fold in folds:
            log_path = get_log_string(config) / f'fold_{fold}'
            flag = redo
            # Check if this config has all results
            for dataset_file in ['task_factor_results.npy',
                                 'task_init_results.npy']:
                flag = flag or (not (log_path / dataset_file).is_file
    ())
            if flag:
                # Initialize model
                model_path = get_log_string(config)
                model_path = model_path / f'fold_{fold}'
                print(model_path)
                model = init_model(model_type, config)
                # Load model from its model path
                model_state_dict = torch.load(
                    model_path / 'model.pt', map_location='cpu')
                model.load_state_dict(model_state_dict)
                model.eval()
                model = model.to(device)

                # Embed the data into initial conditions and factors
                train_inits, train_factors = embed_data(train_loader,
    model)
                valid_inits, valid_factors = embed_data(valid_loader,
    model)
                test_inits, test_factors = embed_data(test_loader,
    model)
                # The shapes of these vectors are:
                # inits: (subjects, num_tasks * num_occurrences,
    latent_dim)
                # factors: (subjects,
                # window_size, num_tasks * num_occurrences, latent_dim
    )

                # Calculate task classification using factors
                train_factors_np = train_factors.cpu().numpy()
                valid_factors_np = valid_factors.cpu().numpy()
                test_factors_np = test_factors.cpu().numpy()
                # Concatenate the training and validation factors
                train_factors_np = np.concatenate(
                    (train_factors_np, valid_factors_np), axis=0)
                task_factor_results = np.zeros((window_size, ))
                for t in range(window_size):
                    # Train a separate logistic regression model
                    # for each timestep
                    lr = LogisticRegression(max_iter=10000, n_jobs=-1)
                    x_train = np.reshape(
                        train_factors_np[:, t],
                        (train_subjects * num_tasks * num_occurrences,
                         config['latent_dim']))
                    x_test = np.reshape(
                        test_factors_np[:, t],
```

```
128                         (test_subjects * num_tasks * num_occurrences,
129                          config['latent_dim']))
130                     # We can create the labels based on the index
131                     # within the multi-dim array
132                     y = np.arange(num_tasks)[np.newaxis, :, np.newaxis
    ]
133                     y_train = np.tile(
134                         y,
135                         (train_subjects, 1, num_occurrences)).flatten
    ()
136                     y_test = np.tile(
137                         y,
138                         (test_subjects, 1, num_occurrences)).flatten()
139                     lr.fit(x_train, y_train)
140                     task_factor_results[t] = lr.score(x_test, y_test)
141                     # Delete the classifier after each timestep
142                     del lr
143 
144                 np.save(log_path / 'task_factor_results.npy',
145                         task_factor_results)
146 
147                 # Calculate task classification for the inits
148                 if config['model'] != 'VAE':
149                     task_init_results = np.zeros((1, ))
150                     train_inits_np = train_inits.cpu().numpy()
151                     valid_inits_np = valid_inits.cpu().numpy()
152                     # Concatenate the training and validation set
153                     train_inits_np = np.concatenate(
154                         (train_inits_np, valid_inits_np), axis=0)
155                     test_inits_np = test_inits.cpu().numpy()
156                     if config['model'] == 'RNN':
157                         train_inits_np = np.reshape(
158                             train_inits_np, (-1, train_inits_np.shape
    [-1]))
159                         test_inits_np = np.reshape(
160                             test_inits_np, (-1, test_inits_np.shape
    [-1]))
161                         pca = PCA(n_components=config['latent_dim'])
162                         # For the RNN model, we need to transform the
163                         # initial conditions to the latent dimension
164                         # because it can't train well with a hidden
    size
165                         # that is equal to the latent dimension. Thus,
166                         # the initial condition is also higher-
    dimensional
167                         # (this would be the first hidden state)
168                         train_inits_np = pca.fit_transform(
    train_inits_np)
169                         test_inits_np = pca.transform(test_inits_np)
170                         train_inits_np = np.reshape(
171                             train_inits_np,
172                             (train_subjects, num_tasks,
173                              num_occurrences, config['latent_dim']))
174                         test_inits_np = np.reshape(
175                             test_inits_np,
176                             (test_subjects, num_tasks,
177                              num_occurrences, config['latent_dim']))
178 
179                     # Perform the classification
180                     lr = LogisticRegression(max_iter=10000, n_jobs=-1)
181                     x_train = np.reshape(
182                         train_inits_np,
183                         (train_subjects * num_tasks * num_occurrences,
184                          config['latent_dim']))
185                     x_test = np.reshape(
```

```
186                             test_inits_np,
187                             (test_subjects * num_tasks * num_occurrences,
188                              config['latent_dim']))
189                     # We can create the labels based on the index
190                     # within the multi-dim array
191                     y = np.arange(num_tasks)[np.newaxis, :, np.newaxis
    ]
192                     y_train = np.tile(
193                         y,
194                         (train_subjects, 1, num_occurrences)).flatten
    ()
195                     y_test = np.tile(
196                         y,
197                         (test_subjects, 1, num_occurrences)).flatten()
198                     lr.fit(x_train, y_train)
199                     task_init_results[0] = lr.score(x_test, y_test)
200                     np.save(log_path / 'task_init_results.npy',
201                             task_init_results)
202     # After finishing with a certain dataset, free the pre-loaded
203     # dataset from memory
204     del train_loader
205     del valid_loader
206     del test_loader
```

4.7.4 embed_results.py

```
1  import torch
2  import importlib
3  import numpy as np
4  from sklearn.decomposition import PCA
5  from sklearn.linear_model import LogisticRegression
6  from utils import (purge_logs, get_log_string, get_default_config,
7                     subset_configs, create_dataloaders, embed_data,
8                     load_model_from_config)
9
10 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
11
12 redo = True
13 datasets = ['HCPLeft', 'HCPMotor', 'HCPRelational', 'HCPVisual', '
    HCPWM']
14 for dataset_name in datasets:
15     # Obtain all experiments that correspond
16     # to any of these variables
17     experiment_config = {
18         'dataset': [dataset_name],
19         'hidden_sizes': [[], [128]],
20         'temporal_hidden_sizes': [[128]],
21         'beta': [1.0],
22         'latent_dim': [2, 4, 8, 16, 32, 64],
23         'model': ['NODE'],
24         'seed': [42, 1337, 9999, 1111],
25         'dropout': [0.1]
26     }
27     data_module = importlib.import_module('dataset')
28     dataset = getattr(data_module, dataset_name)
29
30     # Load the datasets
31     train_dataset = dataset('train')
32     valid_dataset = dataset('valid')
33     test_dataset = dataset('test')
34
35     # Training and validation splits
36     train_df = train_dataset.df.copy()
37     valid_df = valid_dataset.df.copy()
38     test_df = test_dataset.df.copy()
```

```
    # Obtain number of subjects for each split
    train_subjects = len(train_dataset)
    valid_subjects = len(valid_dataset)
    test_subjects = len(test_dataset)

    # For these models we use the training and valid set to train classifiers
    train_subjects = train_subjects + valid_subjects

    # Get the base configuration, and change the input size
    # for the 'motor from visual' task, since it only uses the
    # visual cortex voxels
    base_config = get_default_config([None])
    if dataset_name == 'HCPVisual':
        base_config['input_size'] = 8788

    # Obtain all trained models that correspond to the experiment config
    config_list = purge_logs(base_config)
    config_list = subset_configs(experiment_config, config_list)

    # Get the dataset and use the A40 GPU
    dataset_module = importlib.import_module('dataset')
    dataset = getattr(dataset_module, dataset_name)
    base_config['batch_size'] = 1
    base_config['gpu'] = 'A40'

    # Obtain (and pre-load data into RAM) for each of the folds
    (train_loader, valid_loader, test_loader), _ \
        = create_dataloaders(dataset, base_config)

    # Get information from dataset
    num_tasks = train_dataset.num_tasks
    num_occurrences = train_dataset.num_occurrences
    window_size = train_dataset.window_size

    # Start performing inference for all pre-trained models
    for config in config_list:
        log_path = get_log_string(config)
        flag = redo
        # Check if this config has all results
        for dataset_file in ['task_factor_results.npy',
                             'task_init_results.npy']:
            flag = flag or (not (log_path / dataset_file).is_file())

        if flag:
            # Initialize model
            model = load_model_from_config(config)
            model = model.to(device)

            # Embed the data into initial conditions and factors
            train_inits, train_factors = embed_data(train_loader, model)
            valid_inits, valid_factors = embed_data(valid_loader, model)
            test_inits, test_factors = embed_data(test_loader, model)
            # The shapes of these vectors are:
            # inits: (subjects, num_tasks * num_occurrences, latent_dim)
            # factors: (subjects,
            # window_size, num_tasks * num_occurrences, latent_dim)

            # Calculate task classification using factors
            train_factors_np = train_factors.cpu().numpy()
```

```
        valid_factors_np = valid_factors.cpu().numpy()
        test_factors_np = test_factors.cpu().numpy()
        # Concatenate the training and validation factors
        train_factors_np = np.concatenate(
            (train_factors_np, valid_factors_np), axis=0)
        task_factor_results = np.zeros((window_size, ))
        for t in range(window_size):
            # Train a separate logistic regression model
            # for each timestep
            lr = LogisticRegression(max_iter=10000, n_jobs=-1)
            x_train = np.reshape(
                train_factors_np[:, t],
                (train_subjects * num_tasks * num_occurrences,
                 config['latent_dim']))
            x_test = np.reshape(
                test_factors_np[:, t],
                (test_subjects * num_tasks * num_occurrences,
                 config['latent_dim']))
            # We can create the labels based on the index
            # within the multi-dim array
            y = np.arange(num_tasks)[np.newaxis, :, np.newaxis]
            y_train = np.tile(
                y, (train_subjects, 1, num_occurrences)).flatten()
            y_test = np.tile(
                y, (test_subjects, 1, num_occurrences)).flatten()
            lr.fit(x_train, y_train)
            task_factor_results[t] = lr.score(x_test, y_test)
            # Delete the classifier after each timestep
            del lr
        np.save(log_path / 'task_factor_results.npy',
task_factor_results)

        # Calculate task classification using inits
        if config['model'] != 'VAE':
            task_init_results = np.zeros((1, ))
            train_inits_np = train_inits.cpu().numpy()
            valid_inits_np = valid_inits.cpu().numpy()
            # Concatenate the training and validation set
            train_inits_np = np.concatenate(
                (train_inits_np, valid_inits_np), axis=0)
            test_inits_np = test_inits.cpu().numpy()
            if config['model'] == 'RNN':
                train_inits_np = np.reshape(
                    train_inits_np, (-1, train_inits_np.shape[-1])
)
                test_inits_np = np.reshape(
                    test_inits_np, (-1, test_inits_np.shape[-1]))
                pca = PCA(n_components=config['latent_dim'])
                # For the RNN model, we need to transform the
                # initial conditions to the latent dimension
                # because it can't train well with a hidden size
                # that is equal to the latent dimension. Thus,
                # the initial condition is also higher-dimensional
                # (this would be the first hidden state)
                train_inits_np = pca.fit_transform(train_inits_np)
                test_inits_np = pca.transform(test_inits_np)
                train_inits_np = np.reshape(
                    train_inits_np,
                    (train_subjects, num_tasks, num_occurrences,
                     config['latent_dim']))
                test_inits_np = np.reshape(
                    test_inits_np,
                    (test_subjects, num_tasks, num_occurrences,
                     config['latent_dim']))

```

```
            # Perform the classification
            lr = LogisticRegression(max_iter=10000, n_jobs=-1)
            x_train = np.reshape(
                train_inits_np,
                (train_subjects * num_tasks * num_occurrences,
                 config['latent_dim']))
            x_test = np.reshape(
                test_inits_np,
                (test_subjects * num_tasks * num_occurrences,
                 config['latent_dim']))
            # We can create the labels based on the index
            # within the multi-dim array
            y = np.arange(num_tasks)[np.newaxis, :, np.newaxis]
            y_train = np.tile(
                y, (train_subjects, 1, num_occurrences)).flatten()
            y_test = np.tile(
                y, (test_subjects, 1, num_occurrences)).flatten()
            lr.fit(x_train, y_train)
            task_init_results[0] = lr.score(x_test, y_test)
            np.save(log_path / 'task_init_results.npy',
task_init_results)

        # For the WM dataset we also perform classification
        # over the occurrences
        if dataset_name == 'HCPWM':
            # Calculate task classification using factors
            train_factors_np = train_factors.cpu().numpy()
            valid_factors_np = valid_factors.cpu().numpy()
            test_factors_np = test_factors.cpu().numpy()
            # Concatenate the training and validation set
            train_factors_np = np.concatenate(
                (train_factors_np, valid_factors_np), axis=0)
            occurrence_factor_results = np.zeros((window_size, ))
            for t in range(window_size):
                # Similar to the task classification,
                # except labels are created differently
                lr = LogisticRegression(max_iter=10000, n_jobs=-1)
                x_train = np.reshape(
                    train_factors_np[:, t],
                    (train_subjects * num_tasks * num_occurrences,
                     config['latent_dim']))
                x_test = np.reshape(
                    test_factors_np[:, t],
                    (test_subjects * num_tasks * num_occurrences,
                     config['latent_dim']))
                # Labels are now created based on index of
occurrence
                y = np.arange(num_occurrences)[np.newaxis, np.
newaxis]
                y_train = np.tile(
                    y, (train_subjects, num_tasks, 1)).flatten()
                y_test = np.tile(
                    y, (test_subjects, num_tasks, 1)).flatten()
                lr.fit(x_train, y_train)
                occurrence_factor_results[t] = lr.score(x_test,
y_test)
                # Delete the classifier after each timestep
                del lr
            np.save(log_path / 'occurrence_factor_results.npy',
                    occurrence_factor_results)

            # Init classification for occurrences
            if config['model'] != 'VAE':
                occurrence_init_results = np.zeros((1, ))
                lr = LogisticRegression(max_iter=10000, n_jobs=-1)
```

```
223                     x_train = np.reshape(
224                         train_inits_np,
225                         (train_subjects * num_tasks * num_occurrences,
226                          config['latent_dim']))
227                     x_test = np.reshape(
228                         test_inits_np,
229                         (test_subjects * num_tasks * num_occurrences,
230                          config['latent_dim']))
231                     # Labels are now created based on index of
    occurrence
232                     y = np.arange(num_occurrences)[np.newaxis, np.
    newaxis]
233                     y_train = np.tile(
234                         y, (train_subjects, num_tasks, 1)).flatten()
235                     y_test = np.tile(
236                         y, (test_subjects, num_tasks, 1)).flatten()
237                     lr.fit(x_train, y_train)
238                     occurrence_init_results[0] = lr.score(x_test,
    y_test)
239                     np.save(log_path / 'occurrence_init_results.npy',
240                             occurrence_init_results)
241     # After finishing with a certain dataset, free the pre-loaded
242     # dataset from memory
243     del train_loader
244     del valid_loader
245     del test_loader
```

4.7.5 find_fixed_points.py

```
1  import torch
2  import importlib
3  import numpy as np
4  from torch import nn
5  from torch.optim.lr_scheduler import StepLR
6  from utils import (get_default_config, create_dataloaders,
7                     load_model_from_config)
8  from golub import FixedPoints
9
10
11 # Create default config and then loop over the datasets
12 seeds = [42, 1337, 9999, 1111]
13 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
14 config = get_default_config([''], 0])
15 config['dataset'] = 'HCPRelationalLong'
16 config['temporal_hidden_sizes'] = [128]
17 config['latent_dim'] = 8
18 config['gpu'] = 'V100'
19 config['batch_size'] = 1
20 config['epochs'] = 1000
21 datasets = ['HCPMotorLong', 'HCPWMLong', 'HCPRelationalLong']
22 for (d_ix, dataset_name) in enumerate(datasets):
23     config['dataset'] = dataset_name
24     # Loop over the seeds
25     dataset_module = importlib.import_module('dataset')
26     dataset = getattr(dataset_module, config['dataset'])
27     for (s_ix, seed) in enumerate(seeds):
28         # Create dataloaders
29         (train_loader, valid_loader, test_loader), (_, va_dataset, _)
    \
30             = create_dataloaders(dataset, config)
31         config['seed'] = seed
32         # Load the model
33         model = load_model_from_config(config)
34         # Load the data and the model
35         model = model.to(device)
```

```
        model.eval()
        # Obtain information from dataset
        time = va_dataset.window_size
        num_tasks = va_dataset.num_tasks
        # Initialize the time over which to integrate the NODE
        t_span = torch.linspace(0, 1, time)
        va_subjects = len(va_dataset)
        inits = []
        # Perform inference for 200 subjects
        num_subj = 200
        for (i, batch) in enumerate(train_loader):
            with torch.no_grad():
                # Depending on whether we use DALI dataloader
                # or pre-loaded dataset, we need to handle the
                # batch differently
                if isinstance(batch[0], torch.Tensor):
                    x = batch[0]
                    x = x.to(device, non_blocking=True).float()
                    mask = batch[1]
                    mask = mask.to(device, non_blocking=True).long()
                else:
                    x = batch[0]['fmri'].float()
                    mask = batch[0]['mask'].long()
                # Encode the initial conditions
                init = model.encode_init(x, mask, validation=True)
                inits.append(init)
            if i == (num_subj - 1):
                break
        # Stack all the initial conditions together
        inits = torch.stack(inits, dim=0)
        inits = inits.view(-1, config['latent_dim'])

        # Generate factors from all the initial conditions
        with torch.no_grad():
            _, factors = model.decoder(inits, t_span)

        # Find the fixed points
        torch.manual_seed(seed)
        # Get trajectories
        factors = factors.view(-1, config['latent_dim'])
        factors_detach = factors.detach().clone()
        # Add gaussian to the trajectories
        factors_noise = torch.cat(
            (factors_detach,
             factors_detach + torch.randn(factors_detach.size(),
                                          device=device) * 0.1), dim
=0)
        # Optimize over the latent points
        x = nn.Parameter(factors_noise)
        # We need to move the latent points throughout the space
        # such that the vector field derivative is almost zero
        optimizer = torch.optim.Adam([x], lr=0.01)
        # Initialize scheduler
        scheduler = StepLR(optimizer, step_size=2000, gamma=0.9)
        # Ensure all model parameters do not require gradients
        for p in model.parameters(): p.requires_grad = False

        q_prev = torch.full((x.size(0),), float("nan"), device=device)
        # Perform 10k iterations
        n_iters = 10000
        for i in range(n_iters):
            optimizer.zero_grad()
            # vf is the vector field of the NODE
            # q is its norm
            q = 0.5 * torch.sum(model.decoder.vf(None, x) ** 2, dim=1)
```

```
            loss = q.mean(0)
            loss.backward()
            optimizer.step()
            if i % 1000 == 0:
                print(loss, q.min())
            scheduler.step()
            dq = torch.abs(q - q_prev)
            q_prev = q
        # Create the fixed points
        qstar = q.cpu().detach().numpy()
        all_fps = FixedPoints(
            xstar=x.cpu().detach().numpy().squeeze(),
            x_init=factors_noise.cpu(),
            qstar=qstar,
            dq=dq.cpu().detach().numpy(),
            n_iters=np.full_like(qstar, n_iters),
            tol_unique=1E-1,
        )
        # Find unique fixed points based on tolerance
        unique_fps = all_fps.get_unique()
        # We only use fixed points that have a
        # derivative of less than 1E-10
        best_fps = unique_fps.qstar < 1E-10
        if best_fps.sum() > 0:
            best_fps = FixedPoints(
                xstar=unique_fps.xstar[best_fps],
                x_init=unique_fps.x_init[best_fps],
                qstar=unique_fps.qstar[best_fps],
                dq=unique_fps.dq[best_fps],
                n_iters=unique_fps.n_iters[best_fps],
                tol_unique=1E-1,
            )
            # Use this function to linearize around the fixed point
            func = lambda x: (1/time) * model.decoder.vf(None, x) + x

            # Find the Jacobian linearized around the fixed point
            all_J = []
            x = torch.tensor(best_fps.xstar, device=device)
            for i in range(best_fps.n):
                single_x = x[i, :]
                J = torch.autograd.functional.jacobian(func, single_x)
                all_J.append(J)

            # Recombine and decompose Jacobians for the whole batch
            dFdx = torch.stack(all_J).cpu().detach().numpy()
            best_fps.J_xstar = dFdx
            best_fps.decompose_jacobians()

            print(best_fps.eigval_J_xstar)
            print(best_fps.eigval_J_xstar.shape)
            # Save the Jacobian(s) for each dataset and seed
            # so we can use them to get eigenvalues in plot_figure4c.py
            np.save(f'fixed_point_experiments/fps/{dataset_name}_{seed}.npy',
                    best_fps.eigval_J_xstar)
```

4.7.6 golub.py

```
import pdb
import numpy as np
import pickle


class FixedPoints(object):
```

```
7       '''
8       A class for storing fixed points and associated data.
9       '''
10
11      ''' List of class attributes that represent data corresponding to
    fixed
12      points. All of these refer to Numpy arrays with axis 0 as the
    batch
13      dimension. Thus, each is concatenatable using np.concatenate(...,
    axis=0).
14      '''
15      _data_attrs = [
16              'xstar',
17              'x_init',
18              'inputs',
19              'F_xstar',
20              'qstar',
21              'dq',
22              'n_iters',
23              'J_xstar',
24              'eigval_J_xstar',
25              'eigvec_J_xstar',
26              'is_stable',
27              'cond_id']
28
29      ''' List of class attributes that apply to all fixed points
30      (i.e., these are not indexed per fixed point). '''
31      _nonspecific_attrs = [
32          'dtype',
33          'dtype_complex',
34          'tol_unique',
35          'verbose',
36          'do_alloc_nan']
37
38      def __init__(self,
39                   xstar=None, # Fixed-point specific data
40                   x_init=None,
41                   inputs=None,
42                   F_xstar=None,
43                   qstar=None,
44                   dq=None,
45                   n_iters=None,
46                   J_xstar=None,
47                   eigval_J_xstar=None,
48                   eigvec_J_xstar=None,
49                   is_stable=None,
50                   cond_id=None,
51                   n=None,
52                   n_states=None,
53                   n_inputs=None, # Non-specific data
54                   do_alloc_nan=False,
55                   tol_unique=1e-3,
56                   dtype=np.float32,
57                   dtype_complex=np.complex64,
58                   verbose=False):
59          '''
60          Initializes a FixedPoints object with all input arguments as
    class
61          properties.
62          Optional args:
63              xstar: [n x n_states] numpy array with row xstar[i, :]
64              specifying an the fixed point identified from x_init[i,
    :].
65              Default: None.
66              x_init: [n x n_states] numpy array with row x_init[i, :]
```

