Expanding the Capabilities of Voxelwise Modeling for Naturalistic Brain Decoding

Ryan Ong

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-5
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-5.html

January 15, 2023

Expanding the Capabilities of Voxelwise Modeling for Naturalistic Brain Decoding

by Ryan Bryce Ong

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Shankar Sastry
Research Advisor

01/13/2023

(Date)

* * * * * * *

Professor Allen Yang
Second Reader

1-13-2023

(Date)

Expanding the Capabilities of Voxelwise Modeling for Naturalistic Brain Decoding

by

Ryan Bryce Ong

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Shankar Sastry, Chair
Professor Allen Yang
Professor Jack Gallant

Fall 2022

Abstract

Expanding the Capabilities of Voxelwise Modeling for Naturalistic Brain Decoding

by

Ryan Bryce Ong

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Shankar Sastry, Chair

Autonomous navigation systems have yet to reach the advanced capabilities of the human brain. Further understanding of the brain, through neuroscience, has the potential to facilitate the improvement of modern artificial intelligence systems. Neuroscience is an emergent field that takes a principled approach for investigating the brain. This thesis weaves three projects together into a narrative that follows the pipeline of the voxelwise modeling framework. Voxelwise modeling [63, 80, 112] is a modern, data science-inspired approach for fMRI brain encoding (and, consequently, decoding in the reverse direction) within naturalistic environments. We detail our efforts to customize a driving simulator, CARLA (Unreal Engine 4), for brain decoding/encoding stimuli. Next, we propose and verify a pipeline for non-linearly transforming stimuli into semantic features. Finally, we explore fitting voxelwise encoding models, with multiple feature spaces, to find cortical representations of timescale selectivity.

Contents

List of Figures

List of Tables

Special Thanks

To Jack Gallant:
For graciously guiding me into the Gallant Lab
And every step along the way.
For willingly taking me on.
For being honest and genuine.
For going far above & beyond anything I deserved.
For fostering and facilitating an environment
Where I could learn and grow so, so much.
For gifting sage advice,
Not only for academia, but also for life.
Without your guidance,
I would not have immersed in Neuroscience,
So deeply and thoroughly.
Jack, was the experiment a success?
No matter what,
I am eternally grateful.

To Allen Yang:
For being so understanding
And illuminating my path to Neuroscience.
For leading me to the Gallant Lab.
For making my ventures into graduate research and my Masters
Even possible in the first place!
For lending me an ear,
When I was hopelessly confused.
For being so patient and kindhearted,
And empowering my curiosity.
For entrusting me invaluable opportunities to widen my perspective
And have my heart touched by high-schoolers,
All while gaining a deeper appreciation for AI.
I am indebted to you.

To Tianjiao Zhang,
The Maestro:
For knowing when to guide
And when I needed to learn for myself.
My eyes no longer glaze over,
Whenever I encounter unfamiliar math.
It was thanks to you,
That I gained the courage

To dive in and grasp new concepts!
Never before have I met such a hardworking and persevering mentor.
Thank you for telling me like it is.
Your breadth and achievements are incredible.
I hope I can
Live up to your standards
One day.

To Cheol Jun Cho:
For always putting up with my naive questions,
Indulging in my curiosity,
Patiently guiding me deeper
Into the math of Deep Learning,
And making my autoencoder work possible.
Without your irreplaceable guidance,
I do not know
How I would have gotten through
This thesis.
I hope we can collaborate again
One day.
I am rooting for you!

To Dan Garcia:
For instilling inspiration,
Passion,
And courage in me.
For helping me realize
There is so much more
That is worth learning.

To Michele Winter:
For your incredible patience,
And graciously supporting me
During numerous moments of struggles and embarrassment.

To Cathy Chen & Tom Dupre La Tour:
For guiding me through the world of signal processing
And laying the crucial foundation
That made my timescale research possible.
Cathy, you were right.
Tom, thank you for the ginger ale.
I really hope I can share my animation with you, one day!

To Alicia Zeng, Amanda LeBel, Christine Tseng,
Emily Meschke, Jen Holmberg, Lily Gong,
& Matteo Visconti di Oleggio Castello of the Gallant Lab:
For welcoming me with open arms,
Treating me like your own kin,
And making this an unforgettable experience.
You are all so incredibly talented.
I hope to catch a glimpse of your future endeavors!

To Henry Sun, Jacob Holesinger, Natalie Khamphanh,
& Soohyun Cho:
For gifting me the precious moments
That led me to the world of Neuroscience.

To Karvin, Sally, Valentine, & Vienna Li:
For being so incredibly kind
And warming this worn, disillusioned soul.
For being the source of my conviction
To rally and make the final push
To bring this thesis
To a close.

To the Cindy, Gene, & Kaelyn Ong:
For being the eternal lights in my life.
For being my sword and shield.
I love you all.

To Jack Wu, Art Ong, & Mee Oy Ong:
I miss you all.

Chapter 1

Towards an Enhanced CARLA for Human Navigation Experiments

1.1 Introduction

Neuroscience experiments often use artificial stimuli that are simplified representations of natural systems. The decreased complexity of such results lend to more transparent interpretation. The brain is nonlinear, such that states from a particular stimulus do not necessarily generalize well. Experiments have shown that brain activity is effectively remapped when subjects engaged with a simplified or non-naturalistic experience [18, 21, 74, 78, 112], such as a passive presentation of an environment that they had previously actively navigated [14, 99]. Conversely, experiments have demonstrated that experiments, with naturalistic stimuli, produce results that higher reliability and repeatability [6, 40, 42–45, 54, 75, 111, 113].

To decode human navigation, the stimuli must involve navigation scenarios that are as naturalistic as possible. To date, there has been no other framework that allows for the rich decoding of navigation specifically for the human brain.

Modern statistical techniques allow for the analysis and interpretation of larger sizes and dimensions of data. This affords the usage of complex, naturalistic stimuli. Functional magnetic resonance imaging (fMRI) achieves the best spatial resolution, out of all non-invasive imaging techniques, by measuring hemodynamic activity throughout much of the brain [4]. However, subjects must be stationary within a narrow tube. Therefore, naturalistic stimuli is usually compressed

and transmitted through a device. Constructing virtual realities (VR), as a stimuli, is a promising approach that affords extraction of rich and accurate features. Modern physically-based rendering and scenario scripting can approach the complexity of physical reality. Experiments have created custom virtual environments from scratch, at the cost of time and degree of complexity. As they are project-specific, it is also difficult to continue adding or sharing features [8, 98].

Our approach is to modify CARLA [24], an Unreal Engine-based project for autonomous vehicle simulations, for fMRI experiments. CARLA contains much of the necessary nuts-and-bolts needed for a driving simulator, including traffic scenarios and multi-lane driving. At a high-level, this fork adds logic for the subjects' tasks and classes that extract a multitude of feature spaces while a subject is driving. This custom code extends directly from CARLA's C++ source code in order to maximize performance.

1.2 Background

CARLA [24] is a driving simulator optimized for training self-driving AI programs. However, CARLA provides a rich, dynamic driving environment that can also be used for human driving experiments. To accomplish this, however, we need to modify the simulator for several human considerations:

1. The system must be as performant as possible, to minimize the perceptual speed difference from real-life driving scenarios.

2. The CARLA simulator must be intuitive and seamless to open and start up, preferably with a human-facing single executable that starts up the entire system.

3. The world must be varied and sufficiently large such that the human is immersed in the world and is unlikely to find the boundaries and limitations of the virtual world. The traffic system should be complex, with multi-lane merging and intersection negotiation capabilities, such that the traffic dynamics are reminiscent of the physical world and, from which, rich interactions can emerge.

4. The UI for configuring the virtual world must be intuitive and flexible enough for humans, of a wide range of technical proficiency, to be able to operate.

Ultimately, our contribution will be to optimize the performance of CARLA 0.9.13, opening the path to future research for human navigation in complex, naturalistic environments.

1.3 Related Works

Other experiments use readily-available video games as stimuli and environments, as modern games offer a rich array of features and interactivity. Counter Strike Source was used as a platform for recording the brain activity of subjects in a competitive and highly interactive environment [114]. The idea of a real-life simulator is appealing, as it supports a wide-range of experiments and tasks. To that end, systems have combined Grand Theft Auto V with deep learning to generate near photo-realistic graphics. This has the potential to generate high-fidelity brain activity from human subjects [93]. However, the aforementioned games use proprietary engines, oftentimes making it extremely difficult to modify the game's environment and rules.

1.4 The Unreal Engine 4 Framework

Both versions of CARLA are built upon Unreal Engine 4 [34], a comprehensive 3D game engine.

Upon initialization, Unreal Engine loads a modifiable framework of several classes. Anything that is present within an environment is classified as, or is based on, an *actor*. *Actors* can spin off their capabilities and functionality into modular *components*. The environments can be considered a *level*, several of which are contained within a persistent *world*. This collectively defines the concept of an Unreal Engine *map*.

$$World \rightarrow GameMode/GameState/Level \rightarrow Actors \rightarrow Components$$

The rules and mechanics of the game are defined by the **GameMode** class, which is also a type of *actor*. Information about a particular game instance is stored in the **GameState**, used primarily for networking purposes.

Physically moving actors are usually classified as *pawns*. The player's inputs are routed through a *controller* that can possess and manipulate one *pawn* at a time. Information about the player is stored in the **PlayerState**, used primarily for networking purposes.

Ticking

Temporal game logic, including actor movement, is updated via a class's `tick()` function. Each tick essentially corresponds to a rendered frame. The time difference between each frame, or delta, can determine the extent of an update. This is also known as 'variable timestep', as different frames will take different amounts of time to compute and render. For example, a longer delta results in a thrown ball moving to a farther position in the proceeding frame.

Physics computations are typically subsampled, meaning that each tick involves several updates to an entity's physics state. This alleviates inaccuracies caused by long delta times. Physics is also known to be computationally intensive and a culprit for performance bottlenecks.

Replication and Recording Replay

Actors and components can be marked to 'replicate'. This configures Unreal Engine's built-in networking features to synchronize the changes of these actors and components across all connected game instances.

The record replay feature utilizes this replication synchronization by storing the optimized 'changes' of replicated actors and components. The 'changes' can then be applied to a newly created environment, containing the same actors and components, for replay. Replay affords the extraction of large amounts of data about the state of the world while a human is navigating, without hindering a human's experience of actually driving in the simulator.

1.5 Upgrading from CARLA 0.8.4

A variant of CARLA for fMRI experiments exists (Zhang, 2021) [114]. We use this as a reference for our CARLA derivative. However, the CARLA for fMRI variant is based on CARLA 0.8.4. CARLA 0.8.4 has far less dependence on Python, with most logic implemented within the Unreal Engine project itself. There is no separate *LibCarla* system, and traffic negotiation is handled by an Unreal Engine class. The **AIVehicles** have built-in PID controllers, and there is an explicit class for Player controllers that accepts inputs, such as keyboard or steering wheels. To combat the physics bottleneck, the fMRI variant destroys traffic vehicles that are too far away from the player's vehicle and spawns a set number of nearby vehicles. The CARLA for fMRI variant also adds a custom GUI, allowing the player to intuitively modify much of CARLA's code configurations, such as the number of nearby

traffic vehicles. As the entire system is largely self-contained within Unreal Engine, it is far more feasible to package the simulator into a single package for distribution. This is in contrast to the new version of vanilla CARLA, which depends on several separately-running packages to run simulations.

We want to eventually combine the new features and sophisticated traffic scenarios, from the latest CARLA, with the functionality of the legacy CARLA for fMRI variant. In doing so, the AI agent capabilities can even further approach a naturalistic environment. This should enable real-time multi-agent experiments that can integrate with CARLA autonomous programs built by other researchers.

Our hope is that, once the upgrade is completed, the newly modified CARLA 0.9.13 will be nearly as performant as the CARLA for fMRI variant. In doing so, it will open the path to replicating the feature extraction capabilities from the CARLA for fMRI variant. These include states such as the current distance to the destination and road graph to reflect topologic (relative) route progression. It will support rudimentary objectives via trigger boxes, and allow for human control of any vehicle. Additionally, environments will now have multi-lane roads and capabilities for humans and autonomous agents to simultaneously connect to the same session.

1.6 Major Differences in CARLA 0.9.13

The following are crucial concepts in the latest version of CARLA, that did not exist in the 0.8.4 version.

LibCarla

The LibCarla client library exists at the top-level directory within the `LibCarla/` folder, while the server library exists within the Unreal Engine project dependencies directory
(`Unreal/CarlaUE4/Content/Include/CarlaDependencies`).
Both versions are linked with external Boost libraries. The server library is added as an Unreal Engine external dependency via its **UnrealBuildTool**, defined by the `CarlaUE4.build.cs` file. We attempted to add the client library as an Unreal Engine external dependency, however the classes' numerous namespace definitions conflicted with Unreal Engine's unique restrictions on namespaces.

CARLA provides instructions for creating an executable by building C++ code against the LibCarla client library. The provided **make-**

file packages the LibCarla classes into the target directory's `install/` folder and links the library with the specified C++ (**.cpp**) file. Using this C++ API allows for startup and control of the LibCarla client, while bypassing the installation and usage of Python packages. However, the client library makes use of functions that prevent static linking. Static linking copies complete system libraries into the target executable, producing a portable executable that can run on different systems. Instead, the **makefile** dynamically links the library with the C++ file. This means that, at runtime, the executable requires separately installed system libraries. It may be possible to automate this through a bash or shell script.

Traffic Management

Traffic Management is handled in a centralized, monolithic system that resides in the client-side LibCarla library. The centralized system affords multi-lane support and intersection negotiation. The traffic manager caches the state and trajectory of all registered traffic vehicles. It uses this data to continuously compute and send the next command for each vehicle. It relies on converting the map's OpenDrive [84] data into a R-Tree [37] representation, for quickly finding nearby neighbors, and a grid of waypoints for lane merging.

Vehicles

All steering and trajectory commands are sent through the LibCarla client library. The LibCarla server receives these commands and sends it to the target vehicle, identified by an assigned ID. The vehicle's `CarlaActor` class routes these commands to the Unreal Engine's vehicle class, which translates the command to changes in the vehicle's steering angle or physics trajectory.

Walkers

CARLA now has support for pedestrians, which can be set to move at different speeds and randomly cross streets. The pedestrians are separately controlled and possessed by individual **WalkerAIControllers**, within Unreal Engine, and are not handled by the traffic manager.

Fixed vs Variable Timestep

LibCarla can be configured to handle variable timesteps but operates in a fixed timestep, by default. This means the calculated changes do not take the different lengths of time between frames into account. Instead, a constant time delta is set and applied to each frame's calculations. This is ideal for the system's target autonomous systems, as it allows for deterministic state updates. For real-time player simulations, however, this must be changed to a variable timestep. Otherwise, the vehicles and pedestrians may appear to be moving far too quickly at high framerates.

Synchronous vs Asynchronous Ticking

CARLA recommends that the LibCarla client enforces synchronous ticking, which means that the system effectively stalls the Unreal Engine's state of the world until the client has finished all of its computations and the server receives all commands. The multiple stages and complexity of the traffic manager result in performance bottlenecks under a synchronous architecture. This guarantees that the traffic manager can accurately account for every change of the world's states.

