Sequence Design with a Genetic Algorithm and a CNN Fitness Function

Companion to Lectures 20 and 21

Marcel Turcotte

Version: Nov 22, 2026 20:09

From prediction to sequence design

In Lecture 15, a convolutional neural network (CNN) learned to predict **mean ribosome load (MRL)** from a sequence of 50 nucleotides. MRL is an experimental measure related to translation efficiency: how effectively an RNA molecule is used to produce protein. The sequences describe its **5′ untranslated region (5′ UTR)**, which precedes the protein-coding region.

The prediction problem asked: *What MRL does the model predict for this sequence?* We now ask a different question:

> Can we use that predictor to search for a sequence with a larger predicted MRL?

This self-study notebook connects the CNN regressor from Lecture 15 with the genetic algorithms (GAs) in Lecture 20 and Lecture 21. It runs independently of those notebooks. The CNN becomes a **surrogate objective function**: a learned approximation used when directly evaluating the desired objective would be difficult or expensive. The GA changes the input sequence; the trained CNN's weights remain fixed throughout the search.

The Universal Approximation Theorem motivates the representational capacity of neural networks under its assumptions. It does not guarantee that this particular CNN can represent the desired function, that training will find an accurate approximation, or that predictions will generalize to sequences selected by optimization. We will distinguish an improvement in the model's prediction from an experimentally established improvement in MRL.

The example adapts the GA baseline in **AUGMENT**, a manuscript under review by Siavash Khalaj and Marcel Turcotte. It uses the Keras CNN and random data split from Lecture 15, rather than reproducing the manuscript's complete experimental pipeline.

Learning objectives

By the end of this notebook, you should be able to:

- use a trained regressor as a fixed fitness evaluator;
- choose a sequence representation and compatible genetic operators;
- explain a trade-off between predicted improvement and distance from a reference sequence; and
- distinguish successful optimization of a surrogate from evidence that the desired real-world objective has improved.

Execution and computational cost

Run the cells in order. Required data preparation and CNN training are included in collapsed code cells before the GA. They use TensorFlow/Keras, NumPy, pandas, Matplotlib, and scikit-learn, as in Lecture 15. No AUGMENT installation, research checkpoint, or earlier notebook execution is required.

The first execution downloads the experimental data and trains the CNN on 220,000 sequences, with early stopping and a maximum of 50 epochs. This is the expensive part, and its duration depends on the available hardware. A local cache stores the downloaded data, trained model, target scaler, and training configuration. Later executions reuse a compatible model. The training section explains how to request a new fit.

The subsequent demonstration makes two modest GA runs. Each scores a population in a batch; the CNN is not retrained between generations or runs. The appendix suggests extensions after the main example.

Preparing the CNN fitness function

We reuse the first experiment from Lecture 15: retain the 275,000 sequences with the highest read counts, randomly split them, and train the same CNN to predict mean ribosome load (MRL). The supporting code is included below so that this notebook can run independently. Expand a code block to revisit the implementation.

The first run downloads the data and trains the CNN. Later runs reuse the local model and its target transformation when the settings and target scaling match. Set `FORCE_RETRAIN = True` to train again. The local cache is stored in `.ga_cnn_cache` in the working directory; it does not contain results from the GA.

Imports and reproducible settings

Use a Python environment with TensorFlow, pandas, NumPy, and scikit-learn. Training on 220,000 sequences is the expensive step; a GPU is useful but is not required. The seed fixes the data partition and initializes the random number generators. Numerical results may still vary across hardware.

```
import json
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from tensorflow import keras
from tensorflow.keras import callbacks, layers

SEED = 42
SEQUENCE_LENGTH = 50
N_RETAINED = 275_000
N_TRAIN = 220_000
N_VALIDATION = 27_500
N_TEST = 27_500
BATCH_SIZE = 512
MAX_EPOCHS = 50
PATIENCE = 5
FORCE_RETRAIN = False

CACHE_DIR = Path(".ga_cnn_cache")
DATA_DIR = CACHE_DIR / "data"
MODEL_DIR = CACHE_DIR / "models"
DATA_DIR.mkdir(parents=True, exist_ok=True)

DATA_FILENAME = "GSM3130435_egfp_unmod_1.csv.gz"
DATA_URL = (
    "https://ftp.ncbi.nlm.nih.gov/geo/samples/"
    "GSM3130nnn/GSM3130435/suppl/" + DATA_FILENAME
)

keras.utils.set_random_seed(SEED)
```

