Math 564 - Project 02

Sajjad Uddin Mahmud

October 21, 2025

1 Introduction

2 Data Preparation

The Somerville happiness dataset contained 3,669 survey responses across multiple wards of the city, with one target variable (*Happiness*) and eight ordinal predictor variables representing satisfaction in different aspects of life. Each feature took values from 1 (very unsatisfied) to 5 (very satisfied) as inputs.

Handling Missing and Invalid Entries: Rows with missing target values could not be used for model training or evaluation and were therefore removed. Among the remaining records, rows containing six or more missing feature values (out of eight) were also discarded to prevent distortion caused by heavy imputation and to maintain data reliability.

Hierarchical Median Imputation: For the remaining data, missing values in the eight satisfaction features were imputed using a hierarchical median approach. Specifically, for each feature, the median value within the respondent's ward was used whenever available, reflecting local neighborhood-level satisfaction patterns. When a ward had insufficient data for a given feature, the overall citywide median for that feature was used instead. This approach preserves the ordinal nature of the satisfaction variables while maintaining robustness against outliers.

Normalization of Ordinal Features: All eight satisfaction features were then normalized to the interval $[0, 1]$ using the transformation

$$x_{\text{scaled}} = \frac{x - 1}{4},$$

which maps the original 1–5 Likert scale to a consistent numeric range. This scaling improves numerical conditioning and prevents sigmoid activation saturation during neural network optimization.

Target Encoding: Finally, the target variable (*Happiness*) was encoded as fractional values $\{1/6, 2/6, 3/6, 4/6, 5/6\}$ to align with the sigmoid network output range, allowing the model output to represent the ordered nature of happiness levels.

The whole process is shown in Algorithm 1

Algorithm 1 data preparation for somerville happiness dataset

Require: raw table $\mathcal{D}_{\text{raw}}$ with columns: ward w, target y (happiness $\in \{1, \ldots, 5\}$), and features $\mathcal{F} = \{f_1, \ldots, f_8\}$ (eight 1–5 likert items)

Require: row-drop threshold $m_{\text{th}} \leftarrow 6$ (drop rows with $\geq m_{\text{th}}$ missing features)

Ensure: processed table $\mathcal{D}_{\text{proc}}$; imputation summary $\mathcal{S}$

$\mathcal{D} \leftarrow \mathcal{D}_{\text{raw}}$

drop rows with missing target: $\mathcal{D} \leftarrow \{r \in \mathcal{D} \mid r.y \neq \text{nan}\}$ ▷ remove rows that cannot be used for training or evaluation

drop rows with too many missing features: $\mathcal{D} \leftarrow \{r \in \mathcal{D} \mid |\{f \in \mathcal{F} : r.f = \text{nan}\}| < m_{\text{th}}\}$

compute ward-level medians: for each $f \in \mathcal{F}$ and ward w, let $m_{w,f} \leftarrow \text{median}\{r.f : r \in \mathcal{D}, r.w = w, r.f \neq \text{nan}\}$

compute global medians: for each $f \in \mathcal{F}$, let $m_f \leftarrow \text{median}\{r.f : r \in \mathcal{D}, r.f \neq \text{nan}\}$

```
for each row r ∈ D do
    for each feature f ∈ F do
        if r.f = nan then
            if m_{r.w,f} is defined then
                r.f ← m_{r.w,f}        ▷ impute by ward median when available
            else
                r.f ← m_f              ▷ fallback to global median
            end if
        end if
    end for
end for
```

normalize features to $[0, 1]$: for each $f \in \mathcal{F}$ and row r, set $r.f_{\text{scaled}} \leftarrow \text{clip}(r.f, 1, 5)$ then $r.f_{\text{scaled}} \leftarrow (r.f_{\text{scaled}} - 1)/4$

encode target: for each row r, set $r.y_{\text{enc}} \leftarrow r.y/6$ ▷ $\{1, \ldots, 5\} \mapsto \{1/6, \ldots, 5/6\}$

record summary: $\mathcal{S} \leftarrow$ {initial rows, rows dropped (missing target), rows dropped ($\geq m_{\text{th}}$ missing), global medians m_f, counts of ward medians used}

return $\mathcal{D}_{\text{proc}} \leftarrow \mathcal{D}$, $\mathcal{S}$

3 Training and Test Data Selection

To evaluate the generalization capability of the neural network, the processed dataset was divided into separate training and testing subsets. A stratified random split was employed to preserve the class proportions of the target variable (*Happiness* levels 1–5) in both sets.

