Python Code

```
#!/usr/bin/env python3
"""
Modern Python demonstration of ANI/MCSI stochastic neural classification
applied to KASCADE extensive-air-shower simulations.

Purpose and relation to ANI/MCSI
--------------------------------
This program provides a compact and reproducible Python implementation of
one of the classification examples discussed within the Monte Carlo
Statistical Inference (MCSI) framework. It reproduces the energy-dependent
proton–iron (p–Fe) classification benchmark using KASCADE CORSIKA simulation
data and two basic extensive-air-shower observables:

    TrNe = log10(Ne)
    TrNm = log10(Nmu)

The script is not intended to reproduce the complete historical ANI
(Analysis and Nonparametric Inference) software package. Rather, it
demonstrates in modern Python the central ANI principle of training a
feed-forward neural classifier by stochastic Random Search in parameter
space instead of gradient-based backpropagation.

This NN–Random Search combination illustrates an important methodological
feature of ANI/MCSI: a flexible nonparametric classifier is coupled to a
derivative-free stochastic optimization procedure. Neural-network parameters
are therefore determined by direct optimization of a statistical quality
criterion rather than by gradient calculation.

For transparency and reproducibility, the script also implements classical
LDA and QDA classifiers and a simple physically motivated TrNm-TrNe threshold
as reference methods. Classification performance is evaluated independently
in successive primary-energy intervals using AUC, balanced accuracy, and
class-wise efficiencies.

No experimental KASCADE data are used in this benchmark. The analysis is
performed on simulated extensive-air-shower events.

Classification problem
----------------------
The truth primary code TrPP is used only to define the class label:

    TrPP = 14    -> proton, class 0
    TrPP = 5626  -> iron,   class 1

The simulated primary energy TrEP is used only to divide events into energy
bins. It is deliberately NOT supplied to the classifier as an input feature.
Thus, the event-by-event classification is based exclusively on the simulated
electron and muon shower-size observables TrNe and TrNm.

ANI stochastic optimization
```

```
---------------------------
The implemented classifier is a small feed-forward neural network trained
without backpropagation and without gradient calculation.

At each Random Search iteration, the parameters associated with a selected
hidden neuron are stochastically perturbed. Gaussian or Cauchy perturbations
can be used. A trial configuration is accepted only when it improves the
training objective:

    accept w_new if Q(w_new) < Q(w_old)

where Q is the binary cross-entropy evaluated on the training sample.

An independent validation sample is used to retain the best network
configuration encountered during stochastic optimization. Final performance
is evaluated on a separate test sample.

Reproducibility note
--------------------
Training, validation, and test samples are generated independently within
each energy interval and are balanced between proton and iron events.
Input-variable standardization is determined from the training sample only.
A user-defined random seed permits exact reproduction of the stochastic
splitting and optimization sequence.
"""



"""
ANI-style stochastic neural training for KASCADE proton-iron classification.

Purpose
-------
This script reproduces the energy-dependent p-Fe benchmark using only two
EAS shower-size variables:

    TrNe = log10(Ne)
    TrNm = log10(Nmu)

The truth primary code TrPP is used only to define the class label:
    TrPP = 14    -> proton, class 0
    TrPP = 5626  -> iron,   class 1

The primary energy TrEP is used only to divide events into energy bins.
It is deliberately NOT used as an input feature.

The implemented ANI classifier is a small feed-forward neural network trained
without backpropagation. Training is performed by stochastic neuron-level
perturbations in weight space. A proposed perturbation is accepted only if it
improves the training objective.

Acceptance rule:
    accept w_new if Q(w_new) < Q(w_old)

where Q is binary cross-entropy on the training sample.
```

```
105
106 Input
107 -----
108 A folder containing CSV chunks with at least these columns:
109
110     TrEP, TrPP, TrNe, TrNm
111
112 or one CSV file with the same columns.
113
114 Output
115 ------
116 1. CSV table with energy-dependent classification results.
117 2. CSV table with ANI learning curves.