Downloading and preparing the measurements

We standardize the input column names to `utr`, `mrl`, and `read_count`, check the sequence format, and retain exactly 275,000 measurements. Stable sorting resolves ties at the read-count boundary in the same way as in Lecture 15.

```
def load_utr_dataset(path, expected_length=SEQUENCE_LENGTH):
    """Load a dataset with columns utr, mrl, and read_count."""
    data = pd.read_csv(path)

    if "utr" not in data.columns and "UTR" in data.columns:
        data = data.rename(columns={"UTR": "utr"})

    if "total_reads" in data.columns:
        read_column = "total_reads"
    elif "total" in data.columns:
        read_column = "total"
    else:
        raise ValueError("The dataset has no total_reads or total column.")

    required = {"utr", "rl", read_column}
    missing = required.difference(data.columns)
    if missing:
        raise ValueError(f"Missing required columns: {sorted(missing)}")

    data = (
        data.loc[:, ["utr", "rl", read_column]]
        .rename(columns={"rl": "mrl", read_column: "read_count"})
        .copy()
    )
    data["utr"] = data["utr"].astype("string").str.upper()
    data["mrl"] = pd.to_numeric(data["mrl"], errors="coerce")
    data["read_count"] = pd.to_numeric(data["read_count"], errors="coerce")

    if data[["utr", "mrl", "read_count"]].isna().any().any():
        raise ValueError("Missing or non-numeric values were found.")

    valid_sequence = data["utr"].str.fullmatch(
        rf"[ACGTN]{{{expected_length}}}"
    )
    if not valid_sequence.all():
        n_invalid = int((~valid_sequence).sum())
        raise ValueError(
```

```
            f"Found {n_invalid} sequences that are not length "
            f"{expected_length} strings over A, C, G, T, and N."
        )

    if (data["read_count"] < 0).any():
        raise ValueError("Read counts must be non-negative.")
    if data["utr"].duplicated().any():
        raise ValueError("The dataset contains duplicate UTR sequences.")

    return data.reset_index(drop=True)


def select_top_by_reads(data, n):
    """Return the n highest-read rows and the boundary read count."""
    if not 0 < n <= len(data):
        raise ValueError("n must be between 1 and the number of rows.")

    ranked = data.sort_values(
        "read_count", ascending=False, kind="stable"
    ).reset_index(drop=True)
    cutoff = float(ranked.loc[n - 1, "read_count"])
    return ranked.iloc[:n].copy(), cutoff


data_path = DATA_DIR / DATA_FILENAME
if not data_path.exists():
    data_path = Path(
        keras.utils.get_file(
            fname=DATA_FILENAME,
            origin=DATA_URL,
            cache_dir=str(CACHE_DIR.resolve()),
            cache_subdir="data",
        )
    )

main_data = load_utr_dataset(data_path)
retained_data, read_cutoff = select_top_by_reads(main_data, N_RETAINED)

print(f"Retained {len(retained_data):,} sequences.")
print(f"Read-count boundary: {read_cutoff:,.0f}")
```

```
Retained 275,000 sequences.
```

```
Read-count boundary: 294
```

We use 220,000 sequences for training, 27,500 for validation, and 27,500 for testing. Only the training set determines the target mean and standard deviation. The validation set controls early stopping. The test set remains reserved; this notebook does not repeat the evaluation from Lecture 15.

