In Class Assignment 1(Group)	
Member #1: Arya Bhavesh Shah	Member #2: Mahamood Abdulla
Roll No: 25290002	Roll No: 24510063
arya.shah@iitgn.ac.in	24510063@iitgn.ac.in

Assignment Details:
Design a new model of perceptron that can emulate functionality present in real neurons including but not limited to different ion channels, synaptic connections/neurotransmitters, neuronal morphology. As of now, a well thought of model is required. Extra credits for python based implementation.
This in-class assignment continued from Friday's class can be done in groups of two (individually also allowed) and should be submitted prior to the next class (Tuesday, 10 pm).

Introduction

Before sitting down to brainstorm a new model of perceptron, we took time to understand the original perceptron model and see where we could fill gaps or apply principles of neuroscience further and assess its feasibility backed by research.

We imagined our brain as a panel full of light switches where each switch takes in a few wires, i.e inputs, has electricity passing through it and once there is enough of it, the switch flips on. In this traditional setup of perceptron, we have a single step from inputs to output, no inner workings as such. We had this idea that no matter what, overall the system is powerful when we connect a lot of switches but a single switch itself is pretty "dumb" since it effectively is just a yes/no switch.

Now, we started imagining that instead of a single switch, if each neuron is a mini city with neighbourhoods, i.e branches, streets, i.e connections and traffic lights, i.e inhibitory control along with special factors i.e plateau boosters, we can have messages come into different neighbourhoods to get processed locally and then some of them boost the message and the remaining slow it down. The results then meet in the city centre, i.e soma in order to decide what happens. The thing to note here is that some boosts would only work if certain signals such as the neurotransmitter signals in our context are green i.e have a go ahead and some roads have the equivalent of toll gates,

such as shunting inhibitions that would divide the flow. This means that a single perceptron neuron that we propose can do the job of a small network of traditional perceptrons.

How this matters is that in the conventional perceptron model, you may need many simple switches for solving problems like XOR, etc. Now, with our novel idea, our neuron already has built-in sub-calculations and logic gates so that fewer neurons can do more things in a fast manner. Plus, this works more like a real biological neuron where the signals don't just jump signals, instead they shape, gate and amplify them in a complex fashion before deciding.
Thus, we call this above model of perceptron "Morpho-Conductance Perceptron (MCP)" which is a biologically inspired model that adds branch-local divisive conductance and an inhibition gated plateau augmentation term mapped to a minimal, differentiable implementation.

New Model Details

I. Overall Definition:
   A. Morphology: It splits inputs into branches (dendrites) where each branch gathers a subset of inputs.
   B. Shunting Inhibition (conductance): The inhibitory inputs raise a branch conductance "g" that divisively shrinks the excitatory drive, similar to how opening of ion channels that leak current and reduce gain behaves.
   C. Plateau Augmentation: If excitatory drive on a branch is strong, a positive boost is added. This is suppressed by the gating mechanism (inhibition), so plateaus occur when inhibition is low.
   D. Soma: All branches, i.e dendrite outputs are summed at the soma and passed through sigmoid activation function just like the normal perceptron but this time with refined branch-level preprocessing.

This helps because it lets a single layer unit perform logical operations between branches which a linear perceptron cannot do. We achieve this while being trainable with gradients and simple.

II. Mathematical Definition: The inputs are divided into B Branches through index masks of two types in nature: Excitatory and Inhibitory. Therefore, branch b:

→ Partitioning inputs : $x \in \mathbb{R}^n$

$x_b^E = \{ x_i : i \in exc\_idx[b] \}$ (Excitatory)

$x_b^I = \{ x_i : i \in inh\_idx[b] \}$ (Inhibitory)

→ Partitioning inputs : $x \in \mathbb{R}^n$

$x_b^E = \{ x_i : i \in exc\_idx[b] \}$ (Excitatory)

$x_b^I = \{ x_i : i \in inh\_idx[b] \}$ (Inhibitory)

→ Per branch computation ($b = 1, 2, \dots B$) :