118 3. PNG figures:
119    - energy spectra
120    - Ne-Nmu scatter by energy
121    - AUC vs energy
122    - balanced accuracy vs energy
123    - representative ANI learning curves
124
125 Example
126 -------
127 python kascade_ani_pfe_energy.py --input ./kascade_csv --output ./ani_results
128
129 Author
130 ------
131 Prepared for ANI/MCSI revival benchmark studies.
132 """
133
134 import argparse
135 import glob
136 import os
137 from pathlib import Path
138
139 import numpy as np
140 import pandas as pd
141 import matplotlib.pyplot as plt
142
143
144 # ---------------------------------------------------------------------
145 # Basic numerical functions
146 # ---------------------------------------------------------------------
147
148 def sigmoid(z):
149     """
150     Numerically stable sigmoid activation.
151
152     Large positive or negative arguments are clipped to avoid overflow.
153     This is analogous to overflow protection in older Fortran neural-network
154     routines.
155     """
156     return 1.0 / (1.0 + np.exp(-np.clip(z, -40.0, 40.0)))
157
158
```

```python
def binary_cross_entropy(y_true, y_prob):
    """
    Binary cross-entropy loss.

    Parameters
    ----------
    y_true : array of 0/1 labels
    y_prob : predicted probabilities for class 1

    Returns
    -------
    Mean binary cross-entropy.
    """
    y_prob = np.clip(y_prob, 1e-6, 1.0 - 1e-6)
    return -np.mean(y_true * np.log(y_prob) + (1.0 - y_true) * np.log(1.0 - y_prob))


def auc_mann_whitney(y_true, score):
    """
    Fast AUC calculation using the Mann-Whitney rank statistic.

    Class 1 (iron) is treated as the positive class.

    This avoids requiring scikit-learn and is adequate for this benchmark.
    """
    y_true = np.asarray(y_true, dtype=np.int8)
    score = np.asarray(score, dtype=float)

    n_pos = int(y_true.sum())
    n_neg = len(y_true) - n_pos

    if n_pos == 0 or n_neg == 0:
        return np.nan

    order = np.argsort(score)
    ranks = np.empty(len(score), dtype=float)
    ranks[order] = np.arange(1, len(score) + 1)

    rank_sum_pos = ranks[y_true == 1].sum()
    return (rank_sum_pos - n_pos * (n_pos + 1) / 2.0) / (n_pos * n_neg)


def balanced_accuracy(y_true, y_pred):
    """
    Balanced accuracy for two classes.

    Returns
    -------
    balanced_accuracy, proton_efficiency, iron_efficiency

    proton_efficiency = probability to correctly classify proton events
    iron_efficiency   = probability to correctly classify iron events
    """
    y_true = np.asarray(y_true, dtype=np.int8)
```

```
    y_pred = np.asarray(y_pred, dtype=np.int8)

    proton_mask = y_true == 0
    iron_mask = y_true == 1

    proton_eff = np.mean(y_pred[proton_mask] == 0)
    iron_eff = np.mean(y_pred[iron_mask] == 1)

    return 0.5 * (proton_eff + iron_eff), proton_eff, iron_eff


# ---------------------------------------------------------------------
# ANI neural-network model
# ---------------------------------------------------------------------

def unpack_weights(w, hidden_neurons=4):
    """
    Convert a flat weight vector into matrices/vectors of a 2-H-1 network.

    Network architecture:
        2 input variables: TrNe, TrNm
        H hidden sigmoid neurons
        1 sigmoid output neuron

    Weight vector layout:
        first 2H values       -> input-to-hidden weights
        next H values         -> hidden biases
        next H values         -> hidden-to-output weights
        final 1 value         -> output bias
    """
    H = hidden_neurons

    W1 = w[:2 * H].reshape(2, H)
    b1 = w[2 * H:3 * H]
    W2 = w[3 * H:4 * H].reshape(H, 1)
    b2 = w[-1]

    return W1, b1, W2, b2


def ani_forward(X, w, hidden_neurons=4):
    """
    Forward pass through the small ANI neural network.

    Parameters
    ----------
    X : array, shape (n_events, 2)
        Standardized input variables [TrNe, TrNm].
    w : flat weight vector
    hidden_neurons : number of hidden sigmoid neurons

    Returns
    -------
    y_prob : predicted probability of class 1 (iron)
```

```
    """
    W1, b1, W2, b2 = unpack_weights(w, hidden_neurons)

    hidden = sigmoid(X @ W1 + b1)
    output = sigmoid((hidden @ W2).ravel() + b2)

    return output


def ani_train(
    X_train,
    y_train,
    X_valid,
    y_valid,
    hidden_neurons=4,
    n_iter=1300,
    seed=123,
    perturbation="gaussian",
):
    """
    Train the ANI network by stochastic neuron-level perturbations.