Each length-50 sequence becomes a 50×4 array, with channels ordered as A, C, G, T. An unknown nucleotide (`N`) is represented by four zeros.

```python
def random_partition(data, seed=SEED):
    """Create the agreed random training, validation, and test partitions."""
    if len(data) != N_RETAINED:
        raise ValueError(f"Expected exactly {N_RETAINED:,} retained rows.")

    shuffled = data.sample(frac=1, random_state=seed).reset_index(drop=True)
    train = shuffled.iloc[:N_TRAIN].copy()
    validation = shuffled.iloc[N_TRAIN:N_TRAIN + N_VALIDATION].copy()
    test = shuffled.iloc[N_TRAIN + N_VALIDATION:].copy()

    if (len(train), len(validation), len(test)) != (
        N_TRAIN, N_VALIDATION, N_TEST
    ):
        raise AssertionError("Unexpected partition sizes.")

    train_sequences = set(train["utr"])
    validation_sequences = set(validation["utr"])
    test_sequences = set(test["utr"])
    if not train_sequences.isdisjoint(validation_sequences):
        raise AssertionError("Training and validation sets overlap.")
    if not train_sequences.isdisjoint(test_sequences):
        raise AssertionError("Training and test sets overlap.")
    if not validation_sequences.isdisjoint(test_sequences):
        raise AssertionError("Validation and test sets overlap.")

    return train, validation, test


def one_hot_encode(sequences, sequence_length=SEQUENCE_LENGTH):
    """Encode nucleotide strings as an (n, sequence_length, 4) array."""
    sequences = pd.Series(sequences, dtype="string").str.upper()
    valid = sequences.str.fullmatch(rf"[ACGTN]{{{sequence_length}}}")
    if not valid.all():
```

```
        raise ValueError("Every sequence must have the expected format.")

    characters = np.frombuffer(
        "".join(sequences).encode("ascii"), dtype=np.uint8
    ).reshape(-1, sequence_length)
    channel_lookup = np.full(256, -1, dtype=np.int8)
    for channel, nucleotide in enumerate("ACGT"):
        channel_lookup[ord(nucleotide)] = channel

    channels = channel_lookup[characters]
    encoded = np.zeros(
        (len(sequences), sequence_length, 4), dtype=np.float32
    )
    rows, positions = np.nonzero(channels >= 0)
    encoded[rows, positions, channels[rows, positions]] = 1.0
    return encoded


train_data, validation_data, test_data = random_partition(retained_data)

target_scaler = StandardScaler()
y_train = target_scaler.fit_transform(
    train_data[["mrl"]]
).astype(np.float32)
y_validation = target_scaler.transform(
    validation_data[["mrl"]]
).astype(np.float32)
x_train = one_hot_encode(train_data["utr"])
x_validation = one_hot_encode(validation_data["utr"])

print(
    f"Training: {len(train_data):,}; "
    f"validation: {len(validation_data):,}; "
    f"test (reserved): {len(test_data):,}."
)
```

```
Training: 220,000; validation: 27,500; test (reserved): 27,500.
```

Building and training the CNN

The architecture is unchanged from Lecture 15: four convolutional layers with 160, 160, 160, and 80 filters of width eight, followed by a dense layer with 80 units and a linear output. The output is standardized MRL. Later, we will invert the target transformation to report fitness in MRL units.

```
def build_mrl_cnn(
    sequence_length=SEQUENCE_LENGTH,
    n_filters=160,
    kernel_size=8,
    latent_dimension=80,
):
    """Build the 1D CNN used to predict standardized MRL."""
    model = keras.Sequential(
        [
            keras.Input(shape=(sequence_length, 4), name="one_hot_utr"),
            layers.Conv1D(
                n_filters, kernel_size, padding="same", activation="relu",
                name="convolution_1",
            ),
            layers.Conv1D(
                n_filters, kernel_size, padding="same", activation="relu",
                name="convolution_2",
            ),
            layers.BatchNormalization(name="batch_normalization_1"),
            layers.Dropout(0.2, name="dropout_1"),
            layers.Conv1D(
                n_filters, kernel_size, padding="same", activation="relu",
                name="convolution_3",
            ),
            layers.BatchNormalization(name="batch_normalization_2"),
            layers.Dropout(0.4, name="dropout_2"),
            layers.Conv1D(
                n_filters // 2, kernel_size, padding="same", activation="relu",
                name="convolution_4",
            ),
            layers.BatchNormalization(name="batch_normalization_3"),
            layers.Dropout(0.2, name="dropout_3"),
            layers.Flatten(name="flatten"),
            layers.Dense(
                latent_dimension, activation="relu",
                name="latent_representation",
```

```
            ),
            layers.Dense(1, name="scaled_mrl"),
        ],
        name="mrl_cnn",
    )
    return model
```

We minimize mean squared error using Adam, with a learning rate of 10^{-3} and batches of 512. Training stops after at most 50 epochs, or after five epochs without an improvement in validation loss. The weights from the best validation epoch are restored.