The performance impact is of little consequence to autonomous systems, which do not need to update in real-time. With asynchronous ticking, the traffic manager collects data from the Unreal Engine's simulation state and simultaneously performs its calculations in a separate process. This means that the traffic manager will be unaware of subsequent changes in the state of the world, before it is able to finish its calculations and again poll the state of the world. This may result in more 'jittery' traffic and less sensible routing. As we prioritize a human-centric experience, we sacrifice this accuracy for better performance.

1.7 The CARLA 0.9.13 Framework

The developers of CARLA provide minimal documentation for everything besides high-level overviews and the Python API. The source code was effectively a black box. Therefore, a significant amount of time was spent reverse-engineering and documenting the source code. The following section provides the current understanding of the existing project infrastructure.

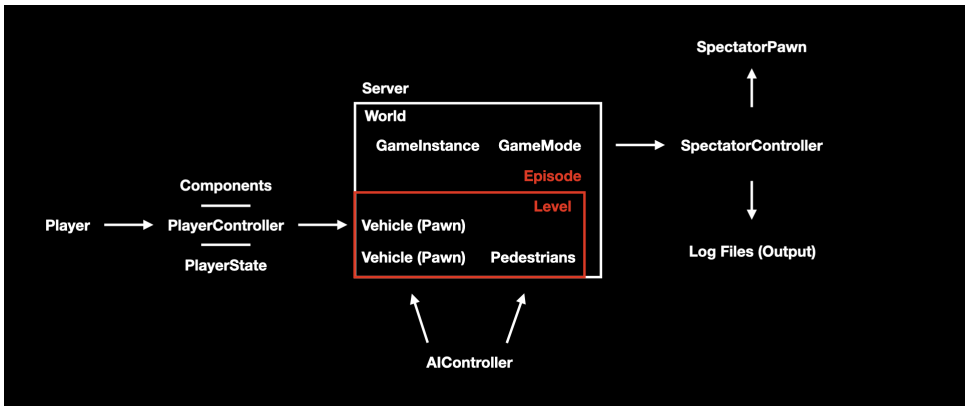

Figure 1.1: An diagram of the combined CARLA and Unreal Engine 4 structure

High-Level Overview

At a high-level, CARLA can be considered to consist of five components. Its (1) Python API is intended to be the client-side interface to ultimately control the server-side Unreal Engine 4 game instance. Between this, the Python API directly interacts with (2) LibCarla, a self-contained collection of C++ processes and libraries. LibCarla represents a massive departure from earlier versions of CARLA, in that LibCarla houses many of the critical computations: traffic management, map processing, pedestrian and vehicle spawning, to name a few. Simpler versions of these features were previously implemented directly into the CARLA project plugin for Unreal Engine. Through network m, remote procedure calls (3, the pimp module), LibCarla and the (4) CARLA project plugin continuously exchange information and commands. The CARLA project plugin contains C++ classes that override the Unreal Engine default game code, effectively controlling the (5) server-side game instance.

Our goal is to implement a human-centric driving simulator, built upon Unreal Engine 4.26 and the latest release of CARLA (0.9.13). A majority of research time was spent investigating CARLA's behemoth of an implementation and consulting its documentation. There is a dramatic system implementation shift between CARLA 0.9+ and previous releases. A major challenge was coming up with creative solutions to overcome the hurdles of CARLA's external dependencies.

Client-Server Model

With every successive release, CARLA further optimizes its pipeline for training and testing autonomous agents. To achieve this, CARLA tries to entirely abstract away the UE4 code with its Python interface. The Python interface, and the added module dependencies (e.g. PyGame), introduce more complexity to the user. Out-of-the-box, Python and Unreal Engine launch separate game windows, which may confound a human user and wastes computational resources. Python also impacts performance, due to its interpreter nature. The Python interface is binded, via the external Boost [7] library, to a C++ library, called LibCarla. LibCarla itself is split into two versions, a client and server. The client communicates with the server over local network *RPC* calls. The LibCarla server then interacts with the UE4 project, which runs the simulation world. This means numerous steps must be taken when controlling a car.

1. A command is entered through the Python script
2. The command is then routed to the LibCarla client library
3. The command is packaged as an *RPC* message
4. The message, as a network packet, to the LibCarla server library
5. The LibCarla server library unpacks the message and passes it along to the Unreal Engine classes
6. Unreal Engine executes the command
7. Unreal Engine updates the state of the world
8. Unreal Engine sends image data from the vehicle's camera to the LibCarla server library
9. The LibCarla server library packages and sends the data over *RPC* to the LibCarla client library
10. Python reconstructs the image and displays it in a separate Python window

A Closer Look at the Client-Server Communication Pipeline

LibCarla

The following is a more technical, involved look at how LibCarla communicates with the CARLA plugin and Unreal Engine 4. To get a better sense of the communication flow, we follow the call stack initiated by the *ALSM* module, part of the traffic manager in LibCarla. The *ALSM* module needs information about all present actors in the UE4 game instance, so it requests this information from the LibCarla **World** class. The LibCarla **World** class then passes the request to the LibCarla **Episode** class, which then performs three main actions: 1. Requests the Identification Numbers for all actors in the game, from the LibCarla **EpisodeState** class 2. Compares the retrieved IDs with its own internal **CachedActorsList** 3. Get information about any non-cached actors through the LibCarla **Client** class The LibCarla **EpisodeState** and Client classes ultimately serve as the gateway to the server-side game instance.

CARLA Plugin + UE4 Project

The Unreal Engine 4 project loads the CARLA project plugin, which overrides the default game C++ classes with its own. The **WorldObserver** class defines a sensor, which spawns in the game and polls for information about the entire game world. It sends information requested by the LibCarla **EpisodeState** class over the established **rpc** channels. Similarly, the **CarlaServer** class acts as the **rpc** messenger between the LibCarla **Client** class and the rest of the Unreal Engine 4 game. For example, the LibCarla **Client** sends specific IDs as a request for its missing actors. The CarlaServer receives this request and returns results from the **Episode** class's registry of actors

Defining Actors

In Carla, *Actors* usually refer to the vehicles present in the server-side game. The information and location of *Actors* is constantly updated in the LibCarla processes, and is used by the traffic manager. Pedestrians, also present in the game, are largely self-contained and are randomly controlled by UE4, instead of the traffic manager. The Carla UE4 project, referring to the server-side game instance, contains a factory, or library, of actor definitions, essentially blueprints or schematics. A definition contains several variations of possible vehicle characteristics,

such as colors or number of wheels. Spawning a valid actor requires an actor description which contains several attributes. These attributes consist of values, including each possible variation of the actor's definition and the actor's role.

Discussed more in-depth in the following sections, an actor's role is usually defined as a hero or traffic. Hero vehicles are intended to be the egoistic frame of reference for experiments, and are usually the focus of the experiment's computation or controls. Traffic vehicles are those handled by the LibCarla traffic manager. This framework for actors is replicated by the client-side LibCarla, which retrieves actor information from the UE4 **CarlaServer** and reconstructs its own mirroring data structures. Relevant classes of interest include the **ActorBlueprint** and **ActorBlueprintFunctionLibrary** classes. These equivalent data structures are what is used by the traffic manager.

The Traffic Manager

The 0.9.13 traffic manager is the largest departure from CARLA 0.8.4, containing entirely new code and offering multi-lane traffic capabilities (CARLA 0.8.4 only supported single-lanes). It is self-contained within LibCarla. At every tick (if in synchronous mode), the TM must retrieve game state information from the server, perform computations, then send control commands to all server-side actors. It has a highly complex structure involving several stages:

1. **Localization** – calculating the relative position of actors (Location, Heading, Velocity, Speed, etc.) in regards to each other and nearby map characteristics, such as multiple lanes or obstacles

2. **Collision** - calculate whether any actors, in their current trajectory, will collide and, if so, mark for steering correction

3. **TrafficLight** - check to see if vehicles are at an intersection and, if so, prepare to coordinate movements of all vehicles, at each different intersection

4. **MotionPlanner** - Given the status of all actors/vehicles in the map, plan and create the control commands for all actors

5. **VehicleLights** - Account and correct for the state of all vehicle lights, according to the weather and driving situation.

These stages can be further decomposed into several smaller classes that make use of C++ libraries, primarily boost data structures. Once

all stages are complete, the control commands are passed by *rpc* to the CarlaServer class, which passes on each command to the designated actors. In synchronous mode, this entire process is done in one tick. This dramatically slows the system down, but ensures that the TM sees every single change in the game. For real-time, it is possible to use asynchronous mode where the server-side game will not wait for the TM to complete, before moving on to the next tick. The TM will still operate, however, the states it receives and commands it sends may be always outdated, with respect to the real-time server-side game instance.

Each stage is specifically bottlenecked, in that all computations of the previous stage must be completed before moving on to the next. This all-or-nothing approach also means that immense maps would require more computation time before ultimately sending out any control commands to any actors.

1.8 Implementation

Replay Recording

CARLA implements its own recorder, which also logs additional information about the state of all actors. We sought to integrate Unreal Engine's built-in replay recording system, which is highly optimized and makes use of its network replication feature. To accomplish this, I followed the Replay System Tutorial from the Unreal Community Wiki. I added the networking-related **DemoNetDriver** to the game project's **DefaultEngine.ini** config file. Several lines of code were added to the **CarlaGameInstance** class and updated to match the specifications of Unreal Engine 4.26. The Replay Recording system records all present actors' commands at each tick and stores the data in an optimized file format. It automatically handles both synchronous and asynchronous ticks and matches the variable timesteps that occur during recording sessions.

Traffic Manager

CARLA is designed such that client-side LibCarla, which contains the traffic manager, and the server-side UE4 project must both be launched, separately. Our goal was to simplify this startup process, such that launching the UE4 project would automatically start the traffic man-

ager. Additional performance considerations make this a non-trivial task.

In collaboration with an undergraduate, we discussed possible methods of TM integrations and their individual drawbacks: 1. The method with optimal performance would be to integrate the traffic management logic into Unreal Engine. We naively assumed this could be done by moving the TM code directly into the CARLA plugin for Unreal Engine. However, we found that the TM is so thoroughly integrated with the rest of LibCarla that it would require substantial rewriting in order to integrate with the server-side data structures. Another point of concern is that the TM makes extensive use of external C++ libraries, e.g. Boost, however these do seem to be integrated into the server-side Unreal Engine project. Reliance on these could induce bottlenecks, as the code does not necessarily interoperate with Unreal Engine-optimized components. Ultimately, this would provide feature parity with the current CARLA, but would be the most time-intensive. 2. An alternative could be to build our own equivalent TM, completely from scratch. The advantage of this would be that we could make use of built-in and highly-optimized Unreal Engine components. However, it is difficult to find pointers or pre-existing documentation on creating Traffic Management systems in Unreal Engine.

There is no guarantee that we can also reach feature parity with the current CARLA. 3. The quickest to verify, but least-performant, option would be to have the Unreal Engine project automatically start the client-side LibCarla. LibCarla is effectively still a separate process, and communicates with CARLA over the default *rpc* ports. In order to access the traffic manager, LibCarla must first be compiled and linked to Python or C++ executables.

Given time constraints, we ultimately chose to pursue the third approach. However, we designed our implementation to be modular, with support for future attempts at native integration. We break down our solution into three components: the C++ API, compilation, and the CARLA plugin for Unreal Engine.

C++ API

CARLA provides extensive documentation for its Python API, which the Boost library translates to CARLA's C++ API. For portability and stability optimizations, we bypass the Python API and directly implement three C++ scripts: `main_tm`, `worker_tm_register`, and `worker_tm_deregister`. The `main_tm` must be executed first and is responsible for starting up LibCarla's traffic manager system. It con-

figures the traffic manager to be in Hybrid Physics mode and controls the computational ticks, by way of its never-ending while loop.

The `worker_tm_register` and `worker_tm_deregister`, as their names imply, are assistants to the traffic manager. They accept lists of actor ID numbers, which it then either registers or deregisters the traffic manager system. The traffic manager only keeps track of and sends commands to registered vehicles. Therefore, destroyed vehicles must be manually deregistered, to ensure accurate future commands.

Compilation and Linking

The C++ API allows us to create standalone executables that do not require installing the Python library. To create the executables, the C++ scripts must be linked with the built LibCarla library during compilation. By default, the compiler will dynamically link the executable, meaning that it will require system-specific library files. To make the executables system-agnostic and truly portable, we can instead statically link the executables. The only viable method we found was to change the C++ API's example **makefile** to use CARLA's **clang++** compiler. The CARLA **makefiles** have been modified to automatically compile the C++-based executables and move them into the CarlaDependencies folder, within the Unreal Engine 4 project directory.

CARLA Plugin for Unreal Engine

Within the CARLA plugin source exists a traffic directory. There, we added two classes:

1. The TrafficManager class is intended to be the highest-level control of traffic, from within the Unreal Engine project. Modular by design, its role is to send high-level commands to any valid assigned *interface*. The *interface* is intended to either handle these computations within Unreal Engine, or pass them off to the LibCarla executables. This class also handles the spawning of vehicles and passes its IDs to the interface, for registration.

2. In this case, we implemented a LibCarla interface that handles the high-level commands from TrafficManager and starts our executables. This class uses Unreal Engine's built-in `FPlatformProcess::CreateProc` to start the `main_tm` executable. When the TM is ready, the interface passes the ID list, as a string, to the `worker_tm_register` executable.

Figure 1.2: Screenshot of the custom-build of CARLA

Performance Optimizations

As physics substepping presents a major computational bottleneck in Unreal Engine, CARLA includes a configuration for Hybrid Physics. To enable this, one vehicle must be designated with the role of the egocentric hero. For our purposes, that will be the vehicle that the player controls. This is specified as an attribute within the description of the actor that the player is assigned to. LibCarla requires a set radius distance from the designated hero, so that all traffic vehicles within that radius will have physics enabled. Any vehicles outside of that radius will have physics disabled, and will instead 'teleport' its position between LibCarla computation steps.

1.9 Evaluation

Replay Recording

The replay recording functionality can be tested via in-game console commands, demorec test and demoplay test. When testing replay, Unreal Engine would respawn all vehicles, pedestrians, traffic lights, and roads. Buildings, flooring, and barriers failed to respawn. This would cause vehicles to fall off the map and into a bottomless void. Upon further testing, it was found that the CARLA project's use of sublevel streaming was the culprit. Different 3D models are grouped into sublevel categories, which are streamed for further performance optimizations. Disabling this streaming and forcing the sublevels to be per-

sistent solved this issue. Replay Recording now is fully functional and properly respawns all 3D models that were present during the recording session.

Traffic Manager

This custom-build of CARLA was informally tested (Figure 1.2) on a system with a 12-core, 24-thread CPU, 48GB of RAM, and GTX 1080 Ti. With Hybrid Physics enabled and asynchronous ticks for LibCarla, the game achieved a roughly 50 fps minimum and an average of 60 fps.

1.10 Discussion

We initially attempted to copy/paste over the contents from the CARLA 0.8.4 project directly into the CARLA 0.9.13 project. Our intention was to manually iron out the dependency errors and conflicts, but it quickly became apparent that there was a fundamental structural rewrite between the two versions. What followed was an extensive process of comparing the implementations of both systems and reading the documentation of each. We tried to copy code line-by-line and manually patch up the errors, as I was worried that the mountains of code had functionality or corner cases that I had yet to understand. However, our graduate mentor advised us to instead try understanding the high-level goals of each system. It took me a long time to understand and accept that I needed to replicate the capabilities of the CARLA 0.8.4 system, but the implementation details could be vastly different. A lot of time was wasted on trying to come up with a coding standard to make future upgrades easier, but eventually I accepted that it was far more important to create a working system.

