**CUED - Engineering Tripos Part IIB 2024-2025** **Module Coursework**

| | | | |
|---|---|---|---|
| Module | 4G3 | Title of report | Computational Neuroscience - Assignment 1 |
| Date submitted: | 2/28/2025 | Assessment for this module is ☐ 100% / ☐ 25% coursework<br>of which this assignment forms _____ % | |

**UNDERGRADUATE and POST GRADUATE STUDENTS**

| | | | | |
|---|---|---|---|---|
| Name: | Mac Walker | College: | | ☐ Undergraduate<br>☒ Post graduate |

| Feedback to the student<br>☐ **See also comments in the text** | | Very good | **Good** | Needs improvmt |
|---|---|---|---|---|
| C O N T E N T | **Completeness, quantity of content:**<br>Has the report covered all aspects of the lab? Has the analysis been carried out thoroughly? | | | |
| | **Correctness, quality of content**<br>Is the data correct? Is the analysis of the data correct? Are the conclusions correct? | | | |
| | **Depth of understanding, quality of discussion**<br>Does the report show a good technical understanding? Have all the relevant conclusions been drawn? | | | |
| | Comments: | | | |
| P R E S E N T A T I O N | **Attention to detail, typesetting and typographical errors**<br>Is the report free of typographical errors? Are the figures/tables/references presented professionally? | | | |
| | Comments: | | | |

Marker: Date:

# Computational Neuroscience - Assignment 1

BGN: 2265V

February 28, 2025

## Contents

# 1 Reinforcement Learning

## 1.1 Question 1

Using a lookup table, the value function is defined as:

$$\hat{V}^\pi(s) = f(s; w) = w_s,$$

where each state $s$ has its own parameter $w_s$. This is a special case of function approximation where the feature vector is one-hot:

$$\phi_k(s) = \begin{cases} 1 & \text{if } k = s, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the lookup table update is:

$$w_s \leftarrow w_s + \epsilon \left[ r_t + \gamma w_{s'} - w_s \right].$$

## 1.2 Question 2

For linear function approximation,

$$\hat{V}^\pi(s) = f(s; \mathbf{w}) = \sum_k w_k \phi_k(s).$$

The TD update is given by:

$$w_k \leftarrow w_k + \epsilon \left[ r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t) \right] \frac{\partial f(s_t, \mathbf{w}}{\partial w_k}$$

Since

$$\frac{\partial f(s_t, \mathbf{w})}{\partial w_k} = \phi_k(s_t),$$

the update becomes:

$$w_k \leftarrow w_k + \epsilon \left[ r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t) \right] \phi_k(s_t).$$

In the one-hot case, $\phi_k(s_t) = 1$ only for $k = s_t$, so this reduces to the lookup table update. (See Appendix for full treatment).

## 1.3 Question 3

### 1.3.1 (a) Modelling the Reward and Stimulus

In Figure 1, we see the reward and stimulus plotted against time. The stimulus occurs as a sharp spike at $t = 10s$, while the reward follows a Gaussian profile centred at $t = 20s$.

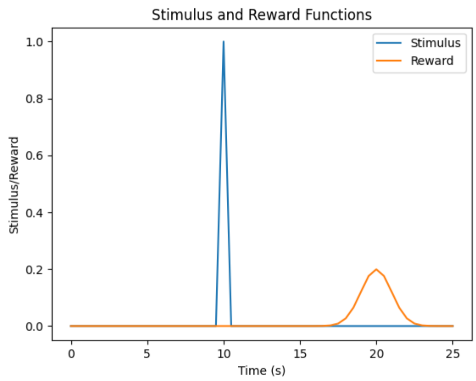


Figure 1: Reward and stimulus functions plotted against time (in seconds).

### 1.3.2 (b) Tapped Delay Line Simulation

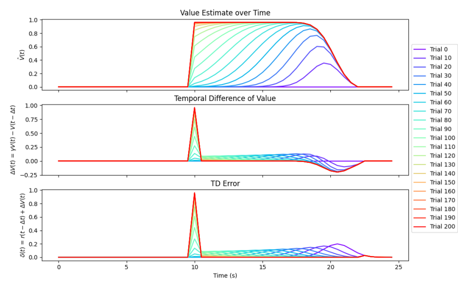


Figure 2: Tapped delay line representation

### 1.3.3 (c) & (d) Results Discussion

**(c) Value $\hat{V}(t)$.** Before the agent can learn from its environment, the value estimate $\hat{V}(t)$ is stable at $\hat{V}(t) = 0$ because the model hasn't interacted with the reward or stimulus yet. As the simulations progress, the value estimate begins by approximating the reward function, by being nearly Gaussian at the time of the reward. As the agent is conditioned to associate the stimulus with the reward, the value estimate tends towards a function that closely resembles the orthogonal nature of the $\delta$ function at the time of the stimulus, having a stable maximum value estimate until it slopes down to 0 again after the reward is delivered.

**Temporal difference $\Delta\hat{V}(t) = \hat{V}(t) - \hat{V}(t - \Delta t)$.**

Initially, $\Delta\hat{V}(t)$ is nearly zero, since $\hat{V}(t)$ itself is zero and does not change over time. Later, as $\hat{V}(t)$ transitions sharply from low to high right after the stimulus, $\Delta\hat{V}(t)$ exhibits a pronounced positive peak near $t = 10\,\text{s}$. As the agent learns, the temporal difference decreases at the point of the reward.

**TD error $\delta(t) = r(t - \Delta t) + \Delta\hat{V}(t)$.** Before the agent learns, $\delta(t)$ shows a rounded positive increase, similar to the reward signal, around $t = 20\,\text{s}$, since the reward is unpredicted. Over many trials, the agent learns that the stimulus at $10\,\text{s}$ predicts future reward, so $\delta(t)$'s peak moves earlier and more closely resembles the $\delta$ function that models the stimulus. By the final trials, a strong positive spike appears at the stimulus time (when the model updates from expecting little to expecting a substantial future reward), and little or no spike remains at $t = 20\,\text{s}$ (since the reward has become fully predicted).

**(d)** The TD error, $\delta(t)$, is taken to represent dopamine activity in the brain. Early in training, $\delta(t)$ (and thus dopamine activity) is at reward delivery, matching the surprise of an unpredicted reward. Over repeated trials, as the agent learns, the positive response shifts to the stimulus time, and $\delta(t)$ no longer spikes at reward delivery. This matches the empirical observation that dopamine neurons shift their burst of activity from the time of reward to the time of the predictive cue, which is supported by the $\delta(t)$ signal at trial 200 in the simulation.

## 1.4 Question 4

### 1.4.1 (a) Box-car Simulation

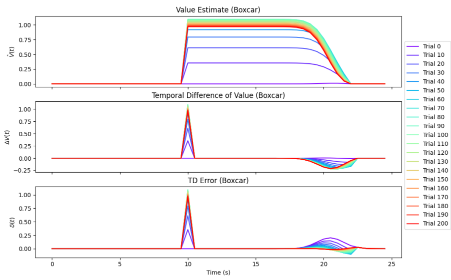


Figure 3: Box-car representation

### 1.4.2 (b), (c) & (d) Results Discussion

**(b)**

The transition in $\hat{V}(t)$ and $\delta(t)$, instead of shifting from the reward signal to the stimulus signal gradually, immediately follows the shape of the final distribution. This is because the boxcar feature sums over a window of past stimuli, the agent updates multiple weight components at once, altering the exact shape of the curves. This difference arises because the boxcar representation is effectively a coarser temporal encoding of the recent stimulus history.

In Figure 4, the initial and final simulations for both the boxcar and the tapped-delay line are plotted. As can be seen, the final simulation is the same for both the boxcar and tapped delay line simulation. This is because both representations are approximating the same true value function in the same environment.

Their initial differences arise because tapped-delay represents distinct time steps, allowing precise stimulus onset encoding, while boxcar accumulates past stimulus presence, causing a broader, less localised response.

**(c)** Despite the more coarse representation, the key observation remains that the large positive TD error $\delta(t)$ moves from reward time to the time of the stimulus as learning proceeds. Hence, as in the tapped-delay case, the model reproduces the empirical "dopamine shift": at the start of learning, $\delta(t)$ peaks when the reward arrives, but by the end of learning, that peak has shifted to the onset of the predictive cue, with little response at reward delivery.

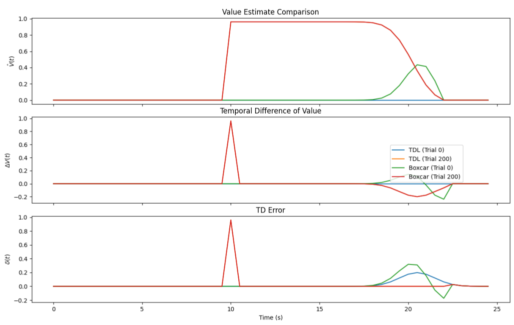


Figure 4: Difference between initial and final simulations

**(d)** The learning rate $\epsilon$ controls the step size in weight updates. In tapped-delay, each feature only detects the stimuli at a single time step (or single time step difference), enforcing localised updates. In the boxcar representation, features accumulate past stimulus occurences such that each weight influences multiple time steps. Thus, the boxcar representation amplifies the effect of weight updates, which without proper hyperparameter treatment, could lead to unstable learning. Hence, if $\epsilon$ was kept unchanged at 0.2, the weight updates may have gotten too large, leading to instability or oscillation in the learned value function. Reducing $\epsilon$ to 0.01 reduces this risk: this highlights the importance of hyperparameter tuning in model development.