    This is not backpropagation. No gradients are calculated.

    At each iteration:
        1. Select one hidden neuron.
        2. Perturb:
           - its two incoming weights,
           - its bias,
           - its output weight.
        3. Evaluate binary cross-entropy on the training sample.
        4. Accept the proposal only if the training loss decreases.
        5. Track the best validation loss.

    Parameters
    ----------
    X_train, y_train : training sample
    X_valid, y_valid : validation sample
    hidden_neurons : number of hidden neurons
    n_iter : number of stochastic proposals
    seed : random seed
    perturbation : "gaussian" or "cauchy"

    Returns
    -------
    best_w : best weight vector according to validation loss
    info : dictionary with accepted-step count and learning curve
    """
    rng = np.random.default_rng(seed)

    H = hidden_neurons
    n_parameters = 4 * H + 1

    # Initial random weights.
```

```python
    w = rng.normal(loc=0.0, scale=0.45, size=n_parameters)

    # Initial training and validation loss.
    current_loss = binary_cross_entropy(y_train, ani_forward(X_train, w, H))
    best_w = w.copy()
    best_valid_loss = binary_cross_entropy(y_valid, ani_forward(X_valid, w, H))

    accepted_steps = 0
    learning_curve = []

    for it in range(n_iter):
        # Gradually reduce perturbation amplitude.
        # This mimics coarse-to-fine stochastic search.
        cf = 0.45 * (1.0 - it / n_iter) + 0.05

        proposal = w.copy()

        # Select one hidden neuron for a block update.
        h = rng.integers(0, H)

        # Indices of the selected neuron's incoming weights, hidden bias,
        # and outgoing weight.
        idx = np.array([
            h,          # weight from input 1 to hidden neuron h
            H + h,      # weight from input 2 to hidden neuron h
            2 * H + h,  # hidden bias of neuron h
            3 * H + h,  # output weight from hidden neuron h
        ])

        if perturbation == "cauchy":
            # Heavy-tailed jumps. Clipping prevents rare enormous jumps.
            delta = np.clip(rng.standard_cauchy(len(idx)), -4.0, 4.0) * cf * 0.22
        elif perturbation == "gaussian":
            # Standard Gaussian local jumps.
            delta = rng.normal(loc=0.0, scale=cf, size=len(idx))
        else:
            raise ValueError("perturbation must be 'gaussian' or 'cauchy'")

        proposal[idx] += delta

        proposal_loss = binary_cross_entropy(y_train, ani_forward(X_train, proposal, H))

        # ANI stochastic hill-climbing acceptance rule.
        if proposal_loss < current_loss:
            w = proposal
            current_loss = proposal_loss
            accepted_steps += 1

            valid_loss = binary_cross_entropy(y_valid, ani_forward(X_valid, w, H))
            if valid_loss < best_valid_loss:
                best_valid_loss = valid_loss
                best_w = w.copy()

```

```
        # Store learning curve every 200 iterations.
        if (it + 1) % 200 == 0:
            learning_curve.append({
                "iteration": it + 1,
                "training_loss": current_loss,
                "best_validation_loss": best_valid_loss,
                "accepted_steps": accepted_steps,
            })

    info = {
        "accepted_steps": accepted_steps,
        "learning_curve": learning_curve,
    }

    return best_w, info


# ---------------------------------------------------------------------
# Classical comparison methods: LDA and QDA
# ---------------------------------------------------------------------

def lda_score_predict(X_train, y_train, X_test):
    """
    Linear discriminant analysis implemented explicitly.

    The returned score is positive for iron-like events.
    """
    X0 = X_train[y_train == 0]
    X1 = X_train[y_train == 1]

    m0 = X0.mean(axis=0)
    m1 = X1.mean(axis=0)

    S0 = np.cov(X0, rowvar=False)
    S1 = np.cov(X1, rowvar=False)

    pooled = ((len(X0) - 1) * S0 + (len(X1) - 1) * S1) / (len(X0) + len(X1) - 2)
    pooled += np.eye(2) * 1e-8

    invS = np.linalg.inv(pooled)

    w = invS @ (m1 - m0)
    b = -0.5 * (m1 @ invS @ m1 - m0 @ invS @ m0)

    score = X_test @ w + b
    prediction = (score >= 0.0).astype(np.int8)

    return score, prediction


def qda_score_predict(X_train, y_train, X_test):
    """
    Quadratic discriminant analysis implemented explicitly.