Chapter 2

Semantic Segmentation for Source Engine Games

2.1 Introduction

In order to perform brain decoding/encoding, within the Voxelwise modeling framework [63, 80, 112], time-based stimuli must be non-linearly transformed into features. The following chapter details an implementation of transforming such features by extracting semantic information from immersive stimuli.

2.2 Background

Semantic segmentation involves assigning entities to categories, and providing a method to identify which categories are perceived, at different points in time. These categorical labels provide useful representations about the world for machines and humans to better interpret data. Ground-truth labels are especially important for training better machine models and providing the most possibly accurate interpretations of data.

Counter Strike: Source [107] is a ubiquitous multiplayer game, affording rich interactions, and continues to be supported by a robust modding community. Its extensive documentation and plethora of custom content affords great flexibility in crafting custom environments and rules, making this a viable platform for generating semantic segmentation datasets. However, the game does not feature semantic segmentation out-of-the-box.

One workaround is to split each rendered frame into smaller regions and assign each region to the category that shares the most similar

group of pixel color values. This avoids any game modification, but it is difficult to accurately predict on pixel values that constantly fluctuate from fancy rendering effects, such as interpolation and shadows. A common alternative approach is to find the target entities and assign each of them to an optimal category. For each category, the rendered pixels of all assigned entities are replaced with a unique solid color.

To achieve ground-truth labels, we pursue an automated method of directly replacing the textures of every 3D model. These textures are accessed by locating the asset paths for each model. The models are then grouped into various semantic categories, based on the names of their parent directories. Distinct solid colors are generated, for each semantic category, and replace the original texture files. The game configuration settings must then be tweaked to disable various cosmetic features, such as pixel interpolation and fog passes, that can alter the rendered color of textures. We essentially want to produce frames that are equivalent to a rasterized scene with shading that ignores all physical effects, including light sources and shadows. This creates "ground-truth" semantic segmentation labels within the game itself. We focus on a pipeline for the `de_dust2` map, but this method can generalize to any other Counter-Strike: Source map.

2.3 Related Work

Previous works have explored approaches to generating semantic segmentation datasets and applying them for scientific interpretation.

Krahenbuhl (2018) [60] developed an approach that is able to generate live semantic labels from signatures and tokens, during runtime. This is achieved by reverse engineering the DirectX rendering code, specifically the HLSL binary shader, that is shared by many games. Code is injected such that, at runtime, the rendered objects' tokens are used to assign groups and generate semantic labels. The result is a game-agnostic system, with the caveat being that the game must render using DirectX, a primarily Windows-dependent library. At a high-level, our pipeline determines semantic groups based on a texture's location in its file hierarchy and its parent directories. This is similar to how Krahenbuhl's code determines semantic groups based on the rendered shader's tokenized variable name.

Huth et al. (2012) [53] hand-labeled semantic groups present in individual frames of naturalistic movies. These movies were presented, as stimuli, to participants whilst their brain's BOLD-data was recorded, via an fMRI machine. A novel linear regression model, dubbed 'voxel-

wise modeling', decoded brain representations of different semantic categories.

Demonstrating the viability of Counter-Strike: Source as an experimental paradigm, Zhang et al. (2021) [114] simultaneously recorded screen captures and fMRI BOLD data from participants. Voxelwise modeling was performed to regress hand-labeled behavioral states, from the screen capture frames, onto the corresponding BOLD voxels. They are able to recover a low-dimensional behavioral state space that can predict on unseen data, despite the existence of strong correlations between the different states. The behaviors were also projected back on to the cortical surface to identify two general groups consistently corresponding to specific brain regions.

2.4 Preliminaries

The Counter Strike game uses Valve-specific file formats, requiring the use of several specific programming tools. This section breaks the game down into the components that need modification, their corresponding file formats, and the tools used accordingly.

Source Engine Compiled Maps (Dust II)

Upon launching Counter Strike, the underlying Source Engine [106] will load all 3D models from a VPK package. A VPK, also known as a Valve Pak, optimizes for real-time loading by storing the game contents, and their corresponding directories, within several smaller archives. These smaller archives are collectively referenced and managed by a master VPK file ending in `_dir.vpk`. GCFScape, a third-party tool, is able to combine all of the archives and export the original assets and directories.

Originally, the plan was to visually identify the semantic categories of the textures, based on their associated 3D models, by inspecting the `de_dust2` map. Counter Strike games can load one of several maps. Map files contain information about landscape geometry and which 3D assets the game engine should load. These assets include prop objects and materials for all 3D models in the map. Maps are compiled into commercial BSP format files, found in the `maps` directory, extracted from the VPKs. The third-party Crafty application can open these compiled map files, but it neither loads or identifies any textures. Unofficial BSP decompiler software could retrieve the original texture references, but we did not choose to pursue this method.

The Components of each 3D Model

Landscape geometry is encoded as vertices, faces, and normals within the map file, while reference prop models are compiled into the MDL format. Models are found in the extracted `models/` directory. Each of these references its own material.

Materials are Encoded as VTF and VMT Files

Like the landscape geometry, 3D model files contain information about their vertices, faces, and normals. Their 'aesthetic' information is contained in separate material and texture files. Materials define the 'qualities' of the surface, such as the amount of reflectivity or transparency, and are defined in a VMT ("Valve Material Type") text format file. Textures are akin to defining what colors are 'painted' on the surface of the object, and are encoded in an accompanying commercial VTF ("Valve Texture Format") file. The texture and material files, for each object, are found side-by-side in the extracted `materials/` directory. This directory will be the focus of our semantic segmentation approach.

As VTF files are a commercial format, texture files are first saved as a standard image format, such as PNG. VTFEdit, part of Nem's Tools [36] VTFLib software, can batch convert the PNG files into VTF files, in-place.

VMT files each define a single material, which consists of a single shader-type with configurable parameters. Different models use various shaders, each with its own inherent properties. Shaders determine how an object is rendered onto the screen. Complex shaders can handle real-time shadows and refractions. However, such physical rendering produces non-uniform pixel values for rendered objects, which is undesirable for semantic segmentation. The **LightMappedGeneric** shader is typically used for 2D brushes and lightmaps, while **VertexLitGeneric** can be found in most materials for 3D models. For our purposes, we want to assign all materials to have the **UnlitGeneric** shader (Figure 2.1), as it is invariant to light propagation and is capable of producing flat shading. It contains a `$basetexture` parameter that accepts an albedo VTF file.

As a special exception, the map's floor cannot use the **UnlitGeneric** shader and its material instead uses a unique **WorldVertexTransition** shader. This shader blends between two textures. To work around this, both textures can be identical. To replicate the flat shading of **UnlitGeneric**, the **WorldVertexTransition** (Figure 2.2) shader is set to be self-illuminating, via its `$selfillum` parameter.

```
UnlitGeneric
{
    "$basetexture" "de_dust/ducrtlrgtp"
}
```

Figure 2.1: An example UnlitGeneric VMT File, which is invariant to light states and is capable of producing flat shading.

```
WorldVertexTransition
{
 "$baseTexture" "de_dust/groundsand03"
 "$basetexture2" "de_dust/groundsand03a"
 "$selfillum" "1"
 "%keywords" "cstrike"
}
```

Figure 2.2: An example WorldVertexTransition VMT File, which blends between two textures.

Texture Mods

When loading all assets, including maps, the Source Engine will first check and see if any identical paths and files are present in the `custom/` directory. This allows for modding, as any custom assets will take precedence over the default version, so that the default file will be effectively ignored. For our purposes, the exact directory structure, containing the textures and materials loaded into the `de_dust2` map, must be replicated within the `custom/` directory. This will replace all textures and materials with our custom variants with flat shading and uniform colors.

Source Engine rendering settings

The Source Engine employs physical rendering features, such as fog and draw distance, that involve interpolation of the on-screen pixels.

These additional rendering passes alter the original pixel color values of textures. Therefore, it is imperative to disable as many of these features as possible, through console commands, in order to retrieve accurate semantic categories.

Operating Systems and Libraries

Our pipeline was set up on Ubuntu 18.04. Counter Strike: Source was installed via Steam and runs on the Proton software layer. Nem's Tools [36] and the .NET [76] library dependencies were installed with Wine [56]. The NET [76] Framework was installed through Winetricks [27]. The custom pipeline script used Python [29] with PIL [15, 70] and CSV libraries.

2.5 Semantic Segmentation Pipeline

VPK extraction

In the Counter Strike game's folder, we locate the vpk file ending in `_dir.vpk`. This contains references to all other vpk files that contain the total sum of assets used within the various maps. GCFscape is used to extract all directories from the vpk files. These directories contain all assets, including the textures and materials necessary for our semantic segmentation pipeline.

CS:S console commands

While there are parent directories that match the names of specific maps, they only contain a subset of the assets used. Many textures and materials are shared between all maps, with some contained in recycled directories from an older game, Half Life 2. Rather than exhaustively modifying every single texture, running the command `mat_showtextures` from the console while playing in the `de_dust2` map outputs the paths of all currently loaded textures.

Folder hierarchy corresponds to a semantic group color (RGB)

The list of paths is a subset of all possible textures, resulting in a more compact hierarchy with names that correspond to possible semantic groups. Visualizing the hierarchy as a tree diagram, leaf nodes

that share the same parent are considered to be in a unique semantic group. Higher-level parent nodes can be manually assigned to a semantic group, so that all of its child leaves now share the same label. This reduces the total number of semantic categories and increases ease of interpretability. Manual assignment was done by assigning category names to each texture in a CSV format file.

An example assignment method would be to increment the RGB value for each next semantic group, starting with R and moving on to G, once the value limit of R has been reached.

Python script with Functional Programming

A functional pipeline (Figure 2.3), written in Python [29], reads the CSV file. It constructs the relevant directories to mirror the original game's path structure, and generates the textures with corresponding VMT material files in the correct locations. The textures are generated as solid color PNGs of an arbitrary size. Their color is dependent on the semantic category, which is converted into a unique corresponding RGB value. The VMT material files are generated according to the standard text format, with material type set to 'UnlitGeneric'. This material type is not dependent on lighting and produces the flat shading needed for semantic segmentation. This ensures that rendering features, such as reflection and specular lighting, do not affect any textures.

Assigning colors to semantic categories

A possible approach to assigning unique colors to each semantic category is to maintain a counter variable c that assigns its number to the next semantic category and increments thereafter. The semantic category's assigned number is converted to the RGB range and floats (decimals) are truncated to an integer (floor) (Equation 2.1).

$$R = \lfloor \frac{c}{256^2} \rfloor, G = \lfloor \frac{c}{256} \bmod 256 \rfloor, B = c \bmod 256 \tag{2.1}$$

Batch converting PNGs to VTFs with VTFEdit

The generated output structure is then batch converted via VTFEdit, which in-place converts all of the PNGs into Source Engine's VTF texture format. The original PNG files are removed, and the completed structure is moved into the game's `custom/` directory.

```
load_csv ← Load texture paths and
           assigned semantic categories
           from csv file

for texture, semantic category in load_csv:
  generate all parent directories
  if semantic category is unseen:
    assign new RGB color to category
    increment assignment counter
  find corresponding RGB color to semantic category
  generate (4096x4096) PNG in correct texture path
  generate corresponding VMT file in correct path
```

Figure 2.3: Pseudocode for the functional pipeline that generates materials and creates mappings to semantic categories

Moving the custom files into the game's directories

The generated texture files can also be compressed into a VPK file and placed into the `custom` directory, but initial testing demonstrated empirically worse performance compared to moving over the full, uncompressed structure of textures and running the game.

Source Engine rendering settings

Console commands were found online, but lacked complete descriptions. Each command was manually tested for functionality. These console commands can be inserted into a config (Figure 2.4) file that is loaded by the game engine, at runtime.

1. `sv_cheats`: allow cheats on server – game sessions are run on servers, several settings are considered "cheats" which require this setting to be enabled

2. `mat_motion_blur_enabled` – toggles motion blur which may cause unwanted interpolation of screen per-pixel RGB values between frames

3. `mat_disable_fancy_blending` – toggles blending of textures, in scenarios such as overlapping

```
//custom
sv_cheats "1"
mat_motion_blur_enabled "0"
mat_disable_fancy_blending "1"
mat_disable_bloom "1"
mat_disable_lightwarp "1"
r_drawsprites "0"
r_drawdecals "0"
r_shadows "0"
r_drawparticles "0"
r_drawbeams "0"
mat_fullbright "1"
cl_drawhud "0"
mat_bumpmap "0"
fog_override "1"
fog_start "30000"
fog_startskybox "30000"
fog_end "30000"
fog_endskybox "30000"
r_farz "30000"
mat_antialias \0"
```

Figure 2.4: Contents of the config file, containing console commands that modify Counter Strike: Source in-game settings.

4. `mat_disable_bloom` – toggles lighting bloom, which blends with the textures and changes screen pixel RGB values
5. `mat_disable_lightwarp` – toggles miscellaneous lighting effects that can affect screen pixel values
6. `r_drawsprites` – toggles 2D textures that are layered on assets, such as graffiti on walls, and fog layers that are projected into the sky
7. `r_drawdecals` – toggles 2D textures that appear on assets, such as gunshot indentations
8. `r_shadows` – toggles the rendering of shadows, which change the on-screen pixel values of rendered textures

9. `r_drawparticles` – toggles particle rendering, which can occlude or blend with textures and produce altered screen pixel RGB color values

10. `r_drawbeams` – toggles light-beams, which can alter textures in a manner similar to that of particles

11. `mat_fullbright` – enabling this will cause all materials and 3D models to effectively self-illuminate and be invariant to other lighting sources. This uniform lighting should produce pixel RGB values that match that of each model's original textures.

12. `cl_drawhud` – toggles the HUD elements, such as the minimap and HP values

13. `mat_bumpmap` – toggles using the material normals to alter the rendered appearance of textures, usually by altering light and shadow scattering. This too may cause undesirable modifications to the original texture's per-pixel RGB values

14. `fog_override` – overrides the default rendering settings of game fog, which blends with the textures and modifies the original texture pixel RGB values.

15. `fog_start, fog_startskybox, fog_end, fog_endskybox` – parameters for the fog location and size, we maximize these values so that the fog is not present within the map itself

16. `r_farz` – overrides the clipping plane, the distance cutoff point at which the fog should obfuscate any models that are clipped. This is intended for performance optimizations. We maximize the possible value so that effectively no models are clipped

17. `mat_antialias` – toggles the antialiasing of edges and textures. This smooths and blends what is rendered, which modifies the original pixel RGB values of rendered textures.

2.6 Evaluation

The pipeline needs to be visually tested before it can be used to generate semantic labels. In this section, we walk through our approach of gradually testing and verifying the semantic labels of an increasingly greater number of categories.

Figure 2.5: Pipeline verification - Replace a few materials

To verify the functionality of the pipeline, we focused on replacing the texture and material for a single asset – the hands (Figure 2.5). Once the hands successfully changed into a flat color, we applied the pipeline to all loaded textures and materials (Figure 2.6). At this point, the ground is not set to the correct **WorldVertexTransition** shader and rendering passes have yet to be disabled. Therefore, there is noticeable pixel blending and interpolation on the screen.