## 1.5 Question 5

### 1.5.1 (a) Rewarded vs Unrewarded Stimuli Simulation

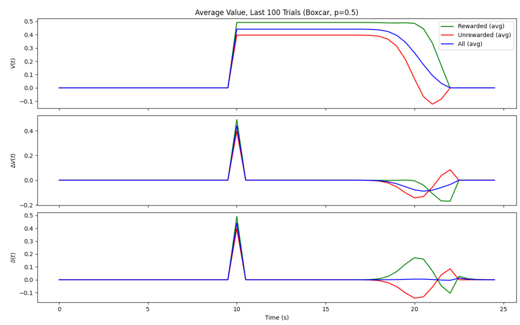


Figure 5: Average values between rewarded and unrewarded stimuli

### 1.5.2 (b) Results and Discussion

For all variables $\hat{V}(t)$, $\Delta\hat{V}(t)$ and $\delta(t)$, there is a clear difference between rewarded and unrewarded simulations (and trivially therefore between each of these and the average).

For $\hat{V}(t)$, the rewarded simulation average demonstrates the characteristic profile of a learned reward function, stabilising at approximately 0.45 before declining to 0 after the reward is completed. In contrast, the unrewarded signal stabilises around 0.4

but drops below 0 (negative reward) when the expected reward is absent, before returning to 0.

For both $\Delta\hat{V}(t)$ and $\delta(t)$, we observe characteristic patterns. The stimulus-associated spikes demonstrate that the stimulus has acquired predictive value through learning. At reward time, the rewarded signal shows a positive prediction error because receiving a reward with only 50% probability represents an outcome "better than expected." Conversely, the unrewarded signal displays a negative prediction error, indicating an outcome "worse than expected." This matches observed dopamine neuron behaviour in partial reinforcement, where neurons respond to both the reward-predicting stimulus and to the probabilistic reward itself.

#### 1.5.3 (c) Asymmetric Dynamic Range Simulation

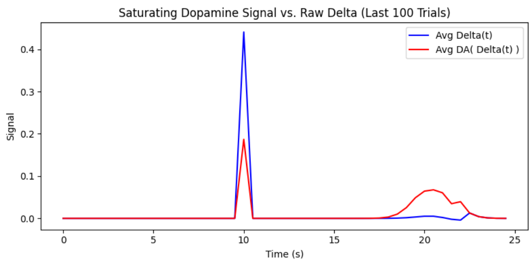


Figure 6: Assymetric dynamic range of dopamine

At stimulus onset, the large positive prediction error spike is compressed in the dopamine signal. At reward time, the slight fluctuations in prediction error are transformed into a more prominent positive bump in the dopamine signal. This occurs because positive prediction errors remain unscaled while negative errors are diminished by factor $\alpha = 6$.

This asymmetric encoding creates a biologically realistic model. Dopamine neurons show limited ability to signal negative prediction errors. They maintain sensitivity to small positive rewards. The model also prevents excessive firing for large positive prediction errors.

## 1.6 Question 6

The figure reveals how dopamine signaling changes with reward probability. The pattern shows the classic temporal difference learning behavior: the more predictable the reward (higher $p$), the more the dopamine response shifts to the earliest reliable predictor (the stimulus) and diminishes at the actual reward time.

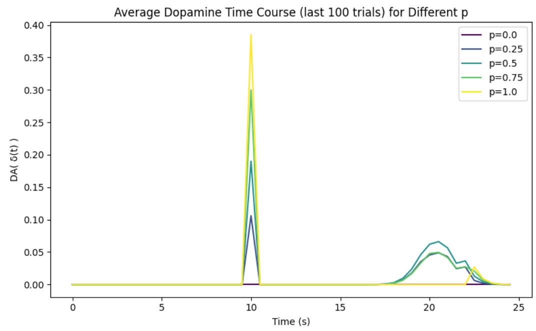


Figure 7: Different Reward Probability

## 1.7 Question 7

### 1.7.1 7. Dopamine Response as a Function of Reward Probability

The stimulus-related dopamine peak increases approximately linearly with reward probability, representing the expected value of future reward ($p \times$ reward_magnitude).

This demonstrates how the system learns to assign value to predictive stimuli in proportion to their reliability.

The reward-related dopamine peak, however, follows an inverted U-shape, maximising around $p = 0.5$ and diminishing as $p$ approaches either 0 or 1. This pattern precisely corresponds to reward prediction error. When reward is either certain ($p = 1$) or impossible ($p = 0$), there is no prediction error at reward time. Maximum uncertainty occurs at $p = 0.5$, producing the largest prediction errors on average. The inverted U-shape notably resembles the entropy function, suggesting dopamine signals not only prediction error but also encodes information about reward uncertainty.

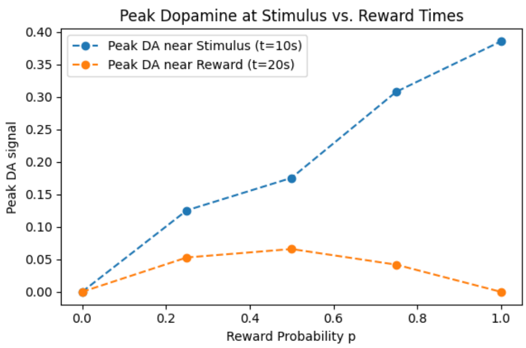


Figure 8: Average level of dopamine at stimulus and reward

## 1.8 Question 8

The relationship seen in Q7 may support the hypothesis that dopamine signals uncertainty rather than prediction error. This pattern could be explained through the prediction error. The uncertainty coding may emerge as a mathematical consequence of averaging asymmetrically encoded prediction errors across probabilistic outcomes. Our earlier analysis (Question 5b) revealed distinct dopamine responses for rewarded versus unrewarded trials - a finding inconsistent with a pure uncertainty signal, which would be identical regardless of outcome. A more nuanced interpretation is that the dopamine system simultaneously communicates multiple types of information, including both prediction error and uncertainty.

# 2 Representational Learning

## 2.1 Question 1

### 2.1.1 (i) Empirical Marginal Distributions

To assess whether the components are sparse, the empirical marginal probability distribution $p(x_k)$ for selected components $k$ ($k = [125, 129, 133, 137, 141]$ is estimated by forming histograms of the values $\{x_{n,k}\}_{n=1}^{N}$. We visualise the results in Figure 9

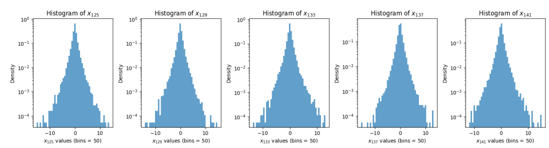

Figure 9: Histograms of selected latent components $x_k$.

Sparsely distributed component should exhibit a peaked distribution with heavy tails. The histograms reveal this, suggesting that they are sparsely activated, consistent with the sparse coding objective.

### 2.1.2 (ii) Joint and Conditional Distributions

We further investigate components 125 and 141, as they have a similar orientation and are located in a similar spatial area, as shown in Figure 10.

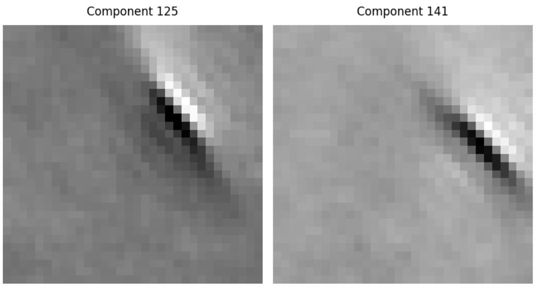


Figure 10: The two components, as represented by their raw patches.

To examine whether these components are independent, we estimate their joint distribution $p(x_{125}, x_{141})$ using a 2D histogram of the component pairs $\{x_{n,125}, x_{n,141}\}$.

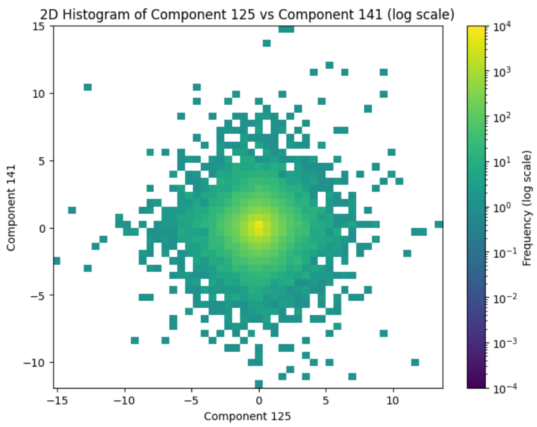


Figure 11: 2D histogram of component pairs $(x_{125}, x_{141})$. Color intensity represents frequency on a logarithmic scale.