In this approach, the data were first grouped by their discrete happiness category. Within each group, approximately 30% of the samples were randomly assigned to the test set, while the remaining 70% were used for training. Stratification was selected to prevent bias toward dominant classes (e.g., “4” and “5”), ensuring that rare classes (such as “1”) were represented proportionally in both subsets.

4 Results

4.1 Data Preparation Results

After applying the above preprocessing steps, the dataset was cleaned and normalized. The resulting record counts are summarized in Table 4.1:

Table 4.1: Number of data points

Description	**Count**
Initial number of rows	3,669
Rows dropped (missing target)	34
Rows dropped (too many missing features)	8
Final number of usable rows	**3,627**

Thus, a total of 42 records (approximately 1.1% of the dataset) were excluded, leaving 3,627 fully processed entries for model development.

4.2 Train–Test Split Results

After applying the stratified 70/30 division, the resulting sample counts are summarized in Table 4.2.

Table 4.2: Stratified train–test split summary.

Statistic	**Training Set**	**Test Set**
Total rows	2,539	1,088
Happiness = 1	46	20
Happiness = 2	88	38
Happiness = 3	472	202
Happiness = 4	1,271	544
Happiness = 5	662	284

4.3 Confusion Matrix

The feed-forward neural network (FFNN) model, consisting of eight input features, two hidden layers of 12 and 10 neurons, and a single sigmoid output neuron, was trained using two optimization methods: (i) **BFGS with Strong Wolfe line search**, and (ii) **Gradient Descent with Strong Wolfe line search**.

Both models were evaluated on the stratified test dataset. The resulting confusion matrices are shown in Table 4.3 and 4.3.

Table 4.3: Confusion Matrix — BFGS + Strong Wolfe

True \ Predicted	**1**	**2**	**3**	**4**	**5**
1	1	2	3	12	2
2	2	2	6	25	3
3	2	11	12	163	14
4	3	13	16	475	37
5	6	6	6	211	55

Table 4.4: Confusion Matrix — GD + Strong Wolfe

True \ Predicted	**1**	**2**	**3**	**4**	**5**
1	0	3	4	13	0
2	0	0	11	27	0
3	0	0	24	178	0
4	0	0	21	523	0
5	0	0	7	277	0

The confusion matrices in Tables 4.3 and 4.3 reveal consistent prediction patterns across both optimization methods. In both cases, the network displays a strong tendency to classify most samples as belonging to the middle or higher satisfaction levels, particularly class 4 (*Satisfied*). This behavior reflects the underlying imbalance in the dataset, where classes 4 and 5 together account for more than 70% of all responses in the training set.

5 Exploration of Alternative FFNN Constructions

Beyond the baseline architecture of two hidden layers with 12 and 10 neurons, several alternative network structures can be explored to assess model performance. A couple of things may be explored like:

- Increasing or decreasing the number of hidden units
- Adding or removing hidden layers
- Variations in activation functions (e.g., ReLU instead of sigmoid)

6 Repository

All code and iteration logs for this project are available in a public GitHub repository:

`https://github.com/sajjad30148/WSU_Math564_Fall2025`

The repository includes the code, and result folders containing iterations from each run.

7 Code

The full Python implementation is provided below. It can also be found in the repository.