GIMP

As a quick, informal, form of verification, randomly sampled frames were loaded into the GIMP [100] image editor. RGB values from random pixel locations were picked via the eyedropper tool. The associated semantic category was compared with the on-screen object that con-

tained the pixel. Correct labeling was visually confirmed by human intuition.

Multi-frame verification

All additional rendering passes and advanced engine features were successfully toggled off (Figure 2.7). For consistency and accuracy, batches of frames were cropped and the pixel color values were compared with their corresponding semantic categories. These were again visually verified by humans.

2.7 Application

Reliable semantic labels are useful for a wide variety of applications that involve representations. As an example, we detail how combining Voxelwise modeling and semantic segmentation enables advanced interpretation of brain decoders.

Figure 2.6: Pipeline verification - Replace all loaded textures and materials

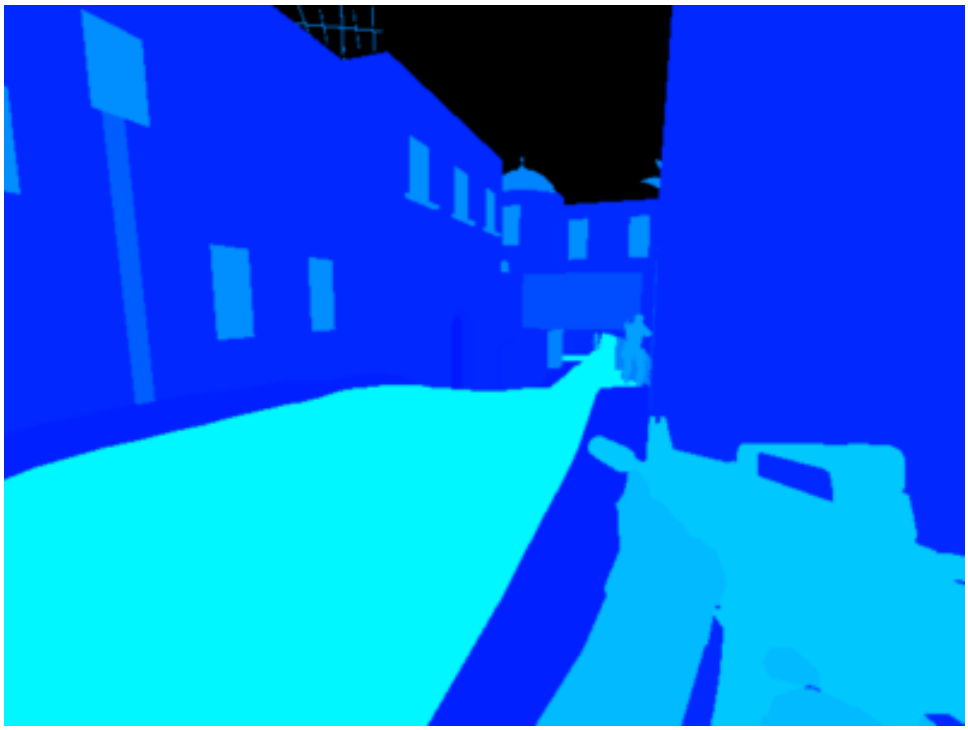

Figure 2.7: Pipeline verification - Fully replaced textures and materials, with all additional rendering passes and advanced engine features successfully toggled off

fMRI Voxelwise Modeling

As this pipeline works at the game level, we are able to extract frame-wise semantic categories from recorded sessions that are replayed. This data was fit, via Voxelwise modeling [80, 112], with accompanying fMRI recordings of the player's brain. The produced model recovers how the representation of the different semantic categories are distributed across the cortical surface, at the resolution of thousands of voxels.

2.8 Discussion

Bugs caused by Source Engine rendering features

A bug was discovered when illogical semantic categories were being reported in pixel clusters of very small sizes. Finding the location of these erroneous pixels confirmed that there was still an antialiasing

rendering feature that had yet to be toggled off. This resulted in an interpolation of pixel values, producing presumably gaussian noise that corresponded to incorrect semantic categories being identified across several frames. Toggling the engine feature off removed these erroneous pixels.

The majority of development time was spent researching the Source Engine system. A significant amount of time was spent trial-and-error testing different console commands to understand their functionality, due to their scarce documentation.

Chapter 3

Interpretable Cortical Representations of Timescale Selectivity for Active Navigation

3.1 Introduction

Coming full circle, this chapter details the creation of feature spaces from CARLA experiments. These feature spaces are then applied to the Voxelwise Modeling framework [63, 80, 112] to explore cortical representations of timescale selectivity.

3.2 Background

Mounting empirical evidence suggests that the human brain is capable of disentangling multiple timescales. In the occipital lobe, the ventral and dorsal streams have demonstrated representations of increasingly abstract information [65, 101, 108]. Increasing levels of abstraction intuitively correlate with increasingly longer timescales. Bayesian-related hypotheses provide impetus for brain processes to be defined as a hierarchical dynamical modeling system.

Since the ancient times, the brain is persistently theorized to be a Bayesian inference machine [19, 32, 49, 65]. A Bayesian inference machine's persisting objective is to minimize surprise, which is made possible by modeling its environment. The mechanisms behind how the brain implements this and, consequently, how the fundamental objective is represented within the brain is opaque.

Converging schools of thought ostensibly study different sides of the same coin. The Neural Sampling Hypothesis [28, 52] postulates that neurons sample from each other and the environment, as an efficient means to estimate the true state of the environment. Such a process could explain the emergence of our perception of 'time' [94], whose artifice is hypothesized to be a symptom of our limited range of sampling rates and thus loss of information, from smoothing [72]. The Neural Sampling Hypothesis has been considered, by some, to be subsumed by the unifying *Free-Energy Principle* [31, 33].

Building upon the Bayesian theory, Karl Friston [31, 33] proposed that the brain is driven by the 'free-energy principle'. The brain does not have an omniscient understanding of its environment, so it estimates the state from sensory input sampling. This estimation is then compared with the brain's expectation, a prediction based on the brain's internal representation of the environment. Honoring the Bayesian school of thought, the human brain's ultimate goal is hypothesized to minimize surprise about its environment and to optimize the evidence, or marginal likelihood, of its internal model. This persistent minimization objective motivates the brain to constantly update its internal representations, enabling online learning.

To explain the brain's ability to maintain temporally stable representations of diverse dynamic environments, converging hypotheses suggest that the human brain is fundamentally organized as a hierarchical dynamical system [12, 58, 59, 64] that can model different levels of dynamical complexity. In this hierarchy, successive levels model dynamics on increasingly longer timescales that reflect the complex systems of our world. These dynamics then serve as feedback control parameters for shaping manifolds, upon which consecutive lower levels unfold their more relatively transient dynamics.

One Bayesian implementation theory, Predictive Coding (PC) [31, 77, 90], posits that feed-forward communication is encoded as the residual difference from the higher cognitive function's prediction. The brain's objective is to optimize energy efficiency by minimizing prediction error. This goal can be regarded as being effectively the same as minimizing free energy. [32] In order to predict, it intuitively follows that higher orders of cognitive function would require information within longer timescales. This pursuit of energy minimization supports the emergence of the brain's hierarchical dynamical system. The act of predicting also involves going backwards in time, possibly explaining the phenomena of brain memory reactivation [30, 79, 82, 86, 109, 110] and our ability to mental time travel [102, 103].

In fact, modeling dynamics is entangled with the notion of memory,

in which longer timescales are intuitively dependent on maintained past context. In fact, the hierarchical dynamical system offers a perspective on memory: it emerges from the different timescales distributed across the brain's neurons. Corroborating this emergence, recent studies argue that all neurons display some degree of 'memory' capability [9, 41, 89].

The hierarchical dynamical system is hypothesized to be the governing principle behind the anatomical formation of the brain. A fMRI autocorrelation study purportedly found that the human brain exhibits hierarchical timescale 'gradients' parallel to the cerebral cortex, motivating differentiation between cortical networks [91]. Observed activity in cortical regions furthest from sensory input took relatively 'longer' to unfold, via slower temporal autocorrelation decay.

Recent research on timescales, investigating both listening and reading modalities [13, 55], converged on shared cortical representations. Voxelwise modeling identified gradients and patterns, at an unprecedented granularity, that also corroborate with findings from stimulus scrambling [66]. The decomposition of natural language into hierarchical timescales uncovered common cortical representations that reflect characteristics of commonly accepted regions of interest.

Translating this approach to the video domain, we propose and test a modular pipeline (Table 3.1) for identifying the neuroanatomical organization of hierarchical timescale selectivity exhibited during naturalistic navigation.

3.3 The Voxewise Modeling Framework

Voxelwise modeling [63, 80, 112] is a modern, data science-inspired approach for fMRI brain encoding (and, consequently, decoding in the reverse direction) within rich, naturalistic environments. Time-based stimuli are nonlinearly transformed into features – typically crafted manually or extracted from a machine learning model [2] – that are linearly projected [112] onto the corresponding recorded BOLD signals. Blood-Oxygen-Level-Dependent signals, measured during fMRI, reflect the varying levels of blood pressure in 3D voxels across the cortex [68, 69]. Each recorded voxel measures changes in blood pressure that are related to changes in activity of a contained neuron population. The BOLD signals therefore serve as proxies for neural activity, with each voxel's signal fit in a separate ridge regression model against the created features. The trained model is then evaluated on a held-out test set, encouraging generalization and reproducibility. The model weights, for each voxel, are treated as 'tuning weights' that reflect the average fea-

ture selectivity for the corresponding neural population. The features are represented by the distribution of tuning weights and prediction performance across the voxels. The effectiveness of this approach rests on the accuracy, and interpretability, of its features.

For natural language timescales associated with listening, Jain et al. [55] approached the feature engineering challenge by processing time-aligned audio [105] transcripts through a long short-term memory (LSTM) architecture. The architecture is trained to be a language model, in that it predicts future words, and explicitly represents different timescales within its internal units. These long short-term memory (LSTM) [50, 71] units are set to different forget-gate biases, approximating a power-law memory decay for the language domain [67]. Forget gate biases imply how long it takes units to 'forget' previous input and allows those units to explicitly represent different timescales. The hidden state activations are extracted from these LSTM units and are utilized as the features for Voxelwise modeling. The timescale selectivity for each voxel is estimated by essentially computing a weighted average, scaling the timescales of each hidden unit by its associated tuning weights.

Instead of enforcing an explicit separation of timescales, Chen et al. [13] demonstrated an alternative signal-based approach of applying custom linear filters to extract information, at different timescales, from a language model's vector embedding. A neural language model ("BERT") [22] projects the stimulus onto a relatively-smaller contextual word embedding space. Linear filters are then convolved on the resulting vector embeddings to separate linguistic information into components that each are still in the LM's embedding space and vary in timescale.

We translate these feature engineering approaches to the video domain by extracting activity from the latent spaces of a 3-dimensional Autoencoder and the Clockwork Variational Autoencoder [96].

3D Variational Autoencoder

Autoencoders [5, 95] involve some form of compression. Inputs, from a multi-dimensional space, are translated into internal representations within a latent space. This latent space is dependent on the embedding space that exists between the encoder and decoder. For imagery, encoders and decoders are typically narrowing and expanding convolutional neural networks (CNNs) [83], respectively. Images are considered 2D RGB pixel values. For videos, 3D autoencoders model spatiotemporal information by adding time as an additional dimension. Vari-

ational Autoencoders (VAE) [23] impose a prior constraint on the latent space that encourages a normal distribution. This is observed to improve distribution centering and a lower effective rank compared with vanilla autoencoders.

Clockwork Variational Autoencoder

The Clockwork Variational Autoencoder (CWVAE) [96] is a hierarchical latent dynamics model that is designed for long-term video prediction. When applied as a language model, CWVAE was found to outperform other video prediction models and reduce the accuracy gap to deterministic speech models [46]. The latent variables of multiple autoencoders are daisy-chained together, with each subsequent autoencoder acting as the successive level. Each level transitions its internal state at a different fixed clock speed that slows down by an exponential factor of k, for each successive level. This temporal abstraction factor k approximates the power law, a recurring statistic in nature [11, 47, 73, 85, 104], by maintaining constant power at each level via a constant internal state size. The CWVAE process can be broken down into two steps (Figure 3.1):

1. The autoencoder at each level, i, embeds k^{i-1} context frames for inference. All levels, excluding the topmost, receive an additional output, posterior q_{i+1}, from the next level up. Together, the embedding and passed-down posterior jointly condition the level's internal distribution of possible futures, represented by a stochastic cell. A sample from this distribution is treated as this level's posterior and, during the same step, is passed to the next level down. Each level proceeds to update its internal state every k^{i-1} steps. An update consists of a deterministic component, h, that combines the current level's previous posterior, as temporal context, combined with the passed-down posterior, q_{i+1}, from the next level up. The current encoded context frame and deterministic h is then jointly conditioned on the internal stochastic cell. A sample of the resulting distribution is the next posterior.

2. Once every level has seen a total of k^{i-1} context frames, CWVAE is ready to predict future frames. Each level's internal stochastic cell has also been solely conditioning on the deterministic h, without any observations, and outputting a prior, p_i. The prior now replaces the role of the posteriors. At the lowest level, a decoder transforms its prior into a video frame. Taken altogether,

this approach encourages the model to separate information into multiple levels of features, in an end-to-end manner.

3.4 Methods

An ongoing experiment collects first-person screen recordings, and accompanying fMRI data, from sessions of human subjects navigating inside a customized driving simulator. Each session is split into several runs. The original screen recordings have a refresh rate of 30 frames-per-second, downsampled to 15 frames-per-second, and a resolution of 1080 x 720 pixels. The fMRI data has a temporal resolution (TR) of 2 seconds and smooths out activity that is under the 2 second threshold. For one particular human subject, which we will designate as Subject x, a subset of each run is set aside for testing. The remainder are combined with the sessions from all other human subjects. The test set has a total of 18 run subsets that are approximately 600 seconds each. A 3DVAE and CWVAE were subsequently trained on the combined dataset.

3DVAE

The encoder and decoder, for 3DVAE, is composed of 3D convolutional neural networks (3DCNN) with residual connections. As the training dataset is fed into the autoencoder, the recordings are downsampled to 3.75 frames-per-second, at 120 x 90 pixels. This direct feed prevents loss of interleaved information and allows the model to maintain the original 15 hz sampling rate. 8 consecutive frames are combined, as a single input, for the 3DVAE. The input is a rolling window, with a window size that roughly matches the TR of the fMRI data. The latent space is a 256-dimensional vector that combines spatial and temporal information together.

The vector embeddings, of the test set, are extracted[1]. Discrete Cosine Transform (DCT-II) [3] is applied on each of the 128 channels, which are treated as separate signals. DCT-II converts time-series data into an equivalent number of coefficients, each being a scaling power for a corresponding cosine wave. The corresponding cosine waves are equally spaced apart, as frequency bins, and range from 0, the mean,

[1]The vector embeddings were provided by Cheol Jun Cho, an Electrical Engineering & Computer Science graduate student in UC Berkeley and the Gallant Lab. Cheol Jun Cho also designed and trained the 3DVAE.