If the components were truly independent, their joint distribution would be circularly symmetric, meaning the values of $x_{125}$ and $x_{141}$ would be uncorrelated. However, the diamond shape observed in Figure 11 suggests a dependency between these components, indicating that certain values of $x_{125}$ are more likely to co-occur with specific values of $x_{141}$.

Each slice of this histogram is then normalised to estimate the conditional probability $p(x_{141} \mid x_{125})$, which is visualised in Figure 12.

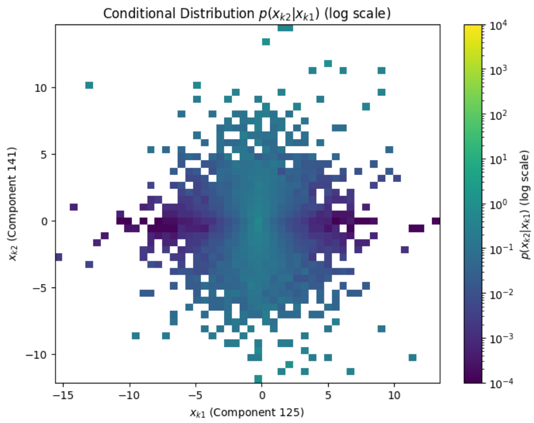


Figure 12: Conditional distribution $p(x_{141} \mid x_{125})$. Independence would imply uniform color distribution across $x_{125}$.

If $x_{125}$ and $x_{141}$ were independent, the conditional probability would be uniformly spread across different values of $x_{125}$. Instead, we observe a structured pattern where the variance of $x_{141}$ increases with the magnitude of $x_{125}$, suggesting a statistical dependency.

These findings indicate that while the sparse coding model enforces sparsity, it does not achieve full independence between components. This motivates the need for a more expressive model, where the variance of each component depends on latent variables.

## 2.2 Question 2

In Question 1, our analysis of joint and conditional distributions showed that the latent components are *not fully independent*. This challenges the assumptions of the ICA, suggesting a more expressive model is needed - we need to introduce some notion of 'coupling' the latent variables.

We use a conditional model where we sample $x_{n,k}$, given the other coordinates $\{x_{n,j} : j \neq k\}$, from a zero-mean Gaussian with a variance that depends *quadratically* on the other coordinates.

A full mathematical treatment of this problem can be seen in the Appendix.

The derivatives of the objective function, w.r.t. the parameters $\{a_{k,j}, b_k\}$, are:

$$\frac{\partial \mathcal{L}}{\partial a_{k,j}} = \sum_{n=1}^{N} \frac{\partial \ell_{n,k}}{\partial a_{k,j}} = \sum_{n=1}^{N} \left[ x_{n,j}^2 \cdot \frac{\partial \ell_{n,k}}{\partial (\sigma_{n,k}^2)} \right],$$

$$\frac{\partial \mathcal{L}}{\partial b_k} = \sum_{n=1}^{N} \frac{\partial \ell_{n,k}}{\partial b_k} = \sum_{n=1}^{N} \left[ \frac{\partial \ell_{n,k}}{\partial (\sigma_{n,k}^2)} \right].$$

The code for this implementation can be seen in the Appendix. A brief overview of the implementation is as follows. The parameters are initialised. The negative conditional log-likelihood is evaluated for each sample in the data, and then the chain rule is used on $\log a_{k,j}$ and $\log b_k$ to enforce the positivity constraints.The parameters are then updated iteratively using the L-BFGS-B algorithm.

Due to resource constraints, the full optimisiation was unable to be completed. Due to *report author foolishness*, the objective values were not saved during optimisation, only the updated parameter estimates.

## 2.3 Question 3

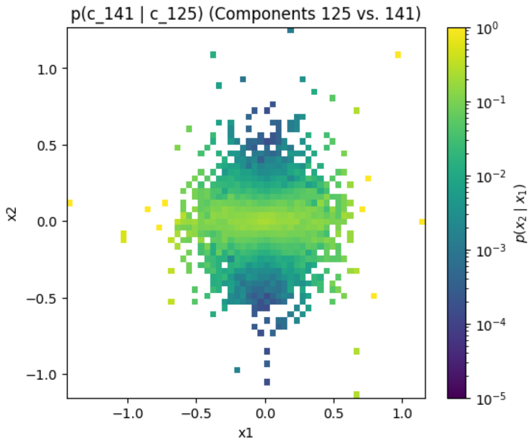


Figure 13: Conditional Distribution of normalised variables 125 and 141

As can be seen when comparing Figure 12 and Figure 13, after normalisation, the conditional probability distribution exhbiits more independence. This can be seen through the more circular shape of Figure 13 - although some dependence still remains. This could be due to optimisation not fully completing, as described previously.

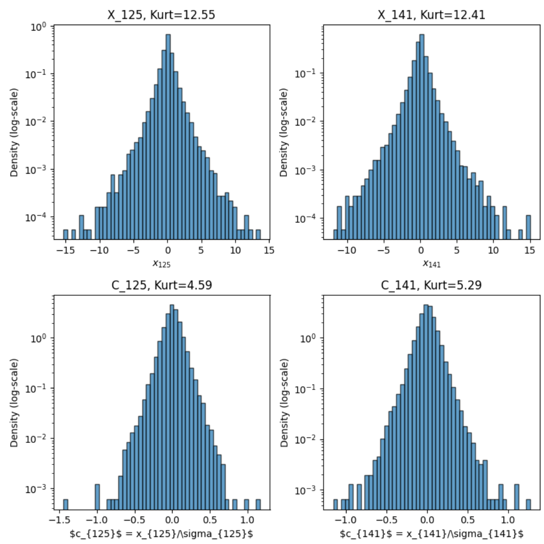


Figure 14: Excess kurtosis of latent variables 125 and 141

The excess kurtosis of the latent variables before and after normalisation, given in Figure 14 show that the kurtosis is reduced by normalisation, hence the normalised latent variables become more 'Gaussian' in shape. This can also be seen in the Figure, where extreme values are rarer and the peak of the graph is less steep. Hence, as the kurtosis has decreased, the latent variables are less sparse, because their interdependence has increased.

## 2.4 Question 4

In Figure 15, we see the 10 generative weights associated with the normalised latent variable $c_{125}$.

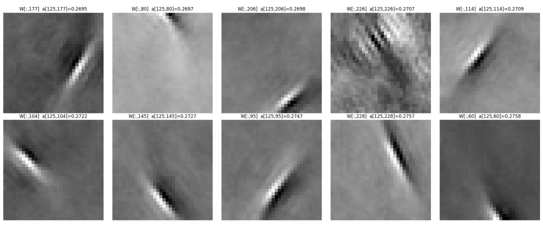

Figure 15: Top 10 generative weights associated with latent variable 125

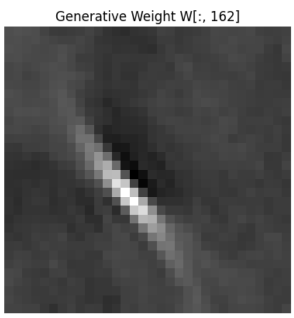


Figure 16: Generative weight of latent variable 162

In Figure 16 we see the generative weight associated with the normalised variable $c_{162}$, and in Figure 17 we observe its 10 greatest associated generative weights. In both cases, the weights corresponding to latent

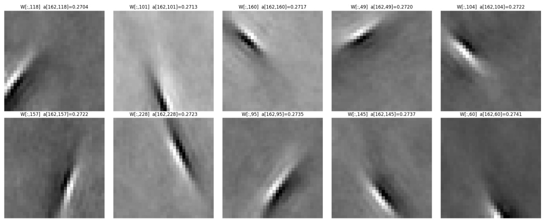

Figure 17: Top 10 generative weights associated with latent variable 162

variables with the highest covariance are either similarly oriented or perpendicular. This indicates that the model captures the fact that edges with similar or orthogonal orientations tend to co-occur, reflecting complementary features. Such a pattern is consistent with the inherent structure and redundancy in natural image statistics.

## 2.5 Question 5

The figure illustrates how a V1 neuron's firing rate is suppressed by a parallel masker but remains unaffected by an orthogonal one. In our model, the neuron's response can be identified with a single latent variable $x_k$.

When the surround contains parallel edges, aligned latent variables $x_j$ increase $\sigma_k^2$, reducing $x_k$ and suppressing the neuron's response. Orthogonal edges do not significantly affect $\sigma_k^2$, so the firing rate remains high. While Question 4 shows that orthogonal filters co-occur in natural images, this does not imply suppression in a single neuron.