1. Excitatory sum

$$s_b^E = w_b^E \cdot x_b^E$$

2. Inhibitory sum

$$s_b^I = w_b^I \cdot x_b^I$$

3. Shunting conductance

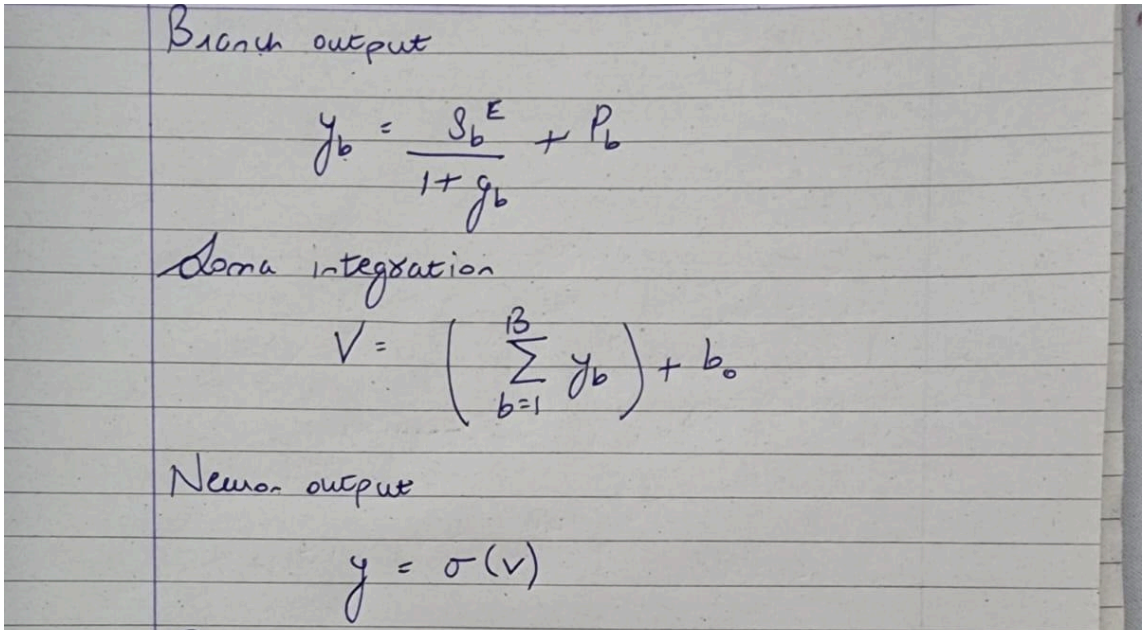
$$g_b = \sigma(\alpha_b s_b^I + \beta_b), \quad g_b \in (0, 1)$$