```
67              specifying the initial state from which xstar[i, :] was optimized.
68              Default: None.
69              inputs: [n x n_inputs] numpy array with row inputs[i, :]
70              specifying the input to the RNN during the optimization of
71              xstar[i, :]. Default: None.
72              F_xstar: [n x n_states] numpy array with F_xstar[i, :]
73              specifying RNN state after transitioning from the fixed point in
74              xstar[i, :]. If the optimization succeeded (e.g., to 'tol') and
75              identified a stable fixed point, the state should not move
76              substantially from the fixed point (i.e., xstar[i, :] should be
77              very close to F_xstar[i, :]). Default: None.
78              qstar: [n,] numpy array with qstar[i] containing the
79              optimized objective (1/2)(x-F(x))^T(x-F(x)), where
80              x = xstar[i, :]^T and F is the RNN transition function (with the
81              specified constant inputs). Default: None.
82              dq: [n,] numpy array with dq[i] containing the absolute
83              difference in the objective function after (i.e., qstar[i]) vs
84              before the final gradient descent step of the optimization of
85              xstar[i, :]. Default: None.
86              n_iters: [n,] numpy array with n_iters[i] as the number of
87              gradient descent iterations completed to yield xstar[i, :].
88              Default: None.
89              J_xstar: [n x n_states x n_states] numpy array with
90              J_xstar[i, :, :] containing the Jacobian of the RNN state
91              transition function at fixed point xstar[i, :]. Default: None,
92              which results in an appropriately sized numpy array of NaNs.
93              Default: None.
94              eigval_J_xstar: [n x n_states] numpy array with
95              eigval_J_xstar[i, :] containing the eigenvalues of
96              J_xstar[i, :, :].
97              eigvec_J_xstar: [n x n_states x n_states] numpy array with
98              eigvec_J_xstar[i, :, :] containing the eigenvectors of
99              J_xstar[i, :, :].
100             is_stable: [n,] numpy array with is_stable[i] indicating as bool
101             whether xstar[i] is a stable fixed point.
102             do_alloc_nan: Bool indicating whether to initialize all data
103             attributes (all optional args above) as NaN-filled numpy arrays.
104             Default: False.
105                 If True, n, n_states and n_inputs must be provided. These
106                 values are otherwise ignored:
107                 n: Positive int specifying the number of fixed points to
108                 allocate space for.
109                 n_states: Positive int specifying the dimensionality of the
110                 network state (a.k.a. the number of hidden units).
111                 n_inputs: Positive int specifying the dimensionality of the
112                 network inputs.
113             tol_unique: Positive scalar specifying the numerical precision
```

```
114             required to label two fixed points as being unique from
    one
115             another. Two fixed points are considered unique if the 2-
    norm of
116             the difference between their concatenated (xstar, inputs)
    is
117             greater than this tolerance. Default: 1e-3.
118             dtype: Data type for representing all of the object's data
    .
119             Default: numpy.float32.
120             cond_id: [n,] numpy array with cond_id[i] indicating the
    condition ID corresponding to inputs[i].
121             verbose: Bool indicating whether to print status updates.
122         Note:
123             xstar, x_init, inputs, F_xstar, and J_xstar are all numpy
    arrays,
124             regardless of whether that type is consistent with the
    state type
125             of the rnncell from which they originated (i.e., whether
    or not
126             the rnncell is an LSTM). This design decision reflects
    that a
127             Jacobian is most naturally expressed as a single matrix (
    as
128             opposed to a collection of matrices representing
    interactions
129             between LSTM hidden and cell states). If one requires
    state
130             representations as type LSTMStateCell, use
131             FixedPointFinder._convert_to_LSTMStateTuple.
132         Returns:
133             None.
134         '''
135 
136         # These apply to all fixed points
137         # (one value each, rather than one value per fixed point).
138         self.tol_unique = tol_unique
139         self.dtype = dtype
140         self.dtype_complex = dtype_complex
141         self.do_alloc_nan = do_alloc_nan
142         self.verbose = verbose
143 
144         if do_alloc_nan:
145 
146             if n is None:
147                 raise ValueError('n must be provided if '
148                                  'do_alloc_nan == True.')
149             if n_states is None:
150                 raise ValueError('n_states must be provided if '
151                                  'do_alloc_nan == True.')
152             if n_inputs is None:
153                 raise ValueError('n_inputs must be provided if '
154                                  'do_alloc_nan == True.')
155 
156             self.n = n
157             self.n_states = n_states
158             self.n_inputs = n_inputs
159 
160             self.xstar = self._alloc_nan((n, n_states))
161             self.x_init = self._alloc_nan((n, n_states))
162             self.inputs = self._alloc_nan((n, n_inputs))
163             self.F_xstar = self._alloc_nan((n, n_states))
164             self.qstar = self._alloc_nan((n))
165             self.dq = self._alloc_nan((n))
166             self.n_iters = self._alloc_nan((n))
```

```
            self.J_xstar = self._alloc_nan((n, n_states, n_states))

            self.eigval_J_xstar = self._alloc_nan(
                (n, n_states), dtype=dtype_complex)
            self.eigvec_J_xstar = self._alloc_nan(
                (n, n_states, n_states), dtype=dtype_complex)

            # not forcing dtype to bool yet, since np.bool(np.nan) is True,
            # which could be misinterpreted as a valid value.
            self.is_stable = self._alloc_nan((n))

            self.cond_id = self._alloc_nan((n))

        else:
            if xstar is not None:
                self.n, self.n_states = xstar.shape
            elif x_init is not None:
                self.n, self.n_states = x_init.shape
            elif F_xstar is not None:
                self.n, self.n_states = F_xstar.shape
            elif J_xstar is not None:
                self.n, self.n_states, _ = J_xstar.shape
            else:
                self.n = None
                self.n_states = None

            if inputs is not None:
                self.n_inputs = inputs.shape[1]
                if self.n is None:
                    self.n = inputs.shape[0]
            else:
                self.n_inputs = None

            self.xstar = xstar
            self.x_init = x_init
            self.inputs = inputs
            self.F_xstar = F_xstar
            self.qstar = qstar
            self.dq = dq
            self.n_iters = n_iters
            self.J_xstar = J_xstar
            self.eigval_J_xstar = eigval_J_xstar
            self.eigvec_J_xstar = eigvec_J_xstar
            self.is_stable = is_stable
            self.cond_id = cond_id

        self.assert_valid_shapes()

    def __setitem__(self, index, fps):
        '''Implements the assignment operator.
        All compatible data from fps are copied. This excludes tol_unique,
        dtype, n, n_states, and n_inputs, which retain their original values.
        Usage:
            fps_to_be_partially_overwritten[index] = fps
        '''

        assert isinstance(fps, FixedPoints),\
            ('fps must be a FixedPoints object but was %s.' % type(fps))

        if isinstance(index, int):
```

```
227             # Force the indexing that follows to preserve numpy array
    ndim
228             index = list(range(index, index+1))
229
230         manual_data_attrs = ['eigval_J_xstar', 'eigvec_J_xstar', '
    is_stable']
231
232         # This block added for testing 9/17/20 (replaces commented
    code below)
233         for attr_name in self._data_attrs:
234             if attr_name not in manual_data_attrs:
235                 attr = getattr(self, attr_name)
236                 if attr is not None:
237                     attr[index] = getattr(fps, attr_name)
238
239         ''' Previous version of block above:
240         if self.xstar is not None:
241             self.xstar[index] = fps.xstar
242         if self.x_init is not None:
243             self.x_init[index] = fps.x_init
244         if self.inputs is not None:
245             self.inputs[index] = fps.inputs
246         if self.F_xstar is not None:
247             self.F_xstar[index] = fps.F_xstar
248         if self.qstar is not None:
249             self.qstar[index] = fps.qstar
250         if self.dq is not None:
251             self.dq[index] = fps.dq
252         if self.J_xstar is not None:
253             self.J_xstar[index] = fps.J_xstar
254         '''
255
256         # This manual handling no longer seems necessary, but I'll
    save that
257         # change and testing for a rainy day.
258         if self.has_decomposed_jacobians:
259             self.eigval_J_xstar[index] = fps.eigval_J_xstar
260             self.eigvec_J_xstar[index] = fps.eigvec_J_xstar
261             self.is_stable[index] = fps.is_stable
262
263     def __getitem__(self, index):
264         '''Indexes into a subset of the fixed points and their
    associated data.
265         Usage:
266             fps_subset = fps[index]
267         Args:
268             index: a slice object for indexing into the FixedPoints
    data.
269         Returns:
270             A FixedPoints object containing a subset of the data from
    the
271             current FixedPoints object, as specified by index.
272         '''
273
274         if isinstance(index, int):
275             # Force the indexing that follows to preserve numpy array
    ndim
276             index = list(range(index, index+1))
277
278         kwargs = self._nonspecific_kwargs
279         manual_data_attrs = ['eigval_J_xstar', 'eigvec_J_xstar', '
    is_stable']
280
281         for attr_name in self._data_attrs:
282
```

```
283             attr_val = getattr(self, attr_name)
284
285             # This manual handling no longer seems necessary, but I'll
     save
286             # that change and testing for a rainy day.
287             if attr_name in manual_data_attrs:
288                 if self.has_decomposed_jacobians:
289                     indexed_val = self._safe_index(attr_val, index)
290                 else:
291                     indexed_val = None
292             else:
293                 indexed_val = self._safe_index(attr_val, index)
294
295             kwargs[attr_name] = indexed_val
296
297         indexed_fps = FixedPoints(**kwargs)
298
299         return indexed_fps
300
301     def __len__(self):
302         '''Returns the number of fixed points stored in the object.'''
303         return self.n
304
305     def __contains__(self, fp):
306         '''Checks whether a specified fixed point is contained in the
    object.
307         Args:
308             fp: A FixedPoints object containing exactly one fixed
    point.
309         Returns:
310             bool indicating whether any fixed point matches fp.
311         '''
312
313         idx = self.find(fp)
314
315         return idx.size > 0
316
317     def get_unique(self):
318         '''Identifies unique fixed points. Among duplicates identified
    ,
319         this keeps the one with smallest qstar.
320         Args:
321             None.
322         Returns:
323             A FixedPoints object containing only the unique fixed
    points and
324             their associated data. Uniqueness is determined down to
    tol_unique.
325         '''
326         assert (self.xstar is not None),\
327             ('Cannot find unique fixed points because self.xstar is
    None.')
328
329         if self.inputs is None:
330             data_nxd = self.xstar
331         else:
332             data_nxd = np.concatenate((self.xstar, self.inputs), axis
    =1)
333
334         idx_keep = []
335         idx_checked = np.zeros(self.n, dtype=np.bool_)
336         for idx in range(self.n):
337
338             if idx_checked[idx]:
```

```
339                 # If this FP matched others, we've already determined
    which
340                 # of those matching FPs to keep. Repeating would
    simply
341                 # identify the same FP to keep.
342                 continue
343 
344             # Don't compare against FPs we've already checked
345             idx_check = np.where(~idx_checked)[0]
346             fps_check = self[idx_check] # only check against these FPs
347             idx_idx_check = fps_check.find(self[idx]) # indexes into
    fps_check
348             idx_match = idx_check[idx_idx_check] # indexes into self
349 
350             if len(idx_match)==1:
351                 # Only matches with itself
352                 idx_keep.append(idx)
353             else:
354                 qstars_match = self.qstar[idx_match]
355                 idx_candidate = idx_match[np.argmin(qstars_match)]
356                 idx_keep.append(idx_candidate)
357                 idx_checked[idx_match] = True
358 
359         return self[idx_keep]
360 
361     def transform(self, U, offset=0.):
362         ''' Apply an affine transformation to the state-space
    representation.
363         This may be helpful for plotting fixed points in a given
    linear
364         subspace (e.g., PCA or an RNN readout space).
365         Args:
366             U: shape (n_states, k) numpy array projection matrix.
367             offset (optional): shape (k,) numpy translation vector.
    Default: 0.
368         Returns:
369             A FixedPoints object.
370         '''
371         kwargs = self.kwargs
372 
373         # These are all transformed. All others are not.
374         for attr_name in ['xstar', 'x_init', 'F_xstar']:
375             kwargs[attr_name] = np.matmul(getattr(self, attr_name), U)
     + offset
376 
377         if self.has_decomposed_jacobians:
378             kwargs['eigval_J_xstar'] = self.eigval_J_xstar
379             kwargs['eigvec_J_xstar'] = \
380                 np.matmul(U.T, self.eigvec_J_xstar) + offset
381 
382         transformed_fps = FixedPoints(**kwargs)
383 
384         return transformed_fps
385 
386     def find(self, fp):
387         '''Searches in the current FixedPoints object for matches to a
388         specified fixed point. Two fixed points are defined as
    matching
389         if the 2-norm of the difference between their concatenated (
    xstar,
390         inputs) is within tol_unique).
391         Args:
392             fp: A FixedPoints object containing exactly one fixed
    point.
393         Returns:
```

```
394             shape (n_matches,) numpy array specifying indices into the
     current
395             FixedPoints object where matches to fp were found.
396         '''
397 
398         # If not found or comparison is impossible (due to type or
    shape),
399         # follow convention of np.where and return an empty numpy
    array.
400         result = np.array([], dtype=int)
401 
402         if isinstance(fp, FixedPoints):
403             if fp.n_states == self.n_states and fp.n_inputs == self.
    n_inputs:
404 
405                 if self.inputs is None:
406                     self_data_nxd = self.xstar
407                     arg_data_nxd = fp.xstar
408                 else:
409                     self_data_nxd = np.concatenate(
410                         (self.xstar, self.inputs), axis=1)
411                     arg_data_nxd = np.concatenate(
412                         (fp.xstar, fp.inputs), axis=1)
413 
414                 norm_diffs_n = np.linalg.norm(
415                     self_data_nxd - arg_data_nxd, axis=1)
416 
417                 result = np.where(norm_diffs_n <= self.tol_unique)[0]
418 
419         return result
420 
421     def update(self, new_fps):
422         ''' Combines the entries from another FixedPoints object into
    this
423         object.
424         Args:
425             new_fps: a FixedPoints object containing the entries to be
426             incorporated into this FixedPoints object.
427         Returns:
428             None
429         Raises:
430             AssertionError if the non-fixed-point specific attributes
    of
431             new_fps do not match those of this FixedPoints object.
432             AssertionError if any data attributes are found in one but
     not both
433             FixedPoints objects (especially relevant for decomposed
    Jacobians).
434             AssertionError if the updated object has inconsistent data
     shapes.
435         '''
436 
437         self._assert_matching_nonspecific_attrs(self, new_fps)
438 
439         for attr_name in self._data_attrs:
440 
441             this_has = hasattr(self, attr_name)
442             that_has = hasattr(new_fps, attr_name)
443 
444             assert this_has == that_has,\
445                 ('One but not both FixedPoints objects have %s. '
446                  'FixedPoints.update does not currently support this '
447                  'configuration.' % attr_name)
448 
449             if this_has and that_has:
```

```
450                 cat_attr = np.concatenate(
451                     (getattr(self, attr_name),
452                     getattr(new_fps, attr_name)),
453                     axis=0)
454                 setattr(self, attr_name, cat_attr)
455 
456         self.n = self.n + new_fps.n
457         self.assert_valid_shapes()
458 
459     def decompose_jacobians(self, do_batch=True, str_prefix=''):
460         '''Adds the following fields to the FixedPoints object:
461         eigval_J_xstar: [n x n_states] numpy array with eigval_J_xstar
    [i, :]
462         containing the eigenvalues of J_xstar[i, :, :].
463         eigvec_J_xstar: [n x n_states x n_states] numpy array
    containing with
464         eigvec_J_xstar[i, :, :] containing the eigenvectors of
465         J_xstar[i, :, :].
466         Args:
467             do_batch (optional): bool indicating whether to perform a
    batch
468             decomposition. This is typically faster as long as
    sufficient
469             memory is available. If False, decompositions are
    performed
470             one-at-a-time, sequentially, which may be necessary if the
     batch
471             computation requires more memory than is available.
    Default: True.
472             str_prefix (optional): String to be pre-pended to print
    statements.
473         Returns:
474             None.
475         '''
476 
477         if self.has_decomposed_jacobians:
478             print('%sJacobians have already been decomposed, '
479                 'not repeating.' % str_prefix)
480             return
481 
482         n = self.n # number of FPs represented in this object
483         n_states = self.n_states # dimensionality of each state
484 
485         if do_batch:
486             # Batch eigendecomposition
487             print('%sDecomposing Jacobians in a single batch.' %
    str_prefix)
488 
489             # Check for NaNs in Jacobians
490             valid_J_idx = ~np.any(np.isnan(self.J_xstar), axis=(1,2))
491 
492             if np.all(valid_J_idx):
493                 # No NaNs, nothing to worry about.
494                 e_vals_unsrt, e_vecs_unsrt = np.linalg.eig(self.
    J_xstar)
495             else:
496                 # Set eigen-data to NaN if there are any NaNs in the
497                 # corresponding Jacobian.
498                 e_vals_unsrt = self._alloc_nan(
499                     (n, n_states), dtype=self.dtype_complex)
500                 e_vecs_unsrt = self._alloc_nan(
501                     (n, n_states, n_states), dtype=dtype_complex)
502 
503                 e_vals_unsrt[valid_J_idx], e_vecs_unsrt[valid_J_idx] =
     \
```

```
504                 np.linalg.eig(self.J_xstar[valid_J_idx])
505
506     else:
507         print('%sDecomposing Jacobians one-at-a-time.' %
str_prefix)
508         e_vals = []
509         e_vecs = []
510         for J in self.J_xstar:
511
512             if np.any(np.isnan(J)):
513                 e_vals_i = self._alloc_nan((n_states,))
514                 e_vecs_i = self._alloc_nan((n_states, n_states))
515             else:
516                 e_vals_i, e_vecs_i = np.linalg.eig(J)
517
518             e_vals.append(np.expand_dims(e_vals_i, axis=0))
519             e_vecs.append(np.expand_dims(e_vecs_i, axis=0))
520
521         e_vals_unsrt = np.concatenate(e_vals, axis=0)
522         e_vecs_unsrt = np.concatenate(e_vecs, axis=0)
523
524     print('%sSorting by Eigenvalue magnitude.' % str_prefix)
525     # For each FP, sort eigenvectors by eigenvalue magnitude
526     # (decreasing order).
527     mags_unsrt = np.abs(e_vals_unsrt) # shape (n,)
528     sort_idx = np.argsort(mags_unsrt)[:,::-1]
529
530     # Apply the sort
531     # There must be a faster way, but I'm too lazy to find it at
the moment
532     self.eigval_J_xstar = \
533         self._alloc_nan((n, n_states), dtype=self.dtype_complex)
534     self.eigvec_J_xstar = \
535         self._alloc_nan((n, n_states, n_states), dtype=self.
dtype_complex)
536     self.is_stable = np.zeros(n, dtype=np.bool_)
537
538     for k in range(n):
539         sort_idx_k = sort_idx[k]
540         e_vals_k = e_vals_unsrt[k][sort_idx_k]
541         e_vecs_k = e_vecs_unsrt[k][:, sort_idx_k]
542         self.eigval_J_xstar[k] = e_vals_k
543         self.eigvec_J_xstar[k] = e_vecs_k
544
545         # For stability, need only to look at the leading
eigenvalue
546         self.is_stable[k] = np.abs(e_vals_k[0]) < 1.0
547
548     self.assert_valid_shapes()
549
550 def save(self, save_path):
551     '''Saves all data contained in the FixedPoints object.
552     Args:
553         save_path: A string containing the path at which to save
554         (including directory, filename, and arbitrary extension).
555     Returns:
556         None.
557     '''
558     if self.verbose:
559         print('Saving FixedPoints object.')
560
561     self.assert_valid_shapes()
562
563     file = open(save_path,'wb')
564     file.write(pickle.dumps(self.__dict__))
```

```
565         file.close()
566 
567     def restore(self, restore_path):
568         '''Restores data from a previously saved FixedPoints object.
569         Args:
570             restore_path: A string containing the path at which to
    find a
571             previously saved FixedPoints object (including directory,
    filename,
572             and extension).
573         Returns:
574             None.
575         '''
576         if self.verbose:
577             print('Restoring FixedPoints object.')
578         file = open(restore_path,'rb')
579         restore_data = file.read()
580         file.close()
581         self.__dict__ = pickle.loads(restore_data)
582 
583         # Hacks to bridge between different versions of saved data
584         if not hasattr(self, 'do_alloc_nan'):
585             self.do_alloc_nan = False
586 
587         if not hasattr(self, 'eigval_J_xstar'):
588             n = self.n
589             n_states = self.n_states
590             dtype_complex = np.complex64
591             self.eigval_J_xstar = self._alloc_nan(
592                 (n, n_states), dtype=dtype_complex)
593             self.eigvec_J_xstar = self._alloc_nan(
594                 (n, n_states, n_states), dtype=dtype_complex)
595 
596             self.is_stable = self._alloc_nan((n))
597 
598             self.cond_id = self._alloc_nan((n))
599 
600         self.assert_valid_shapes()
601 
602     def print_summary(self):
603         '''Prints a summary of the fixed points.
604         Args:
605             None.
606         Returns:
607             None.
608         '''
609 
610         print('\nThe q function at the fixed points:')
611         print(self.qstar)
612 
613         print('\nChange in the q function from the final iteration '
614               'of each optimization:')
615         print(self.dq)
616 
617         print('\nNumber of iterations completed for each optimization:
    ')
618         print(self.n_iters)
619 
620         print('\nThe fixed points:')
621         print(self.xstar)
622 
623         print('\nThe fixed points after one state transition:')
624         print(self.F_xstar)
625         print('(these should be very close to the fixed points)')
626 
```

```
627         if self.J_xstar is not None:
628             print('\nThe Jacobians at the fixed points:')
629             print(self.J_xstar)
630 
631     def print_shapes(self):
632         ''' Prints the shapes of the data attributes of the fixed
    points.
633         Args:
634             None.
635         Returns:
636             None.
637         '''
638 
639         for attr_name in FixedPoints._data_attrs:
640             attr = getattr(self, attr_name)
641             print('%s: %s' % (attr_name, str(attr.shape)))
642 
643     def assert_valid_shapes(self):
644         ''' Checks that all data attributes reflect the same number of
     fixed
645         points.
646         Raises:
647             AssertionError if any non-None data attribute does not
    have
648             .shape[0] as self.n.
649         '''
650         n = self.n
651         for attr_name in FixedPoints._data_attrs:
652             data = getattr(self, attr_name)
653             if data is not None:
654                 assert data.shape[0] == self.n,\
655                     ('Detected %d fixed points, but %s.shape is %s '
656                     '(shape[0] should be %d' %
657                     (n, attr_name, str(data.shape), n))
658 
659     @staticmethod
660     def concatenate(fps_seq):
661         ''' Join a sequence of FixedPoints objects.
662         Args:
663             fps_seq: sequence of FixedPoints objects. All FixedPoints
    objects
664             must have the following attributes in common:
665                 n_states
666                 n_inputs
667                 has_decomposed_jacobians
668         Returns:
669             A FixedPoints objects containing the concatenated
    FixedPoints data.
670         '''
671 
672         assert len(fps_seq) > 0, 'Cannot concatenate empty list.'
673         FixedPoints._assert_matching_nonspecific_attrs(fps_seq)
674 
675         kwargs = {}
676 
677         for attr_name in FixedPoints._nonspecific_attrs:
678             kwargs[attr_name] = getattr(fps_seq[0], attr_name)
679 
680         for attr_name in FixedPoints._data_attrs:
681             if all((hasattr(fps, attr_name) for fps in fps_seq)):
682 
683                 cat_list = [getattr(fps, attr_name) for fps in fps_seq
    ]
684 
685                 if all([l is None for l in cat_list]):
```