```

```
    The returned score is log-likelihood(iron) - log-likelihood(proton).
    """
    X0 = X_train[y_train == 0]
    X1 = X_train[y_train == 1]

    m0 = X0.mean(axis=0)
    m1 = X1.mean(axis=0)

    S0 = np.cov(X0, rowvar=False) + np.eye(2) * 1e-6
    S1 = np.cov(X1, rowvar=False) + np.eye(2) * 1e-6

    inv0 = np.linalg.inv(S0)
    inv1 = np.linalg.inv(S1)

    _, logdet0 = np.linalg.slogdet(S0)
    _, logdet1 = np.linalg.slogdet(S1)

    d0 = X_test - m0
    d1 = X_test - m1

    q0 = np.einsum("ij,jk,ik->i", d0, inv0, d0)
    q1 = np.einsum("ij,jk,ik->i", d1, inv1, d1)

    score = (-0.5 * logdet1 - 0.5 * q1) - (-0.5 * logdet0 - 0.5 * q0)
    prediction = (score >= 0.0).astype(np.int8)

    return score, prediction


# ---------------------------------------------------------------------
# Data loading and preprocessing
# ---------------------------------------------------------------------

def load_kascade_csv(input_path):
    """
    Load KASCADE p-Fe CSV data.

    input_path can be:
        - one CSV file,
        - a directory containing CSV files.

    Required columns:
        TrEP, TrPP, TrNe, TrNm
    """
    input_path = Path(input_path)

    if input_path.is_file():
        csv_files = [input_path]
    elif input_path.is_dir():
        csv_files = sorted(input_path.glob("*.csv"))
    else:
        raise FileNotFoundError(f"Input path not found: {input_path}")

    if len(csv_files) == 0:
```

```
        raise FileNotFoundError(f"No CSV files found in {input_path}")

    usecols = ["TrEP", "TrPP", "TrNe", "TrNm"]

    frames = []
    for f in csv_files:
        if f.name.startswith("._"):
            continue
        frames.append(pd.read_csv(f, usecols=usecols))

    df = pd.concat(frames, ignore_index=True)

    # Remove non-finite or missing values.
    df = df.replace([np.inf, -np.inf], np.nan)
    df = df.dropna(subset=usecols)

    # Keep only proton and iron.
    df = df[df["TrPP"].isin([14, 5626])].copy()

    # Class label: 0 = proton, 1 = iron.
    df["label"] = (df["TrPP"] == 5626).astype(np.int8)

    return df


def make_balanced_splits(df_bin, rng, n_train_max=2500, n_valid_max=1000,
n_test_max=12000):
    """
    Build balanced train/validation/test splits inside one energy bin.

    The same number of proton and iron events is used in each subset.
    This prevents the classifier from being biased by unequal class counts.
    """
    proton = df_bin[df_bin["label"] == 0]
    iron = df_bin[df_bin["label"] == 1]

    min_count = min(len(proton), len(iron))

    if min_count < 200:
        return None

    n_train = min(n_train_max, max(150, int(0.35 * min_count)))
    n_valid = min(n_valid_max, max(80, int(0.15 * min_count)))
    n_test = min(n_test_max, max(200, min_count - n_train - n_valid))

    if n_test < 100:
        return None

    proton_idx = rng.permutation(proton.index.to_numpy())
    iron_idx = rng.permutation(iron.index.to_numpy())

    train_idx = np.r_[proton_idx[:n_train], iron_idx[:n_train]]
    valid_idx = np.r_[proton_idx[n_train:n_train + n_valid],
                      iron_idx[n_train:n_train + n_valid]]
```

```
    test_idx = np.r_[proton_idx[n_train + n_valid:n_train + n_valid + n_test],
                     iron_idx[n_train + n_valid:n_train + n_valid + n_test]]

    rng.shuffle(train_idx)
    rng.shuffle(valid_idx)
    rng.shuffle(test_idx)

    return train_idx, valid_idx, test_idx


# ---------------------------------------------------------------------
# Main benchmark
# ---------------------------------------------------------------------

def run_benchmark(args):
    """
    Run the full energy-dependent p-Fe benchmark.