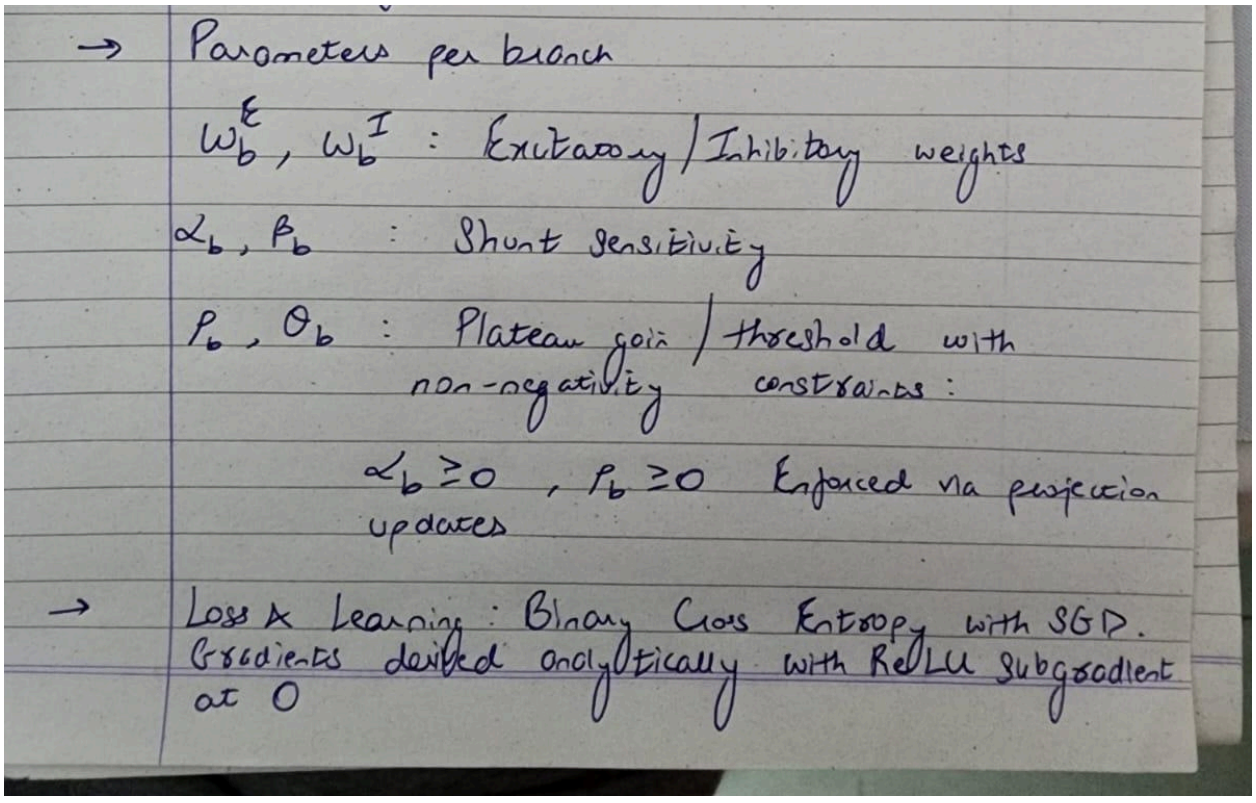
4. Plateau core

$$P_b^{core} = P_b \, ReLU(s_b^E - \theta_b)$$

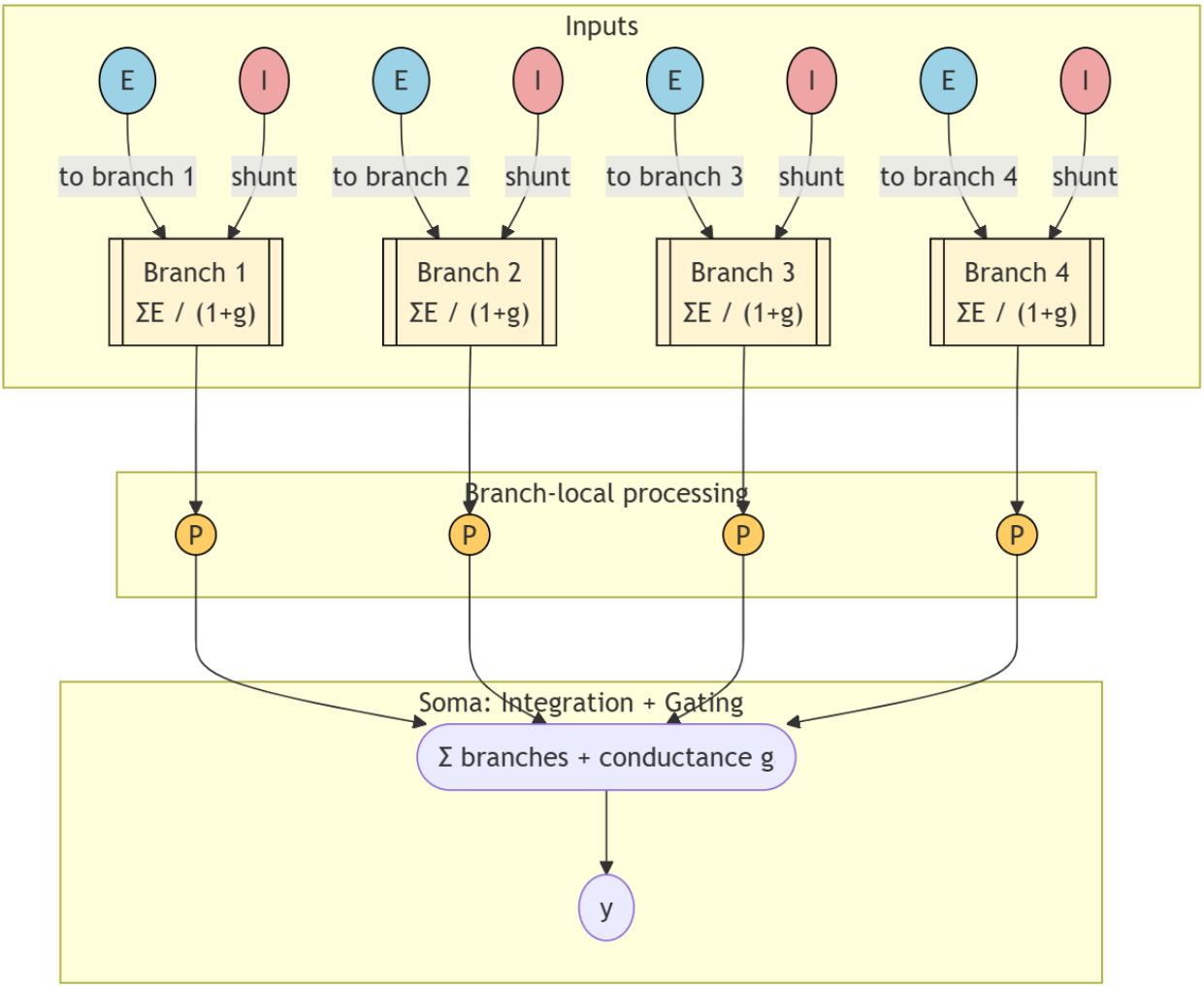
5. Gated Plateau

$$P_b = P_b^{core} \cdot (1 - g_b)$$





We even show the diagram of the model visually as follows: (Mermaid Code for creating this visualization present in the appendix)



Python Code and Preliminary Results

You can find the Python code in the appendix section along with the results. Here we will be explaining the script in brief and discuss the results.

In our script, we implement and benchmark our novel perceptron model Morpho-Conductance Perceptron (MCP) and we compare it against a standard perceptron with Logistic Regression on a synthetic dataset in order to highlight the benefits of branch-local non linear processing.