```
686                     cat_attr = None
687                 elif any([l is None for l in cat_list]):
688                     # E.g., attempting to concat cond_id when it
    exists for
689                     # some fps but not for others. Better handling of
    this
690                     # would be nice. And yes, this would catch the all
     above,
691                     # but I'm keeping these cases separate to
    facilitate an
692                     # eventual refinement.
693                     cat_attr = None
694                 else:
695                     cat_attr = np.concatenate(cat_list, axis=0)
696
697                 kwargs[attr_name] = cat_attr
698
699         return FixedPoints(**kwargs)
700
701     @property
702     def is_single_fixed_point(self):
703         return self.n == 1
704
705     @property
706     def has_decomposed_jacobians(self):
707
708         if not hasattr(self, 'eigval_J_xstar'):
709             return False
710
711         return self.eigval_J_xstar is not None
712
713     @property
714     def kwargs(self):
715         ''' Returns dict of keyword arguments necessary for
    reinstantiating a
716         (shallow) copy of this FixedPoints object, i.e.,
717         fp_copy  = FixedPoints(**fp.kwargs)
718         '''
719
720         kwargs = self._nonspecific_kwargs
721
722         for attr_name in self._data_attrs:
723             kwargs[attr_name] = getattr(self, attr_name)
724
725         return kwargs
726
727     def _alloc_nan(self, shape, dtype=None):
728         '''Returns a nan-filled numpy array.
729         Args:
730             shape: int or tuple representing the shape of the desired
    numpy
731             array.
732         Returns:
733             numpy array with the desired shape, filled with NaNs.
734         '''
735         if dtype is None:
736             dtype = self.dtype
737
738         result = np.zeros(shape, dtype=dtype)
739         result.fill(np.nan)
740         return result
741
742     @staticmethod
743     def _assert_matching_nonspecific_attrs(fps_seq):
744
```

```
        for attr_name in FixedPoints._nonspecific_attrs:
            items = [getattr(fps, attr_name) for fps in fps_seq]
            for item in items:
                assert item == items[0],\
                    ('Cannot concatenate FixedPoints because of
    mismatched %s '
                     '(%s is not %s)' %
                     (attr_name, str(items[0]), str(item)))

    @staticmethod
    def _safe_index(x, idx):
        '''Safe method for indexing into a numpy array that might be
    None.
        Args:
            x: Either None or a numpy array.
            idx: Positive int or index-compatible argument for
    indexing into x.
        Returns:
            Self explanatory.
        '''
        if x is None:
            return None
        else:
            return x[idx]

    @property
    def _nonspecific_kwargs(self):
        # These are not specific to individual fixed points.
        # Thus, simple copy, no indexing required
        return {
            'dtype': self.dtype,
            'tol_unique': self.tol_unique
            }
```

4.7.7 main_baseline.py

```
import sys
import copy
import torch
import random
import importlib
import pandas as pd
import numpy as np
from pathlib import Path
from sklearn.decomposition import PCA
from utils import get_default_config_baseline, mask_input
from sklearn.linear_model import LogisticRegression

# Load the default config and set seeds for commputations
config = get_default_config_baseline(sys.argv)
random.seed(config['seed'])
np.random.seed(config['seed'])
# Load the name of the dataset
data_module = importlib.import_module('dataset')
dataset = getattr(data_module, config['dataset'])

# For the Supplement, we perform experiments across folds
# if there is no fold key in the config, then do not
# pass the fold keyword arg to the dataset
if 'fold' in config.keys():
    train_dataset = dataset('train', fold=config['fold'])
    valid_dataset = dataset('valid', fold=config['fold'])
    test_dataset = dataset('test', fold=config['fold'])
else:
    train_dataset = dataset('train')
```

```
30     valid_dataset = dataset('valid')
31     test_dataset = dataset('test')
32 
33 # Training and validation splits
34 train_df = train_dataset.df.copy()
35 valid_df = valid_dataset.df.copy()
36 test_df = test_dataset.df.copy()
37 
38 # Obtain the number of subjects for each split
39 train_subjects = len(train_dataset)
40 valid_subjects = len(valid_dataset)
41 test_subjects = len(test_dataset)
42 
43 # Obtain the number of tasks and occurrences for this dataset
44 num_tasks = train_dataset.num_tasks
45 num_occurrences = train_dataset.num_occurrences
46 # Load data
47 
48 # Load all the training data
49 x_tr = []
50 i = 0
51 for (_, row) in train_df.iterrows():
52     fmri = torch.from_numpy(np.load(
53         row['fmri']).astype(np.float32)).unsqueeze(0)
54     # The mask is used to obtain the timeseries corresponding
55     # to the specific sub-block
56     mask = torch.from_numpy(np.load(
57         row['targets'])).unsqueeze(0)
58     fmri = mask_input(fmri, mask).view(
59         -1, num_tasks, num_occurrences, config['input_size'])
60     x_tr.append(fmri)
61 
62 # We get the following shape:
63 # (train_subjects, window_size, num_tasks, num_occurrences, input_size
    )
64 x_tr = torch.stack(x_tr, dim=0).numpy()
65 
66 # Load all the validation data
67 x_va = []
68 i = 0
69 for (_, row) in valid_df.iterrows():
70     fmri = torch.from_numpy(np.load(
71         row['fmri']).astype(np.float32)).unsqueeze(0)
72     # The mask is used to obtain the timeseries corresponding
73     # to the specific sub-block
74     mask = torch.from_numpy(np.load(
75         row['targets'])).unsqueeze(0)
76     fmri = mask_input(fmri, mask).view(
77         -1, num_tasks, num_occurrences, config['input_size'])
78     x_va.append(fmri)
79 
80 # We get the following shape:
81 # (valid_subjects, window_size, num_tasks, num_occurrences, input_size
    )
82 x_va = torch.stack(x_va, dim=0).numpy()
83 
84 # For the baseline, we concatenate the training
85 # and validation data together
86 x_tr = np.concatenate((x_tr, x_va), axis=0)
87 train_subjects = train_subjects + valid_subjects
88 train_df = pd.concat((train_df, valid_df), axis=0)
89 
90 # Load all the test data
91 x_te = []
92 i = 0
```

```
93 for (_, row) in test_df.iterrows():
94     fmri = torch.from_numpy(np.load(
95         row['fmri']).astype(np.float32)).unsqueeze(0)
96     # The mask is used to obtain the timeseries corresponding
97     # to the specific sub-block
98     mask = torch.from_numpy(np.load(
99         row['targets'])).unsqueeze(0)
100     fmri = mask_input(fmri, mask).view(
101         -1, num_tasks, num_occurrences, config['input_size'])
102     x_te.append(fmri)
103
104 # We get the following shape:
105 # (test_subjects, window_size, num_tasks, num_occurrences, input_size)
106 x_te = torch.stack(x_te, dim=0).numpy()
107
108 # Get the window size for the sub-blocks
109 window_size = x_te.shape[1]
110
111 # Create a directory to save the results to
112 name_log = f'{config["dataset"]}_{config["transform"]}_{config["
    latent_dim"]}'
113 log_dir = Path('baseline_logs') / Path(name_log)
114 log_dir.mkdir(parents=True, exist_ok=True)
115 if 'fold' in config.keys():
116     log_dir = log_dir / f'fold_{config["fold"]}'
117     log_dir.mkdir(parents=True, exist_ok=True)
118
119 # Reshape before performing a transformation so
120 # PCA is performed over the input size
121 x_tr = np.reshape(
122     x_tr,
123     (train_subjects * window_size * num_tasks * num_occurrences, -1))
124 x_te = np.reshape(
125     x_te,
126     (test_subjects * window_size * num_tasks * num_occurrences, -1))
127
128 # Perform PCA and save the components + mean
129 if config['transform'] == 'PCA':
130     pca = PCA(n_components=config['latent_dim'],
131               whiten=config['whiten'],
132               svd_solver='arpack')
133     x_tr_transform = pca.fit_transform(x_tr)
134     np.save(log_dir / 'components.npy', pca.components_)
135     np.save(log_dir / 'mean.npy', pca.mean_)
136     x_te_transform = pca.transform(x_te)
137 else:
138     x_tr_transform = x_tr
139     x_te_transform = x_te
140
141 # Reshape back to multi-dimensional array
142 x_tr_transform = np.reshape(
143     x_tr_transform,
144     (train_subjects, window_size, num_tasks, num_occurrences, -1))
145 x_te_transform = np.reshape(
146     x_te_transform,
147     (test_subjects, window_size, num_tasks, num_occurrences, -1))
148
149 results = np.zeros((2,))
150 lr = LogisticRegression(max_iter=10000, n_jobs=-1)
151 factor_ls = []
152 for t in range(window_size):
153     # Create a training and test set for each timestep
154     x_train = np.reshape(
155         x_tr_transform[:, t],
156         (train_subjects * num_tasks * num_occurrences, -1))
```

```
157         x_test = np.reshape(
158             x_te_transform[:, t],
159             (test_subjects * num_tasks * num_occurrences, -1))
160         # We can create the labels based on the index within the multi-dim
             array
161         y = np.arange(num_tasks)[np.newaxis, :, np.newaxis]
162         y_train = np.tile(y, (train_subjects, 1, num_occurrences)).flatten
            ()
163         y_test = np.tile(y, (test_subjects, 1, num_occurrences)).flatten()
164         # Ensure a new model is fit for each timestep
165         modelIter = copy.deepcopy(lr)
166         modelIter.fit(x_train, y_train)
167         factor_ls.append(modelIter.score(x_test, y_test))
168
169 results[0] = np.mean(factor_ls)
170 results_df = pd.Series(results, index=['factor_avg_acc', 'factor_acc'
        ])
171 results_df.iloc[1] = str(factor_ls)
172
173 # For the WM dataset we also perform classification over the
        occurrences
174 if (config['dataset'] == 'HCPWM'):
175     factor_ls_occ = []
176     for t in range(window_size):
177         # Similar to the task classification, except labels are
        created
178         # differently
179         lr = LogisticRegression(max_iter=10000, n_jobs=-1)
180         x_train = np.reshape(
181             x_tr_transform[:, t],
182             (train_subjects * num_tasks * num_occurrences, -1))
183         x_test = np.reshape(
184             x_te_transform[:, t],
185             (test_subjects * num_tasks * num_occurrences, -1))
186         # Labels are now created based on index of occurrence
187         y = np.arange(num_occurrences)[np.newaxis, np.newaxis]
188         y_train = np.tile(y, (train_subjects, num_tasks, 1)).flatten()
189         y_test = np.tile(y, (test_subjects, num_tasks, 1)).flatten()
190         lr.fit(x_train, y_train)
191         factor_ls_occ.append(lr.score(x_test, y_test))
192         # Delete the classifier after each timestep
193         del lr
194     occ_results = pd.Series(
195         np.zeros((2, )),
196         index=['factor_occ_avg_acc', 'factor_occ_acc'])
197     occ_results.iloc[0] = np.mean(factor_ls_occ)
198     occ_results.iloc[1] = str(factor_ls_occ)
199     # Append dataframe to previous dataframe
200     results_df = results_df.append(occ_results, ignore_index=False)
201
202 # Save results
203 results_df.to_csv(log_dir / 'results.csv')
```

4.7.8 main.py

```
1 import sys
2 import torch
3 import random
4 import importlib
5 import numpy as np
6 from train import Trainer
7 from utils import get_default_config, get_log_string
8
9
10 if __name__ == '__main__':
```

```
    # Load the config based on command line arguments
    config = get_default_config(sys.argv)
    # Ensure reproducibility
    torch.manual_seed(config['seed'])
    random.seed(config['seed'])
    np.random.seed(config['seed'])
    # Set device, and load criterion, optimizer, model, and dataset
    device = torch.device('cuda' if torch.cuda.is_available() else '
cpu')
    criterion_module = importlib.import_module('criterion')
    criterion = getattr(criterion_module, config['criterion'])(config[
'beta'])
    optimizer = getattr(torch.optim, config['optimizer'])
    optimizer_params = {'lr': config['learning_rate'],
                        'weight_decay': config['weight_decay']}
    model_module = importlib.import_module('model')
    data_module = importlib.import_module('dataset')
    # Initialize the trainer
    trainer = Trainer(
        model=getattr(model_module, config['model']),
        optimizer=optimizer,
        optimizer_params=optimizer_params,
        criterion=criterion,
        device=device,
        dataset=getattr(data_module, config['dataset']),
        log_dir=get_log_string(config))
    # Train the model
    trainer.train(
        config=config,
        epochs=config['epochs'],
        batch_size=config['batch_size'])
```

4.7.9 model.py

```
import torch
import importlib
from torch import nn
from torch import distributions as D
from torch.nn import functional as F
from torchdyn.core import NeuralODE
from utils import mask_input


class RNN(nn.Module):
    def __init__(self, encoder_type, decoder_type, encoder_args,
decoder_args,
                 temporal_hidden_sizes):
        super().__init__()
        # Encoder args look like:
        # (input_size, hidden_sizes, output_size, activation,
        # normalization, dropout)
        self.input_size = encoder_args[0]
        self.latent_dim = encoder_args[2]
        # For the RNN, we don't need multiple layers
        # to define the vector field, so we just use
        # the first hidden size
        self.hidden_size = temporal_hidden_sizes[0]
        self.dropout_val = encoder_args[-1]
        self.dropout = nn.Dropout(self.dropout_val)
        modules = importlib.import_module('modules')
        # Initialize the spatial encoder and decoder
        self.spatial_encoder = getattr(modules, encoder_type)(
encoder_args)
        self.spatial_decoder = getattr(modules, decoder_type)(
decoder_args)
```

```
29         # Initialize the temporal encoder
30         self.encoder = nn.GRU(
31             input_size=self.latent_dim,
32             hidden_size=self.hidden_size, bidirectional=True)
33         # Learn the first hidden state for the encoder
34         self.encoder_h = nn.Parameter(
35             torch.randn(2, 1, self.hidden_size) * 0.1)
36         # The mu-layer for the initial hidden state
37         self.encoder_mu = nn.Linear(2 * self.hidden_size, self.
    hidden_size)
38         nn.init.xavier_normal_(self.encoder_mu.weight, 1.)
39         nn.init.constant_(self.encoder_mu.bias, 0.)
40         self.encoder_mu = nn.utils.weight_norm(self.encoder_mu)
41         # The standard deviation layer for the initial hidden state
42         self.encoder_lv = nn.Linear(2 * self.hidden_size, self.
    hidden_size)
43         nn.init.xavier_normal_(self.encoder_lv.weight, 1.)
44         nn.init.constant_(self.encoder_lv.bias, 0.)
45         self.encoder_lv = nn.utils.weight_norm(self.encoder_lv)
46         self.decoder = nn.GRU(input_size=1, hidden_size=self.
    hidden_size)
47         # The layer that maps from the hidden state to the latent
    dimension
48         self.factor_layer = nn.Linear(self.hidden_size, self.
    latent_dim)
49         nn.init.xavier_normal_(self.factor_layer.weight, 1.)
50         nn.init.constant_(self.factor_layer.bias, 0.)
51         # Add weight normalization to the factor layer
52         self.factor_layer = nn.utils.weight_norm(self.factor_layer)
53 
54     def forward(self, x, mask, validation=False):
55         x = mask_input(x, mask)
56         # x shape:
57         # (timesteps, num_tasks * num_occurences, voxels)
58         # These are the embeddings
59         z = self.spatial_encoder(x)
60         # z shape:
61         # (timesteps, num_tasks * num_occurences, latent_dim)
62         _, h_enc = self.encoder(z, self.encoder_h.repeat(1, x.size(1),
     1))
63         # This is the final hidden state
64         h_enc = self.dropout(h_enc)
65         h_enc = torch.reshape(
66             h_enc.permute(1, 0, 2),
67             (x.size(1), self.hidden_size * 2))
68         # Use the final temporal hidden state of the temporal encoder
69         # to infer a distribution for the initial hidden state
70         mu = self.encoder_mu(h_enc)
71         sd = torch.exp(0.5 * self.encoder_lv(h_enc.squeeze(0)))
72         dist = D.Normal(mu, sd)
73         if validation:
74             z = dist.mean
75         else:
76             z = dist.rsample()
77         # We do not use any input for the temporal decoder (all zeros)
78         in_dec = torch.zeros((x.size(0), x.size(1), 1), device=x.
    device)
79         # Unroll the temporal decoder based ONLY on the initial hidden
80         # state
81         h_dec, _ = self.decoder(in_dec, z.unsqueeze(0))
82         # Normalize the factor layer weights
83         with torch.no_grad():
84             self.factor_layer.weight.data \
85                 = F.normalize(self.factor_layer.weight.data, dim=1)
86         # Obtain the latent factors (z)
```

```
87          factors = self.factor_layer(h_dec)
88          # Map the latent factors to the brain
89          x_hat = self.spatial_decoder(factors)
90          return {
91              'dist': dist,
92              'x_hat': x_hat,
93              'x': x,
94              'classes': None,
95              'factors': factors,
96              'h_0': z
97          }
98 
99      def encode_init(self, x, mask, validation=True):
100         x = mask_input(x, mask)
101         # x shape:
102         # (timesteps, num_tasks * num_occurences, voxels)
103         # These are the embeddings
104         z = self.spatial_encoder(x)
105         # (timesteps, num_tasks * num_occurences, latent_dim)
106         _, h_enc = self.encoder(z, self.encoder_h.repeat(1, x.size(1),
     1))
107         # Use the final hidden state of the temporal encoder to infer
    the
108         # distribution over the initial hidden state
109         h_enc = self.dropout(h_enc)
110         h_enc = torch.reshape(
111             h_enc.permute(1, 0, 2), (x.size(1), self.hidden_size * 2))
112         h_enc = self.dropout(h_enc)
113         mu = self.encoder_mu(h_enc)
114         sd = torch.exp(0.5 * self.encoder_lv(h_enc.squeeze(0)))
115         dist = D.Normal(mu, sd)
116         if validation:
117             z = dist.mean
118         else:
119             z = dist.rsample()
120         return z
121 
122 
123 class NODE(nn.Module):
124     def __init__(self, encoder_type, decoder_type, encoder_args,
    decoder_args,
125                  temporal_hidden_sizes):
126         super().__init__()
127         # Encoder args look like:
128         # (input_size, hidden_sizes, output_size, activation,
129         # normalization, dropout)
130         self.input_size = encoder_args[0]
131         self.latent_dim = encoder_args[2]
132         self.dropout_val = encoder_args[-1]
133         self.hidden_size = temporal_hidden_sizes[0]
134         modules = importlib.import_module('modules')
135         # Initialize the spatial encoder and decoder
136         self.spatial_encoder = getattr(modules, encoder_type)(
    encoder_args)
137         self.spatial_decoder = getattr(modules, decoder_type)(
    decoder_args)
138         # Initialize the temporal encoder
139         self.encoder = nn.GRU(input_size=self.latent_dim,
140                               hidden_size=self.hidden_size,
    bidirectional=True)
141         # Learn the first hidden state for the encoder
142         self.encoder_h = nn.Parameter(torch.randn(2, 1, self.
    hidden_size))
143         # The mu-layer for the initial conditions (z0)
```

```
144         self.encoder_mu = nn.Linear(2 * self.hidden_size, self.
    latent_dim)
145         nn.init.xavier_normal_(self.encoder_mu.weight, 1.)
146         nn.init.constant_(self.encoder_mu.bias, 0.)
147         self.encoder_mu = nn.utils.weight_norm(self.encoder_mu)
148         # The standard deviation layer for the initial hidden state
149         self.encoder_lv = nn.Linear(2 * self.hidden_size, self.
    latent_dim)
150         nn.init.xavier_normal_(self.encoder_lv.weight, 1.)
151         nn.init.constant_(self.encoder_lv.bias, 0.)
152         self.encoder_lv = nn.utils.weight_norm(self.encoder_lv)
153         mlp_hidden_dims = temporal_hidden_sizes
154         # This loop is to create the vector field MLP for the NODE
155         layers = []
156         layer_in_dim = self.latent_dim
157         for layer_out_dim in mlp_hidden_dims:
158             layers.extend([nn.Linear(layer_in_dim, layer_out_dim), nn.
    Tanh()])
159             nn.init.xavier_normal_(layers[-2].weight, 5/3)
160             nn.init.constant_(layers[-2].bias, 0.)
161             layer_in_dim = layer_out_dim
162         layers.extend([nn.Linear(layer_in_dim, self.latent_dim)])
163         nn.init.xavier_normal_(layers[-1].weight, 1.0)
164         nn.init.constant_(layers[-1].bias, 0.)
165         vector_field = nn.Sequential(*layers)
166         # Initialize the NODE
167         self.decoder = NeuralODE(vector_field, solver='DormandPrince45
    ')
168         self.dropout = nn.Dropout(self.dropout_val)
169 
170     def forward(self, x, mask, validation=False):
171         x = mask_input(x, mask)
172         timesteps = x.size(0)
173         # x shape:
174         # (timesteps, num_tasks * num_occurences, voxels)
175         # These are the embeddings
176         z = self.spatial_encoder(x)
177         # z shape:
178         # (timesteps, num_tasks * num_occurences, latent_dim)
179         _, h_enc = self.encoder(z, self.encoder_h.repeat(1, x.size(1),
     1))
180         h_enc = torch.reshape(
181             h_enc.permute(1, 0, 2), (x.size(1), self.hidden_size * 2))
182         # This is the final hidden state
183         h_enc = self.dropout(h_enc)
184         # Use the final temporal hidden state of the temporal encoder
185         # to infer a distribution for the initial hidden state
186         mu = self.encoder_mu(h_enc)
187         sd = torch.exp(0.5 * self.encoder_lv(h_enc.squeeze(0)))
188         # Distribution of the initial condition
189         dist = D.Normal(mu, sd)
190         # (num_tasks, hidden_size)
191         if validation:
192             z_0 = dist.mean
193         else:
194             z_0 = dist.rsample()
195         t_span = torch.linspace(0, 1, timesteps)
196         # Unroll the NODE based ONLY on the initial conditions
197         _, factors = self.decoder(z_0, t_span)
198         # Map the latent factors to the brain
199         x_hat = self.spatial_decoder(factors)
200         return {
201             'dist': dist,
202             'x_hat': x_hat,
203             'x': x,
```

```
204             'classes': None,
205             'factors': factors,
206             'h_0': z_0
207         }
208
209     def encode_init(self, x, mask, validation=True):
210         x = mask_input(x, mask)
211         # x shape:
212         # (timesteps, num_tasks * num_occurences, voxels)
213         # These are the embeddings
214         z = self.spatial_encoder(x)
215         # (timesteps, num_tasks * num_occurences, latent_dim)
216         _, h_enc = self.encoder(z, self.encoder_h.repeat(1, x.size(1),
     1))
217         h_enc = torch.reshape(
218             h_enc.permute(1, 0, 2), (x.size(1), self.hidden_size * 2))
219         # Use the final hidden state of the temporal encoder to infer
    the
220         # distribution over the initial hidden state
221         h_enc = self.dropout(h_enc)
222         mu = self.encoder_mu(h_enc)
223         sd = torch.exp(0.5 * self.encoder_lv(h_enc.squeeze(0)))
224         dist = D.Normal(mu, sd)
225         # (num_tasks, hidden_size)
226         if validation:
227             z_0 = dist.mean
228         else:
229             z_0 = dist.rsample()
230         return z_0
231
232     def reconstruct(self, factors):
233         x_hat = self.spatial_decoder(factors)
234         return x_hat
235
236
237 class VAE(nn.Module):
238     def __init__(self, encoder_type, decoder_type, encoder_args,
    decoder_args,
239                  temporal_hidden_sizes):
240         super().__init__()
241         # Encoder args look like:
242         # (input_size, hidden_sizes, output_size, activation,
243         # normalization, dropout)
244         self.input_size = encoder_args[0]
245         self.latent_dim = encoder_args[2]
246         # Need mean and logvar for VAE (so latent_dim * 2)
247         encoder_args = (*encoder_args[:2], self.latent_dim * 2,
248                         *encoder_args[3:])
249         self.dropout_val = encoder_args[-1]
250         modules = importlib.import_module('modules')
251         self.spatial_encoder = getattr(modules, encoder_type)(
    encoder_args)
252         self.spatial_decoder = getattr(modules, decoder_type)(
    decoder_args)
253
254     def forward(self, x, mask, validation=False):
255         x = mask_input(x, mask)
256         # x shape:
257         # (timesteps, num_tasks * num_occurences, voxels)
258         # These are the embeddings
259         z = self.spatial_encoder(x)
260         # Split up the embeddings into mu and sd
261         mu, logvar = torch.split(z, self.latent_dim, dim=-1)
262         sd = torch.exp(0.5 * logvar).clamp(1E-9, 5)
263         dist = D.Normal(mu, sd)
```