    """
    output_dir = Path(args.output)
    output_dir.mkdir(parents=True, exist_ok=True)

    rng = np.random.default_rng(args.seed)

    df = load_kascade_csv(args.input)

    print("Loaded events:")
    print(df["TrPP"].value_counts().rename(index={14: "proton", 5626: "iron"}))

    energy_bins = np.arange(args.energy_min, args.energy_max + 1e-9, args.energy_step)

    result_rows = []
    curve_rows = []

    for bin_index, (lo, hi) in enumerate(zip(energy_bins[:-1], energy_bins[1:])):
        df_bin = df[(df["TrEP"] >= lo) & (df["TrEP"] < hi)]

        split = make_balanced_splits(df_bin, rng)
        if split is None:
            print(f"Skipping bin {lo:.1f}-{hi:.1f}: not enough events")
            continue

        train_idx, valid_idx, test_idx = split

        X_train_raw = df.loc[train_idx, ["TrNe", "TrNm"]].to_numpy(float)
        y_train = df.loc[train_idx, "label"].to_numpy(np.int8)

        X_valid_raw = df.loc[valid_idx, ["TrNe", "TrNm"]].to_numpy(float)
        y_valid = df.loc[valid_idx, "label"].to_numpy(np.int8)

        X_test_raw = df.loc[test_idx, ["TrNe", "TrNm"]].to_numpy(float)
        y_test = df.loc[test_idx, "label"].to_numpy(np.int8)

        # Standardization is fitted on training data only.
        mean = X_train_raw.mean(axis=0)
```

```
        std = X_train_raw.std(axis=0)
        std[std == 0] = 1.0

        X_train = (X_train_raw - mean) / std
        X_valid = (X_valid_raw - mean) / std
        X_test = (X_test_raw - mean) / std

        # ------------------------------------------------------------
        # Physical baseline: threshold on TrNm - TrNe
        # ------------------------------------------------------------
        contrast_train = X_train_raw[:, 1] - X_train_raw[:, 0]
        contrast_test = X_test_raw[:, 1] - X_test_raw[:, 0]

        threshold_grid = np.quantile(contrast_train, np.linspace(0.02, 0.98, 101))

        best_threshold = None
        best_ba = -np.inf

        for threshold in threshold_grid:
            pred_train = (contrast_train >= threshold).astype(np.int8)
            ba, _, _ = balanced_accuracy(y_train, pred_train)
            if ba > best_ba:
                best_ba = ba
                best_threshold = threshold

        pred_test = (contrast_test >= best_threshold).astype(np.int8)
        ba, p_eff, fe_eff = balanced_accuracy(y_test, pred_test)

        result_rows.append({
            "energy_bin": f"{lo:.1f}-{hi:.1f}",
            "E_center": 0.5 * (lo + hi),
            "method": "TrNm-TrNe threshold",
            "N_bin": len(df_bin),
            "N_test": len(y_test),
            "balanced_accuracy": ba,
            "AUC": auc_mann_whitney(y_test, contrast_test),
            "proton_efficiency": p_eff,
            "iron_efficiency": fe_eff,
            "accepted_steps": np.nan,
        })

        # ------------------------------------------------------------
        # LDA and QDA comparisons
        # ------------------------------------------------------------
        for method_name, method_func in [
            ("LDA", lda_score_predict),
            ("QDA", qda_score_predict),
        ]:
            score, pred = method_func(X_train, y_train, X_test)
            ba, p_eff, fe_eff = balanced_accuracy(y_test, pred)

            result_rows.append({
                "energy_bin": f"{lo:.1f}-{hi:.1f}",
                "E_center": 0.5 * (lo + hi),
```

```
                "method": method_name,
                "N_bin": len(df_bin),
                "N_test": len(y_test),
                "balanced_accuracy": ba,
                "AUC": auc_mann_whitney(y_test, score),
                "proton_efficiency": p_eff,
                "iron_efficiency": fe_eff,
                "accepted_steps": np.nan,
            })

        # ------------------------------------------------------------
        # ANI stochastic neural training
        # ------------------------------------------------------------
        for method_name, perturbation in [
            ("ANI neuron Gaussian", "gaussian"),