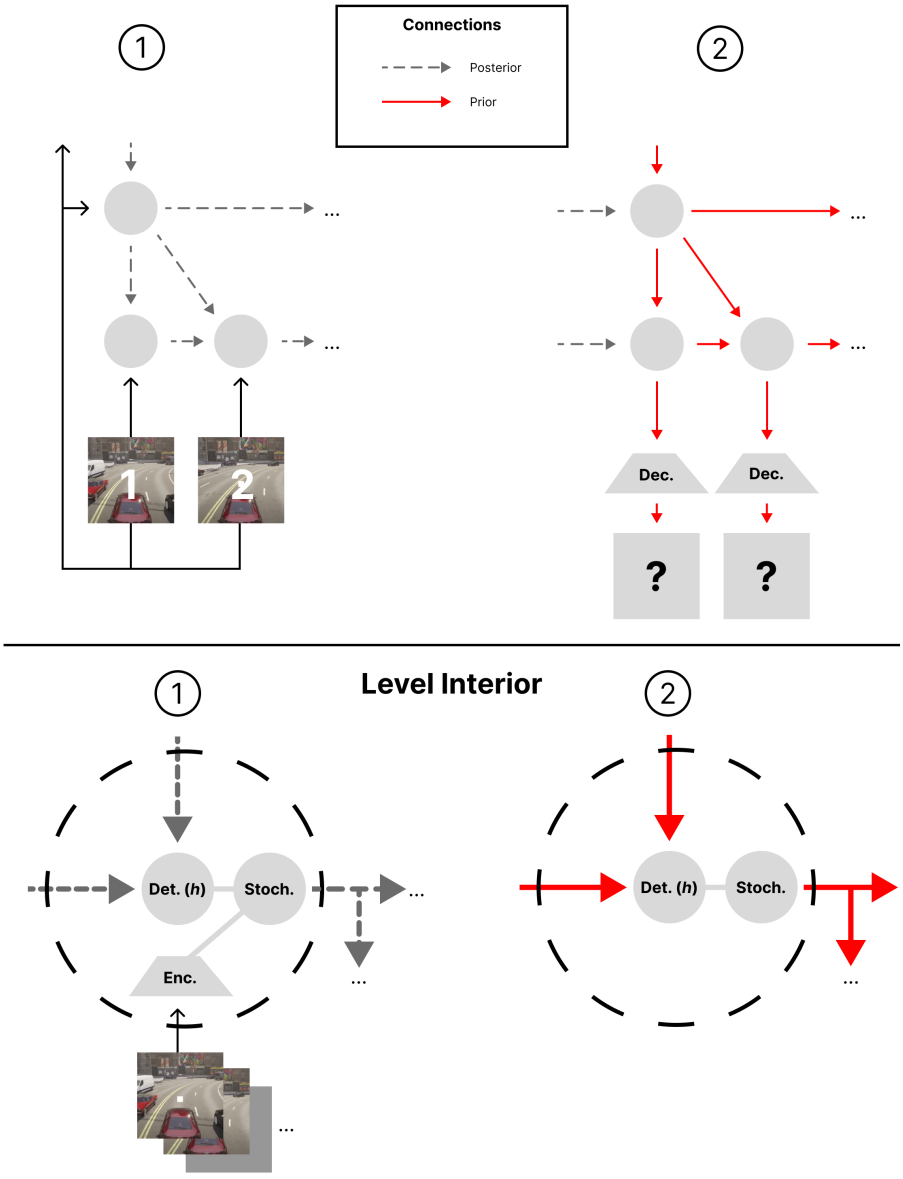

Figure 3.1: Diagram of the CWVAE process, as an unrolled loop.

up to half of the original sampling frequency, known as the Nyquist frequency. In this case, DCT-II yields coefficients that represent frequencies between the 0 and 7.5 Hz, or 0.13 seconds per frame. Characteristically, the coefficients generally follow a power law and the majority of power is condensed in the lower frequencies.

Inverse DCT (DCT-III) [3] linearly sums the cosine waves, scaled by their coefficients, back in the time-series domain. Modifying or masking different coefficients can approximate the effect of applying a custom Bandpass (DCT) to the original data. We pursue three approaches, with DCT, to separate the spatial temporal information into components that vary in timescales (Figures 3.2, 3.3).

- **Bandpass (DCT)s**: Divide the frequency by powers of 2, creating a logarithmic scale that reflects the 1/f spectra [57]. Break up the coefficients at these frequencies, creating components with timescales that are between these frequencies. The separation of frequencies reflects the hierarchical dynamical system hypothesis.
- **Log-Normal DCTs**: Each of the aforementioned "power" frequencies is the average, μ, for a normal distribution of power versus frequency bin. The frequency bins are logarithmically scaled and their corresponding powers are multiplied with the DCT coefficients. The intent is to improve the approximation of voxel characteristics observed by the natural language domain [55]. The normal distribution reflects the averaging [55] of timescale selectivity within the neuronal population of a voxel. The logarithmic scaling emulates how information, within a voxel, has been observed to decay according to a power law [67].

CWVAE

Custom Tensorflow [1] datasets were created for the training and testing. Each sample contains approximately 67 seconds of footage. The encoder and decoders are convolutional neural networks with 800 layers each. To maintain consistency with the DCT [3] analysis of the 3DVAE, the CWVAE [96] temporal abstraction factor is set to k = 2. The internal stochastic cell size, for each level, is set to be 100-dimensional. Subsequently, the prior and posterior are both 100-dimensional vectors.

The validation set is fed, as a rolling window, into a CWVAE that has 10 levels and requires 512 context frames (34 seconds). The first sampled prior is extracted from each level, containing information at a specific timescale.

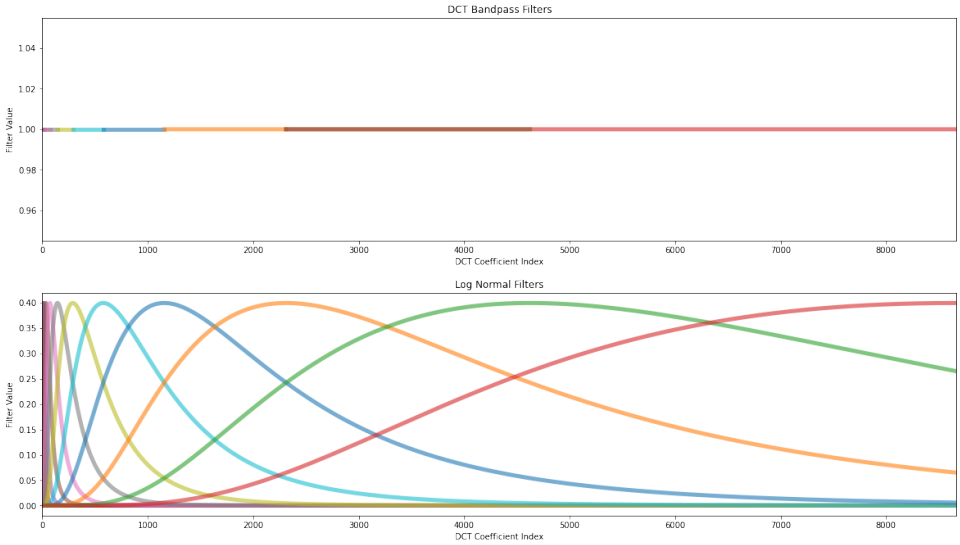

Figure 3.2: Plots of the different filter methods. Each of the colored lines (frequency bands) ultimately produce a unique feature space.

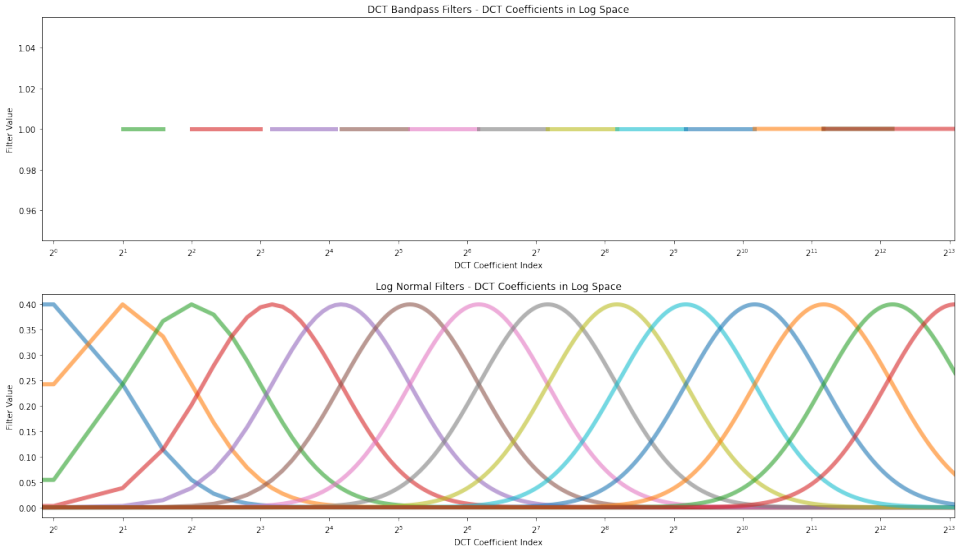

Figure 3.3: Plots of the different filter methods, with the DCT coefficient indices **in log space**. Each of the colored lines, and involved frequency bands, ultimately produce a unique feature space.

Voxelwise Modeling with Banded Ridge Regression

Each of these approaches produce components that are used as separate feature spaces for Voxelwise Modeling with Banded Ridge Regression, implemented by the Himalaya Python Library [29, 61]. Banded Ridge Regression [26] extends Ridge Regression [51] by simultaneously fitting multiple feature spaces, each with separate regularization terms. This enables the combination of all feature space's predictions and the consequent disentanglement of contributions from each feature space (Equation 3.1, 3.2) [25].

Estimating the Timescale Represented by Each Feature Space

- **DCT 'Band-pass filters'**: For each feature space, a constant power is applied to all frequencies within a frequency band. Consequently, a feature space's representative timescale is approximated by the midpoint frequency period of its corresponding frequency band.
- **DCT 'Log-Normal (DCT)s'**: As each filter is essentially a normal distribution, the representative timescale period is estimated by its mean μ, before logarithmic scaling is applied.
- **CWVAE**: The timescale for a level is approximated by dividing the level's temporal abstraction k-value by the known frames-per-second of the dataset.

Permutation Testing

The predicted BOLD activity, from each banded ridge regression model, is split into contiguous blocks of 10 TRs (20 seconds). These blocks are sampled-without-replacement 1000 times. The R^2 scores of these 1000 variations are compared with the actual prediction's R^2 scores to calculate each voxel's p-value. Significantly predicted voxels, with a p-score of $\alpha \leq 0.05$, are kept and the rest are discarded.

R2 Score and Thresholding

The R^2 score provides a metric for the per-voxel prediction accuracy of the Voxelwise models. For each model, Banded Ridge Regression [26]

Approach	**Frequency Bands (Hz)**	**Estimated Timescale (seconds)**
Band-pass Filters	$0 - \frac{1}{1024}$, $\frac{1}{1024} - \frac{1}{512}$, $\frac{1}{512} - \frac{1}{256}$, $\frac{1}{256} - \frac{1}{128}$, $\frac{1}{128} - \frac{1}{64}$, $\frac{1}{64} - \frac{1}{32}$, $\frac{1}{32} - \frac{1}{16}$, $\frac{1}{16} - \frac{1}{8}$, $\frac{1}{8} - \frac{1}{4}$, $\frac{1}{4} - \frac{1}{2}$, $\frac{1}{2} - 1, 1 - 2$, $2 - 4, 2 - 7.5$	1024, 768, 384, 192, 96, 48, 24, 12, 6, 3, 1.5, 0.8, 0.4, 0.3
Log-Normal (DCT)s	N/A	1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1, 0.5, 0.2, 0.1

Table 3.1: Summary of Feature Space Approaches

is able to calculate the joint prediction $\hat{y}$ via a weighted combination of predictions from all feature spaces (Equation 3.1). Consequently, the per-voxel R^2 scores $\tilde{R}_i^2$, for all $i = \{1, ..., m\}$ individual feature spaces, are corrected so that their sum is equivalent to the joint R^2 score of the full prediction $\hat{y}$ (Equation 3.2) [25, 26, 61, 81]. This also enables per-voxel comparison of each feature space's relative contribution.

$$\hat{y} = \sum_{1}^{m} \hat{y}_i \tag{3.1}$$

$$R^2 = \sum_{1}^{m} \tilde{R}_i^2 \tag{3.2}$$

All approaches yield voxels with a range of negative and positive $\tilde{R}_i^2$ scores for their predictions of different features spaces (information at different timescales) for the held-out test set (Figure 3.4, 3.5). The negative scores are thresholded out by replacing them with zeroes (Equation 3.3).

$$\tilde{R}^2_{i,thresholded} = \max(0, \tilde{R}^2_i) \tag{3.3}$$

Scaling

For the same feature space, different voxels exhibit different magnitudes of R^2 scores. This could result from differences in Signal-to-Noise ratios between voxels. It is also plausible that the information extracted from the autoencoders and DCT are not strongly represented by the voxels with lower R^2 scores. However, the voxels may still be selective of the timescales represented by the feature spaces. Therefore, each voxel is independently rescaled to approximate their individual relative timescale selectivity. For each voxel, the individual feature space $\tilde{R}^2_{i,thresholded}$ scores are divided by their composite $R^2_{thresholded}$ score (Equation 3.4). This allows for the estimation of a voxel's representative mean timescale selectivity, via its weighted average.

$$\tilde{R}^2_{i,scaled} = \frac{\tilde{R}^2_{i,thresholded}}{R^2_{thresholded}} = \frac{\tilde{R}^2_{i,thresholded}}{\sum_i^m \tilde{R}^2_{i,thresholded}} \tag{3.4}$$

3.5 Results

Joint R^2 Score Distributions

The log-normal approach yields the highest maximum R^2 score (Figure 3.6). Compared with the band-pass filter approach, the log-normal approach yields more voxels with higher R^2 scores and a narrower range of negative R^2 scores, ostensibly outperforming the band-pass filter approach. The 9-level CWVAE predictive priors approach yields the lowest maximum R^2 score.