This script implements and benchmarks a novel neural model called the Morpho-Conductance Perceptron (MCP), inspired by dendritic computation in biological neurons. It compares MCP

against a standard perceptron (logistic regression) on a synthetic dataset designed to highlight the strengths of branch-local nonlinear processing.

The benchmark task is 4D Branch-XOR which contains 4 binary inputs, grouped into two branches: (x0,x1) and (x2,x3).
Label: 1 if exactly one branch is "active" (sum > 0), else 0 (i.e., XOR at branch level). We also introduce gaussian noise to simulate realism.

The following models are compared:
Standard Perceptron: classic logistic regression.

- MCP (full): all mechanisms enabled.
- MCP Ablations:
- No shunt: disables divisive inhibition.
- No plateau: disables plateau boost.
- No gating: disables inhibition gating of plateau.

We got the following benchmark results:

Model	Train Accuracy	Test Accuracy
Standard Perceptron	**0.812**	**0.829**
MCP (Full)	**0.989**	**0.988**
MCP (No Shunt)	**0.834**	**0.829**
MCP (No Plateau)	**0.916**	**0.900**
MCP (No Gating)	**0.918**	**0.900**

We derive the following interpretation based on the script results:

1. MCP (full) achieves almost perfect accuracy, showing its ability to solve nonlinear branch-level XOR tasks that defeat standard perceptrons.
2. In terms of the ablations we performed, we saw that when having no shunt, the performance drops to perceptron level, showing divisive inhibition is critical for nonlinear separation and secondly, when having no plateau/gating, the performance is better than perceptron, but not perfect, indicating plateau augmentation and its gating are important for full nonlinear capacity.

APPENDIX

A. Mermaid Code for Generating the Visualization Chart of MCP

```
flowchart TD
  subgraph INPUTS [Inputs]
    direction TB
    E1([E]) -->|to branch 1| B1
    I1([I]) -->|shunt| B1
    E2([E]) -->|to branch 2| B2
    I2([I]) -->|shunt| B2
    E3([E]) -->|to branch 3| B3
    I3([I]) -->|shunt| B3
    E4([E]) -->|to branch 4| B4
    I4([I]) -->|shunt| B4
  end

  subgraph BRANCHES [Branch-local processing]
    direction LR
    B1[["Branch 1<br/>ΣE / (1+g)"]] --> P1((P))
    B2[["Branch 2<br/>ΣE / (1+g)"]] --> P2((P))
    B3[["Branch 3<br/>ΣE / (1+g)"]] --> P3((P))
    B4[["Branch 4<br/>ΣE / (1+g)"]] --> P4((P))
  end

  subgraph SOMA [Soma: Integration + Gating]
    direction TB
    P1 --> S([Σ branches + conductance gating])
    P2 --> S
    P3 --> S
    P4 --> S
    S --> Y([y])
  end

  %% Styling hints
  classDef exc fill:#9fd3e6,stroke:#000;
  classDef inh fill:#f4a6a6,stroke:#000;
  classDef branch fill:#fff6d6,stroke:#000;
  classDef plateau fill:#ffd166,stroke:#000;
  class E1,E2,E3,E4 exc;
```

```
  class I1,I2,I3,I4 inh;
  class B1,B2,B3,B4 branch;
  class P1,P2,P3,P4 plateau;
```