```
264         # Disttribution in the latent space
265         # (num_tasks, hidden_size)
266         if validation:
267             z = dist.mean
268         else:
269             z = dist.rsample()
270         # Map the latent factors to the brain
271         x_hat = self.spatial_decoder(z)
272         return {
273             'dist': dist,
274             'x_hat': x_hat,
275             'x': x,
276             'classes': None,
277             'factors': z,
278             'h_0': None
279         }
```

4.7.10 modules.py

```
1 import torch
2 from torch import nn
3 from math import sqrt
4 from typing import List
5
6
7 # Create a base class for the modules
8 class BaseClass(nn.Module):
9     def __init__(self, input_size: int, hidden_sizes: List[int],
10                  output_size: int, activation: str, normalization: str
    ,
11                  dropout: float):
12         super().__init__()
13         self.num_layers = len(hidden_sizes)
14         self.activation = activation
15         self.normalization = normalization
16         self.input_size = input_size
17         self.output_size = output_size
18         self.hidden_sizes = hidden_sizes
19         self.dropout = dropout
20
21     def forward(self, x: torch.Tensor):
22         raise NotImplementedError
23
24
25 # This is for linear spatial encoder and decoders
26 # We noticed the weight_norm was incredibly important
27 # to ensure stable training
28 class Linear(BaseClass):
29     def __init__(self, args):
30         super(Linear, self).__init__(*args)
31         self.lin = nn.Linear(self.input_size, self.output_size)
32         nn.init.xavier_normal_(self.lin.weight, 1.)
33         nn.init.constant_(self.lin.bias, 0.0)
34         self.lin = nn.utils.weight_norm(self.lin)
35
36     def forward(self, x):
37         return self.lin(x)
38
39
40 # This is a simple MLP Block with
41 # a residual connection (input = output) for these
42 # blocks
43 class MLP(BaseClass):
44     def __init__(self, args):
45         super(MLP, self).__init__(*args)
```

```python
        self.layers = nn.Sequential(
            nn.Linear(self.input_size, self.output_size),
            nn.GELU())
        nn.init.xavier_normal_(self.layers[0].weight, sqrt(2))
        nn.init.constant_(self.layers[0].bias, 0.0)
        self.layers[0] = nn.utils.weight_norm(self.layers[0])
        self.dropout_l = nn.Dropout(self.dropout)

    def forward(self, x):
        x, x_res = x
        return (self.dropout_l(self.layers(x) + x), x_res)


# Stack multiple of the residual MLP blocks (above)
# together
class MLPs(BaseClass):
    def __init__(self, args):
        super(MLPs, self).__init__(*args)
        self.mlps = []
        input_size = self.input_size
        # First layer should map to the hidden size, all other
        # hidden sizes are assumed to be the same
        self.lin = nn.Linear(self.input_size, self.hidden_sizes[0])
        nn.init.xavier_normal_(self.lin.weight, sqrt(2))
        nn.init.constant_(self.lin.bias, 0.0)
        self.lin = nn.utils.weight_norm(self.lin)
        input_size = self.hidden_sizes[0]
        for (_, hidden_size) in enumerate(self.hidden_sizes):
            self.mlps.append(MLP((input_size, [], hidden_size, *args
    [3:])))
            input_size = hidden_size
        self.final_size = hidden_size
        self.mlps = nn.Sequential(*self.mlps)

    def forward(self, x):
        pass


# Decoder that uses the MLP blocks
class MLPDecoder(MLPs):
    def __init__(self, args):
        super(MLPDecoder, self).__init__(args)
        # Final layer maps from hidden size to the number of voxels
        self.out_layer = nn.Linear(self.final_size, self.output_size)
        nn.init.xavier_normal_(self.out_layer.weight, 1.0)
        nn.init.constant_(self.out_layer.bias, 0.)
        self.out_layer = nn.utils.weight_norm(self.out_layer)
        self.dropout_l = nn.Dropout(self.dropout)

    def forward(self, x):
        x = self.dropout_l(self.lin(x))
        x_res = x
        x, _ = self.mlps((x, x_res))
        x = self.out_layer(x)
        return x


# Encoder that uses the MLP blocks
class MLPEncoder(MLPs):
    def __init__(self, args):
        super(MLPEncoder, self).__init__(args)
        # Layer that maps from the hidden size to the latent dimension
        self.out_layer = nn.Linear(self.final_size, self.output_size)
        nn.init.xavier_normal_(self.out_layer.weight, 1.0)
        nn.init.constant_(self.out_layer.bias, 0.)
```

```
        self.out_layer = nn.utils.weight_norm(self.out_layer)
        self.dropout_l = nn.Dropout(self.dropout)

    def forward(self, x):
        x = self.dropout_l(self.lin(x))
        x_res = x
        x, _ = self.mlps((x, x_res))
        x = self.out_layer(x)
        return x
```

4.7.11 plot_figure1.py

```
import torch
import importlib
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from utils import (get_default_config, load_model_from_config,
                   create_dataloaders, mask_input)
matplotlib.use('Agg')

# Set the model config
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
config = get_default_config([''])
config['dataset'] = 'HCPLeft'
config['temporal_hidden_sizes'] = [128]
config['latent_dim'] = 8
config['gpu'] = 'V100'
config['learning_rate'] = 0.001
config['model'] = 'NODE'
config['seed'] = 42
model = load_model_from_config(config)
# Add extra subsamples to the data (normally this is window_size)
time = 284 * 1
t_span = torch.linspace(0, 1, time)
# Load the data and the model
model = model.to(device)
dataset_module = importlib.import_module('dataset')
dataset = getattr(dataset_module, config['dataset'])
config['batch_size'] = 1
# Create dataloaders
(train_loader, valid_loader, test_loader), (_, va_dataset, _) \
    = create_dataloaders(dataset, config)
factors = []
inits = []
xs = []
num_subjects = 100
# Embed test set
for (i, batch) in enumerate(test_loader):
    with torch.no_grad():
        # Depending on whether we use DALI dataloader
        # or pre-loaded dataset, we need to handle the
        # batch differently
        if isinstance(batch[0], torch.Tensor):
            x = batch[0]
            x = x.to(device, non_blocking=True).float()
            mask = batch[1]
            mask = mask.to(device, non_blocking=True).long()
        else:
            x = batch[0]['fmri'].float()
            mask = batch[0]['mask'].long()
        # Obtain the factors and the initial conditions
        model_output = model(x, mask, validation=True)
        factors.append(model_output['factors'])
```

```
54          inits.append(model_output['h_0'])
55          xs.append(mask_input(x, mask).cpu())
56      if i == (num_subjects - 1):
57          break
58 
59 # Set the colors for the plot (the two different tasks)
60 colors = ['#A65B8C', '#B8BF80']
61 # Reshape factors and initial conditions
62 factors = torch.stack(factors, dim=0)
63 # The number of tasks and occurrences are both 2 in this case
64 # (num_subjects * num_timesteps * num_tasks * num_occurrences,
    latent_dim)
65 num_timesteps = factors.size(1)
66 # Map data to 3 dimensions instead of 8
67 factors = factors.view(
68     num_subjects * num_timesteps * 2 * 2, config['latent_dim']).cpu()
69 pca = PCA(n_components=3, svd_solver='full')
70 factors_pca = pca.fit_transform(factors)
71 factors_pca = np.reshape(factors_pca, (num_subjects, num_timesteps, 2,
     2, 3))
72 
73 # Plot factors
74 fig, ax = plt.subplots(1, 1, figsize=(10, 10),
75                        subplot_kw=dict(projection='3d'))
76 task_1_factors = factors_pca[:, :, 0]
77 task_2_factors = factors_pca[:, :, 1]
78 for i in range(num_subjects):
79     # Task 1, first occurence
80     ax.plot(
81         task_1_factors[i, :, 0, 0],
82         task_1_factors[i, :, 0, 1],
83         task_1_factors[i, :, 0, 1], color=colors[0], alpha=0.5,
    linewidth=3)
84     # Task 1, second occurence
85     ax.plot(
86         task_1_factors[i, :, 1, 0],
87         task_1_factors[i, :, 1, 1],
88         task_1_factors[i, :, 1, 2], color=colors[0], alpha=0.5,
    linewidth=3)
89     # Task 2, first occurrence
90     ax.plot(
91         task_2_factors[i, :, 0, 0],
92         task_2_factors[i, :, 0, 1],
93         task_2_factors[i, :, 0, 2], color=colors[1], alpha=0.5,
    linewidth=3)
94     # Task 2, second occurrence
95     ax.plot(
96         task_2_factors[i, :, 1, 0],
97         task_2_factors[i, :, 1, 1],
98         task_2_factors[i, :, 1, 2], color=colors[1], alpha=0.5,
    linewidth=3)
99 # Turn off x and y ticks
100 ax.set_xticks([])
101 ax.set_yticks([])
102 ax.set_zticks([])
103 plt.tight_layout()
104 # Save the figure
105 fig.savefig('paper_figures/figure1/factors.png',
106             bbox_inches=0, transparent=True)
107 plt.clf()
108 plt.close(fig)
109 
110 # Plot the initial conditions
111 inits = torch.stack(inits, dim=0)
112 # Num tasks and occurrences in this case are both 2
```

```
113 inits = inits.view(num_subjects * 2 * 2, config['latent_dim']).cpu()
114 # Map initial conditions to lower-dim space (3)
115 pca = PCA(n_components=3, svd_solver='full')
116 inits_pca = pca.fit_transform(inits)
117 inits_pca = np.reshape(inits_pca, (num_subjects, 2, 2, 3))
118
119 fig, ax = plt.subplots(1, 1, figsize=(10, 10),
120                        subplot_kw=dict(projection='3d'))
121 task_1_inits = inits_pca[:, 0]
122 task_2_inits = inits_pca[:, 1]
123 for i in range(num_subjects):
124     # Task 1, first occurrence
125     ax.scatter(
126         task_1_inits[i, 0, 0],
127         task_1_inits[i, 0, 1],
128         task_1_inits[i, 0, 1], color=colors[0], alpha=0.75, s=50)
129     # Task 1, second occurrence
130     ax.scatter(
131         task_1_inits[i, 1, 0],
132         task_1_inits[i, 1, 1],
133         task_1_inits[i, 1, 2], color=colors[0], alpha=0.75, s=50)
134     # Task 2, first occurrence
135     ax.scatter(
136         task_2_inits[i, 0, 0],
137         task_2_inits[i, 0, 1],
138         task_2_inits[i, 0, 2], color=colors[1], alpha=0.75, s=50)
139     # Task 2, second occurrence
140     ax.scatter(
141         task_2_inits[i, 1, 0],
142         task_2_inits[i, 1, 1],
143         task_2_inits[i, 1, 2], color=colors[1], alpha=0.75, s=50)
144 # Turn off the x and y ticks
145 ax.set_xticks([])
146 ax.set_yticks([])
147 ax.set_zticks([])
148 plt.tight_layout()
149 fig.savefig('paper_figures/figure1/initial_conditions.png',
150             bbox_inches=0, transparent=True)
151 plt.clf()
152 plt.close(fig)
153
154 # Plot the PCA for these two tasks
155 xs = torch.stack(xs, dim=0)
156 # Get the components and mean for 3-dim PCA
157 components = np.load('baseline_logs/HCPLeft_PCA_3/components.npy').T
158 mean = np.load('baseline_logs/HCPLeft_PCA_3/mean.npy')
159 # Map input size to 3-dimensions, using the PCA formula
160 xs = xs.view(-1, config['input_size']).numpy()
161 pca_factors = (xs - mean) @ components
162 # Reshape (2 tasks, 2 occurrences)
163 pca_factors = np.reshape(pca_factors, (num_subjects, num_timesteps, 2,
    2, 3))
164
165 # Plot PCA factors
166 fig, ax = plt.subplots(1, 1, figsize=(10, 10),
167                        subplot_kw=dict(projection='3d'))
168 task_1_factors = pca_factors[:, :, 0]
169 task_2_factors = pca_factors[:, :, 1]
170 for i in range(num_subjects):
171     # Task 1, first occurrence
172     ax.plot(
173         task_1_factors[i, :, 0, 0],
174         task_1_factors[i, :, 0, 1],
175         task_1_factors[i, :, 0, 1], color=colors[0], alpha=0.5,
    linewidth=3)
```

```
    # Task 1, second occurrence
    ax.plot(
        task_1_factors[i, :, 1, 0],
        task_1_factors[i, :, 1, 1],
        task_1_factors[i, :, 1, 2], color=colors[0], alpha=0.5,
    linewidth=3)
    # Task 2, first occurrence
    ax.plot(
        task_2_factors[i, :, 0, 0],
        task_2_factors[i, :, 0, 1],
        task_2_factors[i, :, 0, 2], color=colors[1], alpha=0.5,
    linewidth=3)
    # Task 2, second occurrence
    ax.plot(
        task_2_factors[i, :, 1, 0],
        task_2_factors[i, :, 1, 1],
        task_2_factors[i, :, 1, 2], color=colors[1], alpha=0.5,
    linewidth=3)
# Turn off x and y ticks
ax.set_xticks([])
ax.set_yticks([])
ax.set_zticks([])
plt.tight_layout()
fig.savefig('paper_figures/figure1/pca_factors.png',
            bbox_inches=0, transparent=True)
plt.clf()
plt.close(fig)
```

4.7.12 plot_figure2.py

```
import matplotlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from utils import (get_default_config, get_log_string)
matplotlib.use('Agg')
plt.rcParams.update({'font.size': 32})

# Get default config and set all the important variables
# for the plot
config = get_default_config([None])
seeds = [42, 1337, 9999, 1111]
models = ['NODE', 'NODE-nl', 'RNN', 'RNN-nl', 'VAE', 'VAE-nl']
datasets = ['HCPLeft', 'HCPMotor', 'HCPWM',
            'HCPRelational', 'HCPVisual', 'HCPWM-occ']
latent_dims = [2, 4, 8, 16, 32, 64]
factor_results = np.zeros(
    (len(models) + 1,
     len(datasets),
     len(latent_dims),
     2))  # 2 is for mean and SD over seeds
inits_results = np.zeros(
    (len(models),
     len(datasets),
     len(latent_dims),
     2))  # 2 is for mean and SD over seeds

# Color dictionaries
model_dict_factors = {
    'NODE': ('#A65B8C', 'solid'),
    'NODE-nl': ('#A65B8C', 'dashed'),
    'RNN': ('#F2A477', 'solid'),
    'RNN-nl': ('#F2A477', 'dashed'),
    'VAE': ('#A65C41', 'solid'),
    'VAE-nl': ('#A65C41', 'dashed'),
```

```
    'PCA': ('#D9CBBA', 'solid')
}
model_dict_inits = {
    'NODE': ('#A65B8C', 'solid'),
    'NODE-nl': ('#A65B8C', 'dashed'),
    'RNN': ('#F2A477', 'solid'),
    'RNN-nl': ('#F2A477', 'dashed')
}

# Loop over datasets, latent dimensions, models, and seeds
for (d, dataset) in enumerate(datasets):
    for (ld, latent_dim) in enumerate(latent_dims):
        # For the occurrence results for WM
        if dataset == 'HCPWM-occ':
            # Load the results from the log
            pca_path = f'baseline_logs/HCPWM_PCA_{latent_dim}/results.
    csv'
            pca_results = pd.read_csv(pca_path, index_col=0)
            if 'factor_occ_avg_acc' in pca_results.index.values:
                factor_results[-1, d, ld, 0] \
                    = pca_results.loc['factor_occ_avg_acc', '0']
        else:
            # Load the results from the log
            pca_path = f'baseline_logs/{dataset}_PCA_{latent_dim}/
    results.csv'
            pca_results = pd.read_csv(pca_path, index_col=0)
            factor_results[-1, d, ld, 0] \
                = pca_results.loc['factor_avg_acc', '0']
        for (m, model) in enumerate(models):
            seed_results_factors = np.zeros((len(seeds), ))
            seed_results_inits = np.zeros((len(seeds), ))
            for (s, seed) in enumerate(seeds):
                if dataset == 'HCPWM-occ':
                    config['dataset'] = 'HCPWM'
                else:
                    config['dataset'] = dataset
                config['seed'] = seed
                config['latent_dim'] = latent_dim
                if 'VAE' in model:
                    config['learning_rate'] = 0.00005
                    config['temporal_hidden_sizes'] = [128]
                elif 'NODE' in model:
                    config['learning_rate'] = 0.001
                    config['temporal_hidden_sizes'] = [128]
                elif 'RNN' in model:
                    config['learning_rate'] = 0.001
                    config['temporal_hidden_sizes'] = [128, 128]
                if '-nl' in model:
                    config['encoder_type'] = 'MLPEncoder'
                    config['decoder_type'] = 'MLPDecoder'
                    config['hidden_sizes'] = [128]
                    config['model'] = model[:-3]
                else:
                    config['encoder_type'] = 'Linear'
                    config['decoder_type'] = 'Linear'
                    config['hidden_sizes'] = []
                    config['model'] = model
                config_path = get_log_string(config)
                if dataset == 'HCPWM-occ':
                    # Load the results from the log
                    factor_results_path \
                        = config_path / 'occurrence_factor_results.npy
    '
                else:
                    factor_results_path \
```

```
98                          = config_path / 'task_factor_results.npy'
99                  if factor_results_path.is_file():
100                     model_results = np.load(factor_results_path)
101                     seed_results_factors[s] = np.mean(model_results)
102                 if dataset == 'HCPWM-occ':
103                     inits_results_path \
104                         = config_path / 'occurrence_init_results.npy'
105                 else:
106                     inits_results_path \
107                         = config_path / 'task_init_results.npy'
108                 if inits_results_path.is_file():
109                     seed_results_inits[s] = np.load(inits_results_path
    )[0]
110                 else:
111                     print(config)
112                     print(config_path)
113             # Take the mean and SD over the seeds for each model
114             factor_results[m, d, ld, 0] = np.mean(seed_results_factors
    )
115             factor_results[m, d, ld, 1] = np.std(seed_results_factors)
116             inits_results[m, d, ld, 0] = np.mean(seed_results_inits)
117             inits_results[m, d, ld, 1] = np.std(seed_results_inits)
118 
119     # Plot the results with a the SD as a bar around the mean
120     fig, ax = plt.subplots(1, 1, figsize=(10, 10))
121     for (m, plot_args) in enumerate(model_dict_factors.values()):
122         color, linestyle = plot_args
123         ax.plot(
124             latent_dims,
125             factor_results[m, d, :, 0],
126             alpha=0.9, color=color, linewidth=4, linestyle=linestyle)
127         ax.fill_between(
128             latent_dims,
129             factor_results[m, d, :, 0] + factor_results[m, d, :, 1],
130             factor_results[m, d, :, 0], alpha=0.25, color=color)
131         ax.fill_between(
132             latent_dims,
133             factor_results[m, d, :, 0],
134             factor_results[m, d, :, 0] - factor_results[m, d, :, 1],
135             alpha=0.25, color=color)
136         ax.set_xticks([0, 8, 16, 32, 64])
137         ax.set_ylim([0.2, 1.0])
138         ax.set_box_aspect(1)
139 
140     if d > 0:
141         ax.set_yticks([])
142     ax.set_frame_on(False)
143     fig.savefig(f'paper_figures/figure2/{dataset}_factors_results.png'
    ,
144                 bbox_inches=0, transparent=True, dpi=400)
145     plt.clf()
146     plt.close(fig)
147 
148     # Plot the initial condition results
149     fig, ax = plt.subplots(1, 1, figsize=(10, 10))
150     for (m, plot_args) in enumerate(model_dict_inits.values()):
151         color, linestyle = plot_args
152         ax.plot(
153             latent_dims,
154             inits_results[m, d, :, 0],
155             alpha=0.9, color=color, linewidth=4, linestyle=linestyle)
156         ax.fill_between(
157             latent_dims,
158             inits_results[m, d, :, 0] + inits_results[m, d, :, 1],
159             inits_results[m, d, :, 0], alpha=0.25, color=color)
```

```
        ax.fill_between(
            latent_dims,
            inits_results[m, d, :, 0],
            inits_results[m, d, :, 0] - inits_results[m, d, :, 1],
            alpha=0.25, color=color)
        ax.set_xticks([0, 8, 16, 32, 64])
        ax.set_ylim([0.2, 1.0])
        ax.set_box_aspect(1)
    if d > 0:
        ax.set_yticks([])
    ax.set_frame_on(False)
    fig.savefig(f'paper_figures/figure2/{dataset}_inits_results.png',
                bbox_inches=0, transparent=True, dpi=400)
    plt.clf()
    plt.close(fig)
```

4.7.13 plot_figure3_group.py

```
import torch
import matplotlib
import numpy as np
import nibabel as nb
import matplotlib.pyplot as plt
from torch import nn
from sklearn.linear_model import LinearRegression
from utils import (get_default_config, load_model_from_config)
matplotlib.use('Agg')
plt.rcParams.update({'font.size': 22})


def calculate_var_explained(clf_map, group_avg):
    lr = LinearRegression()
    mask = (np.abs(group_avg) >= 0.2)
    quantile_group_avg = 1 - (mask.sum() / mask.size)
    map_quantile = np.quantile(np.abs(clf_map), quantile_group_avg)
    map_mask = (np.abs(clf_map) >= map_quantile)
    clf_map = map_mask * clf_map
    group_avg = mask * group_avg
    lr.fit(clf_map, group_avg)
    return lr.score(clf_map, group_avg)


device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
config = get_default_config([''])
config['dataset'] = 'HCPMotor'
config['learning_rate'] = 0.001
# The group maps and their index + name
group_avg = nb.load(
    '/path/to/
    HCP_S1200_997_tfMRI_ALLTASKS_level2_cohensd_hp200_s2_MSMAll.
    dscalar.nii').get_fdata()
group_avg_labels = [
    (36, 'Visual cue'),
    (37, 'Left foot'),
    (38, 'Left hand'),
    (39, 'Right foot'),
    (40, 'Right hand'),
    (41, 'Tongue')]

seeds = [42, 1337, 9999, 1111]
latent_dims = [8, 16, 32, 64]
# (num_models, latent dims, mean + sd)
scores = np.zeros((4, len(latent_dims), len(group_avg_labels), 2))
for (latent_ix, latent_dim) in enumerate(latent_dims):
    # Load PCA components
```