Weighted Averages Yield Consistent Relative Timescale Selectivity Distributions Across Voxels

To compare with previous natural language timescale results, a voxel's mean timescale selectivity $\overline{P}$ is estimated by averaging an approach's

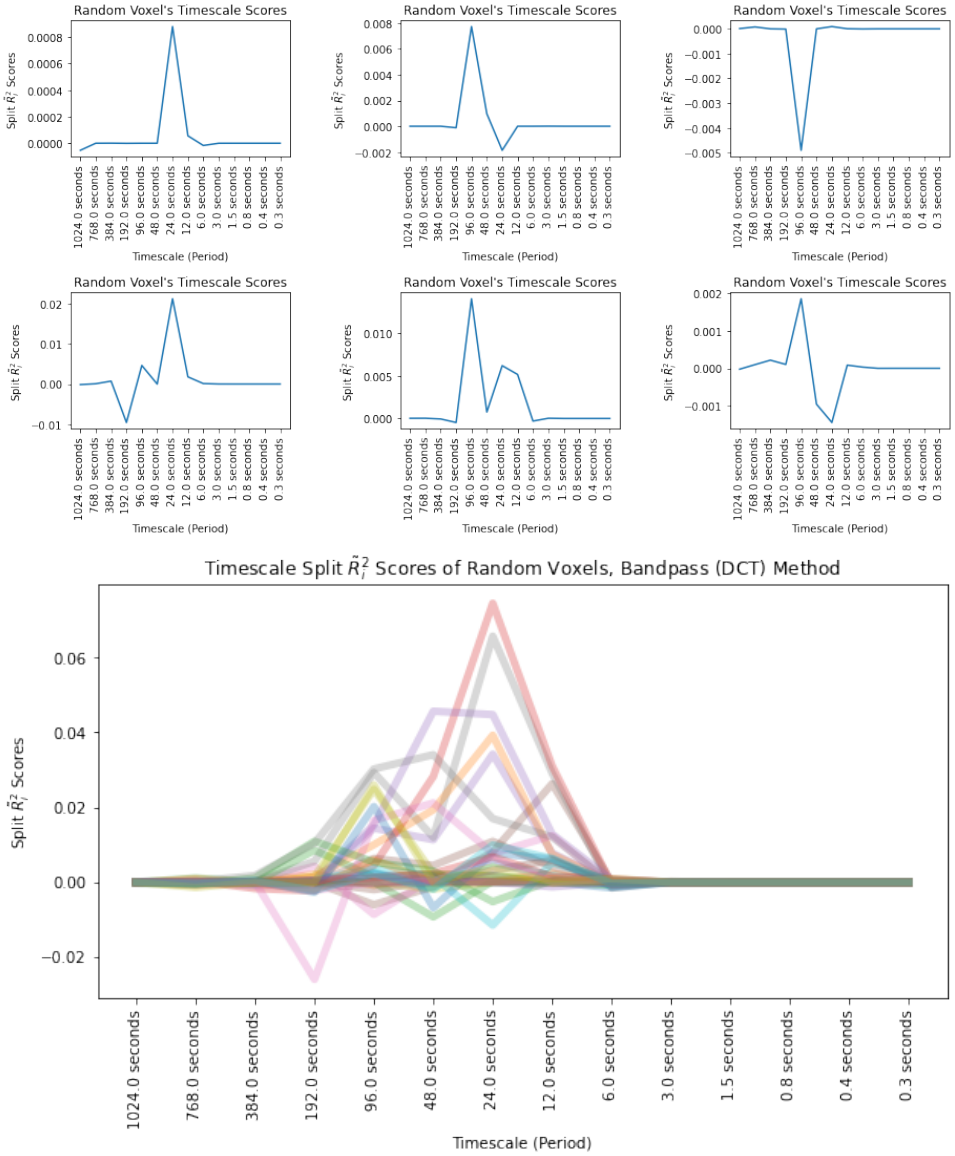

Figure 3.4: Split R_i^2 scores for randomly-selected voxels, using the **Bandpass (DCT)** method. In the second plot, each voxel is represented by a line of distinct color.

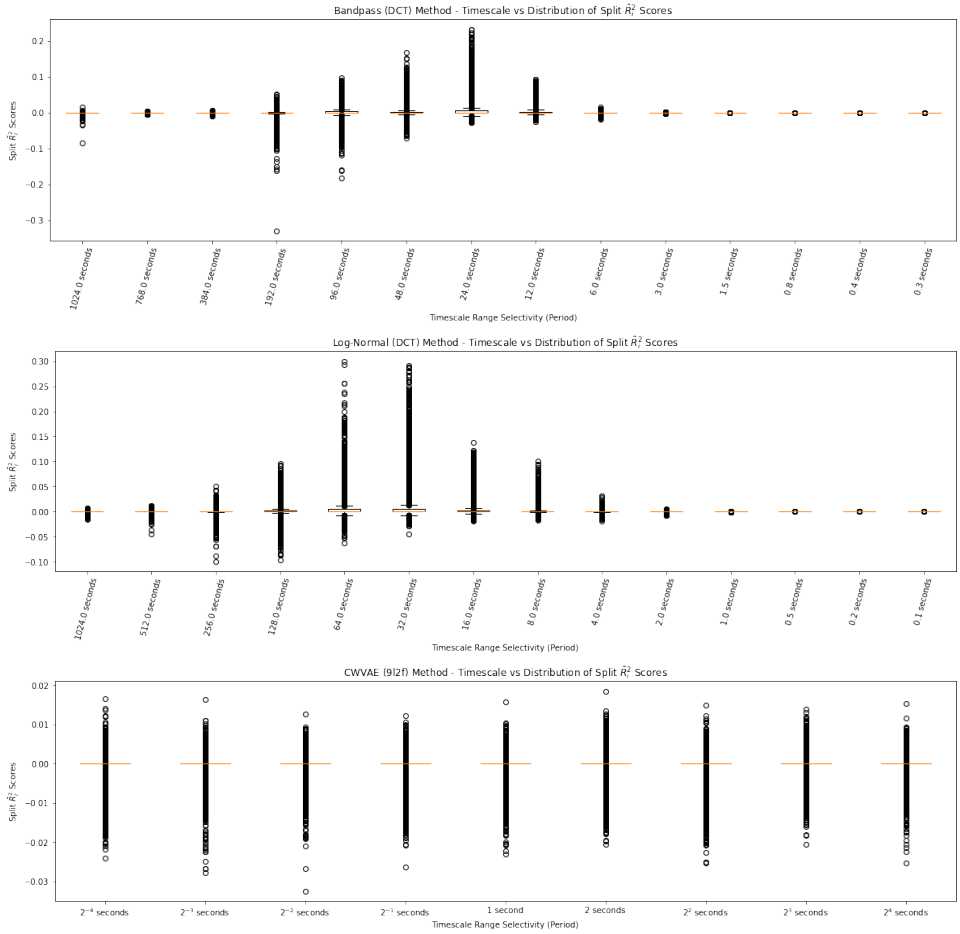

Figure 3.5: Boxplot of each method's split R_i^2 scores

predefined timescale periods P_i. Each period, corresponding to a feature space $i = \{1, ..., m\}$, is scaled by their corresponding relative $\tilde{R}^2_{i,scaled}$ scores (Equation 3.5).

$$\overline{P} = \sum_{1}^{m} \frac{1}{\tilde{R}^2_{i,scaled}} P_i \tag{3.5}$$

For visualization, the per-voxel weighted averages are converted to colors and mapped to vertices across the cortex (Figures 3.7, 3.8, 3.9). This was accomplished by the Pycortex Python-based toolkit [35, 62]. For all approaches, the relative distribution of voxel-timescale selectivity, across the cortical surface, exhibits remarkable similarities with the distributions found in the natural language domain [13, 55]. When

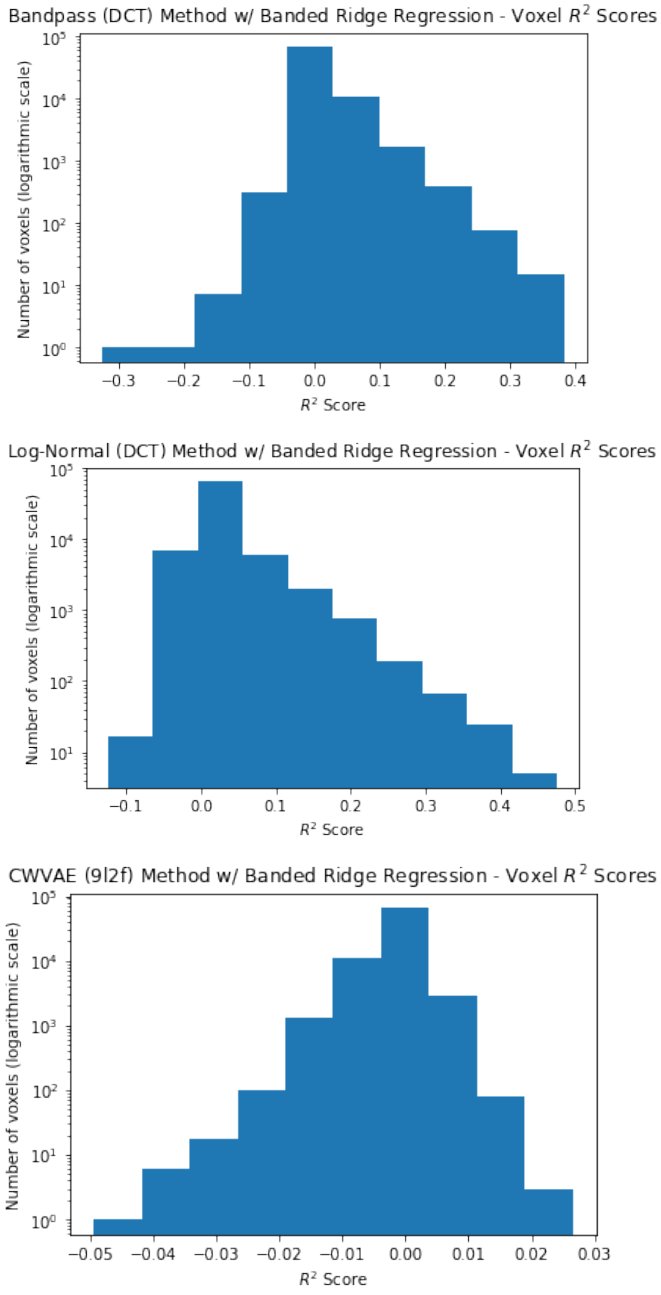

Figure 3.6: Histograms of each method's joint R^2 scores

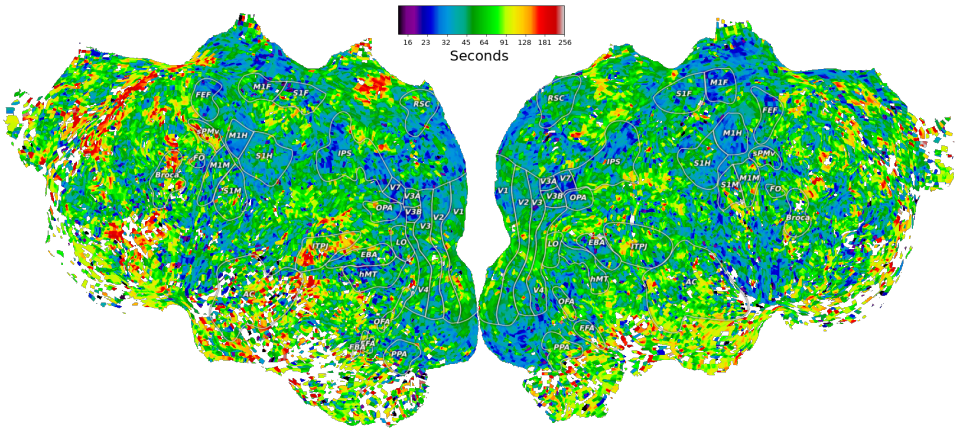

Figure 3.7: **Bandpass (DCT)** Method - Flatmap of Weighted Averages

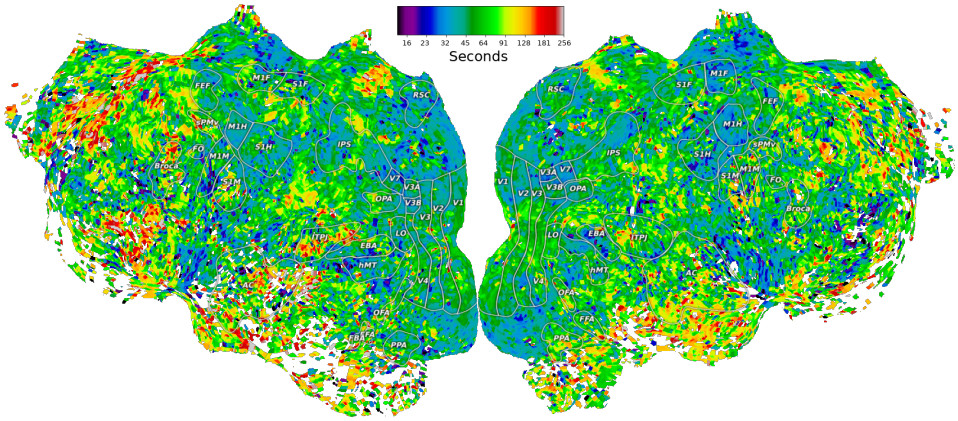

Figure 3.8: **Log-Normal DCT** Method - Flatmap of Weighted Averages

mapped to colors in a logarithmic range, the Early Visual Cortex exhibits a selectivity for short-to-medium relative timescales during both driving and reading [13]. Similarly, all modalities preferentially represent information at relatively longer timescales in the Prefrontal Cortex, Temporoparietal Junction, and Medial Parietal Cortex.

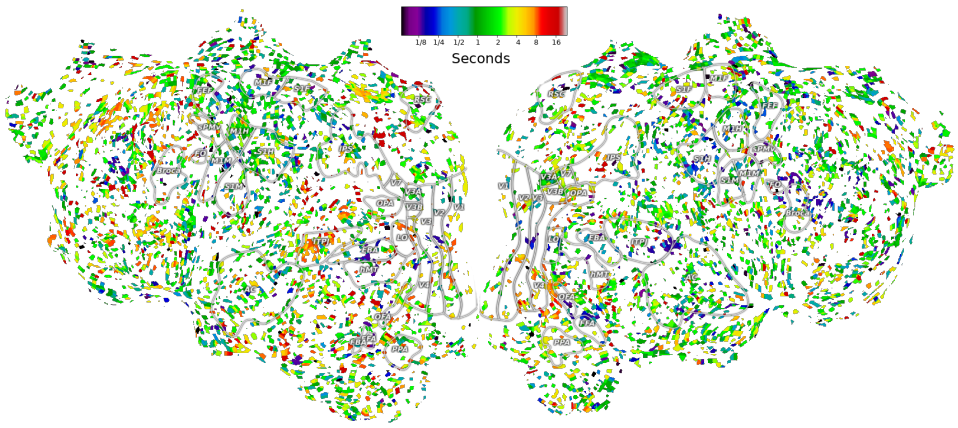

Figure 3.9: **CWVAE** Method - Flatmap of Weighted Averages

The Distribution of Driving Timescales is Consistent with Gradients Observed in Previous Research

For the timescales created from the two DCT filter approaches, their distributions both roughly exhibit a posterior-anterior gradient of progressively longer timescales. The approximate voxel-timescale selectivity increases in an outward-radial fashion, away from the Early Visual Cortex and towards the Ventral, Anterior, and Dorsal directions. This gradient corroborates with those observed from stimulus scrambling experiments [66] and temporal autocorrelation decay [91].

Significant Representation of Predictive Priors is Distributed Across the Cortical Surface

Despite having a lower maximum timescale period, the distribution of timescale selectivity, for predictive priors, exhibits relative timescale patterns that are similar to those from the two DCT filter approaches and the natural language modalities [13, 55]. Relatively longer timescale selectivity is exhibited in the Prefrontal Cortex, Temporoparietal Junction, and Medial Parietal Cortex.

3.6 Discussion

A Diffusion Map Uncovers a Plausible Representation of Cortical Hierarchical Organization from Timescale Selectivity

Diffusion maps are a nonlinear dimensionality reduction technique that can uncover an underlying lower-dimensional manifold that embeds the dataset [16]. This involves defining the Euclidean distance between points as their diffusion distance in the original feature space. The diffusion distance is a measurement of graph connectivity in a time-dependent diffusion process that can uncover geometric structures, of the dataset, at different scales [20].