7.1 Main Python Script

Listing 1: project01_main.py

```
# ================================================================
# Project 02 Main Script
# ================================================================

import os
from datetime import datetime
import numpy as np
import pandas as pd
from pathlib import Path
import sys
import matplotlib.pyplot as plt

workspace_root = Path(__file__).resolve().parent.parent
sys.path.insert(0, str(workspace_root))

import functions as fn

# ------------------------------------------------------------
# user settings
# ------------------------------------------------------------

# data file path
data_path = r"D:\One_Drive_Sajjad\OneDrive - Washington State University (email.wsu.edu)\Documents\Sajjad_Uddin_Mahmud\Courses\5. Fall 2025\MATH_564\Projects\WSU_Math564_Fall2025\Project_02\happiness.csv"

# data preparation settings
ward_col = "Ward/Neighborhood"
target_col = "Happiness"
feature_cols = [
    "Beauty of Neighborhood",
    "Convenience of Getting Around",
    "Housing Condition",
    "Street and Sidewalk Maintenance",
    "Public Schools",
```

```python
    "Police Department",
    "Community Events",
    "City Services Information",
]
drop_row_missing_threshold = 6 # rows with >= this many missing features will be dropped
keep_raw_features = True # set to False to overwrite original columns

# train/test split settings
test_size = 0.30
random_state = 42
stratify = True
strat_col = target_col

# ------------------------------------------------------------
# data preparation function
# ------------------------------------------------------------

def prepare_happiness_data(
    df,
    ward_col = "Ward/Neighborhood",
    target_col = "Happiness",
    feature_cols = None,
    drop_row_missing_threshold = 6,
    keep_raw_features = True,
):
    """
    prepare the somerville happiness dataset for modeling

    inputs
        df pandas dataframe with raw data
        ward_col column name for ward / neighborhood
        target_col column name for happiness (1..5)
        feature_cols list of the eight satisfaction feature column names
        drop_row_missing_threshold rows with >= this many missing features will be dropped
        keep_raw_features if true, original 1..5 features are kept; scaled features
                                    are added with suffix '_scaled'. if false, originals are
                                    overwritten by scaled values.

    outputs
        df_out processed dataframe
```

```
        prep_info dictionary with imputation and drop statistics for reporting
    """

    if feature_cols is None:
        # set the expected eight feature names here
        feature_cols = [
            "Beauty of Neighborhood",
            "Convenience of Getting Around",
            "Housing Condition",
            "Street and Sidewalk Maintenance",
            "Public Schools",
            "Police Department",
            "Community Events",
            "City Services Information",
        ]

    df = df.copy()

    # ------------------------------------------------------------
    # drop unusable rows
    # ------------------------------------------------------------

    # remove rows with missing target
    n0 = len(df)
    mask_missing_target = df[target_col].isna()
    n_missing_target = int(mask_missing_target.sum())
    df = df.loc[~mask_missing_target].reset_index(drop = True)

    # remove rows with too many missing features
    feature_missing_count = df[feature_cols].isna().sum(axis = 1)
    mask_too_many_missing = feature_missing_count >= drop_row_missing_threshold
    n_drop_too_many_missing = int(mask_too_many_missing.sum())
    if n_drop_too_many_missing > 0:
        df = df.loc[~mask_too_many_missing].reset_index(drop = True)

    # ------------------------------------------------------------
    # compute medians (ward-level and global)
    # ------------------------------------------------------------

    # ward-level medians
    ward_medians = {}
    for col in feature_cols:
        ward_medians[col] = df.groupby(ward_col)[col].median()
```

```python

    # global medians
    global_medians = {col: float(df[col].median()) for col in
        feature_cols}

    # ------------------------------------------------------------
    # impute feature missing values by ward -> global
    # ------------------------------------------------------------

    # try ward-level median first, then global median
    for col in feature_cols:
        ward_median_col = df.groupby(ward_col)[col].transform("median")
        df[col] = df[col].fillna(ward_median_col)

        # fallback to global median if ward median was nan or still missing
        df[col] = df[col].fillna(global_medians[col])

    # ------------------------------------------------------------
    # normalize features to [0, 1] by (x - 1) / 4
    # ------------------------------------------------------------

    # scale function with clamping
    def scale_to_unit_interval(x):
        x = x.astype(float)

        # clamp to [1, 5] for safety, then scale
        x = np.clip(x, 1.0, 5.0)

        return (x - 1.0) / 4.0

    # apply scaling
    if keep_raw_features:
        for col in feature_cols:
            df[f"{col}_scaled"] = scale_to_unit_interval(df[col])
        scaled_feature_cols = [f"{c}_scaled" for c in feature_cols]
    else:
        for col in feature_cols:
            df[col] = scale_to_unit_interval(df[col])
        scaled_feature_cols = feature_cols

    # ------------------------------------------------------------
    # encode target as fractions {1/6, ..., 5/6}
    # ------------------------------------------------------------
    # happiness expected in {1, 2, 3, 4, 5}
    # encode as k / 6 where k in {1..5}
```

```
    df["Happiness_Encoded"] = df[target_col].astype(float) / 6.0

    # assemble prep info for reporting
    df_out = df
    prep_info = {
        "n_rows_initial" : n0,
        "n_missing_target_dropped" : n_missing_target,
        "n_drop_too_many_missing" : n_drop_too_many_missing,
        "feature_cols" : feature_cols,
        "scaled_feature_cols" : scaled_feature_cols,
        "ward_medians_available_for_cols" : {
            col: int(ward_medians[col].notna().sum()) for col in
                feature_cols
        },
        "global_medians" : global_medians,
        "scaling" : "(x - 1) / 4 with clamp to [1, 5]",
        "target_encoding" : "Happiness_Encoded = Happiness / 6",
        "ward_col" : ward_col,
        "target_col" : target_col,
    }

    return df_out, prep_info


if __name__ == "__main__":


    # ------------------------------------------------------------
    # load raw data
    # ------------------------------------------------------------
    data_path_obj = Path(data_path)
    df_raw = pd.read_csv(data_path_obj)

    # ------------------------------------------------------------
    # prepare data (no saving inside the function)
    # ------------------------------------------------------------
    df_proc, prep_info = prepare_happiness_data(
        df_raw,
        ward_col = ward_col,
        target_col = target_col,
        feature_cols = feature_cols,
        drop_row_missing_threshold = drop_row_missing_threshold,
        keep_raw_features = keep_raw_features,
    )