```
46      pca_map = np.load(
47          f'baseline_logs/{config["dataset"]}_PCA_{config["latent_dim
        "]}/components.npy').T
48      # Load maps for NODE
49      config['model'] = 'NODE'
50      config['latent_dim'] = latent_dim
51      config['temporal_hidden_sizes'] = [128]
52      config['learning_rate'] = 0.001
53      out_layers = []
54      for seed in seeds:
55          print(config)
56          config['seed'] = seed
57          model = load_model_from_config(config)
58          # Need to remove weight norm to get the normal matrix
59          lin = nn.utils.remove_weight_norm(model.out_lin.lin)
60          out_layers.append(lin.weight.detach().cpu().numpy())
61      # Load maps for RNN
62      out_layers_rnn = []
63      config['model'] = 'RNN'
64      config['temporal_hidden_sizes'] = [128, 128]
65      config['learning_rate'] = 0.001
66      for seed in seeds:
67          config['seed'] = seed
68          print(config)
69          model = load_model_from_config(config)
70          lin = nn.utils.remove_weight_norm(model.out_lin.lin)
71          out_layers_rnn.append(lin.weight.detach().cpu().numpy())
72      # Load maps for VAE
73      out_layers_vae = []
74      config['model'] = 'VAE'
75      config['temporal_hidden_sizes'] = [128]
76      config['learning_rate'] = 5e-5
77      for seed in seeds:
78          config['seed'] = seed
79          print(config)
80          model = load_model_from_config(config)
81          lin = nn.utils.remove_weight_norm(model.out_lin.lin)
82          out_layers_vae.append(lin.weight.detach().cpu().numpy())
83      # Calculate variance explained for each model
84      for (i, (group_ix, name)) in enumerate(group_avg_labels):
85          seed_scores = np.zeros((len(seeds), ))
86          for (s, out_layer) in enumerate(out_layers):
87              seed_scores[s] \
88                  = calculate_var_explained(out_layer, group_avg[
        group_ix])
89          # Record mean and SD for NODE
90          scores[0, latent_ix, i, 0] = np.mean(seed_scores)
91          scores[0, latent_ix, i, 1] = np.std(seed_scores)
92          seed_scores = np.zeros((len(seeds), ))
93          for (s, out_layer) in enumerate(out_layers_rnn):
94              seed_scores[s] \
95                  = calculate_var_explained(out_layer, group_avg[
        group_ix])
96          # Record mean and SD for RNN
97          scores[1, latent_ix, i, 0] = np.mean(seed_scores)
98          scores[1, latent_ix, i, 1] = np.std(seed_scores)
99          seed_scores = np.zeros((len(seeds), ))
100         for (s, out_layer) in enumerate(out_layers_vae):
101             seed_scores[s] \
102                 = calculate_var_explained(out_layer, group_avg[
        group_ix])
103         # Record mean ans SD for VAE
104         scores[2, latent_ix, i, 0] = np.mean(seed_scores)
105         scores[2, latent_ix, i, 1] = np.std(seed_scores)
106         # PCA results
```

```
107         scores[3, latent_ix, i, 0] \
108             = calculate_var_explained(pca_map, group_avg[group_ix])
109 
110 colors = ['#A65B8C', '#F2A477', '#A65C41', '#D9CBBA']
111 
112 fig, ax = plt.subplots(1, 6, figsize=(5 * 6, 5), sharey=True)
113 for i in range(4):
114     for j in range(6):
115         ax[j].plot(
116             latent_dims,
117             scores[i, :, j, 0],
118             alpha=0.9, linewidth=3.5, color=colors[i])
119         ax[j].fill_between(
120             latent_dims,
121             scores[i, :, j, 0] + scores[i, :, j, 1],
122             scores[i, :, j, 0], alpha=0.25, color=colors[i])
123         ax[j].fill_between(
124             latent_dims,
125             scores[i, :, j, 0],
126             scores[i, :, j, 0] - scores[i, :, j, 1],
127             alpha=0.25, color=colors[i])
128         ax[j].plot(
129             np.arange(latent_dims[0], latent_dims[-1]),
130             np.ones(latent_dims[-1] - latent_dims[0]) * 0.9,
131             'r--', alpha=0.5, linewidth=3)
132         ax[j].set_xticks([8, 16, 32, 64])
133         ax[j].set_box_aspect(1)
134 plt.tight_layout()
135 fig.savefig('./paper_figures/figure3/group.png', bbox_inches=0, dpi
    =400)
```

4.7.14 plot_figure3_interp.py

```
1 import torch
2 import importlib
3 import matplotlib
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import hcp_utils as hcp
7 from utils import (get_default_config, load_model_from_config,
    mask_input,
8                    create_dataloaders)
9 from nilearn import plotting
10 matplotlib.use('Agg')
11 plt.rcParams.update({'font.size': 22})
12 
13 # Set the config
14 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
15 config = get_default_config([''])
16 config['dataset'] = 'HCPMotor'
17 config['temporal_hidden_sizes'] = [128]
18 config['learning_rate'] = 0.001
19 config['model'] = 'NODE'
20 config['latent_dim'] = 32
21 config['gpu'] = 'V100'
22 config['seed'] = 42
23 model = load_model_from_config(config)
24 time = 284
25 # Load the data and the model
26 model = model.to(device)
27 dataset_module = importlib.import_module('dataset')
28 dataset = getattr(dataset_module, config['dataset'])
29 config['batch_size'] = 1
30 # Create dataloaders
```

```
31 (train_loader, valid_loader, test_loader), (_, va_dataset,
    test_dataset) \
32     = create_dataloaders(dataset, config)
33 va_subjects = len(va_dataset)
34 factors = []
35 inits = []
36 xs = []
37 num_subjects = 300
38 for (i, batch) in enumerate(test_loader):
39     with torch.no_grad():
40         # Depending on whether we use DALI dataloader
41         # or pre-loaded dataset, we need to handle the
42         # batch differently
43         if isinstance(batch[0], torch.Tensor):
44             x = batch[0]
45             x = x.to(device, non_blocking=True).float()
46             mask = batch[1]
47             mask = mask.to(device, non_blocking=True).long()
48         else:
49             x = batch[0]['fmri'].float()
50             mask = batch[0]['mask'].long()
51         model_output = model(x, mask, validation=True)
52         factors.append(model_output['factors'])
53         inits.append(model_output['h_0'])
54         xs.append(mask_input(x, mask).cpu())
55     if i == (num_subjects - 1):
56         break
57
58 inits = torch.stack(inits, dim=0)
59 num_subjects = inits.size(0)
60 window_size = test_dataset.window_size
61 inits = inits.view(num_subjects, 5, 2, config['latent_dim'])
62 # Take left hand and right hand sub-tasks (see prep_motor.py)
63 avg_task1_init = torch.reshape(
64     inits[:, 1], (-1, config['latent_dim'])).mean(0).clone()
65 avg_task2_init = torch.reshape(
66     inits[:, 3], (-1, config['latent_dim'])).mean(0).clone()
67 num_steps = 6
68 time_steps = 5
69 # Interpolating between initial conditions
70 initial_conditions = torch.zeros(
71     (num_steps, config['latent_dim']), device=device)
72 for i in range(config['latent_dim']):
73     initial_conditions[:, i] = torch.linspace(
74         avg_task1_init[i], avg_task2_init[i], num_steps, device=device
    )
75
76 # Generate factors from the initial conditions
77 # and then reconstructions
78 model.eval()
79 with torch.no_grad():
80     t_span = torch.linspace(0, 1, time_steps, device=device)
81     t_span_long = torch.linspace(0, 1, window_size, device=device)
82     _, factors = model.decoder(initial_conditions, t_span)
83     _, factors_long = model.decoder(initial_conditions, t_span_long)
84     reconstructions = model.out_lin(factors).cpu().numpy()
85     reconstructions_long = model.out_lin(factors_long).cpu().numpy()
86
87 # For every interpolation step, create figure
88 for i in range(num_steps):
89     fig, ax = plt.subplots(1, 1, figsize=(15, 2))
90     reconstruction_left = np.zeros_like(reconstructions_long[:, i])
91     reconstruction_right = np.zeros_like(reconstructions_long[:, i])
92     reconstruction_left[:, hcp.struct.cortex_left] \
93         = reconstructions_long[:, i, hcp.struct.cortex_left].copy()
```

```
    reconstruction_right[:, hcp.struct.cortex_right] \
        = reconstructions_long[:, i, hcp.struct.cortex_right].copy()
    print(hcp.yeo7)
    print(reconstruction_left.shape, reconstruction_right.shape)
    reconstruction_left_motor \
        = reconstruction_left[:, hcp.yeo7['map_all'] == 2].mean(-1)
    reconstruction_right_motor \
        = reconstruction_right[:, hcp.yeo7['map_all'] == 2].mean(-1)
    ax.plot(reconstruction_left_motor,
            color='#B8BF80', alpha=0.75, linewidth=8)
    ax.plot(reconstruction_right_motor,
            color='#A65B8C', alpha=0.75, linewidth=8)
    ax.axis('off')
    fig.savefig(f'paper_figures/figure3/interpolation_line_{i}.png',
                bbox_inches=0, transparent=True, dpi=400)
    plt.clf()
    plt.close(fig)

hemisphere = 'right'
view = 'dorsal'
for i in range(num_steps):
    fig, axs = plt.subplots(1, time_steps * 2, figsize=(20, 2),
                            subplot_kw=dict(projection='3d'))
    for t_ix in range(time_steps * 2):
        t = t_ix // 2
        print(f'Time: {t}, step: {i}')
        cortex = hcp.cortex_data(reconstructions[t, i])
        hemisphere = 'right' if t_ix % 2 == 0 else 'left'
        plotting.plot_surf_stat_map(
            hcp.mesh.midthickness, cortex, hcp.mesh.sulc, axes=axs[
    t_ix],
            vmax=np.max(np.abs(reconstructions)), colorbar=False,
            threshold=0.01, alpha=1.0, bg_on_data=True,
            darkness=1.0, hemi=hemisphere, view='lateral',
            cmap=plt.get_cmap('jet'))
        axs[t].axis('off')
    fig.savefig(f'paper_figures/figure3/interpolation_{i}.png',
                bbox_inches=0, transparent=True, dpi=300)
    plt.clf()
    plt.close(fig)
```

4.7.15 plot_figure4a.py

```
import torch
import importlib
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from torch import nn
from pathlib import Path
from torch.optim.lr_scheduler import StepLR
from sklearn.decomposition import PCA
from utils import (get_default_config, load_model_from_config,
                   create_dataloaders)
from golub import FixedPoints
matplotlib.use('Agg')
plt.rcParams.update({'font.size': 22})

# This code is very similar to find_fixed_points.py
# a more in-depth explanation of the code can be found there
seeds = [42, 1337, 9999, 1111]
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
config = get_default_config(['', 0])
config['dataset'] = 'HCPLeft'
config['temporal_hidden_sizes'] = [128]
```

```
23 config['latent_dim'] = 4
24 config['gpu'] = 'V100'
25 config['batch_size'] = 1
26 config['epochs'] = 500
27 datasets = ['HCPLeft']
28 for (s_ix, seed) in enumerate(seeds):
29     dataset_module = importlib.import_module('dataset')
30     dataset = getattr(dataset_module, config['dataset'])
31     (train_loader, valid_loader, test_loader), (_, va_dataset, _) \
32         = create_dataloaders(dataset, config)
33     config['seed'] = seed
34     model = load_model_from_config(config)
35     # Load the data and the model
36     model = model.to(device)
37     time = va_dataset.window_size
38     num_tasks = va_dataset.num_tasks
39     num_occurrences = va_dataset.num_occurrences
40     t_span = torch.linspace(0, 1, time)
41     va_subjects = len(va_dataset)
42     inits = []
43     num_subj = 200
44     for (i, batch) in enumerate(train_loader):
45         with torch.no_grad():
46             if isinstance(batch[0], torch.Tensor):
47                 x = batch[0]
48                 x = x.to(device, non_blocking=True).float()
49                 mask = batch[1]
50                 mask = mask.to(device, non_blocking=True).long()
51             else:
52                 x = batch[0]['fmri'].float()
53                 mask = batch[0]['mask'].long()
54             init = model.encode_init(x, mask, validation=True)
55             inits.append(init)
56         if i == (num_subj - 1):
57             break
58     inits = torch.stack(inits, dim=0)
59     inits = inits.view(-1, config['latent_dim'])
60     model = model.eval()
61     with torch.no_grad():
62         _, factors = model.decoder(inits, t_span)
63
64     # Optimize
65     torch.manual_seed(seed)
66     # Get trajectories
67     factors = factors.view(-1, config['latent_dim'])
68     factors_detach = factors.detach().clone()
69     # Add gaussian noise to the trajectories
70     factors_noise = torch.cat(
71         (factors_detach,
72          factors_detach + torch.randn(factors_detach.size(),
73                                       device=device) * 0.1), dim=0)
74     x = nn.Parameter(factors_noise)
75     optimizer = torch.optim.Adam([x], lr=0.01)
76     scheduler = StepLR(optimizer, step_size=2000, gamma=0.9)
77     for p in model.parameters(): p.requires_grad = False
78
79     j = 0
80     q_prev = torch.full((x.size(0),), float("nan"), device=device)
81     n_iters = 20000
82     for i in range(n_iters):
83         optimizer.zero_grad()
84         q = 0.5 * torch.sum(model.decoder.vf(None, x) ** 2, dim=1)
85         loss = q.mean(0)
86         loss.backward()
87         optimizer.step()
```

```
88          if i % 1000 == 0:
89              print(loss, q.min())
90          scheduler.step()
91          dq = torch.abs(q - q_prev)
92          q_prev = q
93      qstar = q.cpu().detach().numpy()
94      all_fps = FixedPoints(
95          xstar=x.cpu().detach().numpy().squeeze(),
96          x_init=factors_noise.cpu(),
97          qstar=qstar,
98          dq=dq.cpu().detach().numpy(),
99          n_iters=np.full_like(qstar, n_iters),
100         tol_unique=1E-1,
101     )
102     unique_fps = all_fps.get_unique()
103     best_fps = unique_fps.qstar < 1E-8
104     if best_fps.sum() > 0:
105         best_fps = FixedPoints(
106             xstar=unique_fps.xstar[best_fps],
107             x_init=unique_fps.x_init[best_fps],
108             qstar=unique_fps.qstar[best_fps],
109             dq=unique_fps.dq[best_fps],
110             n_iters=unique_fps.n_iters[best_fps],
111             tol_unique=1E-4,
112         )
113         func = lambda x: (1/time) * model.decoder.vf(None, x) + x
114 
115         all_J = []
116         x = torch.tensor(best_fps.xstar, device=device)
117         for i in range(best_fps.n):
118             single_x = x[i, :]
119             J = torch.autograd.functional.jacobian(func, single_x)
120             all_J.append(J)
121         # Recombine and decompose Jacobians for the whole batch
122         dFdx = torch.stack(all_J).cpu().detach().numpy()
123         best_fps.J_xstar = dFdx
124         best_fps.decompose_jacobians()
125         print(best_fps.eigval_J_xstar)
126         print(best_fps.eigval_J_xstar.shape)
127         # Save the fixed point for each seed
128         np.save(f'fixed_point_experiments/fps/HCPLeft_{seed}.npy',
129                 best_fps.eigval_J_xstar)
130 
131         # Plot the trajectories for left hand vs left foot
132         fp_colors = ['#A65B8C', '#B8BF80']
133         if s_ix == 0:
134             fig = plt.figure(figsize=(10, 10))
135             ax = fig.add_subplot(1, 1, 1, projection='3d')
136             factors_detach = factors_detach.view(
137                 time, num_subj, num_tasks, num_occurrences,
138                 config['latent_dim'])
139             # Show the first 100 subjects
140             factors_detach = factors_detach[:, :, :, 0]
141             factors_detach = factors_detach[:, :100]
142             factors_np = factors_detach.contiguous().view(
143                 time * 100 * num_tasks, config['latent_dim']).cpu().
    numpy()
144             pca = PCA(n_components=3)
145             num_fps = best_fps.xstar.shape[0]
146             factors_tsne = pca.fit_transform(factors_np)
147             fp_tsne = pca.transform(best_fps.xstar)
148             factors_tsne = np.reshape(factors_tsne, (time, 100,
    num_tasks, 3))
149 
150             for i in range(num_fps):
```

```
                ax.scatter(
                    fp_tsne[i, 0],
                    fp_tsne[i, 1],
                    fp_tsne[i, 2], color='#F2A477', s=100, alpha=0.75)

            for i in range(100):
                for j in range(num_tasks):
                    ax.plot(
                        factors_tsne[:, i, j, 0],
                        factors_tsne[:, i, j, 1],
                        factors_tsne[:, i, j, 2],
                        alpha=0.4, color=fp_colors[j])
            ax.set_xticks([])
            ax.set_yticks([])
            ax.set_zticks([])
            plt.savefig('paper_figures/figure4/dynamicsa.png',
                        bbox_inches=0, transparent=True)
            plt.clf()
            plt.close(fig)

# Get the fixed points
fixed_points = np.zeros((4, 4), dtype='complex')
seeds = [42, 1337, 9999, 1111]
for (s, seed) in enumerate(seeds):
    fps_p = Path(f'fixed_point_experiments/fps/HCPLeft_{seed}.npy')
    if fps_p.is_file():
        fp = np.load(fps_p)
        ix = np.argsort(-np.abs(fp[0].imag))
        fixed_points[s] = fp[0][ix]

# Plot the eigenvalue plot for the left hand vs left foot task
real_max = np.abs(fixed_points.real).max() + 0.05
imag_max = np.abs(fixed_points.imag).max() + 0.1
markers = ['X', 's', 'o', 'v']
colors = ['#A65B8C', '#F2A477', 'black', '#D9CBBA']
t = np.linspace(0, np.pi*2, 100)
fig, ax = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
for i in range(4):
    ax.scatter(
        fixed_points[:, i].real,
        fixed_points[:, i].imag,
        marker='X', alpha=0.75, c=colors[i//2])
# Add the zero line
ax.plot(
    np.linspace(0.9, 1.10, 100),
    [0] * 100, linewidth=1, color='b', linestyle='dashed', alpha=0.5)
# Add init circle
ax.plot(np.cos(t), np.sin(t), color='black', alpha=0.5)
ax.set_xlim([0.9, 1.10])
ax.set_ylim([-0.3, 0.3])
ax.set_box_aspect(1)
plt.tight_layout()
fig.savefig('paper_figures/figure4/fixed_pointsa.png',
            dpi=400, bbox_inches=0, transparent=True)
```

4.7.16 plot_figure4b.py

```
import torch
import importlib
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
from torch import nn
from pathlib import Path
from torch.optim.lr_scheduler import StepLR
```

```
from utils import (get_default_config, create_dataloaders, init_model)
from golub import FixedPoints
matplotlib.use('Agg')
plt.rcParams.update({'font.size': 22})

# This is the same as plot_figure4a.py, except we use
# randomly initialized (or untrained) versions of our model
seeds = [42, 1337, 9999, 1111]
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
config = get_default_config(['', 0])
config['dataset'] = 'HCPLeft'
config['temporal_hidden_sizes'] = [128]
config['latent_dim'] = 4
config['gpu'] = 'V100'
config['batch_size'] = 1
config['epochs'] = 500

for (s_ix, seed) in enumerate(seeds):
    config['seed'] = seed
    dataset_module = importlib.import_module('dataset')
    dataset = getattr(dataset_module, config['dataset'])
    (train_loader, valid_loader, test_loader), (_, va_dataset, _) \
        = create_dataloaders(dataset, config)
    model_module = importlib.import_module('model')
    model_type = getattr(model_module, config['model'])
    model = init_model(model_type, config)
    # Load the data and the model
    model = model.to(device)
    time = va_dataset.window_size
    num_tasks = va_dataset.num_tasks
    num_occurrences = va_dataset.num_occurrences
    t_span = torch.linspace(0, 1, time)
    va_subjects = len(va_dataset)
    inits = []
    num_subj = 200
    for (i, batch) in enumerate(train_loader):
        with torch.no_grad():
            if isinstance(batch[0], torch.Tensor):
                x = batch[0]
                x = x.to(device, non_blocking=True).float()
                mask = batch[1]
                mask = mask.to(device, non_blocking=True).long()
            else:
                x = batch[0]['fmri'].float()
                mask = batch[0]['mask'].long()
            init = model.encode_init(x, mask, validation=True)
            inits.append(init)
        if i == (num_subj - 1):
            break
    inits = torch.stack(inits, dim=0)
    inits = inits.view(-1, config['latent_dim'])
    print(inits.size())
    model = model.eval()
    with torch.no_grad():
        # TODO: Sample from init distribution
        _, factors = model.decoder(inits, t_span)

    # Optimize
    torch.manual_seed(seed)
    # Get trajectories
    factors = factors.view(-1, config['latent_dim'])
    factors_detach = factors.detach().clone()
    # Add gaussian to the trajectories
    factors_noise = torch.cat((
        factors_detach,
```

```
        factors_detach + torch.randn(factors_detach.size(),
                                     device=device) * 0.1), dim=0)
    x = nn.Parameter(factors_noise)
    optimizer = torch.optim.Adam([x], lr=0.01)
    scheduler = StepLR(optimizer, step_size=2000, gamma=0.9)
    for p in model.parameters(): p.requires_grad = False

    j = 0
    q_prev = torch.full((x.size(0),), float("nan"), device=device)
    n_iters = 20000
    for i in range(n_iters):
        optimizer.zero_grad()
        q = 0.5 * torch.sum(model.decoder.vf(None, x) ** 2, dim=1)
        loss = q.mean(0)
        loss.backward()
        optimizer.step()
        if i % 1000 == 0:
            j += 1
            print(loss, q.min())
        scheduler.step()
        dq = torch.abs(q - q_prev)
        q_prev = q
    qstar = q.cpu().detach().numpy()
    all_fps = FixedPoints(
        xstar=x.cpu().detach().numpy().squeeze(),
        x_init=factors_noise.cpu(),
        qstar=qstar,
        dq=dq.cpu().detach().numpy(),
        n_iters=np.full_like(qstar, n_iters),
        tol_unique=1E-1,
    )
    unique_fps = all_fps.get_unique()
    best_fps = unique_fps.qstar < 1E-8
    if best_fps.sum() > 0:
        best_fps = FixedPoints(
            xstar=unique_fps.xstar[best_fps],
            x_init=unique_fps.x_init[best_fps],
            qstar=unique_fps.qstar[best_fps],
            dq=unique_fps.dq[best_fps],
            n_iters=unique_fps.n_iters[best_fps],
            tol_unique=1E-4,
        )

        # We use the TR (0.72)
        func = lambda x: (1/time) * model.decoder.vf(None, x) + x

        all_J = []
        x = torch.tensor(best_fps.xstar, device=device)
        for i in range(best_fps.n):
            single_x = x[i, :]
            J = torch.autograd.functional.jacobian(func, single_x)
            all_J.append(J)
        # Recombine and decompose Jacobians for the whole batch
        dFdx = torch.stack(all_J).cpu().detach().numpy()
        best_fps.J_xstar = dFdx
        best_fps.decompose_jacobians()

        print(best_fps.eigval_J_xstar)
        print(best_fps.eigval_J_xstar.shape)
        np.save(f'fixed_point_experiments/fps/Random_{seed}.npy',
                best_fps.eigval_J_xstar)

fixed_points = np.zeros((4, 4), dtype='complex')
seeds = [42, 1337, 9999, 1111]
for (s, seed) in enumerate(seeds):
```