A particular implementation [17, 38, 39, 92] uses Scikit-Learn [10, 88] k-nearest neighbor search [97], with k indirectly serving as the diffusion pseudotime, in order to calculate diffusion distances. For each of n total vectors, the split feature-space $R^2_{i,scaled}$ scores serve as the coordinates of a multi-dimensional point. With a reasonably high k-value, limited by memory constraints, we can approach $k = N$.

With the R^2_i scores from the band-pass filter method, the first-dimension embedding values of the Diffusion map yields a plausible hierarchical organization (Figures 3.10, 3.11, 3.13). For the voxels in the Early Visual Cortex, the presence of low embedding values generally decrease and are overtaken by increasingly higher values from the V1 region to V4. This trend continues into the occipital place area, parahippocampal place area, temporoparietal junction, and onward. The voxels in the prefrontal cortex region generally have the highest embedding values. Further investigation is needed for the interpretation of embedding coordinate values.

Conclusion

We explore different methods, involving DCT filters and CWVAE, of creating multiple feature spaces. Each feature space contains information within a different range of timescales. Weighted averages of the resulting split R^2_i scores and Diffusion map coordinates are calculated from the various voxelwise models. The results produce flatmaps that reveal relative mean timescale selectivity distributions corroborating with previous research.

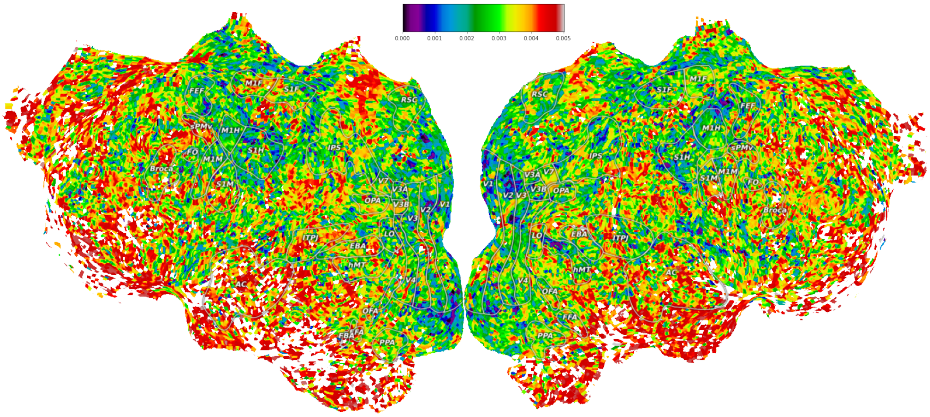

Figure 3.10: **Bandpass (DCT)** Method's Flatmap of Diffusion Embedding Values. Importantly the values are flipped. Given a voxel $i = \{0, ..., n\}$ and its first Diffusion Component value $DC_{1,i}$: $FlatmapValue_i = \max(DC_{1,0}, ..., DC_{1,n}) - DC_{1,i}$

.

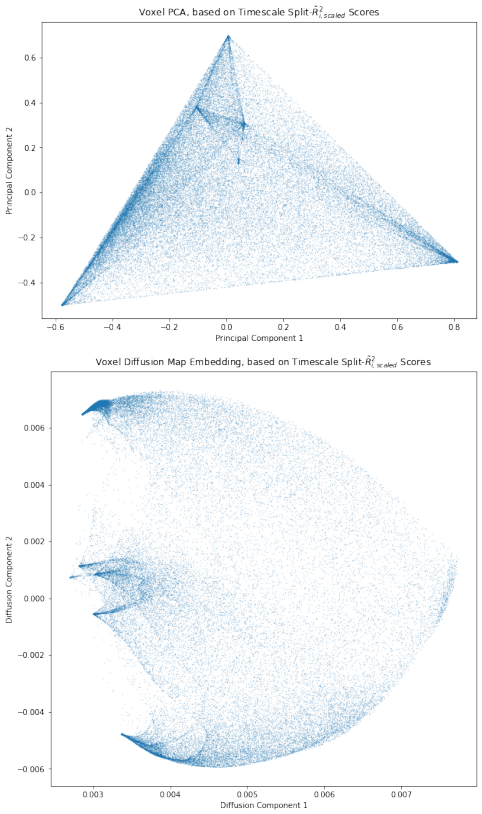

Figure 3.11: Scatter plots of Voxel Diffusion Map Components and Principal Components, using the **Bandpass (DCT) Method**. Principal Component Analysis is a linear dimensionality reduction technique [48, 87], implemented by Scikit-learn [10, 88]. The Diffusion Map algorithm is able to flatten the PCA manifold and push out the areas of high-density towards the edges of the manifold.

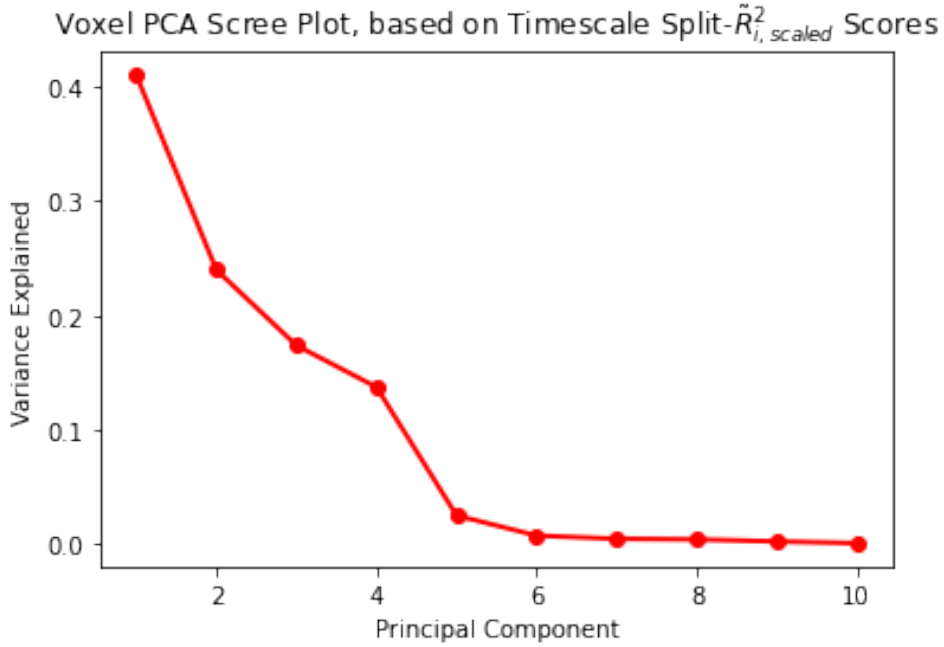

Figure 3.12: Scree plot of the Voxel Principal Components, using the **Bandpass (DCT) Method**. This shows that the first two principal components can explain a little over 60% of the variance.

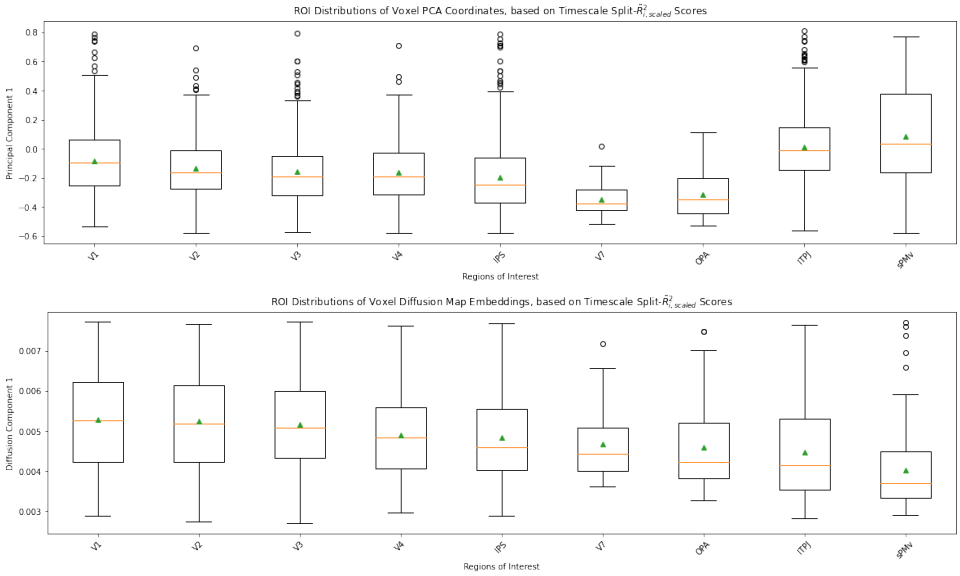

Figure 3.13: Boxplots of the voxels' Diffusion Map First Component and Principal Component, using the **Bandpass (DCT) Method**. The distribution of Diffusion Map First Components for voxels, within regions V1 through V4, display a clear linear trend for its medians (red lines) and means (green triangles).

Bibliography

[1] Martın Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Pulkit Agrawal et al. "Convolutional Neural Networks Mimic the Hierarchy of Visual Representations in the Human Brain". In: ().

[3] N. Ahmed, T. Natarajan, and K.R. Rao. "Discrete Cosine Transform". In: *IEEE Transactions on Computers* C-23.1 (1974), pp. 90–93. DOI: 10.1109/T-C.1974.223784.

[4] Alexander G. Anderson et al. "High-acuity vision from retinal image motion". In: *Journal of Vision* 20.7 (July 2020), pp. 34–34. ISSN: 1534-7362. DOI: 10.1167/jov.20.7.34. eprint: https://arvojournals.org/arvo/content_public/journal/jov/938480/i1534-7362-722-1-06487.pdf. URL: https://doi.org/10.1167/jov.20.7.34.

[5] Pierre Baldi. "Autoencoders, Unsupervised Learning, and Deep Architectures". In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. Ed. by Isabelle Guyon et al. Vol. 27. Proceedings of Machine Learning Research. Bellevue, Washington, USA: PMLR, Feb. 2012, pp. 37–49. URL: https://proceedings.mlr.press/v27/baldi12a.html.

[6] Andrei Belitski et al. "Low-frequency local field potentials and spikes in primary visual cortex convey independent visual information". In: *Journal of Neuroscience* 28.22 (2008), pp. 5696–5709.

[7] Boost. *The Boost C++ Libraries*. URL: https://www.boost.org/.

[8] Thackery I Brown et al. "Prospective representation of navigational goals in the human hippocampus". In: *Science (New York, N.Y.)* 352.6291 (June 2016), pp. 1323–1326. ISSN: 0036-8075. DOI: 10.1126/science.aaf0784. URL: https://doi.org/10.1126/science.aaf0784.

[9] Bradley R Buchsbaum and Mark D'Esposito. "The search for the phonological store: from loop to convolution". In: *Journal of Cognitive Neuroscience* 20.5 (2008), pp. 762–778.

[10] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[11] Gyorgy Buzsaki and Andreas Draguhn. "Neuronal oscillations in cortical networks". In: *science* 304.5679 (2004), pp. 1926–1929.

[12] Ryan T Canolty et al. "High gamma power is phase-locked to theta oscillations in human neocortex". In: *science* 313.5793 (2006), pp. 1626–1628.

[13] Catherine Chen et al. "The Cortical Representation of Language Timescales is Shared between Reading and Listening". In: *bioRxiv* (2023). DOI: 10.1101/2023.01.06.522601. eprint: https://www.biorxiv.org/content/early/2023/01/06/2023.01.06.522601.full.pdf. URL: https://www.biorxiv.org/content/early/2023/01/06/2023.01.06.522601.

[14] Guifen Chen et al. "How vision and movement combine in the hippocampal place code". In: *Proceedings of the National Academy of Sciences* 110.1 (2013), pp. 378–383. ISSN: 0027-8424. DOI: 10.1073/pnas.1215834110. eprint: https://www.pnas.org/content/110/1/378.full.pdf. URL: https://www.pnas.org/content/110/1/378.

[15] Alex Clark. *Pillow*. URL: https://python-pillow.org/.

[16] Ronald R Coifman and Stéphane Lafon. "Diffusion maps". In: *Applied and computational harmonic analysis* 21.1 (2006), pp. 5–30.

[17] Ronald R Coifman et al. "Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps". In: *Proceedings of the national academy of sciences* 102.21 (2005), pp. 7426–7431.

[18] Henk R. Cremers, Tor D. Wager, and Tal Yarkoni. "The relation between statistical power and inference in fMRI". In: *PLOS ONE* 12.11 (Nov. 2017), pp. 1–20. DOI: 10.1371/journal.pone.0184923. URL: https://doi.org/10.1371/journal.pone.0184923.

[19] Peter Dayan et al. "The helmholtz machine". In: *Neural computation* 7.5 (1995), pp. 889–904.

[20] J De la Porte et al. "An introduction to diffusion maps". In: *Proceedings of the 19th symposium of the pattern recognition association of South Africa (PRASA 2008), Cape Town, South Africa*. 2008, pp. 15–25.

[21] John Desmond and Gary Glover. "Estimating sample size in functional MRI (fMRI) neuroimaging studies". In: *Journal of neuroscience methods* 118 (Sept. 2002), pp. 115–28. DOI: 10.1016/S0165-0270(02)00121-8.

[22] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[23] Carl Doersch. "Tutorial on variational autoencoders". In: *arXiv preprint arXiv:1606.05908* (2016).

[24] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[25] T. Dupré La Tour, M. Visconti di Oleggio Castello, and J. L. Gallant. *Voxelwise modeling tutorials: an encoding model approach to functional MRI analysis. In preparation.* URL: https://gallantlab.github.io/voxelwise_tutorials/index.html.

[26] Tom Dupré la Tour et al. "Feature-space selection with banded ridge regression". In: *NeuroImage* 264 (2022), p. 119728. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2022.119728. URL: https://www.sciencedirect.com/science/article/pii/S1053811922008497.

[27] Austin English and Winetrick authors. *Wintricks*. URL: https://github.com/Winetricks/winetricks.

[28] József Fiser et al. "Statistically optimal perception and learning: from behavior to neural representations". In: *Trends in Cognitive Sciences* 14.3 (2010), pp. 119–130. ISSN: 1364-6613. DOI: https://doi.org/10.1016/j.tics.2010.01.003. URL: https://www.sciencedirect.com/science/article/pii/S1364661310000045.

[29] Python Software Foundation. *Python*. URL: https://www.python.org/.

[30] Paul W Frankland, Sheena A Josselyn, and Stefan Köhler. "The neurobiological foundation of memory retrieval". In: *Nature neuroscience* 22.10 (2019), pp. 1576–1585.

[31] Karl Friston. "A theory of cortical responses". In: *Philosophical transactions of the Royal Society B: Biological sciences* 360.1456 (2005), pp. 815–836.

[32] Karl Friston. "The history of the future of the Bayesian brain". In: *NeuroImage* 62.2 (2012). 20 YEARS OF fMRI, pp. 1230–1233. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2011.10.004. URL: https://www.sciencedirect.com/science/article/pii/S1053811911011657.

[33] Karl Friston, James Kilner, and Lee Harrison. "A free energy principle for the brain". In: *Journal of Physiology-Paris* 100.1 (2006). Theoretical and Computational Neuroscience: Understanding Brain Functions, pp. 70–87. ISSN: 0928-4257. DOI: https://doi.org/10.1016/j.jphysparis.2006.10.001. URL: https://www.sciencedirect.com/science/article/pii/S092842570600060X.