Thus, parallel edges in the surround increase the conditional variance of $x_k$ (leading to suppression), while orthogonal edges do not. This aligns with orientation-specific surround modulation in V1.

# Appendix

A full exposition of the code used in this assignment would be repetitive. I have included the code for simulation, modelling and machine learning as this is the crux of the assignment. Code for the plotting and other such tasks can be found in the GitHub (not linked to due to anonymity requirements). Please request access from the author. Please see the Generative AI statement for full details around usage.

# A Reinforcement Learning

## A.1 Mathematical Derivations

### A.1.1 Question 1

In classical temporal-difference (TD) learning, the value function $\hat{V}^\pi(s)$ is estimated using a **lookup table**, where each state $s$ has an independent value estimate:

$$f(s; w) = w_s$$

This approach is infeasible for large or continuous state spaces because storing and updating an independent value for every state would require excessive memory and data. Instead, we use function approximation to generalize across states.

A common approach is to approximate $\hat{V}^\pi(s)$ with a parametric function $f(s; \mathbf{w})$, where $\mathbf{w}$ represents a set of learnable parameters. This allows generalization across similar states, rather than treating each state as independent.

We introduce a mapping $\phi(s) : \mathbb{R}^D \to \mathbb{R}^k$ that extracts useful features from the state representation. This allows us to approximate the value function as:

$$\hat{V}^\pi(s) = f(s; \mathbf{w}) = \sum_k w_k \phi_k(s).$$

A special case of this is the **one-hot encoding** approach, where each state has a unique feature representation:

$$\phi_k(s) = \begin{cases} 1 & \text{if } k = s \\ 0 & \text{otherwise} \end{cases}$$

Under this encoding, function approximation reduces to the classical TD lookup table update:

$$w_s \leftarrow w_s + \epsilon \left[ r_t + \gamma w_{s'} - w_s \right]$$

showing that the lookup table method is just a special case of function approximation.

A more practical and scalable approach is *linear function approximation*, where we define the value function as a weighted sum of predefined feature functions:

$$f(s_t; \mathbf{w}) = \sum_k w_k \phi_k(s_t).$$

This formulation allows generalization across states and is widely used in reinforcement learning.

### A.1.2 Question 2

In the case of *linear function approximation*, the value function is parameterised as:

$$\hat{V}^\pi(s) = f(s; \mathbf{w}) = \sum_k w_k \phi_k(s).$$

The general temporal-difference (TD) update rule, given in Equation (**??**), is:

$$w_k \leftarrow w_k + \epsilon \left[ r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t) \right] \frac{\partial f(s_t, \mathbf{w})}{\partial w_k}.$$

To compute this update, we differentiate the function approximation:

$$\frac{\partial f(s_t, \mathbf{w})}{\partial w_k} = \frac{\partial}{\partial w_k} \sum_j w_j \phi_j(s_t).$$

Since differentiation is linear, only the term involving $w_k$ remains, leading to:

$$\frac{\partial f(s_t, \mathbf{w})}{\partial w_k} = \phi_k(s_t).$$

Substituting this into the update rule, we obtain:

$$w_k \leftarrow w_k + \epsilon \left[ r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t) \right] \phi_k(s_t).$$

This update rule adjusts the weight in proportion to the temporal difference error and the feature activation.

## A.2 Python Code

### A.2.1 Parameter and Function Setup

```
#========== Input Variables ==========#
num_trials = 201 # number of trials
t = 25 # in seconds
dt = 0.5 # time step in seconds
time_steps = int(t/dt) # Number of time steps per trial
```

```python
stimulus_time = 10 # time of stimulus presentation in seconds
gamma = 1 # currently no discounting
epsilon = 0.2 # learning rate
T_mem = 12 # memory span, in seconds
mu = 20 # mean of gaussian function
sigma = 1 # standard deviation of gaussian function
memory_steps = int(T_mem / dt) # Discretised memory span
#========== Functions ==========#
def g(x): # delta function
    if x == 0:
        return 1
    else:
        return 0

def y(t): # stimulus function
    return g(t-stimulus_time)

def gaussian(x): # normal distribution
    return np.exp(-(x-mu)**2/(2*sigma**2))/(sigma*np.sqrt(2*np.pi)
    )

def r(t): # reward function
    return (1/2)*gaussian(t)
```

### A.2.2 Queston 3a: Reward and Stimulus Plotting

```python
time_array = np.arange(time_steps) * dt
time = np.arange(0, t+dt, dt)

stimulus_values = list(map(y, time))
reward_values = list(map(r, time))
```

### A.2.3 Queston 3b: Tapped Delay Line Simulation

```python
def feature_vector(n):
    phi = np.zeros(memory_steps)
    for tau in range(memory_steps):
        t_val = (n - tau) * dt
        if t_val < 0:
            phi[tau] = 0.0
        else:
```

```
            phi[tau] = y(t_val)
    return phi

w = np.zeros(memory_steps)
plot_trials = range(0, num_trials, 10)
all_V = {}
all_dV = {}
all_delta = {}

for trial in range(num_trials):
    V = np.zeros(time_steps)
    phi_0 = feature_vector(0)
    V[0] = np.dot(w, phi_0)

    for n in range(1, time_steps):
        phi_n = feature_vector(n)
        V[n] = np.dot(w, phi_n)
        td_error = r((n - 1) * dt) + gamma * V[n] - V[n - 1]
        phi_prev = feature_vector(n - 1)
        w += epsilon * td_error * phi_prev

    if trial in plot_trials:
        dV_arr = np.zeros(time_steps)
        delta_arr = np.zeros(time_steps)
        dV_arr[0] = gamma * V[0]
        delta_arr[0] = r(-dt) + dV_arr[0]
        for n in range(1, time_steps):
            dV_arr[n] = gamma * V[n] - V[n - 1]
            delta_arr[n] = r((n - 1) * dt) + dV_arr[n]
        all_V[trial] = V
        all_dV[trial] = dV_arr
        all_delta[trial] = delta_arr
```

### A.2.4 Queston 4: Box-car Simulation

```
def boxcar_feature_vector(n):
    phi = np.zeros(memory_steps)
    for tau in range(memory_steps):
        s = 0.0
        for u in range(tau + 1):
            idx = n - u
```

```
            if idx >= 0:
                s += y(idx * dt)
        phi[tau] = s
    return phi

w = np.zeros(memory_steps)  # Initialize weights

plot_trials = range(0, num_trials, 10)
all_V = {}
all_dV = {}
all_delta = {}

for trial in range(num_trials):
    V = np.zeros(time_steps)
    phi_0 = boxcar_feature_vector(0)
    V[0] = np.dot(w, phi_0)

    for n in range(1, time_steps):
        phi_n = boxcar_feature_vector(n)
        V[n] = np.dot(w, phi_n)
        td_error = r((n - 1) * dt) + gamma * V[n] - V[n - 1]
        phi_prev = boxcar_feature_vector(n - 1)
        w += epsilon * td_error * phi_prev

    if trial in plot_trials:
        dV_arr = np.zeros(time_steps)
        delta_arr = np.zeros(time_steps)
        dV_arr[0] = gamma * V[0]
        delta_arr[0] = r(-dt) + dV_arr[0]
        for n in range(1, time_steps):
            dV_arr[n] = gamma * V[n] - V[n - 1]
            delta_arr[n] = r((n - 1) * dt) + dV_arr[n]
        all_V[trial] = V
        all_dV[trial] = dV_arr
        all_delta[trial] = delta_arr
```