```
    fps_p = Path(f'fixed_point_experiments/fps/Random_{seed}.npy')
    if fps_p.is_file():
        fp = np.load(fps_p)
        ix = np.argsort(-np.abs(fp[0].imag))
        fixed_points[s] = fp[0][ix]

real_max = np.abs(fixed_points.real).max() + 0.05
imag_max = np.abs(fixed_points.imag).max() + 0.1
markers = ['X', 's', 'o', 'v']
colors = ['#A65B8C', '#F2A477', 'black', '#D9CBBA']
t = np.linspace(0, np.pi*2, 100)
fig, ax = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
for i in list(range(4))[::-1]:
    ax.scatter(
        fixed_points[:, i].real,
        fixed_points[:, i].imag,
        marker='X', alpha=0.75, c=colors[i//2])
# Add the zero line
ax.plot(
    np.linspace(0.9, 1.10, 100),
    [0] * 100, linewidth=1, color='b', linestyle='dashed', alpha=0.5)
# Add init circle
ax.plot(np.cos(t), np.sin(t), color='black', alpha=0.5)
ax.set_xlim([0.9, 1.10])
ax.set_ylim([-0.3, 0.3])
ax.set_box_aspect(1)
plt.tight_layout()
fig.savefig('paper_figures/figure4/fixed_pointsb.png',
            dpi=400, bbox_inches=0, transparent=True)
```

4.7.17 plot_figure4c.py

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from pathlib import Path
matplotlib.use('Agg')
plt.rcParams.update({'font.size': 22})

fixed_points = np.zeros((3, 4, 8), dtype='complex')
datasets = ['HCPMotorLong', 'HCPWMLong', 'HCPRelationalLong']
seeds = [42, 1337, 9999, 1111]
for (d, dataset_name) in enumerate(datasets):
    for (s, seed) in enumerate(seeds):
        fps_p = Path(f'fixed_point_experiments/fps/{dataset_name}_{
    seed}.npy')
        if fps_p.is_file():
            fp = np.load(fps_p)
            ix = np.argsort(-np.abs(fp[0].imag))
            fixed_points[d, s] = fp[0][ix]

print(fixed_points.shape)
print(fixed_points[0])
hcp_motor = fixed_points[0]
hcp_wm = fixed_points[1]
hcp_relational = fixed_points[2]
t = np.linspace(0, np.pi*2, 100)
# Add offset for figure
real_max = np.array([
    np.abs(hcp_motor.real).max(),
    np.abs(hcp_wm.real).max(),
    np.abs(hcp_relational.real).max()]).max() + 0.05
imag_max = np.array([
    np.abs(hcp_motor.imag).max(),
    np.abs(hcp_wm.imag).max(),
```

```
33     np.abs(hcp_relational.imag).max()]).max() + 0.1
34 markers = ['X', 's', 'o', 'v']
35 colors = ['#A65B8C', '#F2A477', 'black', '#D9CBBA']
36 fps_ls = [hcp_motor, hcp_wm, hcp_relational]
37 datasets = ['HCPMotor', 'HCPWM', 'HCPRelational']
38 fig, axs = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
39 for (fp_ix, fps) in enumerate(fps_ls):
40     fig, axs = plt.subplots(1, 1, figsize=(5, 5), sharey=True)
41     for i in range(8):
42         if fp_ix == 0:
43             axs.scatter(
44                 fps[0, i].real,
45                 fps[0, i].imag, marker='o', alpha=0.3, c=colors[i//2])
46             axs.scatter(
47                 fps[1:, i].real,
48                 fps[1:, i].imag, marker='X', alpha=0.75, c=colors[i
    //2])
49         elif fp_ix == 1:
50             axs.scatter(
51                 fps[:, i].real,
52                 fps[:, i].imag, marker='X', alpha=0.75, c=colors[i
    //2])
53         elif fp_ix == 2:
54             axs.scatter(
55                 fps[:3, i].real,
56                 fps[:3, i].imag, marker='X', alpha=0.75, c=colors[i
    //2])
57             axs.scatter(
58                 fps[3, i].real,
59                 fps[3, i].imag, marker='o', alpha=0.3, c=colors[i//2])
60         # Add zero line
61         axs.plot(
62             np.linspace(-real_max, real_max, 100),
63             [0] * 100, linewidth=1, color='b', linestyle='dashed',
    alpha=0.5)
64         # Add unit circle
65         axs.plot(np.cos(t), np.sin(t), color='black', alpha=0.5)
66         axs.set_xlim([0.8, real_max])
67         axs.set_ylim([-imag_max, imag_max])
68         axs.set_box_aspect(1)
69     plt.tight_layout()
70     fig.savefig(f'paper_figures/figure4/{datasets[fp_ix]}c.png',
71                 dpi=400, bbox_inches=0, transparent=True)
72     plt.clf()
73     plt.close(fig)
```

4.7.18 plot_supplement_folds.py

```
1 import matplotlib
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from utils import (get_default_config, get_log_string)
6 matplotlib.use('Agg')
7 plt.rcParams.update({'font.size': 32})
8
9 config = get_default_config([None])
10 seeds = [42]
11 models = ['NODE', 'NODE-nl', 'RNN', 'RNN-nl', 'VAE', 'VAE-nl']
12 latent_dims = [2, 4, 8, 16, 32, 64]
13 factor_results = np.zeros(
14     (len(models) + 1,
15      len(latent_dims),
16      2))  # 2 is for mean and SD over folds
17 inits_results = np.zeros(
```

```
18     (len(models),
19      len(latent_dims),
20      2))  # 2 is for mean and SD over folds
21
22 # Color dictionaries
23 model_dict_factors = {
24     'NODE': ('#A65B8C', 'solid'),
25     'NODE-nl': ('#A65B8C', 'dashed'),
26     'RNN': ('#F2A477', 'solid'),
27     'RNN-nl': ('#F2A477', 'dashed'),
28     'VAE': ('#A65C41', 'solid'),
29     'VAE-nl': ('#A65C41', 'dashed'),
30     'PCA': ('#D9CBBA', 'solid')
31 }
32 model_dict_inits = {
33     'NODE': ('#A65B8C', 'solid'),
34     'NODE-nl': ('#A65B8C', 'dashed'),
35     'RNN': ('#F2A477', 'solid'),
36     'RNN-nl': ('#F2A477', 'dashed')
37 }
38 folds = list(range(10))
39 dataset = 'HCPVisual'
40 for (ld, latent_dim) in enumerate(latent_dims):
41     fold_results = np.zeros((10, ))
42     # Load results for each fold
43     for fold in folds:
44         pca_path = f'baseline_logs/{dataset}_PCA_{latent_dim}/fold_{
    fold}/results.csv'
45         pca_results = pd.read_csv(pca_path, index_col=0)
46         fold_results[fold] = pca_results.loc['factor_avg_acc', '0']
47     # Average and SD over folds
48     factor_results[-1, ld, 0] = np.mean(fold_results)
49     factor_results[-1, ld, 1] = np.std(fold_results)
50     for (m, model) in enumerate(models):
51         fold_results_factors = np.zeros((10, ))
52         fold_results_inits = np.zeros((10, ))
53         for fold in folds:
54             config['dataset'] = dataset
55             config['seed'] = 42
56             config['latent_dim'] = latent_dim
57             if 'VAE' in model:
58                 config['learning_rate'] = 0.00005
59                 config['temporal_hidden_sizes'] = [128]
60             else:
61                 config['learning_rate'] = 0.001
62                 config['temporal_hidden_sizes'] = [128]
63             if '-nl' in model:
64                 config['encoder_type'] = 'MLPEncoder'
65                 config['decoder_type'] = 'MLPDecoder'
66                 config['hidden_sizes'] = [128]
67                 config['model'] = model[:-3]
68             else:
69                 config['encoder_type'] = 'Linear'
70                 config['decoder_type'] = 'Linear'
71                 config['hidden_sizes'] = []
72                 config['model'] = model
73             config_path = get_log_string(config)
74             print(config_path)
75             factor_results_path \
76                 = config_path / f'fold_{fold}' / 'task_factor_results.
    npy'
77             if factor_results_path.is_file():
78                 model_results = np.load(factor_results_path)
79                 fold_results_factors[fold] = np.mean(model_results)
80             inits_results_path \
```

```
81                  = config_path / f'fold_{fold}' / 'task_init_results.
    npy'
82              if inits_results_path.is_file():
83                  fold_results_inits[fold] = np.load(inits_results_path)
    [0]
84              else:
85                  print(config)
86                  print(config_path)
87          print(dataset)
88          factor_results[m, ld, 0] = np.mean(fold_results_factors)
89          factor_results[m, ld, 1] = np.std(fold_results_factors)
90          inits_results[m, ld, 0] = np.mean(fold_results_inits)
91          inits_results[m, ld, 1] = np.std(fold_results_inits)
92
93  fig, ax = plt.subplots(1, 1, figsize=(10, 10))
94  for (m, plot_args) in enumerate(model_dict_factors.values()):
95      color, linestyle = plot_args
96      ax.plot(
97          latent_dims,
98          factor_results[m, :, 0],
99          alpha=0.9,
100         color=color, linewidth=4, linestyle=linestyle)
101     ax.fill_between(
102         latent_dims,
103         factor_results[m, :, 0] + factor_results[m, :, 1],
104         factor_results[m, :, 0], alpha=0.25, color=color)
105     ax.fill_between(
106         latent_dims,
107         factor_results[m, :, 0],
108         factor_results[m, :, 0] - factor_results[m, :, 1],
109         alpha=0.25, color=color)
110     ax.set_xticks([0, 8, 16, 32, 64])
111     ax.set_ylim([0.2, 1.0])
112     ax.set_box_aspect(1)
113 ax.set_frame_on(False)
114 fig.savefig('paper_supplements/folds/supplement_folds_factors_results.
    png',
115             bbox_inches=0, transparent=True, dpi=400)
116 plt.clf()
117 plt.close(fig)
118
119 fig, ax = plt.subplots(1, 1, figsize=(10, 10))
120 for (m, plot_args) in enumerate(model_dict_inits.values()):
121     color, linestyle = plot_args
122     ax.plot(
123         latent_dims,
124         inits_results[m, :, 0],
125         alpha=0.9, color=color, linewidth=4, linestyle=linestyle)
126     ax.fill_between(
127         latent_dims,
128         inits_results[m, :, 0] + inits_results[m, :, 1],
129         inits_results[m, :, 0], alpha=0.25, color=color)
130     ax.fill_between(
131         latent_dims,
132         inits_results[m, :, 0],
133         inits_results[m, :, 0] - inits_results[m, :, 1],
134         alpha=0.25, color=color)
135     ax.set_xticks([0, 8, 16, 32, 64])
136     ax.set_ylim([0.2, 1.0])
137     ax.set_box_aspect(1)
138 ax.set_yticks([])
139 ax.set_frame_on(False)
140 fig.savefig('paper_supplements/folds/supplement_folds_inits_results.
    png',
141             bbox_inches=0, transparent=True, dpi=400)
```

```
142 plt.clf()
143 plt.close(fig)
```

4.7.19 plot_supplement_group.py

```
1 import torch
2 import matplotlib
3 import numpy as np
4 import nibabel as nb
5 import matplotlib.pyplot as plt
6 from torch import nn
7 from sklearn.linear_model import LinearRegression
8 from utils import (get_default_config, load_model_from_config)
9 matplotlib.use('Agg')
10 plt.rcParams.update({'font.size': 22})
11
12
13 # This file is almost the same as plot_figure3_group.py
14 def calculate_var_explained(clf_map, group_avg):
15     lr = LinearRegression()
16     mask = (np.abs(group_avg) >= 0.2)
17     quantile_group_avg = 1 - (mask.sum() / mask.size)
18     map_quantile = np.quantile(np.abs(clf_map), quantile_group_avg)
19     map_mask = (np.abs(clf_map) >= map_quantile)
20     clf_map = map_mask * clf_map
21     group_avg = mask * group_avg
22     lr.fit(clf_map, group_avg)
23     return lr.score(clf_map, group_avg)
24
25
26 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
27 config = get_default_config([''])
28 config['dataset'] = 'HCPMotor'
29 config['learning_rate'] = 0.001
30 group_avg = nb.load(
31     '/path/to/
    HCP_S1200_997_tfMRI_ALLTASKS_level2_cohensd_hp200_s2_MSMAll.
    dscalar.nii').get_fdata()
32 group_avg_labels = [
33     (36, 'Visual cue'),
34     (37, 'Left foot'),
35     (38, 'Left hand'),
36     (39, 'Right foot'),
37     (40, 'Right hand'),
38     (41, 'Tongue')]
39
40 seeds = [42, 1337, 9999, 1111]
41 latent_dims = [2, 4, 8, 16, 32, 64]
42 # (num_models, latent dims, mean + sd)
43 scores = np.zeros((4, len(latent_dims), len(group_avg_labels), 2))
44 for (latent_ix, latent_dim) in enumerate(latent_dims):
45     config['model'] = 'NODE'
46     config['latent_dim'] = latent_dim
47     config['temporal_hidden_sizes'] = [128]
48     config['learning_rate'] = 0.001
49     pca_map = np.load(
50         f'baseline_logs/{config["dataset"]}_PCA_{config["latent_dim
    "]}/components.npy').T
51     out_layers = []
52     for seed in seeds:
53         config['seed'] = seed
54         model = load_model_from_config(config)
55         lin = nn.utils.remove_weight_norm(model.out_lin.lin)
56         out_layers.append(lin.weight.detach().cpu().numpy())
57     out_layers_rnn = []
```

```
    config['model'] = 'RNN'
    config['temporal_hidden_sizes'] = [128, 128]
    config['learning_rate'] = 0.001
    for seed in seeds:
        config['seed'] = seed
        model = load_model_from_config(config)
        lin = nn.utils.remove_weight_norm(model.out_lin.lin)
        out_layers_rnn.append(lin.weight.detach().cpu().numpy())
    out_layers_vae = []
    config['model'] = 'VAE'
    config['temporal_hidden_sizes'] = [128]
    config['learning_rate'] = 5e-5
    for seed in seeds:
        config['seed'] = seed
        model = load_model_from_config(config)
        lin = nn.utils.remove_weight_norm(model.out_lin.lin)
        out_layers_vae.append(lin.weight.detach().cpu().numpy())
    for (i, (group_ix, name)) in enumerate(group_avg_labels):
        seed_scores = np.zeros((len(seeds), ))
        for (s, out_layer) in enumerate(out_layers):
            seed_scores[s] \
                = calculate_var_explained(out_layer, group_avg[
    group_ix])
        # Record mean and SD for NODE
        scores[0, latent_ix, i, 0] = np.mean(seed_scores)
        scores[0, latent_ix, i, 1] = np.std(seed_scores)
        seed_scores = np.zeros((len(seeds), ))
        for (s, out_layer) in enumerate(out_layers_rnn):
            seed_scores[s] \
                = calculate_var_explained(out_layer, group_avg[
    group_ix])
        scores[1, latent_ix, i, 0] = np.mean(seed_scores)
        scores[1, latent_ix, i, 1] = np.std(seed_scores)
        seed_scores = np.zeros((len(seeds), ))
        for (s, out_layer) in enumerate(out_layers_vae):
            seed_scores[s] \
                = calculate_var_explained(out_layer, group_avg[
    group_ix])
        scores[2, latent_ix, i, 0] = np.mean(seed_scores)
        scores[2, latent_ix, i, 1] = np.std(seed_scores)
        # PCA results
        scores[3, latent_ix, i, 0] \
            = calculate_var_explained(pca_map, group_avg[group_ix])

colors = ['#A65B8C', '#F2A477', '#A65C41', '#D9CBBA']

fig, ax = plt.subplots(1, 6, figsize=(5 * 6, 5), sharey=True)
for i in range(4):
    for j in range(6):
        ax[j].plot(
            latent_dims,
            scores[i, :, j, 0],
            alpha=0.9, linewidth=3.5, color=colors[i])
        ax[j].fill_between(
            latent_dims,
            scores[i, :, j, 0] + scores[i, :, j, 1],
            scores[i, :, j, 0], alpha=0.25, color=colors[i])
        ax[j].fill_between(
            latent_dims,
            scores[i, :, j, 0],
            scores[i, :, j, 0] - scores[i, :, j, 1],
            alpha=0.25, color=colors[i])
        ax[j].plot(
            np.arange(latent_dims[0], latent_dims[-1]),
            np.ones(latent_dims[-1] - latent_dims[0]) * 0.9,
```

```
120             'r--', alpha=0.5, linewidth=3)
121         ax[j].set_xticks([0, 8, 16, 32, 64])
122         ax[j].set_box_aspect(1)
123 plt.tight_layout()
124 fig.savefig('./paper_supplements/spatial_variance/group.png',
125             bbox_inches=0, dpi=400)
```

4.7.20 plot_supplement_visual.py

```
1 import matplotlib
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from utils import (get_default_config, get_log_string)
6 matplotlib.use('Agg')
7 plt.rcParams.update({'font.size': 32})
8
9
10 # This function plots the results for the 'motor from visual'
11 # task, where we show classification accuracies over time
12 # in the supplement
13 config = get_default_config([None])
14 seeds = [42, 1337, 9999, 1111]
15 models = ['NODE', 'NODE-nl', 'RNN', 'RNN-nl', 'VAE', 'VAE-nl']
16 latent_dims = [2, 4, 8, 16, 32, 64]
17 factor_results = np.zeros(
18     (len(models) + 1,
19      len(latent_dims),
20      23,  # Window length
21      2))  # 2 is for mean and SD over seeds
22 time = np.arange(23)
23
24 # Color dictionaries
25 model_dict_factors = {
26     'NODE': ('#A65B8C', 'solid'),
27     'NODE-nl': ('#A65B8C', 'dashed'),
28     'RNN': ('#F2A477', 'solid'),
29     'RNN-nl': ('#F2A477', 'dashed'),
30     'VAE': ('#A65C41', 'solid'),
31     'VAE-nl': ('#A65C41', 'dashed'),
32     'PCA': ('#D9CBBA', 'solid')
33 }
34 model_dict_inits = {
35     'NODE': ('#A65B8C', 'solid'),
36     'NODE-nl': ('#A65B8C', 'dashed'),
37     'RNN': ('#F2A477', 'solid'),
38     'RNN-nl': ('#F2A477', 'dashed')
39 }
40 dataset = 'HCPVisual'
41 for (ld, latent_dim) in enumerate(latent_dims):
42     pca_path = f'baseline_logs/{dataset}_PCA_{latent_dim}/results.csv'
43     pca_results = pd.read_csv(pca_path, index_col=0)
44     factor_results[-1, ld, :, 0] = eval(pca_results.loc['factor_acc',
       '0'])
45     for (m, model) in enumerate(models):
46         seed_results = np.zeros((len(seeds), 23))
47         for (s, seed) in enumerate(seeds):
48             config['dataset'] = dataset
49             config['seed'] = seed
50             config['latent_dim'] = latent_dim
51             if 'VAE' in model:
52                 config['learning_rate'] = 0.00005
53                 config['temporal_hidden_sizes'] = [128]
54             elif 'NODE' in model:
55                 config['learning_rate'] = 0.001
```

```
56                  config['temporal_hidden_sizes'] = [128]
57              elif 'RNN' in model:
58                  config['learning_rate'] = 0.001
59                  config['temporal_hidden_sizes'] = [128, 128]
60              if '-nl' in model:
61                  config['encoder_type'] = 'MLPEncoder'
62                  config['decoder_type'] = 'MLPDecoder'
63                  config['hidden_sizes'] = [128]
64                  config['model'] = model[:-3]
65              else:
66                  config['encoder_type'] = 'Linear'
67                  config['decoder_type'] = 'Linear'
68                  config['hidden_sizes'] = []
69                  config['model'] = model
70              config_path = get_log_string(config)
71              factor_results_path = config_path / 'task_factor_results.
    npy'
72              if factor_results_path.is_file():
73                  model_results = np.load(factor_results_path)
74                  seed_results[s] = model_results
75          factor_results[m, ld, :, 0] = np.mean(seed_results, axis=0)
76          factor_results[m, ld, :, 1] = np.std(seed_results, axis=0)
77
78      fig, ax = plt.subplots(1, 1, figsize=(10, 10))
79      for (m, plot_args) in enumerate(model_dict_factors.values()):
80          color, linestyle = plot_args
81          ax.plot(
82              time,
83              factor_results[m, ld, :, 0],
84              alpha=0.9, color=color, linewidth=4, linestyle=linestyle)
85          ax.fill_between(
86              time,
87              factor_results[m, ld, :, 0] + factor_results[m, ld, :, 1],
88              factor_results[m, ld, :, 0], alpha=0.25, color=color)
89          ax.fill_between(
90              time,
91              factor_results[m, ld, :, 0],
92              factor_results[m, ld, :, 0] - factor_results[m, ld, :, 1],
93              alpha=0.25, color=color)
94          ax.set_xticks([0, 23])
95          ax.set_ylim([0.2, 0.8])
96          ax.set_box_aspect(1)
97          ax.plot(
98              [4] * 100, np.linspace(0.2, 0.8, 100),
99              color='r', linestyle='dashed', alpha=0.8, linewidth=2)
100     if latent_dim not in [2, 4]:
101         ax.set_yticks([])
102     ax.set_frame_on(False)
103     fig.savefig(
104         f'paper_supplements/visual/supplement_visual_factors_results_{
    latent_dim}.png',
105         bbox_inches=0, transparent=True, dpi=400)
106     plt.clf()
107     plt.close(fig)
```

4.7.21 prep_motor_long.py

```
1 import pandas as pd
2 import numpy as np
3 from pathlib import Path
4
5 task_name = 'motor'
6 # Save all the files as .npy files and in np.float16 so it's fast to
    load them
7 # using our dataloader
```

```
main_path = Path('/path/to/HCP/task_data')
subjects = list(main_path.iterdir())

sync_file = open(subjects[3] / 'MOTOR_LR/EVs/Sync.txt', 'r')
sync_val = float(sync_file.readlines()[0])

# The start and end times for each longer task
tab_df = pd.read_csv(subjects[3] / 'MOTOR_LR/tfMRI_MOTOR_LR_tab.txt',
                     delimiter="\t")
print(tab_df.loc[tab_df['Fixdot.OnsetTime'].notna(),
                 'Fixdot.OnsetTime'] / 1000 - sync_val - 12)
print(tab_df.loc[tab_df['BLANK.OnsetTime'].notna(),
                 'BLANK.OnsetTime'].values[::10] / 1000 - sync_val)

# TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
TR = 0.72
# Length response based on times
# 20 seconds based on motor response and 3 for visual addition
num_ix_relaxation = int(27.2 / TR) + 1 + 3
# Rounded up from (30.2/TR) + 3 Although the maximum time is 30.2,
# the last fixation window is shorter

subjects = list(main_path.iterdir())
print(f'--- Window size: {num_ix_relaxation} ---')
files, subject_ls, target_files = [], [], []
for subject in subjects:
    fmri_file = subject / Path(
        f'tfMRI_{task_name.upper()}_LR_Atlas_MSMAll.npy')
    target_file = subject / Path('motor_long_mask.npy')
    if fmri_file.is_file() and (subject.name != '144428'):
        # Creation of the masks
        mask = np.zeros((3, 1, 2), dtype=np.int16)
        mask[0, 0, 0] = int((71.5 - 3)/TR)
        mask[0, 0, 1] = mask[0, 0, 0] + num_ix_relaxation
        mask[1, 0, 0] = int((131.80 - 3)/TR)
        mask[1, 0, 1] = mask[1, 0, 0] + num_ix_relaxation
        mask[2, 0, 0] = int((177.1 - 3)/TR)
        mask[2, 0, 1] = mask[2, 0, 0] + num_ix_relaxation
        assert np.max(mask) < 283
        # Ensure that the window size is the same for each task and
    occurrence
        # and that the end_ix in the mask is not larger
        # than the number of frames
        np.save(target_file, mask)
        files.append(fmri_file.resolve())
        subject_ls.append(subject.name)
        target_files.append(target_file.resolve())

files = np.asarray(files)
subjects = np.asarray(subject_ls)
targets = np.asarray(target_files)
arr = np.stack((files, subjects, targets), axis=1)

df = pd.DataFrame(arr,
                  index=subject_ls, columns=['fmri', 'subjects', '
    targets'])
df.to_csv('motor_long.csv')
```