[34] Epic Games. *Unreal Engine 4*. URL: https://www.unrealengine.com.

[35] James S. Gao et al. "Pycortex: an interactive surface visualizer for fMRI". In: *Frontiers in Neuroinformatics* 9 (2015). ISSN: 1662-5196. DOI: 10.3389/fninf.2015.00023. URL: https://www.frontiersin.org/articles/10.3389/fninf.2015.00023.

[36] Ryan Gregg. *Nem's Tools*. URL: https://nemstools.github.io/.

[37] Antonin Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Boston, Massachusetts: Association for Computing Machinery, 1984, pp. 47–57. ISBN: 0897911288. DOI: 10.1145/602259.602266. URL: https://doi.org/10.1145/602259.602266.

[38] Laleh Haghverdi, Florian Buettner, and Fabian J. Theis. "Diffusion maps for high-dimensional single-cell analysis of differentiation data". In: *Bioinformatics* 31.18 (May 2015), pp. 2989–2998. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btv325. eprint: https://academic.oup.com/bioinformatics/article-pdf/31/18/2989/17090122/btv325.pdf. URL: https://doi.org/10.1093/bioinformatics/btv325.

[39] Laleh Haghverdi et al. "Diffusion pseudotime robustly reconstructs lineage branching". In: *Nature methods* 13.10 (2016), pp. 845–848.

[40] Stephen José Hanson, Alessandro D Gagliardi, and Catherine Hanson. "Solving the brain synchrony eigenvalue problem: conservation of temporal dynamics (fMRI) over subjects doing the same task". In: *Journal of computational neuroscience* 27.1 (2009), pp. 103–114.

[41] Uri Hasson, Janice Chen, and Christopher J Honey. "Hierarchical process memory: memory as an integral component of information processing". In: *Trends in cognitive sciences* 19.6 (2015), pp. 304–313.

[42] Uri Hasson, Rafael Malach, and David J. Heeger. "Reliability of cortical activity during natural stimulation". In: *Trends in Cognitive Sciences* 14.1 (2010), pp. 40–48. ISSN: 1364-6613. DOI: https://doi.org/10.1016/j.tics.2009.10.011. URL: https://www.sciencedirect.com/science/article/pii/S1364661309002393.

[43] Uri Hasson et al. "A hierarchy of temporal receptive windows in human cortex". In: *Journal of Neuroscience* 28.10 (2008), pp. 2539–2550.

[44] Uri Hasson et al. "Enhanced intersubject correlations during movie viewing correlate with successful episodic encoding". In: *Neuron* 57.3 (2008), pp. 452–462.

[45] Uri Hasson et al. "Intersubject synchronization of cortical activity during natural vision". In: *science* 303.5664 (2004), pp. 1634–1640.

[46] Jakob D. Havtorn et al. *Benchmarking Generative Latent Variable Models for Speech*. 2022. DOI: 10.48550/ARXIV.2202.12707. URL: https://arxiv.org/abs/2202.12707.

[47] Biyu J He et al. "The temporal structures and functional significance of scale-free brain activity". In: *Neuron* 66.3 (2010), pp. 353–369.

[48] Joe Hellerstein, Bin Yu, and Fernando Perez. *Learning Data Science - Dimensionality Reduction and PCA*. URL: https://learningds.org/ch/a11/contributors.html.

[49] Hermann von Helmholtz. "Concerning the perceptions in general". In: *Visual perception* (2001), pp. 24–44.

[50] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[51] Arthur E Hoerl and Robert W Kennard. "Ridge regression: Biased estimation for nonorthogonal problems". In: *Technometrics* 12.1 (1970), pp. 55–67.

[52] Patrik Hoyer and Aapo Hyvärinen. "Interpreting Neural Response Variability as Monte Carlo Sampling of the Posterior". In: *Advances in Neural Information Processing Systems*. Ed. by S. Becker, S. Thrun, and K. Obermayer. Vol. 15. MIT Press, 2002. URL: https://proceedings.neurips.cc/paper/2002/file/a486cd07e4ac3d270571622f4f316ec5-Paper.pdf.

[53] Alexander Huth et al. "A Continuous Semantic Space Describes the Representation of Thousands of Object and Action Categories across the Human Brain". In: *Neuron* 76 (Dec. 2012), pp. 1210–24. DOI: 10.1016/j.neuron.2012.10.014.

[54] Iiro P Jääskeläinen et al. "Inter-subject synchronization of prefrontal cortex hemodynamic activity during natural viewing". In: *The open neuroimaging journal* 2 (2008), p. 14.

[55] Shailee Jain et al. "Interpretable multi-timescale models for predicting fMRI responses to continuous natural speech". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 13738–13749. URL: https://proceedings.neurips.cc/paper/2020/file/9e9a30b74c49d07d8150c8c83b1ccf07-Paper.pdf.

[56] Alexandre Julliard and Wine authors. *Wine*. URL: https://www.winehq.org/.

[57] M.S. Keshner. "1/f noise". In: *Proceedings of the IEEE* 70.3 (1982), pp. 212–218. DOI: 10.1109/PROC.1982.12282.

[58] Stefan J. Kiebel, Jean Daunizeau, and Karl J. Friston. "A Hierarchy of Time-Scales and the Brain". In: *PLOS Computational Biology* 4.11 (Nov. 2008), pp. 1–12. DOI: 10.1371/journal.pcbi.1000209. URL: https://doi.org/10.1371/journal.pcbi.1000209.

[59] N Kopell et al. "Gamma rhythms and beta rhythms have different synchronization properties". In: *Proceedings of the National Academy of Sciences* 97.4 (2000), pp. 1867–1872.

[60] Philipp Krahenbuhl. "Free Supervision from Video Games". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2955–2964. DOI: 10.1109/CVPR.2018.00312.

[61] The Gallant Lab. *Himalaya*. URL: https://github.com/gallantlab/himalaya.

[62] The Gallant Lab. *Pycortex*. URL: https://gallantlab.github.io/pycortex/.

[63] The Gallant Lab. *Voxelwise Modeling Tutorials*. URL: https://gallantlab.github.io/voxelwise_tutorials/voxelwise_modeling.html.

[64] Peter Lakatos et al. "An oscillatory hierarchy controlling neuronal excitability and stimulus processing in the auditory cortex". In: *Journal of neurophysiology* 94.3 (2005), pp. 1904–1911.

[65] Tai Sing Lee and David Mumford. "Hierarchical Bayesian inference in the visual cortex". In: *JOSA A* 20.7 (2003), pp. 1434–1448.

[66] Yulia Lerner et al. "Topographic mapping of a hierarchy of temporal receptive windows using a narrated story". In: *Journal of Neuroscience* 31.8 (2011), pp. 2906–2915.

[67] Henry W Lin and Max Tegmark. "Critical behavior in physics and probabilistic formal languages". In: *Entropy* 19.7 (2017), p. 299.

[68] Nikos K Logothetis and Josef Pfeuffer. "On the nature of the BOLD fMRI contrast mechanism". In: *Magnetic resonance imaging* 22.10 (2004), pp. 1517–1531.

[69] Nikos K Logothetis and Brian A Wandell. "Interpreting the BOLD signal". In: *Annu. Rev. Physiol.* 66 (2004), pp. 735–769.

[70] Fredrik Lundh and Secret Labs AB. *Python Imaging Library*. URL: `http://www.pythonware.com/products/pil/`.

[71] Shivangi Mahto et al. "Multi-timescale representation learning in LSTM language models". In: *arXiv preprint arXiv:2009.12727* (2020).

[72] Mauro Manassi and David Whitney. "Illusion of visual stability through active perceptual serial dependence". In: *Science Advances* 8.2 (2022), eabk2480. DOI: `10.1126/sciadv.abk2480`. eprint: `https://www.science.org/doi/pdf/10.1126/sciadv.abk2480`. URL: `https://www.science.org/doi/abs/10.1126/sciadv.abk2480`.

[73] Shimon Marom. "Neural timescales or lack thereof". In: *Progress in neurobiology* 90.1 (2010), pp. 16–28.

[74] Pawel J Matusz et al. *Are we ready for real-world neuroscience?* 2019.

[75] Ferenc Mechler et al. "Robust temporal coding of contrast by V1 neurons for transient but not for steady-state stimuli". In: *Journal of Neuroscience* 18.16 (1998), pp. 6583–6598.

[76] Microsoft. *.NET Framework*. URL: `https://dotnet.microsoft.com/en-us/`.

[77] David Mumford. "On the computational architecture of the neocortex". In: *Biological cybernetics* 66.3 (1992), pp. 241–251.

[78] Jeanette A Mumford and Thomas E Nichols. "Power calculation for group fMRI studies accounting for arbitrary design and temporal autocorrelation". In: *Neuroimage* 39.1 (2008), pp. 261–268.

[79] Thomas Naselaris et al. "A voxel-wise encoding model for early visual areas decodes mental images of remembered scenes". In: *Neuroimage* 105 (2015), pp. 215–228.

[80] Thomas Naselaris et al. "Encoding and decoding in fMRI". In: *NeuroImage* 56.2 (2011). Multivariate Decoding and Brain Reading, pp. 400–410. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2010.07.073. URL: https://www.sciencedirect.com/science/article/pii/S1053811910010657.

[81] Anwar O. Nunez-Elizalde, Alexander G. Huth, and Jack L. Gallant. "Voxelwise encoding models with non-spherical multivariate normal priors". In: *NeuroImage* 197 (2019), pp. 482–492. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2019.04.012. URL: https://www.sciencedirect.com/science/article/pii/S1053811919302988.

[82] Lars Nyberg et al. "Reactivation of encoding-related brain activity during memory retrieval". In: *Proceedings of the National Academy of Sciences* 97.20 (2000), pp. 11120–11124.

[83] Keiron O'Shea and Ryan Nash. "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458* (2015).

[84] *OpenDRIVE Format Specification, Rev. 1.5, 1st ed.* VIRES Simulationstechnologie GmbH, Feb. 2019.

[85] Satu Palva and J Matias Palva. "Roles of brain criticality and multiscale oscillations in temporal predictions for sensorimotor processing". In: *Trends in neurosciences* 41.10 (2018), pp. 729–743.

[86] Joel Pearson et al. "Mental imagery: functional mechanisms and clinical applications". In: *Trends in cognitive sciences* 19.10 (2015), pp. 590–602.

[87] Karl Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572.

[88] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[89] Bradley R Postle. "Working memory as an emergent property of the mind and brain". In: *Neuroscience* 139.1 (2006), pp. 23–38.

[90] Rajesh PN Rao and Dana H Ballard. "Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects". In: *Nature neuroscience* 2.1 (1999), pp. 79–87.

[91] Ryan V Raut, Abraham Z Snyder, and Marcus E Raichle. "Hierarchical dynamics as a macroscopic organizing principle of the human brain". In: *Proceedings of the National Academy of Sciences* 117.34 (2020), pp. 20890–20897.

[92] redst4r. *pyDiffusionMaps*. URL: https://github.com/redst4r/pyDiffusionMaps.

[93] Stephan R. Richter, Hassan Abu AlHaija, and Vladlen Koltun. "Enhancing Photorealism Enhancement". In: *arXiv:2105.04619* (2021).

[94] Carlo Rovelli. *The Order of Time*. New York, NY: Riverhead Books, 2018.

[95] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362. ISBN: 026268053X.

[96] Vaibhav Saxena, Jimmy Ba, and Danijar Hafner. "Clockwork Variational Autoencoders". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 29246–29257. URL: https://proceedings.neurips.cc/paper/2021/file/f490d0af974fedf90cb0f1edce8e3dd5-Paper.pdf.

[97] Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. "Nearest-neighbor methods in learning and vision". In: *IEEE Trans. Neural Networks* 19.2 (2008), p. 377.

[98] Katherine R Sherrill et al. "Hippocampus and retrosplenial cortex combine path integration signals for successful navigation". In: *The Journal of neuroscience : the official journal of the Society for Neuroscience* 33.49 (Dec. 2013), pp. 19304–19313. ISSN: 0270-6474. DOI: 10.1523/jneurosci.1825-13.2013. URL: https://europepmc.org/articles/PMC3850045.

[99] Eun Song et al. "Role of active movement in place-specific firing of hippocampal neurons". In: *Hippocampus* 15 (Jan. 2005), pp. 8–17. DOI: 10.1002/hipo.20023.

[100] Peter Mattis Spencer Kimball. *GNU Image Manipulation Program*. URL: https://www.gimp.org/.

[101] Jonathan Stone, Bogdan Dreher, and Audie Leventhal. "Hierarchical and parallel mechanisms in the organization of visual cortex". In: *Brain Research Reviews* 1.3 (1979), pp. 345–394.

[102] Thomas Suddendorf, Donna Rose Addis, and Michael C Corballis. "Mental time travel and the shaping of the human mind". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 364.1521 (2009), pp. 1317–1324.

[103] Thomas Suddendorf and Michael C Corballis. "The evolution of foresight: What is mental time travel, and is it unique to humans?" In: *Behavioral and brain sciences* 30.3 (2007), pp. 299–313.

[104] P Szendro, Gy Vincze, and A Szasz. "Pink-noise behaviour of biosystems". In: *European Biophysics Journal* 30.3 (2001), pp. 227–231.

[105] *The moth radio hour*. 2020. URL: `https://themoth.org`.

[106] Valve. *Source*. URL: `https://developer.valvesoftware.com/wiki/Source`.

[107] Turle Rock Studios Valve. *Counter-Strike: Source*. 2004. URL: `https://store.steampowered.com/app/240/CounterStrike_Source/`.

[108] David C Van Essen and John HR Maunsell. "Hierarchical organization and functional streams in the visual cortex". In: *Trends in neurosciences* 6 (1983), pp. 370–375.

[109] Gerd T Waldhauser, Verena Braun, and Simon Hanslmayr. "Episodic memory retrieval functionally relies on very rapid reactivation of sensory information". In: *Journal of Neuroscience* 36.1 (2016), pp. 251–260.

[110] Mark E Wheeler, Steven E Petersen, and Randy L Buckner. "Memory's echo: vivid remembering reactivates sensory-specific cortex". In: *Proceedings of the National Academy of Sciences* 97.20 (2000), pp. 11125–11129.

[111] Stephen M Wilson, Istvan Molnar-Szakacs, and Marco Iacoboni. "Beyond superior temporal cortex: intersubject correlations in narrative speech comprehension". In: *Cerebral cortex* 18.1 (2008), pp. 230–242.

[112] Michael C.-K. Wu, Stephen V. David, and Jack L. Gallant. "COMPLETE FUNCTIONAL CHARACTERIZATION OF SENSORY NEURONS BY SYSTEM IDENTIFICATION". In: *Annual Review of Neuroscience* 29.1 (2006). PMID: 16776594, pp. 477–505. DOI: `10.1146/annurev.neuro.29.051605.113024`. eprint: `https://doi.org/10.1146/annurev.neuro.29.051605.`

113024. URL: https://doi.org/10.1146/annurev.neuro.29.051605.113024.

[113] Haishan Yao et al. "Rapid learning in cortical coding of visual scenes". In: *Nature neuroscience* 10.6 (2007), pp. 772–778.

[114] Tianjiao Zhang. "Navigational representation in the human brain". PhD thesis. Berkeley, California: University of California, Berkeley, 2021.