```

```
# ----------------------------------------------------------
# save processed data as csv (done here, not in the function)
# ----------------------------------------------------------
out_path = data_path_obj.with_name("happiness_processed.csv")
df_proc.to_csv(out_path, index = False)

# simple console report
print("processed data saved to :", out_path)
print("rows initial :", prep_info["n_rows_initial"])
print("rows dropped (target) :",
    prep_info["n_missing_target_dropped"])
print("rows dropped (>= missing threshold) :",
    prep_info["n_drop_too_many_missing"])

# ----------------------------------------------------------
# train/test split
# ----------------------------------------------------------
if stratify:
    df_train, df_test = fn.stratified_split_df(
        df_proc,
        target_col = strat_col,
        test_size = test_size,
        random_state = random_state,
)
else:
    # simple random split without stratification
    df_shuf = df_proc.sample(frac = 1.0, random_state =
        random_state).reset_index(drop = True)
    n_total = len(df_shuf)
    n_test = int(round(test_size * n_total))
    n_test = max(1, min(n_test, n_total - 1))
    df_test = df_shuf.iloc[:n_test].reset_index(drop = True)
    df_train = df_shuf.iloc[n_test:].reset_index(drop = True)

# simple train/test split report
print("train rows :", len(df_train))
print("test rows :", len(df_test))
print("train class counts :")
print(df_train[strat_col].value_counts().sort_index())
print("test class counts :")
print(df_test[strat_col].value_counts().sort_index())

# select features (scaled) and target (encoded)
feat_cols = prep_info["scaled_feature_cols"]
y_col_enc = "Happiness_Encoded"
```

```python

X_train = df_train[feat_cols].to_numpy(dtype = float)
y_train = df_train[y_col_enc].to_numpy(dtype = float).reshape(-1, 1)
X_test = df_test[feat_cols].to_numpy(dtype = float)
y_test = df_test[y_col_enc].to_numpy(dtype = float).reshape(-1, 1)

# layer sizes for the required architecture
layer_sizes = [X_train.shape[1], 12, 10, 1]

# create feedforward neural network objective
f, g = fn.make_ffnn_objective(X_train, y_train, layer_sizes =
    layer_sizes, l2 = 0.0)

# initialize parameters
w0 = fn.ffnn_init_params(layer_sizes, random_state =
    np.random.default_rng(42))

# ------------------------------------------------------------
# helper: convert scalar outputs in (0,1) to classes {1..5}
# targets are spaced at {1/6, 2/6, 3/6, 4/6, 5/6}
# ------------------------------------------------------------
targets_enc = np.arange(1, 6, dtype = float).reshape(-1, 1) / 6.0 #
    shape (5,1)
targets_cls = np.arange(1, 6, dtype = int).reshape(-1, 1) # shape
    (5,1)

def decode_to_class(yhat):
    # yhat: (n,1) in (0,1); snap to nearest of the five target points
    diffs = np.abs(yhat - targets_enc.T) # (n,5)
    idx = np.argmin(diffs, axis = 1) # (n,)
    return targets_cls[idx, 0] # (n,)


# ------------------------------------------------------------
# run method 1: bfgs + strong wolfe
# ------------------------------------------------------------
bfgs_opts = {"max_iter" : 500, "tol" : 1e-6}
res_bfgs = fn.quasi_newton_bfgs(
    f = f,
    grad = g,
    x0 = w0,
    line_search = fn.strong_wolfe,
    opts = bfgs_opts
)
w_bfgs = res_bfgs["x"]
```

```python