            ("ANI neuron Cauchy", "cauchy"),
        ]:
            w, info = ani_train(
                X_train,
                y_train,
                X_valid,
                y_valid,
                hidden_neurons=args.hidden_neurons,
                n_iter=args.iterations,
                seed=args.seed + 1000 + 13 * bin_index + (0 if perturbation ==
"gaussian" else 7),
                perturbation=perturbation,
            )

            score = ani_forward(X_test, w, args.hidden_neurons)
            pred = (score >= 0.5).astype(np.int8)

            ba, p_eff, fe_eff = balanced_accuracy(y_test, pred)

            result_rows.append({
                "energy_bin": f"{lo:.1f}-{hi:.1f}",
                "E_center": 0.5 * (lo + hi),
                "method": method_name,
                "N_bin": len(df_bin),
                "N_test": len(y_test),
                "balanced_accuracy": ba,
                "AUC": auc_mann_whitney(y_test, score),
                "proton_efficiency": p_eff,
                "iron_efficiency": fe_eff,
                "accepted_steps": info["accepted_steps"],
            })

            for row in info["learning_curve"]:
                row = dict(row)
                row["energy_bin"] = f"{lo:.1f}-{hi:.1f}"
                row["method"] = method_name
                curve_rows.append(row)

        print(f"Finished energy bin {lo:.1f}-{hi:.1f}")
```

```
    results = pd.DataFrame(result_rows)
    curves = pd.DataFrame(curve_rows)

    results_path = output_dir / "kascade_pfe_ani_energy_results.csv"
    curves_path = output_dir / "kascade_pfe_ani_learning_curves.csv"

    results.to_csv(results_path, index=False)
    curves.to_csv(curves_path, index=False)

    print("\nClassification results:")
    print(results.to_string(index=False))
    print(f"\nSaved: {results_path}")
    print(f"Saved: {curves_path}")

    make_plots(df, results, curves, output_dir, energy_bins)


# ---------------------------------------------------------------------
# Plotting
# ---------------------------------------------------------------------

def make_plots(df, results, curves, output_dir, energy_bins):
    """
    Create standard diagnostic plots.
    """
    colors = {
        "TrNm-TrNe threshold": "green",
        "LDA": "blue",
        "QDA": "orange",
        "ANI neuron Gaussian": "red",
        "ANI neuron Cauchy": "purple",
    }

    # Energy spectra
    plt.figure(figsize=(8.3, 5.3))
    plt.hist(
        df[df["label"] == 0]["TrEP"],
        bins=energy_bins,
        histtype="step",
        linewidth=2.2,
        color="blue",
        label="Proton, TrPP=14",
    )
    plt.hist(
        df[df["label"] == 1]["TrEP"],
        bins=energy_bins,
        histtype="step",
        linewidth=2.2,
        color="red",
        label="Iron, TrPP=5626",
    )
    plt.xlabel("TrEP = log10(E/eV)")
    plt.ylabel("Number of events")
```

```
    plt.title("KASCADE simulation energy spectra used for p-Fe classification")
    plt.grid(alpha=0.25)
    plt.legend()
    plt.tight_layout()
    plt.savefig(output_dir / "kascade_pfe_energy_spectra.png", dpi=220)
    plt.close()

    # Ne-Nmu scatter in representative energy bins
    scatter_bins = [(14.0, 14.5), (14.5, 15.0), (15.0, 15.5),
                    (15.5, 16.0), (16.0, 16.5), (16.5, 17.0)]

    fig, axes = plt.subplots(2, 3, figsize=(13, 8))

    for ax, (lo, hi) in zip(axes.ravel(), scatter_bins):
        proton = df[(df["label"] == 0) & (df["TrEP"] >= lo) & (df["TrEP"] < hi)]
        iron = df[(df["label"] == 1) & (df["TrEP"] >= lo) & (df["TrEP"] < hi)]

        proton = proton.sample(min(1200, len(proton)), random_state=int(lo * 100)) if len(proton) else proton
        iron = iron.sample(min(1200, len(iron)), random_state=int(hi * 100)) if len(iron) else iron

        ax.scatter(proton["TrNe"], proton["TrNm"], s=5, alpha=0.28, color="blue", label="p" if lo == 14.0 else None)
        ax.scatter(iron["TrNe"], iron["TrNm"], s=5, alpha=0.28, color="red", label="Fe" if lo == 14.0 else None)
        ax.set_title(f"{lo:.1f} <= TrEP < {hi:.1f}")