#### A.2.5 Queston 5: Partial Reinforcement - Simulation

```
w = np.zeros(memory_steps)

# Time Courses
```

```
V_matrix = np.zeros((num_trials, time_steps))
dV_matrix = np.zeros((num_trials, time_steps))
delta_matrix = np.zeros((num_trials, time_steps))

# Track whether a simulation was rewarded
is_rewarded = np.zeros(num_trials, dtype=bool)

# MAIN LEARNING LOOP
for trial in range(num_trials):
    rewarded = (np.random.rand() < p_reward)
    is_rewarded[trial] = rewarded

    V = np.zeros(time_steps)

    phi_0 = boxcar_feature_vector(0)
    V[0] = np.dot(w, phi_0)

    # Loop over time steps
    for n in range(1, time_steps):
        phi_n = boxcar_feature_vector(n)
        V[n] = np.dot(w, phi_n)

        reward_val = r((n - 1) * dt) if rewarded else 0.0

        # TD error:  (n) = r((n-1)*dt) +  *V(n) - V(n-1)
        td_error = reward_val + gamma * V[n] - V[n - 1]

        phi_prev = boxcar_feature_vector(n - 1)
        w += epsilon * td_error * phi_prev

    dV_arr = np.zeros_like(V)
    delta_arr = np.zeros_like(V)

    # For n=0, assume V(-1) = 0 so dV(0) =  *V(0) and delta(0) = dV(0)
    dV_arr[0] = gamma * V[0]
    delta_arr[0] = dV_arr[0]

    for n in range(1, time_steps):
        dV_arr[n] = gamma * V[n] - V[n - 1]
        r_prev = r((n - 1) * dt) if rewarded else 0.0
```

```
            delta_arr[n] = r_prev + dV_arr[n]

        V_matrix[trial] = V
        dV_matrix[trial] = dV_arr
        delta_matrix[trial] = delta_arr

# AVERAGE OVER LAST 100 TRIALS
last_100_trials = np.arange(num_trials - 100, num_trials)

# Separate rewarded and unrewarded trials
rewarded_mask = last_100_trials[is_rewarded[last_100_trials]]
unrewarded_mask = last_100_trials[~is_rewarded[last_100_trials]]

V_rewarded = V_matrix[rewarded_mask].mean(axis=0) if len(
    rewarded_mask) > 0 else np.zeros(time_steps)
V_unrewarded = V_matrix[unrewarded_mask].mean(axis=0) if len(
    unrewarded_mask) > 0 else np.zeros(time_steps)
V_all = V_matrix[last_100_trials].mean(axis=0)

dV_rewarded = dV_matrix[rewarded_mask].mean(axis=0) if len(
    rewarded_mask) > 0 else np.zeros(time_steps)
dV_unrewarded = dV_matrix[unrewarded_mask].mean(axis=0) if len(
    unrewarded_mask) > 0 else np.zeros(time_steps)
dV_all = dV_matrix[last_100_trials].mean(axis=0)

delta_rewarded = delta_matrix[rewarded_mask].mean(axis=0) if len(
    rewarded_mask) > 0 else np.zeros(time_steps)
delta_unrewarded = delta_matrix[unrewarded_mask].mean(axis=0) if
    len(unrewarded_mask) > 0 else np.zeros(time_steps)
delta_all = delta_matrix[last_100_trials].mean(axis=0)
```

### A.2.6 Queston 5b: Asymmetric Dynamic Range Function

```
def DA(x, alpha=6.0, x_star=0.27, beta=6.0):
    x = np.asarray(x)
    return np.piecewise(
        x,
        [x < 0, (x >= 0) & (x < x_star), x >= x_star],
        [lambda z: z/alpha,
         lambda z: z,
         lambda z: x_star + (z - x_star)/beta]
```

```
    )
```

### A.2.7 Queston 5b: Asymmetric Dynamic Range Simulation

```
avg_delta_all = delta_all

DA_matrix = np.zeros_like(delta_matrix[last_100_trials])
for i, tr in enumerate(last_100_trials):
    DA_matrix[i] = DA(delta_matrix[tr])

DA_avg = DA_matrix.mean(axis=0)
```

### A.2.8 Queston 6: Different Reward Probability

```
p_values = [0.0, 0.25, 0.5, 0.75, 1.0]

da_curves = {}

for p_reward in p_values:
    w = np.zeros(memory_steps)
    delta_matrix = np.zeros((num_trials, time_steps))
    is_rewarded = np.zeros(num_trials, dtype=bool)

    # Main learning loop
    for trial in range(num_trials):
        rewarded = (np.random.rand() < p_reward)
        is_rewarded[trial] = rewarded

        V = np.zeros(time_steps)

        phi_0 = boxcar_feature_vector(0)
        V[0] = np.dot(w, phi_0)

        for n in range(1, time_steps):
            phi_n = boxcar_feature_vector(n)
            V[n] = np.dot(w, phi_n)

            reward_val = r((n - 1) * dt) if rewarded else 0.0
            td_error = reward_val + gamma * V[n] - V[n - 1]

            phi_prev = boxcar_feature_vector(n - 1)
```

```python
            w += epsilon * td_error * phi_prev

        delta_arr = np.zeros(time_steps)
        delta_arr[0] = gamma * V[0]  # For n=0, assume V(-1) = 0
        for n in range(1, time_steps):
            r_prev = r((n - 1) * dt) if rewarded else 0.0
            delta_arr[n] = r_prev + gamma * V[n] - V[n - 1]

        delta_matrix[trial] = delta_arr

    last_100 = np.arange(num_trials - 100, num_trials)
    DA_matrix = np.zeros_like(delta_matrix[last_100])
    for i, tr_idx in enumerate(last_100):
        DA_matrix[i] = DA(delta_matrix[tr_idx])

    DA_avg = DA_matrix.mean(axis=0)
    da_curves[p_reward] = DA_avg
```

#### A.2.9 Queston 7: Dopamine Response as a Function of Reward Probability

```python
stimulus_idx = int(10.0 / dt)  # approx 10 seconds
reward_idx = int(20.0 / dt)    # approx 20 seconds

peak_stim_list = []
peak_reward_list = []

for p in p_values:
    da_array = da_curves[p]
    window_size = 2  # use 2 time steps around target

    # Max value at stimulus time
    stim_start = max(stimulus_idx - window_size, 0)
    stim_end = min(stimulus_idx + window_size + 1, time_steps)
    peak_stim = np.max(da_array[stim_start:stim_end])

    # Max value at reward time
    rew_start = max(reward_idx - window_size, 0)
    rew_end = min(reward_idx + window_size + 1, time_steps)
    peak_rew = np.max(da_array[rew_start:rew_end])

    peak_stim_list.append(peak_stim)
```

```
    peak_reward_list.append(peak_rew)
```

# B Representational Learning

## B.1 Mathematical Derivations

### B.1.1 Q2: Derivation of Loss Function and Derivatives

**Negative Log-Likelihood**

The probability distribution is modelled as:

$$p\big(x_{n,k} \mid x_{n,\neq k}, \theta\big) = \mathcal{N}\Big(x_{n,k} \,\Big|\, 0,\ \sigma_{n,k}^2\Big),$$

$$\mathcal{N}\big(x_{n,k} \mid 0, \sigma_{n,k}^2\big) = \frac{1}{\sqrt{2\pi\,\sigma_{n,k}^2}} \exp\left(-\frac{x_{n,k}^2}{2\,\sigma_{n,k}^2}\right)$$

The log-likelihood of $(x_{n,k})$ under this Gaussian is

$$\log p\big(x_{n,k} \mid x_{n,\neq k}, \theta\big) = -\tfrac{1}{2}\ln\big(2\pi\,\sigma_{n,k}^2\big) \;-\; \tfrac{x_{n,k}^2}{2\,\sigma_{n,k}^2}.$$

Hence, the *negative* log-likelihood for that point is

$$-\log p\big(x_{n,k} \mid x_{n,\neq k}, \theta\big) = \tfrac{1}{2}\ln\big(2\pi\,\sigma_{n,k}^2\big) \;+\; \tfrac{x_{n,k}^2}{2\,\sigma_{n,k}^2}.$$

To get the total negative log-likelihood (NLL) over the entire dataset, we sum over all $n = 1, \ldots, N$ and $k = 1, \ldots, K$:

$$\mathcal{L}(\mathbf{X};\,\theta) = \sum_{n=1}^{N}\sum_{k=1}^{K}\Big[\tfrac{1}{2}\ln\big(2\pi\,\sigma_{n,k}^2\big) + \tfrac{x_{n,k}^2}{2\,\sigma_{n,k}^2}\Big].$$

**Gradients with respect to $\sigma_{n,k}^2$**

First, define

$$\ell_{n,k} = -\log p\big(x_{n,k} \mid x_{n,\neq k}, \theta\big) = \tfrac{1}{2}\ln\big(2\pi\,\sigma_{n,k}^2\big) + \tfrac{x_{n,k}^2}{2\,\sigma_{n,k}^2}.$$

Taking the partial derivative w.r.t. $\sigma_{n,k}^2$:

$$\frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)} = \frac{1}{2}\,\frac{1}{\sigma_{n,k}^2} - \frac{x_{n,k}^2}{2\,(\sigma_{n,k}^2)^2}.$$

For convenience, denote

$$\frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)} = \frac{1}{2\,\sigma_{n,k}^2}\Big[1 - \frac{x_{n,k}^2}{\sigma_{n,k}^2}\Big].$$

**Chain Rule for $a_{k,j}$ and $b_k$**

Recall that

$$\sigma_{n,k}^2 = \sum_{j\neq k} a_{k,j}\, x_{n,j}^2 \;+\; b_k.$$

Hence:

$$\frac{\partial\, \sigma_{n,k}^2}{\partial\, a_{k,j}} = x_{n,j}^2, \quad \text{and} \quad \frac{\partial\, \sigma_{n,k}^2}{\partial\, b_k} = 1.$$

Thus, by the chain rule,

$$\frac{\partial \ell_{n,k}}{\partial a_{k,j}} = \frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)}\,\frac{\partial(\sigma_{n,k}^2)}{\partial a_{k,j}} = \Big[\frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)}\Big]\; x_{n,j}^2,$$

$$\frac{\partial \ell_{n,k}}{\partial b_k} = \frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)}\,\frac{\partial(\sigma_{n,k}^2)}{\partial b_k} = \Big[\frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)}\Big]\;\cdot 1.$$

**Total Gradient**

To get the gradient of the *full* negative log-likelihood $\mathcal{L}(\mathbf{X};\theta)$, we sum over all $n, k$:

$$\frac{\partial \mathcal{L}}{\partial a_{k,j}} = \sum_{n=1}^{N} \frac{\partial \ell_{n,k}}{\partial a_{k,j}} = \sum_{n=1}^{N} \Big[x_{n,j}^2 \;\cdot\; \frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)}\Big],$$

$$\frac{\partial \mathcal{L}}{\partial b_k} = \sum_{n=1}^{N} \frac{\partial \ell_{n,k}}{\partial b_k} = \sum_{n=1}^{N} \Big[\frac{\partial \ell_{n,k}}{\partial(\sigma_{n,k}^2)}\Big].$$

Each term involves a *single scalar* $\sigma_{n,k}^2$, but across all rows $n$ we get a sum of those scalar contributions.