4.7.22 prep_motor.py

```
import pandas as pd
import numpy as np
from pathlib import Path

# https://www.humanconnectome.org/hcp-protocols-ya-3t-imaging
```

```
6 # These are the task start times, we look at the difference
7 # to determine how long the task needs to be
8 task_times = np.array([10.996, 26.123, 41.25,
9                        56.377, 71.504, 101.625,
10                       116.753, 131.88, 162.001, 177.128])
11 print(np.diff(task_times))
12
13 task_name = 'motor'
14 # Save all the files as .npy files and in np.float16 so it's fast to
     load them
15 # using our dataloader
16 main_path = Path('/path/to/HCP/task_data')
17 subjects = list(main_path.iterdir())
18
19 # TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
20 TR = 0.72
21 # Length response based on times
22 length_relaxation = 16
23 num_ix_relaxation = int(length_relaxation / TR) + 1
24
25 print(f'--- Window size: {num_ix_relaxation} ---')
26 files, subject_ls, target_files = [], [], []
27 for subject in subjects:
28     motor_tasks = [subject / Path(f'{task_name.upper()}_LR/EVs/{name}.
     txt')
29                    for name in ['lf', 'lh', 'rf', 'rh', 't']]
30     fmri_file = subject / Path(
31         f'tfMRI_{task_name.upper()}_LR_Atlas_MSMAll.npy')
32     target_file = subject / Path(f'{task_name}_mask.npy')
33     if fmri_file.is_file() and (subject.name != '144428'):
34         dfs_motor = [pd.read_csv(
35             path, sep="\t",
36             header=None, names=['start_time', 'length', 'extra'])
37                    for path in motor_tasks]
38         mask = np.zeros((len(motor_tasks), 2, 2), dtype=np.int16)
39         for (i, df_motor) in enumerate(dfs_motor):
40             for (j, row) in df_motor.iterrows():
41                 # Visual cue 3 seconds before start time
42                 ix_start = int((row['start_time'] - 3) / TR)
43                 ix_end = ix_start + num_ix_relaxation
44                 mask[i, j, 0] = ix_start
45                 mask[i, j, 1] = ix_end
46             assert j == 1
47         assert mask.max() < 284
48         # Ensure that the window size is the same for each task and
     occurrence
49         np.save(target_file, mask)
50         files.append(fmri_file.resolve())
51         subject_ls.append(subject.name)
52         target_files.append(target_file.resolve())
53
54 files = np.asarray(files)
55 subjects = np.asarray(subject_ls)
56 targets = np.asarray(target_files)
57 arr = np.stack((files, subjects, targets), axis=1)
58
59 df = pd.DataFrame(arr,
60                   index=subject_ls, columns=['fmri', 'subjects', '
     targets'])
61 df.to_csv('motor.csv')
62
63 # The preparation for the left hand vs left foot task
64 subjects = list(main_path.iterdir())
65 print(f'--- Window size: {num_ix_relaxation} ---')
66 files, subject_ls, target_files = [], [], []
```

```
for subject in subjects:
    motor_tasks = [
        subject / Path(f'{task_name.upper()}_LR/EVs/{name}.txt')
        for name in ['lf', 'lh']]
    fmri_file = subject / Path(
        f'tfMRI_{task_name.upper()}_LR_Atlas_MSMAll.npy')
    target_file = subject / Path('foot_mask.npy')
    if fmri_file.is_file() and (subject.name != '144428'):
        dfs_motor = [pd.read_csv(
            path, sep="\t", header=None,
            names=['start_time', 'length', 'extra'])
                    for path in motor_tasks]
        mask = np.zeros((len(motor_tasks), 2, 2), dtype=np.int16)
        for (i, df_motor) in enumerate(dfs_motor):
            for (j, row) in df_motor.iterrows():
                # Visual cue 3 seconds before start time
                ix_start = int((row['start_time'] - 3) / TR)
                ix_end = ix_start + num_ix_relaxation
                mask[i, j, 0] = ix_start
                mask[i, j, 1] = ix_end
            assert j == 1
        # Ensure that the window size is the same for each task and
    occurrence
        np.save(target_file, mask)
        files.append(fmri_file.resolve())
        subject_ls.append(subject.name)
        target_files.append(target_file.resolve())

files = np.asarray(files)
subjects = np.asarray(subject_ls)
targets = np.asarray(target_files)
arr = np.stack((files, subjects, targets), axis=1)

df = pd.DataFrame(arr,
                  index=subject_ls, columns=['fmri', 'subjects', '
    targets'])
df.to_csv('left.csv')
```

4.7.23 prep_relational_long.py

```
import pandas as pd
import numpy as np
from pathlib import Path

# Save all the files as .npy files and in np.float16 so it's fast to
    load them
# using our dataloader
task_name = 'relational'
main_path = Path('/path/to/HCP/task_data')
subjects = list(main_path.iterdir())

start_times = np.array([7.997, 26.482, 60.975, 79.447, 113.953,
    132.519])
print(np.diff(start_times))

sync_file = open(subjects[3] / 'RELATIONAL_LR/EVs/Sync.txt', 'r')
sync_val = float(sync_file.readlines()[0])

# The start and end times for each longer task
tab_df = pd.read_csv(
    subjects[3] / 'RELATIONAL_LR/tfMRI_RELATIONAL_LR_tab.txt',
    delimiter="\t")
print(tab_df.loc[tab_df['FixationBlock.OnsetTime'].notna(),
                 'FixationBlock.OnsetTime'] / 1000 - sync_val)
print(tab_df.loc[tab_df['ControlPrompt.OnsetTime'].notna(),
```

```
23                  'ControlPrompt.OnsetTime'] / 1000 - sync_val)
24 print(tab_df.loc[tab_df['RelationalPrompt.OnsetTime'].notna(),
25                  'RelationalPrompt.OnsetTime'] / 1000 - sync_val)
26
27 # TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
28 TR = 0.72
29 # Length response based on times and based on when next cue starts
30 num_ix_relaxation = int(33.8 / TR) + 1
31 # Although the maximum time is 34.51, the last fixation window is
    shorter
32
33 print(f'--- Window size: {num_ix_relaxation} ---')
34 files, subject_ls, target_files = [], [], []
35 for subject in subjects:
36     relational_tasks = [
37         subject / Path(f'{task_name.upper()}_LR/EVs/{name}.txt')
38         for name in ['match', 'relation']]
39     fmri_file = subject / Path(
40         f'tfMRI_{task_name.upper()}_LR_Atlas_MSMAll.npy')
41     target_file = subject / Path(f'{task_name}_long_mask.npy')
42     # Subjects with the wrong time size
43     removed_subjects = ['150423', '929464']
44     if fmri_file.is_file() and not (subject.name in removed_subjects):
45         mask = np.zeros((3, 1, 2), dtype=np.int16)
46         mask[0, 0, 0] = int(26.4 / TR)
47         mask[0, 0, 1] = mask[0, 0, 0] + num_ix_relaxation
48         mask[1, 0, 0] = int(79.4 / TR)
49         mask[1, 0, 1] = mask[1, 0, 0] + num_ix_relaxation
50         mask[2, 0, 0] = int(132.5 / TR)
51         mask[2, 0, 1] = mask[2, 0, 0] + num_ix_relaxation
52         assert np.all(mask[:, :, 1] < 232)
53         # Ensure that the window size is the same for each task and
    occurrence
54         np.save(target_file, mask)
55         files.append(fmri_file.resolve())
56         subject_ls.append(subject.name)
57         target_files.append(target_file.resolve())
58
59 files = np.asarray(files)
60 subjects = np.asarray(subject_ls)
61 targets = np.asarray(target_files)
62 arr = np.stack((files, subjects, targets), axis=1)
63
64 df = pd.DataFrame(arr,
65                   index=subject_ls, columns=['fmri', 'subjects', '
    targets'])
66 df.to_csv('relational_long.csv')
```

4.7.24 prep_relational.py

```
1 import pandas as pd
2 import numpy as np
3 from pathlib import Path
4
5 # Save all the files as .npy files and in np.float16 so it's fast to
    load them
6 # using our dataloader
7 task_name = 'relational'
8 main_path = Path('/path/to/HCP/task_data')
9 subjects = list(main_path.iterdir())
10
11 # These are the task start times, we look at the difference
12 # to determine how long the task needs to be
13 start_times = np.array([7.997, 26.482, 60.975, 79.447, 113.953,
    132.519])
```

```python
print(np.diff(start_times))

# TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
TR = 0.72
# Length response based on times and based on when next cue starts
length_relaxation = 19
num_ix_relaxation = int(length_relaxation / 0.72) + 1

print(f'--- Window size: {num_ix_relaxation} ---')
files, subject_ls, target_files = [], [], []
for subject in subjects:
    relational_tasks = [
        subject / Path(f'{task_name.upper()}_LR/EVs/{name}.txt')
        for name in ['match', 'relation']]
    fmri_file = subject / Path(
        f'tfMRI_{task_name.upper()}_LR_Atlas_MSMAll.npy')
    target_file = subject / Path(f'{task_name}_mask.npy')
    # Subjects with the wrong time size
    removed_subjects = ['150423', '929464']
    if fmri_file.is_file() and not (subject.name in removed_subjects):
        dfs_relational = [pd.read_csv(
            path, sep="\t",
            header=None, names=['start_time', 'length', 'extra'])
            for path in relational_tasks]
        mask = np.zeros((len(relational_tasks), 3, 2), dtype=np.int16)
        for (i, df_relational) in enumerate(dfs_relational):
            for (j, row) in df_relational.iterrows():
                ix_start = int(row['start_time'] / TR)
                ix_end = ix_start + num_ix_relaxation
                mask[i, j, 0] = ix_start
                mask[i, j, 1] = ix_end
                print(i, j, row['start_time'])
            assert j == 2
        assert np.all(mask[:, :, 1] < 232)
        # Ensure that the window size is the same for each task and occurrence
        np.save(target_file, mask)
        files.append(fmri_file.resolve())
        subject_ls.append(subject.name)
        target_files.append(target_file.resolve())

files = np.asarray(files)
subjects = np.asarray(subject_ls)
targets = np.asarray(target_files)
arr = np.stack((files, subjects, targets), axis=1)

df = pd.DataFrame(arr,
                  index=subject_ls, columns=['fmri', 'subjects', 'targets'])
df.to_csv('relational.csv')
```

4.7.25 prep_visual.py

```python
import pandas as pd
import numpy as np
import hcp_utils as hcp
from pathlib import Path

# https://www.humanconnectome.org/hcp-protocols-ya-3t-imaging
# These are the task start times, we look at the difference
# to determine how long the task needs to be
start_times = np.array([10.996, 26.123, 41.25,
                        56.377, 71.504, 101.625,
                        116.753, 131.88, 162.001, 177.128])
print(np.diff(start_times))
```

```
13
14 task_name = 'motor'
15 # Save all the files as .npy files and in np.float16 so it's fast to
      load them
16 # using our dataloader
17 main_path = Path('/path/to/HCP/task_data')
18 subjects = list(main_path.iterdir())
19
20 # TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
21 TR = 0.72
22 # Length response based on times
23 length_relaxation = 16
24 num_ix_relaxation = int(length_relaxation / TR) + 1
25
26 for subject in subjects:
27     fmri_file = subject / Path(
28         f'tfMRI_{task_name.upper()}_LR_Atlas_MSMAll.npy')
29     visual_file = subject / Path(
30         f'tfMRI_{task_name.upper()}_LR_visual.npy')
31     if fmri_file.is_file():
32         # Only select visual region
33         data = np.load(
34             fmri_file).astype(np.float32)[:, hcp.yeo7['map_all'] == 1]
35         # Save as 16-bit float
36         np.save(visual_file, data.astype(np.float16))
37
38 print(f'--- Window size: {num_ix_relaxation} ---')
39 files, subject_ls, target_files = [], [], []
40 for subject in subjects:
41     motor_tasks = [
42         subject / Path(f'{task_name.upper()}_LR/EVs/{name}.txt')
43         for name in ['lf', 'lh', 'rf', 'rh', 't']]
44     fmri_file = subject / Path(f'tfMRI_{task_name.upper()}_LR_visual.
    npy')
45     target_file = subject / Path(f'{task_name}_visual_mask.npy')
46     if fmri_file.is_file() and (subject.name != '144428'):
47         dfs_motor = [pd.read_csv(
48             path, sep="\t",
49             header=None, names=['start_time', 'length', 'extra'])
50             for path in motor_tasks]
51         mask = np.zeros((len(motor_tasks), 2, 2), dtype=np.int16)
52         for (i, df_motor) in enumerate(dfs_motor):
53             for (j, row) in df_motor.iterrows():
54                 # Visual cue 3 seconds before start time
55                 ix_start = int((row['start_time'] - 3) / TR)
56                 ix_end = ix_start + num_ix_relaxation
57                 mask[i, j, 0] = ix_start
58                 mask[i, j, 1] = ix_end
59             assert j == 1
60         assert mask.max() < 284
61         # Ensure that the window size is the same for each task and
    occurrence
62         np.save(target_file, mask)
63         files.append(fmri_file.resolve())
64         subject_ls.append(subject.name)
65         target_files.append(target_file.resolve())
66
67 files = np.asarray(files)
68 subjects = np.asarray(subject_ls)
69 targets = np.asarray(target_files)
70 arr = np.stack((files, subjects, targets), axis=1)
71
72 df = pd.DataFrame(arr,
73                   index=subject_ls, columns=['fmri', 'subjects', '
    targets'])
```

```
df.to_csv('visual.csv')
```

4.7.26 prep_wm_long.py

```
import pandas as pd
import numpy as np
from pathlib import Path


main_path = Path('/path/to/HCP/task_data')
subjects = list(main_path.iterdir())

sync_file = open(subjects[3] / 'WM_LR/EVs/Sync.txt', 'r')
sync_val = float(sync_file.readlines()[0])

tab_df = pd.read_csv(subjects[3] / 'WM_LR/tfMRI_WM_LR_tab.txt',
    delimiter="\t")
# These are the important start times (they are essentially the same
# across subjects)
print(tab_df.loc[
    tab_df['Fix15sec.OnsetTime'].notna(),
    'Fix15sec.OnsetTime'] / 1000 - sync_val - 25)
print(tab_df.loc[
    tab_df['CueTarget.OnsetTime'].notna(),
    'CueTarget.OnsetTime'] / 1000 - sync_val)
print(tab_df.loc[
    tab_df['Cue2Back.OnsetTime'].notna(),
    'Cue2Back.OnsetTime'] / 1000 - sync_val)

# important: https://www.humanconnectome.org/hcp-protocols-ya-task-
    fmri
# TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
TR = 0.72
# Length response based on times
num_ix_relaxation = int(41 / TR) + 1
# Although the maximum time is 43.3, the last fixation window is
    shorter

print(f'--- Window size: {num_ix_relaxation} ---')
files, subject_ls, target_files = [], [], []
for subject in subjects:
    # 5 minutes and 1 second in 405 frames
    # (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
    fmri_file = subject / Path('tfMRI_WM_LR_Atlas_MSMAll.npy')
    target_file = subject / Path('wm_long_mask.npy')
    # Create the mask
    if fmri_file.is_file():
        mask = np.zeros((4, 1, 2), dtype=np.int16)
        mask[0, 0, 0] = int(36.1 / TR)
        mask[0, 0, 1] = mask[0, 0, 0] + num_ix_relaxation
        mask[1, 0, 0] = int(107.5 / TR)
        mask[1, 0, 1] = mask[1, 0, 0] + num_ix_relaxation
        mask[2, 0, 0] = int(178.5 / TR)
        mask[2, 0, 1] = mask[2, 0, 0] + num_ix_relaxation
        mask[3, 0, 0] = int(250.0 / TR)
        mask[3, 0, 1] = mask[3, 0, 0] + num_ix_relaxation
        # Ensure that the window size is the same for each task and
    occurrence
        # and not larger than the number of frames
        assert mask.max() < 405
        np.save(target_file, mask)
        files.append(fmri_file.resolve())
        subject_ls.append(subject.name)
        target_files.append(target_file.resolve())

```

```
58 files = np.asarray(files)
59 subjects = np.asarray(subject_ls)
60 targets = np.asarray(target_files)
61 arr = np.stack((files, subjects, targets), axis=1)
62
63 df = pd.DataFrame(arr,
64                   index=subject_ls, columns=['fmri', 'subjects', '
    targets'])
65 df.to_csv('wm_long.csv')
```

4.7.27 prep_wm.py

```
1 import pandas as pd
2 import numpy as np
3 from pathlib import Path
4
5 # Save all the files as .npy files and in np.float16 so it's fast to
    load them
6 # using our dataloader
7 main_path = Path('/path/to/HCP/task_data')
8 subjects = list(main_path.iterdir())
9
10 # These are the task start times, we look at the difference
11 # to determine how long the task needs to be
12 start_times = np.array([7.997, 36.119, 79.208,
13                         107.503, 150.512, 178.594, 221.83, 250.045])
14 print(np.diff(start_times))
15
16 # important: https://www.humanconnectome.org/hcp-protocols-ya-task-
    fmri
17 # TR (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
18 TR = 0.72
19 # Length response based on times
20 length_relaxation = 29
21 num_ix_relaxation = int(length_relaxation / TR) + 1
22
23 print(f'--- Window size: {num_ix_relaxation} ---')
24 files, subject_ls, target_files = [], [], []
25 for subject in subjects:
26     paths_0bk = [subject / Path(f'WM_LR/EVs/0bk_{name}.txt')
27                  for name in ['faces', 'body', 'places', 'tools']]
28     paths_2bk = [subject / Path(f'WM_LR/EVs/2bk_{name}.txt')
29                  for name in ['faces', 'body', 'places', 'tools']]
30     # 5 minutes and 1 second in 405 frames
31     # (https://humanconnectome.org/hcp-protocols-ya-3t-imaging)
32     fmri_file = subject / Path('tfMRI_WM_LR_Atlas_MSMAll.npy')
33     target_file = subject / Path('wm_mask.npy')
34     if fmri_file.is_file():
35         dfs_0bk = [pd.read_csv(
36             path, sep="\t",
37             header=None, names=['start_time', 'length', 'extra'])
38                    for path in paths_0bk]
39         dfs_2bk = [pd.read_csv(
40             path, sep="\t",
41             header=None, names=['start_time', 'length', 'extra'])
42                    for path in paths_2bk]
43         mask = np.zeros((2, 4, 2), dtype=np.int16)
44         for (i, df_0bk) in enumerate(dfs_0bk):
45             for (j, row) in df_0bk.iterrows():
46                 ix_start = int(row['start_time'] / TR)
47                 ix_end = ix_start + num_ix_relaxation
48                 mask[0, i, 0] = ix_start
49                 mask[0, i, 1] = ix_end
50                 assert j <= 0
51         for (i, df_2bk) in enumerate(dfs_2bk):
```

```
            for (j, row) in df_2bk.iterrows():
                print(i, paths_2bk[i], row['start_time'])
                ix_start = int(row['start_time'] / TR)
                ix_end = ix_start + num_ix_relaxation
                mask[1, i, 0] = ix_start
                mask[1, i, 1] = ix_end
                assert j <= 0
        # Ensure that the window size is the same for each task and
    occurrence
        assert mask.max() < 405
        np.save(target_file, mask)
        files.append(fmri_file.resolve())
        subject_ls.append(subject.name)
        target_files.append(target_file.resolve())

files = np.asarray(files)
subjects = np.asarray(subject_ls)
targets = np.asarray(target_files)
arr = np.stack((files, subjects, targets), axis=1)

df = pd.DataFrame(arr,
                  index=subject_ls, columns=['fmri', 'subjects', '
    targets'])
df.to_csv('wm.csv')
```

4.7.28 preprocess_data.py

```
import sys
import numpy as np
import nibabel as nb
from pathlib import Path
from nilearn import signal

# This code preprocesses the data, and specifically
# saves the files as 16-bit floats to save disk space
# and to increase loading speed.

main_path = Path('/path/to/HCP/task_data')
subjects = list(main_path.iterdir())

print(f'Nuber of subjects: {len(subjects)}')
ix = int(sys.argv[1])

# Perform preprocessing (low and high pass filtering)
subject = subjects[ix]
motor_file = subject / Path(
    'MOTOR_LR/tfMRI_MOTOR_LR_Atlas_MSMAll.dtseries.nii')
motor_file_new = subject / Path('tfMRI_MOTOR_LR_Atlas_MSMAll.npy')
if motor_file.is_file():
    data = nb.load(motor_file).get_fdata(dtype=np.float16)
    data = signal.clean(data, detrend=True,
                        standardize='zscore', t_r=0.72,
                        low_pass=0.25, high_pass=0.008)
    np.save(motor_file_new, data.astype(np.float16))
    motor_file.unlink()
wm_file = subject / Path('WM_LR/tfMRI_WM_LR_Atlas_MSMAll.dtseries.nii'
    )
wm_file_new = subject / Path('tfMRI_WM_LR_Atlas_MSMAll.npy')
if wm_file.is_file():
    data = nb.load(wm_file).get_fdata(dtype=np.float16)
    data = signal.clean(data, detrend=True,
                        standardize='zscore', t_r=0.72,
                        low_pass=0.25, high_pass=0.008)
    np.save(wm_file_new, data.astype(np.float16))
    wm_file.unlink()
```

```
38 relational_file = subject / Path(
39     'RELATIONAL_LR/tfMRI_RELATIONAL_LR_Atlas_MSMAll.dtseries.nii')
40 relational_file_new = subject / Path('tfMRI_RELATIONAL_LR_Atlas_MSMAll
    .npy')
41 if relational_file.is_file():
42     data = nb.load(relational_file).get_fdata(dtype=np.float16)
43     data = signal.clean(data, detrend=True,
44                         standardize='zscore', t_r=0.72,
45                         low_pass=0.25, high_pass=0.008)
46     np.save(relational_file_new, data.astype(np.float16))
47     relational_file.unlink()
```