# ------------------------------------------------------------
# run method 2: gradient descent + strong wolfe
# ------------------------------------------------------------
from functions import gradient_descent

gd_opts = {"max_iter" : 200, "tol" : 1e-6}
res_gd = fn.gradient_descent(
    f = f,
    grad = g,
    x0 = w0,
    line_search = fn.strong_wolfe,
    opts = gd_opts
)
w_gd = res_gd["x"]



# ------------------------------------------------------------
# prediction helper
# ------------------------------------------------------------
def ffnn_predict(X, w, layer_sizes):
    shapes = []
    sizes = []
    for l in range(1, len(layer_sizes)):
        d_prev = layer_sizes[l - 1]
        d_curr = layer_sizes[l]
        shapes.append(((d_prev, d_curr), (d_curr,)))
        sizes.append((d_prev * d_curr, d_curr))
    idx = 0
    slices = []
    for (w_count, b_count) in sizes:
        sW = slice(idx, idx + w_count)
        idx += w_count
        sb = slice(idx, idx + b_count)
        idx += b_count
        slices.append((sW, sb))

    def sigmoid(u):
        return 1.0 / (1.0 + np.exp(-u))

    a = X
    for (sW, sb), (shapeW, shapeb) in zip(slices, shapes):
        W = w[sW].reshape(shapeW)
        b = w[sb].reshape(shapeb)
```

```
            a = sigmoid(a @ W + b)
        return a

    # test predictions for both methods
    yhat_bfgs = ffnn_predict(X_test, w_bfgs, layer_sizes)
    yhat_gd = ffnn_predict(X_test, w_gd, layer_sizes)

    print("GD yhat min/max/mean:", float(yhat_gd.min()),
        float(yhat_gd.max()), float(yhat_gd.mean()))
    print("Unique predicted classes (GD):",
        np.unique(decode_to_class(yhat_gd)))
    print("Are GD weights unchanged from init?:", np.allclose(w0, w_gd))

    ycls_bfgs = decode_to_class(yhat_bfgs)
    ycls_gd = decode_to_class(yhat_gd)
    ytrue_cls = decode_to_class(y_test)

    # ------------------------------------------------------------
    # Confusion matrix
    # ------------------------------------------------------------

    cm_bfgs = pd.crosstab(
        pd.Series(ytrue_cls.flatten(), name = "True"),
        pd.Series(ycls_bfgs.flatten(), name = "Predicted"),
    )
    cm_gd = pd.crosstab(
        pd.Series(ytrue_cls.flatten(), name = "True"),
        pd.Series(ycls_gd.flatten(), name = "Predicted"),
    )

    print("\nConfusion Matrix: BFGS + Strong Wolfe")
    print(cm_bfgs)
    print("\nConfusion Matrix: Gradient Descent + Strong Wolfe")
    print(cm_gd)

    classes = [1, 2, 3, 4, 5]

    # reindex to full 1...5 grid for neat printing/saving
    cm_bfgs = cm_bfgs.reindex(index = classes, columns = classes,
        fill_value = 0)
    cm_gd = cm_gd.reindex(index = classes, columns = classes, fill_value
        = 0)


    # ------------------------------------------------------------
```

```python
    # save to ./results/evaluation.txt
    # ------------------------------------------------------------
    results_dir = Path(__file__).resolve().parent / "results"
    results_dir.mkdir(parents = True, exist_ok = True)
    with open(os.path.join(results_dir, "evaluation.txt"), "w") as f:
        f.write("confusion matrix: bfgs + strong wolfe\n")
        f.write(str(cm_bfgs) + "\n\n")
        f.write("confusion matrix: gradient descent + strong wolfe\n")

    print("saved confusion matrices to ./results/evaluation.txt")

# ================================================================
```