B. Complete Python Implementation

```
from __future__ import annotations

import math
from dataclasses import dataclass
from typing import List, Dict, Sequence, Tuple

import numpy as np
import argparse
try:
    from tqdm import trange
except Exception:
    trange = None  # fallback: no progress bar


@dataclass
class BranchSpec:
    exc_idx: List[int]
    inh_idx: List[int]


class MorphoConductancePerceptron:
    """
    Morpho-Conductance Perceptron (MCP): a simple perceptron variant with
    branch-local shunting (divisive) conductance and plateau augmentation.

    For branch b:
      sE = sum_i wE[i] * x[i]  over exc_idx[b]
      sI = sum_i wI[i] * x[i]  over inh_idx[b]
      g  = sigmoid(alpha_b * sI + beta_b)  in (0, 1)
      plateau = rho_b * relu(sE - theta_b)
      y_b = sE / (1 + g) + plateau

    Soma: V = sum_b y_b + bias;  y = sigmoid(V)
```

```
    Trained via simple gradient descent (MSE loss by default).
    """

    def __init__(
        self,
        num_inputs: int,
        branches: Sequence[BranchSpec],
        lr: float = 0.05,
        seed: int | None = None,
    ) -> None:
        self.num_inputs = int(num_inputs)
        self.branches = list(branches)
        self.lr = float(lr)
        self.rng = np.random.default_rng(seed)

        # Weights for excitatory and inhibitory channels (per input)
        # stronger init to avoid near-zero sE/sI plateaus
        self.wE = self.rng.normal(0.0, 0.5, size=self.num_inputs)
        self.wI = self.rng.normal(0.0, 0.5, size=self.num_inputs)

        # Per-branch ion-like/shunt/plateau parameters
        B = len(self.branches)
        self.alpha = np.abs(self.rng.normal(1.5, 0.2, size=B))  # stronger
shunt sensitivity >=0
        self.beta = self.rng.normal(0.0, 0.1, size=B)            # shunt
offset
        self.rho = np.abs(self.rng.normal(0.6, 0.1, size=B))    # plateau
gain >=0
        self.theta = self.rng.normal(0.2, 0.05, size=B)        # plateau
threshold

        self.bias = 0.0

        # Ablation controls
        self.disable_shunt = False            # if True, g := 0
        self.disable_plateau = False          # if True, plateau := 0
        self.disable_plateau_gating = False    # if True, plateau not
multiplied by (1-g)
```

```
    def set_ablations(self, *, disable_shunt=False, disable_plateau=False,
disable_plateau_gating=False) -> None:
        self.disable_shunt = bool(disable_shunt)
        self.disable_plateau = bool(disable_plateau)
        self.disable_plateau_gating = bool(disable_plateau_gating)

    @staticmethod
    def _sigmoid(z: np.ndarray | float) -> np.ndarray | float:
        return 1.0 / (1.0 + np.exp(-z))

    @staticmethod
    def _relu(z: np.ndarray | float) -> np.ndarray | float:
        return np.maximum(z, 0.0)

         def  forward(self,  x:  np.ndarray)  ->  Tuple[float,  Dict[str,
np.ndarray]]:
        x = np.asarray(x, dtype=float).reshape(-1)
        assert x.shape[0] == self.num_inputs

        B = len(self.branches)
        sE = np.zeros(B)
        sI = np.zeros(B)
        g = np.zeros(B)
        plateau = np.zeros(B)
        yb = np.zeros(B)

        for b, spec in enumerate(self.branches):
            if spec.exc_idx:
                                sE[b]  =  float(np.dot(self.wE[spec.exc_idx],
x[spec.exc_idx]))
            if spec.inh_idx:
                                sI[b]  =  float(np.dot(self.wI[spec.inh_idx],
x[spec.inh_idx]))
                    g_val  =  float(self._sigmoid(self.alpha[b]  *  sI[b]  +
self.beta[b]))
            if self.disable_shunt:
                g_val = 0.0
            g[b] = g_val
             # Plateau gated by inhibition (1 - g): inhibition suppresses
plateau (unless ablated)
```

```
                    plat_core = float(self.rho[b] * self._relu(sE[b] -
self.theta[b]))
            if self.disable_plateau:
                plat_term = 0.0
            else:
                 plat_term = plat_core if self.disable_plateau_gating else
plat_core * (1.0 - g[b])
            plateau[b] = plat_term
            yb[b] = sE[b] / (1.0 + g[b]) + plateau[b]

        V = float(np.sum(yb) + self.bias)
        y = float(self._sigmoid(V))

        cache = {
             "x": x, "sE": sE, "sI": sI, "g": g, "plateau": plateau, "yb":
yb, "V": V, "y": y
        }
        return y, cache

    def learn(self, x: np.ndarray, target: float) -> float:
        # Forward
        y, c = self.forward(x)
        t = float(target)
        # Binary cross-entropy loss with sigmoid: dL/dV = y - t
        eps = 1e-8
         loss = -(t * math.log(y + eps) + (1.0 - t) * math.log(1.0 - y +
eps))
        dLdV = (y - t)

        # Gradients accumulation
        d_wE = np.zeros_like(self.wE)
        d_wI = np.zeros_like(self.wI)
        d_alpha = np.zeros_like(self.alpha)
        d_beta = np.zeros_like(self.beta)
        d_rho = np.zeros_like(self.rho)
        d_theta = np.zeros_like(self.theta)
        d_bias = dLdV

        for b, spec in enumerate(self.branches):
            sE = c["sE"][b]
```

```
            sI = c["sI"][b]
            g = c["g"][b]

            # dyb/dsE and dyb/dsI
            relu_mask = 1.0 if sE > self.theta[b] else 0.0
            sigp = g * (1.0 - g)
            dgdsi = sigp * self.alpha[b]  # sigmoid'(a*sI+b) * a
            # y = sE/(1+g) + rho*relu(sE-theta)*(1 - g)
            if self.disable_plateau:
                plat_sE_term = 0.0
                plat_g_term = 0.0
            elif self.disable_plateau_gating:
                plat_sE_term = self.rho[b] * relu_mask
                plat_g_term = 0.0
            else:
                plat_sE_term = self.rho[b] * relu_mask * (1.0 - g)
                          plat_g_term = - self.rho[b] * (self._relu(sE -
self.theta[b]))