## B.2 Python Code

### B.2.1 Parameter and Function Initialisation

```

print(f"Current working directory: {os.getcwd()}")

mat_data = scipy.io.loadmat(".../representational.mat")
#%%

#============================= EDA ===============================#
```

```
# Extract variables
Y = mat_data['Y']  # 32000    1024 image patches
R = mat_data['R']  # 1024    256 feed-forward weights
W = mat_data['W']  # Another 1024    256 matrix
X = Y @ R # (32000 x 1024) x (1024 x 256) = (32000, 256)

print(f"Y shape: {Y.shape}")
print(f"R shape: {R.shape}")
print(f"W shape: {W.shape}")

# plotIm(W) # plot all of the generative weight filters
```

### B.2.2 Question 1 (i): Sparsity of Components

```
X = Y @ R # (32000 x 1024) x (1024 x 256) = (32000, 256)
print(f"X shape: {X.shape}")

# Selected components inbetween 125 and 141 (which were highly similar in frequenct and orientation from visual inspection)
selected_components = [125, 129, 133, 137, 141]  # Example indices of columns in X

fig, axes = plt.subplots(1, len(selected_components), figsize=(15, 4))

for i, k in enumerate(selected_components):
    axes[i].hist(X[:, k], bins=50, density=True, alpha=0.7, label=f"Component {k}")

    # LaTeX-style titles and axis labels
    # I used generative AI to help me with the syntax for LaTeX style matplotlib plotting
    axes[i].set_title(f"Histogram of $x_{{{k}}}$")  # LaTeX for x_k
    axes[i].set_xlabel(f"$x_{{{k}}}$ values (bins = 50)")  # r"" ensures raw string for LaTeX
    axes[i].set_ylabel(r"Density")  # Keep this standard
    axes[i].set_yscale('log')  # Log scale for better visualization

```

```
plt.tight_layout()
plt.show()
```

### B.2.3 Question 1 (ii): Independence of Components

```
#plot the 30th and 31st 32x32 matric from W

#Create a 2x1 grid of subplots
fig, axes = plt.subplots(1, 2, figsize=(8, 12))

# Plot the 125th and 141st components of W
for i, k in enumerate([125, 141]):
    patch = W[:, k].reshape(32, 32)
    axes[i].imshow(patch, cmap='gray')
    axes[i].axis('off')
    axes[i].set_title(f'Component {k}', pad=10)  # Add title with
       some padding

plt.tight_layout()
plt.show()

#%%

# let us compare compents 125 and 141, using a 2d histogram

#Create a 2D histogram of X[:, 233] vs X[:, 249]
plt.figure(figsize=(8, 6))
plt.hist2d(X[:, 125], X[:, 141], bins=50, cmap='viridis', norm=
    colors.LogNorm(vmax=1e4, vmin=1e-4))  # Add LogNorm
plt.colorbar(label='Frequency (log scale)')
plt.xlabel('Component 125')
plt.ylabel('Component 141')
plt.title('2D Histogram of Component 125 vs Component 141 (log
    scale)')
plt.show()
#%%

#compute and plot the conditional distribution p(x_k2|x_k1)
H, x_edges, y_edges = np.histogram2d(X[:, 125], X[:, 141], bins
    =50)

```

```
# Normalize each column (x_k1 slice) to get conditional
    probability
# Add small constant to avoid division by zero
H_conditional = H / (H.sum(axis=0, keepdims=True) + 1e-10)

# Plot the conditional distribution
plt.figure(figsize=(8, 6))
plt.pcolormesh(x_edges[:-1], y_edges[:-1], H_conditional.T, cmap='
    viridis', norm=colors.LogNorm(vmax=1e4, vmin=1e-4))  # Add
    LogNorm
plt.colorbar(label='$p(x_{k2}|x_{k1})$ (log scale)')
plt.xlabel('$x_{k1}$ (Component 125)')
plt.ylabel('$x_{k2}$ (Component 141)')
plt.title('Conditional Distribution $p(x_{k2}|x_{k1})$ (log scale)
    ')
plt.show()
```

### B.2.4 Question 2

Please see Generative AI usage section of Appendix.

### B.2.5 Question 3: Calculation of Sigma

```
def compute_sigma2(X, a, b):
    # Get number of samples (N) and components (K)
    N, K = X.shape
    sigma2 = np.zeros((N, K))

    # Loop over each component k
    for k_ in range(K):
        # Compute sum_j a[k_, j] * (X[:, j])^2
        sum_over_j = (X**2) @ a[k_, :]
        # Add b[k_] to each sample's sum for component k
        sigma2[:, k_] = sum_over_j + b[k_]
        # Subtract the diagonal term (j == k)
        sigma2[:, k_] -= a[k_, k_] * (X[:, k_]**2)

    return sigma2
```

### B.2.6 Question 3: Normalised Latent Variables

```python
# Compute sigma^2(x) and then normalized coefficients c
sigma2_vals = compute_sigma2(X, a_data, b_data)
sigma_vals = np.sqrt(sigma2_vals)  # shape (N, K)
C = X / sigma_vals

print("Computed normalized coefficients C = X / sigma(x).")
print("C shape:", C.shape)

#------------------------- 3.1: Marginal Distributions -------------------------#
# Compare p(x_k) and p(c_k) for some chosen components
chosen_components = [125, 141]
fig, axes = plt.subplots(2, len(chosen_components), figsize=(4*len(chosen_components), 8))

for idx, k_ in enumerate(chosen_components):
    # Plot X_k
    axes[0, idx].hist(X[:, k_], bins=50, density=True, alpha=0.7, edgecolor='k')
    axes[0, idx].set_yscale('log')
    axes[0, idx].set_xlabel(f"$x_{{{k_}}}$")
    axes[0, idx].set_ylabel("Density (log-scale)")
    kurt_x = kurtosis(X[:, k_])  # Excess kurtosis
    axes[0, idx].set_title(f"X_{k_}, Kurt={kurt_x:.2f}")

    # Plot C_k
    axes[1, idx].hist(C[:, k_], bins=50, density=True, alpha=0.7, edgecolor='k')
    axes[1, idx].set_yscale('log')
    axes[1, idx].set_xlabel(f"$c_{{{k_}}}$ = x_{{{k_}}}/\\sigma_{{{k_}}}$")
    axes[1, idx].set_ylabel("Density (log-scale)")
    kurt_c = kurtosis(C[:, k_])
    axes[1, idx].set_title(f"C_{k_}, Kurt={kurt_c:.2f}")

plt.tight_layout()
plt.show()

#--------------------- 3.2: Conditional Distributions ---------------------#
def plot_conditional_2D(x1, x2, title='', bins=60, lognorm=True):
```

```python
    """
    Plot p(x2 | x1) by forming a 2D histogram and normalizing each vertical slice.
    """
    H, xedges, yedges = np.histogram2d(x1, x2, bins=bins, density=False)
    for i in range(H.shape[0]):
        col_sum = np.sum(H[i, :])
        if col_sum > 0:
            H[i, :] /= col_sum
    plt.figure(figsize=(6, 5))
    if lognorm:
        plt.pcolormesh(xedges, yedges, H.T, cmap='viridis',
                       norm=colors.LogNorm(vmin=1e-5, vmax=H.max()))
    else:
        plt.pcolormesh(xedges, yedges, H.T, cmap='viridis')
    plt.colorbar(label="$p(x_2 \\mid x_1)$")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.title(title)
    plt.show()

k1, k2 = 125, 141

# p(x_{k2} | x_{k1})
plot_conditional_2D(X[:, k1], X[:, k2], title=f"p(x_{k2} | x_{k1}) (Components {k1} vs. {k2})")

# p(c_{k2} | c_{k1})
plot_conditional_2D(C[:, k1], C[:, k2], title=f"p(c_{k2} | c_{k1}) (Components {k1} vs. {k2})")
```