4.7.29 train.py

```
1 import torch
2 import numpy as np
3 import pandas as pd
4 from torch import nn
5 from tqdm import tqdm
6 from pathlib import Path
7 from visualization import Visualizer
8 from utils import (init_model, create_dataloaders)
9
10
11 class Trainer:
12     def __init__(self, model, optimizer, optimizer_params,
13                  criterion, device, dataset,
14                  log_dir=Path('./logs')):
15         self.model = model
16         self.optimizer = optimizer
17         self.optimizer_params = optimizer_params
18         self.criterion = criterion
19         self.device = device
20         self.dataset = dataset
21         tr_dataset = dataset('train')
22         num_tasks = tr_dataset.num_tasks
23         num_occurrences = tr_dataset.num_occurrences
24         self.checkpoint = None
25         self.previous_best = np.inf
26         self.log_dir = log_dir
27         if not self.log_dir.is_dir():
28             self.log_dir.mkdir(parents=True, exist_ok=True)
29         self.visualizer = Visualizer(self.log_dir, num_tasks,
    num_occurrences)
30         self.scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau
31         self.loss_names = [
32             'loss', 'mse', 'kl']
33
34     def train_step(self, model, optimizer, batch):
35         device = torch.device('cuda' if torch.cuda.is_available() else
     'cpu')
36         optimizer.zero_grad()
37         # Depending on whether we use DALI dataloader
38         # or pre-loaded dataset, we need to handle the
39         # batch differently
40         if isinstance(batch[0], torch.Tensor):
41             x = batch[0]
42             x = x.to(device, non_blocking=True).float()
43             mask = batch[1]
44             mask = mask.to(device, non_blocking=True).long()
45         else:
46             x = batch[0]['fmri'].float()
47             mask = batch[0]['mask'].long()
48         output = model(x, mask)
49         loss, losses = self.criterion(output, x)
```

```
50          log = {'train-loss': loss.detach()}
51          for (loss_key, loss_value) in losses.items():
52              log[f'train-{loss_key}'] = loss_value
53          nn.utils.clip_grad_norm_(model.parameters(), max_norm=50)
54          loss.backward()
55          optimizer.step()
56          return loss.detach(), log
57 
58      def valid_step(self, model, optimizer, batch):
59          device = torch.device('cuda' if torch.cuda.is_available() else
     'cpu')
60          with torch.no_grad():
61              # Depending on whether we use DALI dataloader
62              # or pre-loaded dataset, we need to handle the
63              # batch differently
64              if isinstance(batch[0], torch.Tensor):
65                  x = batch[0]
66                  x = x.to(device, non_blocking=True).float()
67                  mask = batch[1]
68                  mask = mask.to(device, non_blocking=True).long()
69              else:
70                  x = batch[0]['fmri'].float()
71                  mask = batch[0]['mask'].long()
72              output = model(x, mask, validation=True)
73              loss, losses = self.criterion(output, x, validation=True)
74              log = {'valid-loss': loss.detach()}
75              for (loss_key, loss_value) in losses.items():
76                  log[f'valid-{loss_key}'] = loss_value
77          return loss, log
78 
79      def visualize_batch(self, model, batch):
80          device = torch.device('cuda' if torch.cuda.is_available() else
     'cpu')
81          with torch.no_grad():
82              # Depending on whether we use DALI dataloader
83              # or pre-loaded dataset, we need to handle the
84              # batch differently
85              if isinstance(batch[0], torch.Tensor):
86                  x = batch[0]
87                  x = x.to(device, non_blocking=True).float()
88                  mask = batch[1]
89                  mask = mask.to(device, non_blocking=True).long()
90              else:
91                  x = batch[0]['fmri'].float()
92                  mask = batch[0]['mask'].long()
93              output = model(x, mask, validation=True)
94              if output['x'] is not None:
95                  self.visualizer.recon(output, x)
96                  if output['h_0'] is not None:
97                      self.visualizer.visualize_initial_conds(output)
98                      self.visualizer.visualize_factors(output)
99 
100     def epoch_step(self, model, epoch, optimizer,
101                    loaders, loader_type: str, scheduler,
102                    df):
103         loader = loaders[loader_type]
104         # get function
105         step = getattr(self, f'{loader_type}_step')
106         loss = 0.0
107         # TODO: improve this
108         dict_loss = {f'{loader_type}-{loss_name}':
109                        0.0 for loss_name in self.loss_names}
110         for (_, batch) in enumerate(tqdm(loader)):
111             step_loss, log_loss = step(model, optimizer, batch)
112             loss += step_loss
```

```
113             for (key, val) in log_loss.items():
114                 dict_loss[key] += (float(val) / len(loader))
115         df.loc[epoch, self.loss_names] = dict_loss.values()
116         print(f'{loader_type.capitalize()} Epoch: {epoch}')
117         for (key, val) in dict_loss.items():
118             print(f'{key.capitalize()}: {val}')
119         if loader_type == 'valid':
120             self.visualize_batch(model, batch)
121             scheduler.step(dict_loss['valid-loss'])
122         # Only save the best model
123         if loader_type == 'valid' and loss <= self.previous_best:
124             self.checkpoint = model.state_dict()
125             self.previous_best = loss
126
127     def train(self, config, epochs, batch_size):
128         train_df = pd.DataFrame(
129             np.zeros((epochs, len(self.loss_names))),
130             index=list(range(epochs)), columns=self.loss_names)
131         valid_df = pd.DataFrame(
132             np.zeros((epochs, len(self.loss_names))),
133             index=list(range(epochs)), columns=self.loss_names)
134         model = init_model(self.model, config).to(self.device)
135         optimizer = self.optimizer(model.parameters(),
136                                    **self.optimizer_params)
137         scheduler = self.scheduler(
138             optimizer, factor=0.95, patience=10, min_lr=1E-5)
139         print(f'Training with config: {config}')
140         g = torch.Generator()
141         g.manual_seed(config['seed'])
142         (train_loader, valid_loader, _), _ \
143             = create_dataloaders(self.dataset, config)
144         print(
145             'Number of model parameters: '
146             f'{sum(p.numel() for p in model.parameters() if p.
    requires_grad)}')
147
148         loaders = {'train': train_loader,
149                    'valid': valid_loader}
150
151         # Create fold logs in case of fold experiment
152         if 'fold' in config.keys():
153             self.log_dir = self.log_dir / f'fold_{config["fold"]}'
154             self.log_dir.mkdir(parents=True, exist_ok=True)
155
156         for epoch in range(epochs):
157             model.train()
158             self.epoch_step(
159                 model, epoch, optimizer,
160                 loaders, 'train', scheduler, train_df)
161             train_df.to_csv(self.log_dir / 'train.csv')
162             model.eval()
163             with torch.no_grad():
164                 self.epoch_step(
165                     model, epoch, optimizer,
166                     loaders, 'valid', scheduler, valid_df)
167             valid_df.to_csv(self.log_dir / 'valid.csv')
168             if valid_df.loc[epoch, 'loss'] <= valid_df.loc[
169                     :epoch, 'loss'].min():
170                 checkpoint_file = self.log_dir / Path('model.pt')
171                 torch.save(self.checkpoint, checkpoint_file)
172                 best_epoch = epoch
173             # Early stopping
174             if epoch - best_epoch > 50 and epoch > 50:
175                 break
```

4.7.30 utils.py

```
import re
import yaml
import torch
import random
import importlib
import numpy as np
import pandas as pd
import nvidia.dali.fn as fn
import nvidia.dali.types as types
from copy import copy
from tqdm import tqdm
from nvidia import dali
from pathlib import Path
from torch.utils.data import DataLoader, TensorDataset
from nvidia.dali.plugin.pytorch import DALIGenericIterator


def init_model(model_type, config):
    # input_size, hidden_sizes, output_size, activation, normalization
    , dropout
    if config['normalization'] == 'None':
        config['normalization'] = None
    if config['activation'] == 'None':
        config['activation'] = None
    encoder_args = (config['input_size'], config['hidden_sizes'],
                    config['latent_dim'], config['activation'],
                    config['normalization'], config['dropout'])
    decoder_args = (config['latent_dim'], config['hidden_sizes'
    ][::-1],
                    config['input_size'], config['activation'],
                    config['normalization'], config['dropout'])
    # encoder_type, decoder_type, encoder_args, decoder_args
    model = model_type(
        encoder_type=config['encoder_type'],
        decoder_type=config['decoder_type'],
        encoder_args=encoder_args,
        decoder_args=decoder_args,
        temporal_hidden_sizes=config['temporal_hidden_sizes'])
    return model


def get_default_config_baseline(args):
    if len(args) > 1:
        with Path(f'baseline_parameters/hyperparam_{int(args[1])}.yaml
    ').open('r') as f:
            default_conf = yaml.safe_load(f)
    else:
        with Path('baseline_parameters/default.yaml').open('r') as f:
            default_conf = yaml.safe_load(f)
    if len(args) > 2:
        default_conf['gpu'] = args[2]
    return dict(default_conf)


def get_default_config(args):
    if len(args) > 1:
        with Path(f'hyperparameters/hyperparam_{int(args[1])}.yaml').
    open('r') as f:
            default_conf = yaml.safe_load(f)
    else:
        with Path('hyperparameters/default.yaml').open('r') as f:
            default_conf = yaml.safe_load(f)
    if len(args) > 2:
```

```
60          default_conf['gpu'] = args[2]
61      return dict(default_conf)
62
63
64  def seed_worker(worker_id):
65      worker_seed = torch.initial_seed() % 2**32
66      np.random.seed(worker_seed)
67      random.seed(worker_seed)
68
69
70  def load_model_from_config(config):
71      model_module = importlib.import_module('model')
72      model_path = get_log_string(config)
73      model_type = getattr(model_module, config['model'])
74      model = init_model(model_type, config)
75      model_state_dict = torch.load(model_path / 'model.pt',
        map_location='cpu')
76      model.load_state_dict(model_state_dict)
77      model.eval()
78      return model
79
80
81  @dali.pipeline_def(batch_size=4, num_threads=5,
82                     device_id=0, set_affinity=True, seed=42)
83  # The seed remains unused because we do not shuffle
84  def dali_pipeline(fmri_paths, target_paths, input_size, num_timesteps)
        :
85      fmri = fn.readers.numpy(
86          bytes_per_sample_hint=input_size * num_timesteps * 2,
87          files=fmri_paths, device='cpu',
88          prefetch_queue_depth=16, read_ahead=True,
89          tensor_init_bytes=input_size * num_timesteps * 2,
90          name='fmri', random_shuffle=False,
91          cache_header_information=True, preserve=True).gpu()
92      target = fn.readers.numpy(
93          bytes_per_sample_hint=16 * 2, files=target_paths, device='cpu'
        ,
94          prefetch_queue_depth=16, read_ahead=True, tensor_init_bytes=16
         * 2,
95          name='mask', random_shuffle=False).gpu()
96      fmri = fn.cast(fmri, dtype=types.FLOAT)
97      return fmri, target
98
99
100 def create_dataloaders(dataset, config):
101     print(config)
102     if 'fold' in config.keys():
103         tr_dataset = dataset('train', fold=config['fold'])
104         va_dataset = dataset('valid', fold=config['fold'])
105         te_dataset = dataset('test', fold=config['fold'])
106     else:
107         tr_dataset = dataset('train')
108         va_dataset = dataset('valid')
109         te_dataset = dataset('test')
110     # Our A40 has >128GB memory, so we can preload
111     # the data in memory
112     if config['gpu'] == 'A40':
113         print('Loading training set into memory')
114         train_df = tr_dataset.df.copy()
115         x_tr = []
116         y_tr = []
117         for (_, row) in tqdm(train_df.iterrows()):
118             fmri = torch.from_numpy(np.load(row['fmri'])).half()
119             mask = torch.from_numpy(np.load(row['targets'])).half()
120             x_tr.append(fmri)
```

```
            y_tr.append(mask)
        x_train = torch.stack(x_tr, dim=0)
        y_train = torch.stack(y_tr, dim=0)
        del x_tr
        del y_tr
        train_dl = DataLoader(TensorDataset(x_train, y_train),
                              batch_size=config['batch_size'],
                              pin_memory=True,
                              shuffle=False,
                              num_workers=5,
                              prefetch_factor=4,
                              persistent_workers=True)
        print('Loading validation set into memory')
        valid_df = va_dataset.df.copy()
        x_va = []
        y_va = []
        for (_, row) in tqdm(valid_df.iterrows()):
            fmri = torch.from_numpy(np.load(row['fmri'])).half()
            mask = torch.from_numpy(np.load(row['targets'])).half()
            x_va.append(fmri)
            y_va.append(mask)
        x_valid = torch.stack(x_va, dim=0)
        y_valid = torch.stack(y_va, dim=0)
        del x_va
        del y_va
        valid_dl = DataLoader(TensorDataset(x_valid, y_valid),
                              batch_size=config['batch_size'],
                              pin_memory=True,
                              shuffle=False,
                              num_workers=5,
                              prefetch_factor=4,
                              persistent_workers=True)
        print('Loading test set into memory')
        test_df = te_dataset.df.copy()
        x_te = []
        y_te = []
        for (_, row) in tqdm(test_df.iterrows()):
            fmri = torch.from_numpy(np.load(row['fmri'])).half()
            mask = torch.from_numpy(np.load(row['targets'])).half()
            x_te.append(fmri)
            y_te.append(mask)
        x_test = torch.stack(x_te, dim=0)
        y_test = torch.stack(y_te, dim=0)
        del x_te
        del y_te
        test_dl = DataLoader(TensorDataset(x_test, y_test),
                             batch_size=config['batch_size'],
                             pin_memory=True,
                             shuffle=False,
                             num_workers=5,
                             prefetch_factor=4,
                             persistent_workers=True)
    # If not A40 GPU, then use DALI dataloaders
    else:
        tr_fmri_files, tr_target_files = tr_dataset.paths
        va_fmri_files, va_target_files = va_dataset.paths
        te_fmri_files, te_target_files = te_dataset.paths
        train_pipe = dali_pipeline(
            batch_size=config['batch_size'],
            seed=config['seed'],
            fmri_paths=tr_fmri_files,
            target_paths=tr_target_files,
            input_size=config['input_size'],
            num_timesteps=config['num_timesteps'])
        valid_pipe = dali_pipeline(
```

```
186             batch_size=config['batch_size'],
187             seed=config['seed'],
188             fmri_paths=va_fmri_files,
189             target_paths=va_target_files,
190             input_size=config['input_size'],
191             num_timesteps=config['num_timesteps'])
192         test_pipe = dali_pipeline(
193             batch_size=config['batch_size'],
194             seed=config['seed'],
195             fmri_paths=te_fmri_files,
196             target_paths=te_target_files,
197             input_size=config['input_size'],
198             num_timesteps=config['num_timesteps'])
199         train_pipe.build()
200         valid_pipe.build()
201         test_pipe.build()
202         train_dl = DALIGenericIterator(
203             train_pipe,
204             output_map=['fmri', 'mask'], reader_name="fmri",
205             auto_reset=True, prepare_first_batch=True)
206         valid_dl = DALIGenericIterator(
207             valid_pipe,
208             output_map=['fmri', 'mask'], reader_name="fmri",
209             auto_reset=True, prepare_first_batch=True)
210         test_dl = DALIGenericIterator(
211             test_pipe,
212             output_map=['fmri', 'mask'], reader_name="fmri",
213             auto_reset=True, prepare_first_batch=True)
214     return (train_dl, valid_dl, test_dl), (tr_dataset, va_dataset,
    te_dataset)
215
216
217 def load_df_from_config(config, data_type):
218     model_path = get_log_string(config)
219     df_path = model_path / f'{data_type}.csv'
220     df = pd.read_csv(df_path, index_col=0)
221     # Remove rows after stopping training
222     df = df.loc[df['loss'] != 0.0, :]
223     return df
224
225
226 # Mask the sub-blocks out based on the mask
227 # created in the prep_*.py files for each
228 # dataset
229 def mask_input(x, mask):
230     voxels = x.size(-1)
231     batch, num_tasks, num_occurrences, _ = mask.size()
232     xs = []
233     for b in range(batch):
234         x_tasks = []
235         for i in range(num_tasks):
236             x_occs = []
237             for j in range(num_occurrences):
238                 window = x[b, mask[b, i, j, 0]:mask[b, i, j, 1]].clone
    ()
239                 x_occs.append(window)
240             x_tasks.append(torch.stack(x_occs, dim=1))
241         xs.append(torch.stack(x_tasks, dim=1))
242     x = torch.stack(xs, dim=1)
243     window_size = x.size(0)
244     x = x.view(window_size, batch * num_tasks * num_occurrences,
    voxels)
245     return x
246
247
```

```
248 # Create a config based on the path name
249 def path_to_config(base_config: dict, p: Path):
250     config = copy(base_config)
251     model_name = str(p.name)
252     try:
253         config['model'] = str(re.search('(.*?)-ds', model_name).group
    (1))
254         config['dataset'] = str(re.search('-ds(.*?)-s', model_name).
    group(1))
255         config['seed'] = int(re.search('-s(.*?)-spar', model_name).
    group(1))
256         config['beta'] = float(
257             re.search('-spar(.*?)-enc', model_name).group(1))
258         config['encoder_type'] = str(
259             re.search('-enc(.*?)-dec', model_name).group(1))
260         config['decoder_type'] = str(
261             re.search('-dec(.*?)-ep', model_name).group(1))
262         config['epochs'] = int(re.search('-ep(.*?)-lr', model_name).
    group(1))
263         config['learning_rate'] = float(
264             re.search('-lr(.*?)-hs', model_name).group(1))
265         config['hidden_sizes'] = re.search(
266             '-hs(.*?)-do', model_name).group(1).strip('[]').split(',')
267         if config['hidden_sizes'] == ['']:
268             config['hidden_sizes'] = []
269         else:
270             config['hidden_sizes'] = list(map(int, config['
    hidden_sizes']))
271         config['dropout'] = float(
272             re.search('-do(.*?)-ac', model_name).group(1))
273         config['activation'] = str(
274             re.search('-ac(.*?)-no', model_name).group(1))
275         config['normalization'] = str(
276             re.search('-no(.*?)-ld', model_name).group(1))
277         config['latent_dim'] = int(
278             re.search('-ld(.*?)-th', model_name).group(1))
279         config['temporal_hidden_sizes'] = re.search(
280             '-th(.*?)-', model_name).group(1).strip('[]').split(',')
281         config['temporal_hidden_sizes'] = list(
282             map(int, config['temporal_hidden_sizes']))
283     except Exception as e:
284         print(f'Wrong model name string!!!, exception: {e}')
285     return config
286 
287 
288 # Obtain all logs corresponding to preset conditions
289 def purge_logs(base_config: dict, long=False):
290     log_dir = Path('./logs')
291     configs = []
292     for model_log in log_dir.iterdir():
293         if (model_log / 'valid.csv').is_file():
294             configs.append(path_to_config(base_config, model_log))
295     return configs
296 
297 
298 def subset_configs(experiment_config, config_list):
299     # The experiment config will have a key corresponding
300     # to the key in the config and a list of possible values
301     subset_list = []
302     for config in config_list:
303         flag = 0
304         for (key, values) in experiment_config.items():
305             if config[key] not in values:
306                 flag += 1
307         if not flag:
```

```
308             subset_list.append(config)
309     return subset_list
310 
311 
312 def embed_data(loader, model, num_subjects=np.inf):
313     inits = []
314     factors = []
315     start_ix = 0
316     device = torch.device('cuda' if torch.cuda.is_available() else '
    cpu')
317     for (i, batch) in enumerate(loader):
318         with torch.no_grad():
319             if isinstance(batch[0], torch.Tensor):
320                 x = batch[0]
321                 x = x.to(device, non_blocking=True).float()
322                 mask = batch[1]
323                 mask = mask.to(device, non_blocking=True).long()
324             else:
325                 x = batch[0]['fmri'].float()
326                 mask = batch[0]['mask'].long()
327             batch_size = x.size(0)
328             end_ix = start_ix + batch_size
329             model_output = model(x, mask, validation=True)
330             inits.append(model_output['h_0'])
331             factors.append(model_output['factors'])
332             start_ix = end_ix
333         if i >= num_subjects:
334             break
335     if model_output['h_0'] is not None:
336         inits = torch.stack(inits, dim=0)
337     else:
338         inits = None
339     factors = torch.stack(factors, dim=0)
340     return inits, factors
341 
342 
343 def get_log_string(config):
344     return Path(f'./logs/{config["model"]}-'
345                 f'ds{config["dataset"]}-'
346                 f's{config["seed"]}-'
347                 f'spar{config["beta"]}-'
348                 f'enc{config["encoder_type"]}-'
349                 f'dec{config["decoder_type"]}-'
350                 f'ep{config["epochs"]}-'
351                 f'lr{config["learning_rate"]}-'
352                 f'hs{config["hidden_sizes"]}-'
353                 f'do{config["dropout"]}-'
354                 f'ac{config["activation"]}-'
355                 f'no{config["normalization"]}-'
356                 f'ld{config["latent_dim"]}-'
357                 f'th{config["temporal_hidden_sizes"]}-')
358 
359 
360 def reconstruct_factors(config, factors):
361     device = torch.device('cuda' if torch.cuda.is_available() else '
    cpu')
362     factors = factors.to(device)
363     model = load_model_from_config(config)
364     model = model.to(device)
365     with torch.no_grad():
366         reconstruction = model.reconstruct(factors)
367     return reconstruction.cpu()
```

4.7.31 visualization.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
matplotlib.use("Agg")


class Visualizer:
    def __init__(self, log_dir, num_tasks, num_occurrences):
        self.num_tasks = num_tasks
        self.num_occurrences = num_occurrences
        self.log_dir = log_dir

    # Reconstruction during training
    def recon(self, model_output, x):
        x = model_output['x'].detach().mean(-1).cpu()
        x_hat = model_output['x_hat'].detach().mean(-1).cpu()
        x = x.view(x.size(0), -1, self.num_tasks, self.num_occurrences
    )
        x_hat = x_hat.view(
            x_hat.size(0), -1, self.num_tasks, self.num_occurrences)
        fig, axs = plt.subplots(
            self.num_tasks * self.num_occurrences, x.size(1),
            figsize=(x_hat.size(1) * 2,
                     self.num_occurrences * self.num_tasks * 2))
        for i in range(x.size(1)):
            for k in range(self.num_tasks):
                for j in range(self.num_occurrences):
                    axs[j + k*self.num_occurrences, i].plot(
                        x[:, i, k, j], c='b', alpha=0.7)
                    axs[j + k*self.num_occurrences, i].plot(
                        x_hat[:, i, k, j], c='r', alpha=0.8)
                axs[k * self.num_occurrences, i].set_title(f'Task: {k}
    ')
        plt.tight_layout()
        plt.savefig(self.log_dir / 'reconstructions.png')
        plt.clf()
        plt.close(fig)

    # Visualizing the initial conditions during training
    def visualize_initial_conds(self, model_output):
        z = model_output['h_0'].detach().cpu()
        pca = PCA(n_components=2)
        z_pca = pca.fit_transform(z)
        z_pca = np.reshape(
            z_pca, (-1, self.num_tasks, self.num_occurrences, 2))
        fig, axs = plt.subplots(1, 1, figsize=(15, 15))
        colors = plt.get_cmap('tab10')
        for i in range(z_pca.shape[0]):
            axs.scatter(
                z_pca[i, 0, :, 0],
                z_pca[i, 0, :, 1], alpha=0.8, color=colors(i))
            axs.scatter(
                z_pca[i, 1, :, 0],
                z_pca[i, 1, :, 1], alpha=0.8, marker='^', color=colors
    (i))
        plt.savefig(self.log_dir / 'zs.png')
        plt.clf()
        plt.close(fig)

    # Visualizing the factors during training
    def visualize_factors(self, model_output):
        factors = model_output['factors'].detach().cpu()
        if factors.shape[-1] > 2:
            pca = PCA(n_components=2)
```

```
63              num_timesteps, occ_tasks, latent_dim = factors.shape
64              factors = np.reshape(
65                  factors, (num_timesteps * occ_tasks, latent_dim))
66              factors = pca.fit_transform(factors)
67              factors = np.reshape(factors, (num_timesteps, occ_tasks,
    2))
68          factors = np.reshape(
69              factors,
70              (factors.shape[0], -1, self.num_tasks, self.
    num_occurrences, 2))
71          fig, axs = plt.subplots(1, 1, figsize=(15, 15))
72          colors = plt.get_cmap('tab10')
73          for i in range(factors.shape[1]):
74              axs.plot(
75                  factors[:, i, 0, 0, 0],
76                  factors[:, i, 0, 0, 1], alpha=0.8, color=colors(i))
77              axs.plot(
78                  factors[:, i, 1, 0, 0],
79                  factors[:, i, 1, 0, 1],
80                  alpha=0.8, linestyle='dashed', color=colors(i))
81          plt.savefig(self.log_dir / 'factors.png')
82          plt.clf()
83          plt.close(fig)
```