            dyb_d_sE = 1.0 / (1.0 + g) + plat_sE_term
            # dy/dsI via g: dy/dg = -sE/(1+g)^2 + plat_g_term
            dyg_dg = -sE / ((1.0 + g) ** 2) + plat_g_term
            dyb_d_sI = dyg_dg * dgdsi

            # Propagate to input weights on this branch
            for i in spec.exc_idx:
                d_wE[i] += dLdV * dyb_d_sE * x[i]
            for i in spec.inh_idx:
                d_wI[i] += dLdV * dyb_d_sI * x[i]

            # Branch parameter grads
            # via g: dy/d(alpha) = (dy/dg) * sig'(.) * sI
            d_alpha[b] += dLdV * (dyg_dg * sigp * sI)
            d_beta[b]  += dLdV * (dyg_dg * sigp * 1.0)
            # plateau params
            if not self.disable_plateau:
                gate = 1.0 if self.disable_plateau_gating else (1.0 - g)
                   d_rho[b]    += dLdV * (self._relu(sE - self.theta[b]) *
gate)
                d_theta[b] += dLdV * (-self.rho[b] * relu_mask * gate)
```

```
        # Parameter update (SGD)
        self.wE -= self.lr * d_wE
        self.wI -= self.lr * d_wI
        self.alpha -= self.lr * d_alpha
        self.beta -= self.lr * d_beta
        self.rho -= self.lr * d_rho
        self.theta -= self.lr * d_theta
        # simple projection to keep parameters in sensible ranges
        self.alpha = np.maximum(self.alpha, 0.0)
        self.rho = np.maximum(self.rho, 0.0)
        self.bias -= self.lr * d_bias

        return float(loss)

    def fit(self, X: np.ndarray, y: np.ndarray, epochs: int = 2000, *,
verbose: bool = False, bar_desc: str = "MCP", log_every: int = 0) ->
List[float]:
        losses: List[float] = []
        if verbose and trange is not None:
            pbar = trange(epochs, desc=bar_desc)
            for e in pbar:
                idx = np.random.permutation(len(X))
                epoch_losses = []
                for k in idx:
                    loss = self.learn(X[k], float(y[k]))
                    losses.append(loss)
                    epoch_losses.append(loss)
                if log_every and ((e + 1) % log_every == 0):
                            avg_loss = float(np.mean(epoch_losses)) if
epoch_losses else float('nan')
                    pbar.set_postfix(avg_loss=f"{avg_loss:.4f}")
        else:
            for e in range(epochs):
                idx = np.random.permutation(len(X))
                epoch_losses = []
                for k in idx:
                    loss = self.learn(X[k], float(y[k]))
                    losses.append(loss)
                    epoch_losses.append(loss)
```

```
                if verbose and log_every and ((e + 1) % log_every == 0):
                                avg_loss = float(np.mean(epoch_losses)) if
epoch_losses else float('nan')
                                 print(f"[{bar_desc}] epoch {e+1}/{epochs}
avg_loss={avg_loss:.4f}", flush=True)
        return losses

     # Convenience: deterministic initialization for XOR in 2D with two
branches.
    def init_xor_params(self) -> None:
        assert self.num_inputs == 2 and len(self.branches) == 2
        # Branch 0: excite x0, shunt via x1
        # Branch 1: excite x1, shunt via x0
        self.wE[:] = 0.0
        self.wI[:] = 0.0
        self.wE[0] = 2.0
        self.wE[1] = 2.0
        self.wI[0] = 3.0
        self.wI[1] = 3.0
        # Strong shunting sensitivity; low offset
        self.alpha[:] = 5.0
        self.beta[:] = 0.0
        # Plateau boosts singles; gated off by inhibition when both active
        self.rho[:] = 1.0
        self.theta[:] = 0.5
        # Bias to keep 0,0 below threshold
        self.bias = -1.5

       def predict(self, X: np.ndarray, threshold: float = 0.5) ->
np.ndarray:
        preds = []
        for x in X:
            yhat, _ = self.forward(x)
            preds.append(1.0 if yhat >= threshold else 0.0)
        return np.array(preds)


#                 ------------------------------                  Demo
------------------------------------ #
```

```
def _xor_demo():
    # XOR in 2D
    X = np.array([
        [0.0, 0.0],
        [0.0, 1.0],
        [1.0, 0.0],
        [1.0, 1.0],
    ], dtype=float)
    y = np.array([0.0, 1.0, 1.0, 0.0], dtype=float)

     # Two branches: each one excites on one feature and shunts via the
other
    branches = [
        BranchSpec(exc_idx=[0], inh_idx=[1]),
        BranchSpec(exc_idx=[1], inh_idx=[0]),
    ]

       # Start from a deterministic XOR-friendly initialization, then
(optionally) fine-tune
       m = MorphoConductancePerceptron(num_inputs=2, branches=branches,
lr=0.05, seed=0)
    m.init_xor_params()
    # Optional fine-tuning (short)
    m.fit(X, y, epochs=200)
    preds = m.predict(X)
    acc = (preds == y).mean()
    print("MCP XOR accuracy:", acc)
    for xi, yi, pi in zip(X, y, preds):
        yh, cache = m.forward(xi)
        print(f"x={xi}, y*={int(yi)}, yhat={yh:.3f}, pred={int(pi)}")


#       --------------------------       Baseline       and       Benchmarks
---------------------- #

class StandardPerceptron:
    """Sigmoid perceptron (logistic regression) baseline."""
     def __init__(self, num_inputs: int, lr: float = 0.1, seed: int | None
= None) -> None:
        rng = np.random.default_rng(seed)
```

```
        self.w = rng.normal(0.0, 0.1, size=num_inputs)
        self.b = 0.0
        self.lr = float(lr)

    @staticmethod
    def _sigmoid(z):
        return 1.0 / (1.0 + np.exp(-z))

    def forward(self, x: np.ndarray) -> float:
        x = np.asarray(x, dtype=float).reshape(-1)
        return float(self._sigmoid(np.dot(self.w, x) + self.b))

    def learn(self, x: np.ndarray, t: float) -> float:
        y = self.forward(x)
        eps = 1e-8
        loss = -(t * np.log(y + eps) + (1 - t) * np.log(1 - y + eps))
        dLdV = y - t