### B.2.7 Question 4: Natural Image Statistics

```python
# For k = 125: Plot main filter and its top-10 associated filters

k = 125

# Plot the main generative weight for component k
main_filter_img = W[:, k].reshape(32, 32)
```

```
plt.figure()
plt.imshow(main_filter_img, cmap='gray')
plt.title(f"Generative Weight W[:, {k}]")
plt.axis('off')
plt.show()

# Get the 10 indices with highest a[k, j] values
row_k = a_data[k, :]
top_10_indices = np.argsort(row_k)[-10:]
print(f"Top 10 j indices for a[k={k}, j]: {top_10_indices}")
print("Corresponding a-values:", row_k[top_10_indices])

# Plot the top-10 filters in a 2x5 grid
fig, axes = plt.subplots(2, 5, figsize=(20, 8))
for idx, j_ in enumerate(top_10_indices):
    row_idx = idx // 5  # Determine row in grid
    col_idx = idx % 5   # Determine column in grid
    img = W[:, j_].reshape(32, 32)
    axes[row_idx, col_idx].imshow(img, cmap='gray')
    axes[row_idx, col_idx].set_title(f"W[:,{j_}]  a[{k},{j_}]={
       row_k[j_]:.4f}")
    axes[row_idx, col_idx].axis('off')
plt.tight_layout()
plt.show()


# For k = 162: Repeat the same process

k = 162

# Plot the main generative weight for component k
main_filter_img = W[:, k].reshape(32, 32)
plt.figure()
plt.imshow(main_filter_img, cmap='gray')
plt.title(f"Generative Weight W[:, {k}]")
plt.axis('off')
plt.show()

# Get the top 10 indices for component k
row_k = a_data[k, :]
top_10_indices = np.argsort(row_k)[-10:]
```

```
print(f"Top 10 j indices for a[k={k}, j]: {top_10_indices}")
print("Corresponding a-values:", row_k[top_10_indices])

# Plot these top-10 filters in a 2x5 grid
fig, axes = plt.subplots(2, 5, figsize=(20, 8))
for idx, j_ in enumerate(top_10_indices):
    row_idx = idx // 5
    col_idx = idx % 5
    img = W[:, j_].reshape(32, 32)
    axes[row_idx, col_idx].imshow(img, cmap='gray')
    axes[row_idx, col_idx].set_title(f"W[:,{j_}]  a[{k},{j_}]={
       row_k[j_]:.4f}")
    axes[row_idx, col_idx].axis('off')
plt.tight_layout()
plt.show()
```

# C Generative AI Usage - Understanding

(For Question 2 Part 2) **Prompt:** We are looping over different components (columns of the X matrix), multiplying all other squared components (which are vectors of size N) by a coefficient and then adding $b_k$. surely this returns a vector of size N as opposed to a singular sd value? or am i thinking of this wrong?

**Model Used:** ChatGPT, 4o [18/02/2025]

**Output:**

```

\subsection{Understanding Conditional Variance in the Model}

A good way to reconcile the ``vector vs. single value'' confusion is to remember that in a conditional-density model, each data point $x_{n}$ has its own conditional variance. In other words, for each $n$ and each dimension $k$,

\[
\sigma^2_{n,k} = \sum_{j \neq k} a_{k,j} x_{n,j}^2 + b_k,
\]

is a single scalar (just a number) that governs the variance of $x_{n,k}$ for that particular row $n$.

Yes, if we look across all $n$ at once, we obtain an entire vector of variances,

\[
\{\sigma^2_{1,k}, \dots, \sigma^2_{N,k}\}.
\]

There is one variance value per row $n$.

However, for a **single row** (i.e., for one data point), $\sigma^2_{n,k}$ is just **one number**, because $x_{n,j}^2$ is simply the square of the $j$th component of that row.

Thus, there is no single global variance for the entire column; instead, the model assumes that each observation $x_{n,k}$ has a **different conditional variance**, depending on the values of its other coordinates $x_{n, \neq k}$.
```

# D Generative AI Usage - Programming

## D.1 LaTeX Template Code

**Prompt:** Please give me an appropriate template for my university assignment in latex. Make sure to include appropriate libraries for graphics, (IEEE style) bibligographies or anything that you think will be useful. Please also provide code snippets for LaTeX and Python in the Appendix.
**Model Used:** ChatGPT, 4o [17/02/2025]
**Output:**

```latex
\documentclass[12pt,a4paper]{article}

%-------------------------%
%      PACKAGE SETUP      %
%-------------------------%

% Page geometry
\usepackage[margin=1in]{geometry}

% Encoding and fonts
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}

% Graphics and floats
\usepackage{graphicx}   % For including figures
\usepackage{float}      % For controlling figure placement
\usepackage{caption}    % Better caption control
\usepackage{subcaption} % Sub-figures
\usepackage{amsmath,amssymb} % Math packages if needed

% Colors (optional)
\usepackage{xcolor}

% References (IEEE style)
% 'cite' is often used with IEEE style
\usepackage{cite}

% Hyperlinks in PDF
\usepackage{hyperref}
\hypersetup{
```

```
    colorlinks = true,
    linkcolor  = black,
    citecolor  = blue,
    urlcolor   = teal
}

%-------------------------%
%  BIBLIOGRAPHY SETTINGS  %
%-------------------------%
% We will use the IEEEtran bibliography style.
% (Remember to run: pdflatex -> bibtex -> pdflatex -> pdflatex)

%-------------------------%
%      TITLE & AUTHOR     %
%-------------------------%

\title{Your Assignment Title Here}
\author{Your Name \\
Department Name, University Name \\
\texttt{your\_email@university.edu}
}
\date{\today}

%-------------------------%
%      BEGIN DOCUMENT     %
%-------------------------%

\begin{document}
\maketitle

\begin{abstract}
A concise summary of the assignment goes here. This might be optional for smaller assignments.
\end{abstract}

\tableofcontents
\newpage

%-------------------------%
%       INTRODUCTION      %
%-------------------------%
```

```latex

\section{Introduction}
Provide a brief introduction to your assignment topic here. Add
    references like \cite{IEEEexample:book}.

%-------------------------%
%   MAIN SECTION EXAMPLE  %
%-------------------------%

\section{Main Discussion}
Explain your approach or the theory behind your work. You may
    include figures (see Figure~\ref{fig:example}).

\subsection{Sub-section}
Further detail, data, or analyses.

\begin{figure}[H]
    \centering
    \includegraphics[width=0.5\textwidth]{example-image}
    \caption{An example figure. Replace with your own.}
    \label{fig:example}
\end{figure}

%-------------------------%
%   RESULTS & DISCUSSION  %
%-------------------------%

\section{Results and Discussion}
Present your findings/results. Discuss their significance,
    limitations, etc.

%-------------------------%
%       CONCLUSIONS       %
%-------------------------%

\section{Conclusion}
Summarize your findings and possibly suggest future work or
    applications.

%-------------------------%
%   BIBLIOGRAPHY          %
```

```
%------------------------%

\bibliographystyle{IEEEtran} % IEEE style
\bibliography{references}    % references.bib file

\end{document}

\end{document}
```

## D.2 Representation Learning - Question 2

**Prompt:** Please look at these functions in Matlab (attached functions given in assignment data), and look at the mathematical derivation of the loss function and partial derivatives with respect to the parameters (attached mathematical derivation seen in Appendix B.1.1).
I want a python function that will run optimally on my university's HPC cluster, such that I can retrieve the estimates of the parameters.

(There was also a further discussion around usage of the HPC and appropriate filepath. I also attached the specifications of Q2).