        ax.grid(alpha=0.25)

    fig.supxlabel("TrNe = log10(Ne)")
    fig.supylabel("TrNm = log10(Nmu)")
    fig.suptitle("KASCADE p-Fe separation in Ne-Nmu plane across energy intervals", y=0.98)
    axes[0, 0].legend()
    plt.tight_layout()
    plt.savefig(output_dir / "kascade_pfe_Ne_Nm_by_energy.png", dpi=220)
    plt.close()

    # AUC and balanced accuracy curves
    for metric, ylabel, title, filename, ylim in [
        (
            "AUC",
            "AUC",
            "Energy dependence of KASCADE p-Fe classification using only TrNe and TrNm",
            "kascade_pfe_ani_auc_vs_energy.png",
            (0.75, 1.01),
        ),
        (
            "balanced_accuracy",
            "Balanced accuracy",
            "Energy dependence of KASCADE p-Fe balanced accuracy using only TrNe and TrNm",
            "kascade_pfe_ani_ba_vs_energy.png",
```

```
            (0.70, 1.01),
        ),
    ]:
        plt.figure(figsize=(8.3, 5.4))

        for method, group in results.groupby("method"):
            plt.plot(
                group["E_center"],
                group[metric],
                marker="o",
                color=colors.get(method, None),
                label=method,
            )

        plt.xlabel("Energy-bin center TrEP = log10(E/eV)")
        plt.ylabel(ylabel)
        plt.title(title)
        plt.ylim(*ylim)
        plt.grid(alpha=0.3)
        plt.legend(fontsize=8)
        plt.tight_layout()
        plt.savefig(output_dir / filename, dpi=220)
        plt.close()

    # Representative ANI learning curves
    if len(curves) > 0:
        plt.figure(figsize=(8.3, 5.4))

        for energy_bin in ["14.0-14.5", "16.5-17.0"]:
            subset = curves[
                (curves["energy_bin"] == energy_bin)
                & (curves["method"] == "ANI neuron Gaussian")
            ]

            if len(subset) > 0:
                plt.plot(
                    subset["iteration"],
                    subset["best_validation_loss"],
                    marker="o",
                    label=energy_bin,
                )

        plt.xlabel("Iteration")
        plt.ylabel("Best validation BCE loss")
        plt.title("Representative ANI neuron-Gaussian learning curves")
        plt.grid(alpha=0.3)
        plt.legend()
        plt.tight_layout()
        plt.savefig(output_dir / "kascade_pfe_ani_learning_curves.png", dpi=220)
        plt.close()


# ---------------------------------------------------------------------
# Command-line interface
```

```python
# ---------------------------------------------------------------------

def parse_args():
    parser = argparse.ArgumentParser(
        description="Energy-dependent ANI stochastic neural benchmark for KASCADE p-Fe classification."
    )

    parser.add_argument(
        "--input",
        required=True,
        help="Input CSV file or directory containing CSV chunks.",
    )

    parser.add_argument(
        "--output",
        default="ani_results",
        help="Output directory for CSV tables and figures.",
    )

    parser.add_argument(
        "--energy-min",
        type=float,
        default=14.0,
        help="Minimum TrEP = log10(E/eV).",
    )

    parser.add_argument(
        "--energy-max",
        type=float,
        default=18.0,
        help="Maximum TrEP = log10(E/eV).",
    )

    parser.add_argument(
        "--energy-step",
        type=float,
        default=0.5,
        help="Energy-bin width in log10(E/eV).",
    )

    parser.add_argument(
        "--hidden-neurons",
        type=int,
        default=4,
        help="Number of hidden sigmoid neurons in ANI network.",
    )

    parser.add_argument(
        "--iterations",
        type=int,
        default=1300,
        help="Number of stochastic ANI proposals per energy bin.",
    )
```

```
    parser.add_argument(
        "--seed",
        type=int,
        default=2026,
        help="Random seed.",
    )

    return parser.parse_args()


if __name__ == "__main__":
    run_benchmark(parse_args())

```