        self.w -= self.lr * dLdV * x
        self.b -= self.lr * dLdV
        return float(loss)

     def fit(self, X: np.ndarray, y: np.ndarray, epochs: int = 500, *,
verbose: bool = False, bar_desc: str = "Perceptron", log_every: int = 0)
-> None:
        if verbose and trange is not None:
            pbar = trange(epochs, desc=bar_desc)
            for e in pbar:
                idx = np.random.permutation(len(X))
                epoch_losses = []
                for k in idx:
                    epoch_losses.append(self.learn(X[k], float(y[k])))
                if log_every and ((e + 1) % log_every == 0):
                                avg_loss = float(np.mean(epoch_losses)) if
epoch_losses else float('nan')
                    pbar.set_postfix(avg_loss=f"{avg_loss:.4f}")
        else:
            for e in range(epochs):
                idx = np.random.permutation(len(X))
                epoch_losses = []
                for k in idx:
```

```
                    epoch_losses.append(self.learn(X[k], float(y[k])))
                if verbose and log_every and ((e + 1) % log_every == 0):
                                avg_loss = float(np.mean(epoch_losses)) if
epoch_losses else float('nan')
                                 print(f"[{bar_desc}] epoch {e+1}/{epochs}
avg_loss={avg_loss:.4f}", flush=True)

        def predict(self, X: np.ndarray, threshold: float = 0.5) ->
np.ndarray:
         return np.array([1.0 if self.forward(x) >= threshold else 0.0 for
x in X])


def gen_branch_xor_dataset(n: int = 400, noise: float = 0.1, seed: int |
None = 0):
     """Generate a 4D dataset with two branches performing XOR at branch
level.
      - Inputs: x0,x1 (branch A), x2,x3 (branch B). Each in {0,1} with
additive Gaussian noise.
    - Label: 1 if exactly one branch is active (sum > 0.5), else 0.
    """
    rng = np.random.default_rng(seed)
    Xb = rng.integers(0, 2, size=(n, 4)).astype(float)
    if noise > 0:
        X = Xb + rng.normal(0.0, noise, size=Xb.shape)
    else:
        X = Xb
    sA = Xb[:, 0] + Xb[:, 1]
    sB = Xb[:, 2] + Xb[:, 3]
    y = ((sA > 0) ^ (sB > 0)).astype(float)
    return X, y


def _mcp_for_4d(branches: Sequence[BranchSpec], lr: float = 0.05, seed:
int | None = None) -> MorphoConductancePerceptron:
        m = MorphoConductancePerceptron(num_inputs=4, branches=branches,
lr=lr, seed=seed)
    return m
```

```
def run_benchmarks():
    # Dataset
    X, y = gen_branch_xor_dataset(n=800, noise=0.15, seed=0)
    # Train/test split
    n = len(X)
    idx = np.random.default_rng(0).permutation(n)
    split = int(0.7 * n)
    tr_idx, te_idx = idx[:split], idx[split:]
    Xtr, Ytr = X[tr_idx], y[tr_idx]
    Xte, Yte = X[te_idx], y[te_idx]

    # Branch layout: A=(0,1), B=(2,3)
    branches = [
        BranchSpec(exc_idx=[0, 1], inh_idx=[2, 3]),
        BranchSpec(exc_idx=[2, 3], inh_idx=[0, 1]),
    ]

    # Baseline perceptron
    print("[RUN] Training baseline perceptron...", flush=True)
    sp = StandardPerceptron(num_inputs=4, lr=0.1, seed=0)
        sp.fit(Xtr,  Ytr,  epochs=1000,  verbose=True,  bar_desc="Baseline
Perceptron", log_every=50)
    print("[DONE] Baseline perceptron.", flush=True)
    sp_tr = (sp.predict(Xtr) == Ytr).mean()
    sp_te = (sp.predict(Xte) == Yte).mean()

    # MCP full
    print("[RUN] Training MCP (full)...", flush=True)
    m_full = _mcp_for_4d(branches, lr=0.05, seed=0)
     m_full.fit(Xtr, Ytr, epochs=2000, verbose=True, bar_desc="MCP full",
log_every=100)
    print("[DONE] MCP (full).", flush=True)
    full_tr = (m_full.predict(Xtr) == Ytr).mean()
    full_te = (m_full.predict(Xte) == Yte).mean()

    # Ablations
    print("[RUN] Training MCP (no shunt)...", flush=True)
    m_no_shunt = _mcp_for_4d(branches, lr=0.05, seed=1)
    m_no_shunt.set_ablations(disable_shunt=True)
```

```
    m_no_shunt.fit(Xtr, Ytr, epochs=2000, verbose=True, bar_desc="MCP no
shunt", log_every=100)
    print("[DONE] MCP (no shunt).", flush=True)
    no_shunt_tr = (m_no_shunt.predict(Xtr) == Ytr).mean()
    no_shunt_te = (m_no_shunt.predict(Xte) == Yte).mean()

    print("[RUN] Training MCP (no plateau)...", flush=True)
    m_no_plat = _mcp_for_4d(branches, lr=0.05, seed=2)
    m_no_plat.set_ablations(disable_plateau=True)
    m_no_plat.fit(Xtr, Ytr, epochs=2000, verbose=True, bar_desc="MCP no
plateau", log_every=100)
    print("[DONE] MCP (no plateau).", flush=True)
    no_plat_tr = (m_no_plat.predict(Xtr) == Ytr).mean()
    no_plat_te = (m_no_plat.predict(Xte) == Yte).mean()

    print("[RUN] Training MCP (no gating)...", flush=True)
    m_no_gate = _mcp_for_4d(branches, lr=0.05, seed=3)
    m_no_gate.set_ablations(disable_plateau_gating=True)
    m_no_gate.fit(Xtr, Ytr, epochs=2000, verbose=True, bar_desc="MCP no
gating", log_every=100)
    print("[DONE] MCP (no gating).", flush=True)
    no_gate_tr = (m_no_gate.predict(Xtr) == Ytr).mean()
    no_gate_te = (m_no_gate.predict(Xte) == Yte).mean()

    print("\nBenchmark: 4D Branch-XOR with noise=0.15 (70/30 split)")
       print(f"Standard perceptron acc    - train: {sp_tr:.3f}    test:
{sp_te:.3f}")
     print(f"MCP (full) acc               - train: {full_tr:.3f}   test:
{full_te:.3f}")
     print(f"MCP ablation - no shunt   - train: {no_shunt_tr:.3f}   test:
{no_shunt_te:.3f}")
     print(f"MCP ablation - no plateau- train: {no_plat_tr:.3f}   test:
{no_plat_te:.3f}")
     print(f"MCP ablation - no gating - train: {no_gate_tr:.3f}   test:
{no_gate_te:.3f}")


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
```

```
        parser.add_argument("--bench",   action="store_true",   help="Run
benchmark and ablation suite")
    args = parser.parse_args()
    if args.bench:
        run_benchmarks()
    else:
        _xor_demo()
```