7.2 Functions

Listing 2: functions.py

```
# ---------------------------------------------------------------
# fast feedforward neural networkjective
# ---------------------------------------------------------------

def make_ffnn_objective(X, y, layer_sizes, l2 = 0.0):
    """
    create closures f(w) and grad(w) for a general ffnn

    inputs
       X array (n, d_in), features
       y array (n,) or (n, 1), targets in {1/6, ..., 5/6}
       layer_sizes list like [d_in, h1, ..., d_out] with d_out = 1
       l2 nonnegative l2 regularization on weights (biases excluded)

    returns
       f callable, f(w) -> scalar loss
       grad callable, grad(w) -> flat gradient vector
    """

    X = np.asarray(X, dtype = float)
    y = np.asarray(y, dtype = float).reshape(-1, 1)

    n, d_in = X.shape
    assert layer_sizes[0] == d_in, "input dimension must match
        layer_sizes[0]"
    assert layer_sizes[-1] == 1, "output dimension must be 1 for this
        setup"

    # precompute parameter slices
    # for each layer l: W in R^{d_{l-1} x d_l}, b in R^{d_l}
    shapes = []
    sizes = []
    for l in range(1, len(layer_sizes)):
       d_prev = layer_sizes[l - 1]
       d_curr = layer_sizes[l]
       w_count = d_prev * d_curr
       b_count = d_curr
       shapes.append(((d_prev, d_curr), (d_curr,)))
       sizes.append((w_count, b_count))

    # compute flat index ranges
    idx = 0
```

```
    slices = []
    for (w_count, b_count) in sizes:
        sW = slice(idx, idx + w_count)
        idx += w_count
        sb = slice(idx, idx + b_count)
        idx += b_count
        slices.append((sW, sb))
    p_expected = idx

    # cached activations for backprop; created locally inside f/grad
    def _sigmoid(u):
        return 1.0 / (1.0 + np.exp(-u))

    def f(w):
        w = np.asarray(w, dtype = float)
        if w.size != p_expected:
            raise ValueError("size of w does not match architecture")

        # unpack and forward pass
        a_list = [X] # a_0
        params = []
        a = X
        for (sW, sb), (shapeW, shapeb) in zip(slices, shapes):
            W = w[sW].reshape(shapeW)
            b = w[sb].reshape(shapeb)
            params.append((W, b))
            z = a @ W + b
            a = _sigmoid(z)
            a_list.append(a)

        yhat = a_list[-1] # (n, 1)
        residual = yhat - y
        loss = 0.5 * np.sum(residual * residual)

        if l2 > 0.0:
            reg = 0.0
            for (W, b) in params:
                reg += np.sum(W * W)
            loss += 0.5 * l2 * reg

        return float(loss)

    def grad(w):
        w = np.asarray(w, dtype = float)
        if w.size != p_expected:
```

```
            raise ValueError("size of w does not match architecture")

        # unpack and forward pass (store activations for backprop)
        a_list = [X]
        z_list = []
        params = []
        a = X
        for (sW, sb), (shapeW, shapeb) in zip(slices, shapes):
            W = w[sW].reshape(shapeW)
            b = w[sb].reshape(shapeb)
            params.append((W, b))
            z = a @ W + b
            a = _sigmoid(z)
            z_list.append(z)
            a_list.append(a)

        yhat = a_list[-1]