**Model Used:** ChatGPT, 4o [24/02/2025]

```python
#!/usr/bin/env python

import numpy as np
import os
import pickle
import matplotlib.pyplot as plt
import scipy.io
from scipy.optimize import minimize

def save_data_for_plotting(data, filename):
    """Save data to file for later plotting locally."""
    with open(filename, 'wb') as f:
        pickle.dump(data, f)
    print(f"Data saved to {filename}")

def conditional_log_likelihood(params, X):
    """
    Compute the negative conditional log-likelihood and its
        gradient
```

```
    for the model:
        p(x_k | x_{-k}) = Normal(0, sum_j a_{k,j} * x_j^2 + b_k).
    'params' is a 1D array of length (K*(K-1) + K),
    containing log(a_{k,j}) and log(b_k).
    """
    N, K = X.shape

    # Unpack parameters
    log_a_flat = params[:K*(K-1)]
    log_b = params[K*(K-1):]

    # Reshape log_a to a K x K matrix (with zeros on diagonal)
    log_a = np.zeros((K, K))
    idx = 0
    for k in range(K):
        for j in range(K):
            if j != k:
                log_a[k, j] = log_a_flat[idx]
                idx += 1

    # Exponentiate to get a_{k,j} and b_k
    a = np.exp(log_a)
    b = np.exp(log_b)

    # Initialize log-likelihood and gradients
    log_likelihood = 0.0
    grad_log_a = np.zeros_like(log_a)
    grad_log_b = np.zeros_like(log_b)

    # Compute log-likelihood and derivatives
    for n in range(N):
        if n % 1000 == 0:
            print(f"Processing sample {n}/{N}...")
        x_n = X[n, :]
        for k in range(K):
            sigma_squared = np.sum(a[k, :] * x_n**2) - a[k, k] *
               x_n[k]**2 + b[k]
            if not np.isfinite(sigma_squared) or sigma_squared <=
               0:
                # Prevent invalid or zero value
                sigma_squared = 1e-6
```

```
                x_nk = x_n[k]
                # Contribution to log-likelihood
                log_likelihood += -0.5 * np.log(2*np.pi) - 0.5 * np.log(sigma_squared) - 0.5*(x_nk**2)/sigma_squared

                # Gradient wrt a_{k,j}
                for j in range(K):
                    if j != k:
                        partial = (-0.5 / sigma_squared + 0.5*(x_nk**2)/(sigma_squared**2)) * a[k, j] * x_n[j]**2
                        grad_log_a[k, j] += partial

                # Gradient wrt b_k
                grad_log_b[k] += (-0.5/sigma_squared + 0.5*(x_nk**2)/(sigma_squared**2)) * b[k]

    # Flatten grad_log_a (excluding diagonal)
    grad_log_a_flat = []
    for k in range(K):
        for j in range(K):
            if j != k:
                grad_log_a_flat.append(grad_log_a[k, j])
    grad_log_a_flat = np.array(grad_log_a_flat)

    grad = np.concatenate([grad_log_a_flat, grad_log_b])

    # We return the NEGATIVE log-likelihood & its gradient, because we minimize
    print(f"Current Loss: {-log_likelihood:.6f}")
    return -log_likelihood, -grad

def estimate_parameters(X, max_iter=100, save_dir="./parameter_estimation"):
    """
    Estimate a_{k,j}, b_k by maximizing the conditional log-likelihood.
    Saves results to estimated_parameters.pkl plus diagnostic plots.
    """
    os.makedirs(save_dir, exist_ok=True)
```

```python

    N, K = X.shape
    # Optionally use fewer samples to speed up (e.g. 8000 or fewer)
    N_train = min(N, 8000)
    X_train = X[:N_train]

    num_a_params = K*(K-1)
    num_b_params = K
    initial_params = np.random.randn(num_a_params + num_b_params) * 0.01

    objective_history = []

    def callback(params):
        obj_val, _ = conditional_log_likelihood(params, X_train)
        objective_history.append(obj_val)
        if len(objective_history) % 5 == 0:
            print(f"Iteration {len(objective_history)}: Loss = {obj_val:.6f}")

    print("Starting Optimization...")

    result = minimize(
        lambda p: conditional_log_likelihood(p, X_train),
        initial_params,
        method='L-BFGS-B',
        jac=True,
        callback=callback,
        options={'maxiter': max_iter, 'disp': True}
    )

    if result.success:
        print(f"Optimization successful! Final loss: {result.fun:.6f}")
    else:
        print(f"WARNING: Optimization did not converge. Message: {result.message}")

    optimized_params = result.x
    log_a_flat = optimized_params[:num_a_params]
```

```python
    log_b = optimized_params[num_a_params:]

    # Reshape
    a = np.zeros((K, K))
    idx = 0
    for k in range(K):
        for j in range(K):
            if j != k:
                a[k, j] = np.exp(log_a_flat[idx])
                idx += 1
    b = np.exp(log_b)

    # Save
    params_data = {
        'a': a,
        'b': b,
        'optimized_params': optimized_params,
        'objective_history': objective_history
    }
    save_data_for_plotting(params_data, os.path.join(save_dir, '
       estimated_parameters.pkl'))

    # Plot objective function
    plt.figure(figsize=(10, 6))
    plt.plot(objective_history, marker='o', linestyle='-')
    plt.title('Negative Conditional Log-Likelihood')
    plt.xlabel('Iteration')
    plt.ylabel('Objective Value')
    plt.grid(True)
    plt.savefig(os.path.join(save_dir, 'convergence.png'))
    plt.close()

    # Visualize the a matrix
    plt.figure(figsize=(10, 8))
    plt.imshow(a, cmap='viridis')
    plt.colorbar(label='a[k,j] values')
    plt.title('Estimated a[k,j] Parameters')
    plt.xlabel('Component j')
    plt.ylabel('Component k')
    plt.savefig(os.path.join(save_dir, 'parameter_a_matrix.png'))
    plt.close()
```

```python

    # Visualize b parameters
    plt.figure(figsize=(10, 5))
    plt.plot(b, marker='o')
    plt.title('Estimated b[k] Parameters')
    plt.xlabel('Component k')
    plt.ylabel('b[k]')
    plt.grid(True)
    plt.savefig(os.path.join(save_dir, 'parameter_b_values.png'))
    plt.close()

    return a, b

def main():
    # Adjust path if needed for your HPC
    mat_file_path = "./representational.mat"
    output_dir = "./representational_output_part2"

    os.makedirs(output_dir, exist_ok=True)

    print(f"Loading data from {mat_file_path} ...")
    mat_data = scipy.io.loadmat(mat_file_path)
    # Extract variables
    Y = mat_data['Y']  # shape (32000, 1024)
    R = mat_data['R']  # shape (1024, 256)
    W = mat_data['W']  # shape (1024, 256)

    # Compute X
    X = Y @ R  # shape (32000, 256)
    print("Data shape:", X.shape)

    # Estimate parameters
    print("\n*** Estimating conditional model parameters (Question
        2) ***")
    a, b = estimate_parameters(
        X,
        max_iter=100,  # you can reduce this if it's too slow
        save_dir=os.path.join(output_dir, "parameter_estimation")
    )
    print("\nDone estimating parameters!")

```

```
if __name__ == "__main__":
    main()
```

## D.3 Extraction of .pkl files

**Prompt:** Please help me extract the pkl files that I have created in my university's HPC. (Full conversation followed about this - attached code was implemented locally)

**Model Used:** ChatGPT, 4o [24/02/2025]
**Output:**

```
import pickle
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from scipy.stats import kurtosis

# Path to your trained parameter checkpoint
pkl_path = "/Users/mac/Documents/Cambridge/Lent/Comp Neuro/Assignments/Assignment 1/HPC_Coursework/representational_output_fast/parameter_estimation_checkpoint/estimated_parameters.pkl"

# Load the dictionary containing 'a' and 'b'
with open(pkl_path, "rb") as f:
    param_dict = pickle.load(f)

# Extract the arrays for 'a' and 'b'. Adjust the keys if your .pkl uses different names.
a_data = param_dict["a"]  # shape should be (K, K)
b_data = param_dict["b"]  # shape should be (K,)

print("Loaded parameters from pickle:")
print("a_data shape:", a_data.shape)
print("b_data shape:", b_data.shape)

# Confirm that X exists (from your code above, X = Y @ R, shape: (N, K))
print("X shape:", X.shape)
```

### D.4 Conversion from .mat Files

**Prompt:** I have .mat files given here: Download the file representational.zip from the module's Moodle page. This file contains a Matlab data file representational.mat and the Matlab functions plotIm.m, checkgrad.m and minimize.m. The data file contains three variables: • Y is a 32000 × 1024 array containing 32000 image patches that are each 32 by 32 pixels in size, but which have been reshaped into the array. Elements of Y are denoted yn,d below. • R is a 1024 × 256 matrix containing the feed-forward weights (corresponding to Wff in the lecture slides) from the simplified sparse coding model that has been trained on a different set of natural image patches. Elements of R are denoted rd,k below. • W is a 1024 × 256 matrix containing. I want to be able to manipulate this data solely in python. how?
**Model Used:** ChatGPT, 4o [22/02/2025]
**Output:**

```
import scipy.io
import numpy as np

# Load the .mat file
mat_data = scipy.io.loadmat("representational.mat")

# Extract variables
Y = mat_data['Y']  # 32000   1024 image patches
R = mat_data['R']  # 1024   256 feed-forward weights
W = mat_data['W']  # Another 1024   256 matrix

# Print shapes to confirm
print(f"Y shape: {Y.shape}")
print(f"R shape: {R.shape}")
print(f"W shape: {W.shape}")
```

## E Use of Generative AI

In this assignment, generative AI (e.g., ChatGPT) was used to assist in creating or refining the LaTeX template. Any direct AI-generated text or code has been carefully reviewed and edited to ensure it meets the assignment requirements and academic integrity guidelines. Almost all coding was done in conjunction with Generative AI tools. Cursor was used, along with ChatGPT and Claude. The reader can assume that practically all matplotlib code was produced using generative AI. Where logic was needed (simulation in either Reinforcement Learning and Representation Learning), the initial logic was written up by a human. When bugs occured, and observation of code documentation didn't result in the errors being corrected, generative AI was consulted. Code completion (through Cursor) was used extensively for boilerplate code. Interaction with Matlab data and functions was also almost entirely done with Generative AI. Generative AI was also used to 'talk to papers'. I talked to a paper about ICA and also the original classical conditioned neuroscience paper.