C. Python Script Results and Evaluation

```
(cogsci) C:\Arya\cogsci\assignment1\deliverable1>python mcp.py --bench
[RUN] Training baseline perceptron...
Baseline Perceptron: 100%|| 1000/1000 [00:03<00:00, 327.26it/s, avg_loss=0.4932]
[DONE] Baseline perceptron.
[RUN] Training MCP (full)...
MCP full: 100%|| 2000/2000 [01:40<00:00, 19.86it/s, avg_loss=0.0339]
[DONE] MCP (full).
[RUN] Training MCP (no shunt)...
MCP no shunt: 100%|| 2000/2000 [01:44<00:00, 19.22it/s, avg_loss=0.4188]
[DONE] MCP (no shunt).
[RUN] Training MCP (no plateau)...
MCP no plateau: 100%|| 2000/2000 [01:15<00:00, 26.43it/s, avg_loss=0.2909]
[DONE] MCP (no plateau).
[RUN] Training MCP (no gating)...
MCP no gating: 100%|| 2000/2000 [01:04<00:00, 30.87it/s, avg_loss=0.2909]
[DONE] MCP (no gating).

Benchmark: 4D Branch-XOR with noise=0.15 (70/30 split)
Standard perceptron acc  - train: 0.812  test: 0.829
MCP (full) acc           - train: 0.989  test: 0.988
MCP ablation - no shunt  - train: 0.834  test: 0.829
MCP ablation - no plateau- train: 0.916  test: 0.900
MCP ablation - no gating - train: 0.918  test: 0.900
```