        # initial delta at output: (yhat - y) * sigma'(z_L)
        # derivative of sigmoid via activation: a * (1 - a)
        delta = (yhat - y) * (yhat * (1.0 - yhat)) # (n, 1)

        # backprop
        dWs = [None] * len(params)
        dbs = [None] * len(params)

        for l in reversed(range(len(params))):
            a_prev = a_list[l] # (n, d_l)
            W, b = params[l] # W: (d_l, d_{l+1}), b: (d_{l+1},)

            dW = a_prev.T @ delta # (d_l, d_{l+1})
            db = np.sum(delta, axis = 0) # (d_{l+1},)
            if l2 > 0.0:
                dW += l2 * W

            dWs[l] = dW
            dbs[l] = db

            if l > 0:
                a_prev_act = a_list[l] # activation at layer l
                delta = (delta @ W.T) * (a_prev_act * (1.0 - a_prev_act))

        # pack grads into a flat vector following the same order
        g = np.empty(p_expected, dtype = float)
        for (sW, sb), dW, db in zip(slices, dWs, dbs):
```

```python
            g[sW] = dW.ravel()
            g[sb] = db.ravel()

        return g

    return f, grad


def ffnn_init_params(layer_sizes, random_state = None):
    """
    xavier/glorot uniform initializer for sigmoid networks

    inputs
        layer_sizes [d_in, h1, ..., d_out]
        random_state seed or numpy random generator or None

    returns
        w0 flat parameter vector
    """
    if random_state is None:
        random_state = np.random.default_rng(42)

    blocks = []
    for l in range(1, len(layer_sizes)):
        d_prev = layer_sizes[l - 1]
        d_curr = layer_sizes[l]
        limit = np.sqrt(6.0 / (d_prev + d_curr))
        W = random_state.uniform(-limit, limit, size = (d_prev, d_curr))
        b = np.zeros((d_curr,), dtype = float)
        blocks.append(W.ravel())
        blocks.append(b)
    return np.concatenate(blocks, axis = 0)




# ------------------------------------------------------------
# Miscellaneous functions
# ------------------------------------------------------------

# stratified train-test split for dataframes
def stratified_split_df(df, target_col, test_size = 0.3, random_state =
    42):
    """
    stratified split by discrete target values
```

```

    inputs
        df dataframe to split
        target_col column with class labels (here: original 1..5 happiness)
        test_size fraction in test
        random_state seed for reproducibility

    outputs
        df_train, df_test
    """
    rng = np.random.default_rng(random_state)

    df_train_parts = []
    df_test_parts = []

    for cls, grp in df.groupby(target_col):
        n = len(grp)
        n_test = int(round(test_size * n))
        n_test = max(1, min(n_test, n - 1)) # at least 1 in each side when possible

        idx = np.arange(n)
        rng.shuffle(idx)

        test_idx = idx[:n_test]
        train_idx = idx[n_test:]

        df_test_parts.append(grp.iloc[test_idx])
        df_train_parts.append(grp.iloc[train_idx])

    df_train = pd.concat(df_train_parts, axis = 0).sample(frac = 1.0, random_state = random_state).reset_index(drop = True)
    df_test = pd.concat(df_test_parts, axis = 0).sample(frac = 1.0, random_state = random_state).reset_index(drop = True)

    return df_train, df_test
```