The cache stores three files together: the CNN, the target mean and scale, and the settings used to obtain them. A cache is reused only when all three are present and match the current settings and target transformation. The files are saved after training finishes.

```
training_config = {
    "schema_version": 1,
    "dataset": DATA_FILENAME,
    "seed": SEED,
    "n_retained": N_RETAINED,
    "n_train": N_TRAIN,
    "n_validation": N_VALIDATION,
    "n_test": N_TEST,
    "architecture": "lecture-15-mrl-cnn-v1",
    "sequence_length": SEQUENCE_LENGTH,
    "n_filters": 160,
    "kernel_size": 8,
    "latent_dimension": 80,
    "learning_rate": 1e-3,
    "batch_size": BATCH_SIZE,
    "max_epochs": MAX_EPOCHS,
    "patience": PATIENCE,
}


def fit_or_load_model(
    x_train,
    y_train,
    x_validation,
    y_validation,
    target_scaler,
    config,
```

```
    model_dir=MODEL_DIR,
    force_retrain=False,
):
    """Return the CNN and its training history (None when loaded)."""
    model_dir = Path(model_dir)
    model_path = model_dir / "model.keras"
    scaler_path = model_dir / "scaler.npz"
    config_path = model_dir / "config.json"
    cache_complete = all(
        path.exists() for path in (model_path, scaler_path, config_path)
    )

    if cache_complete and not force_retrain:
        saved_config = json.loads(config_path.read_text())
        with np.load(scaler_path) as saved_scaler:
            scaler_matches = (
                np.array_equal(saved_scaler["mean"], target_scaler.mean_)
                and np.array_equal(saved_scaler["scale"], target_scaler.scale_)
            )
        if saved_config == config and scaler_matches:
            model = keras.models.load_model(model_path, compile=False)
            print("Loaded the cached CNN and matching target transformation.")
            return model, None

    keras.utils.set_random_seed(config["seed"])
    model = build_mrl_cnn(
        sequence_length=config["sequence_length"],
        n_filters=config["n_filters"],
        kernel_size=config["kernel_size"],
        latent_dimension=config["latent_dimension"],
    )
    model.compile(
        optimizer=keras.optimizers.Adam(
            learning_rate=config["learning_rate"]
        ),
        loss="mean_squared_error",
    )
    early_stopping = callbacks.EarlyStopping(
        monitor="val_loss",
        mode="min",
        patience=config["patience"],
        restore_best_weights=True,
```

```
        verbose=1,
    )
    history = model.fit(
        x_train,
        y_train,
        validation_data=(x_validation, y_validation),
        epochs=config["max_epochs"],
        batch_size=config["batch_size"],
        callbacks=[early_stopping],
        shuffle=True,
        verbose=2,
    )

    model_dir.mkdir(parents=True, exist_ok=True)
    model.save(model_path)
    np.savez(
        scaler_path,
        mean=target_scaler.mean_,
        scale=target_scaler.scale_,
    )
    config_path.write_text(json.dumps(config, indent=2, sort_keys=True))
    return model, history


model, training_history = fit_or_load_model(
    x_train,
    y_train,
    x_validation,
    y_validation,
    target_scaler,
    training_config,
    force_retrain=FORCE_RETRAIN,
)

if training_history is not None:
    best_validation_mse = min(training_history.history["val_loss"])
    n_epochs = len(training_history.history["loss"])
    print(f"Completed {n_epochs} epochs.")
    print(f"Best validation MSE (standardized targets): {best_validation_mse:.4f}")
```

```
Loaded the cached CNN and matching target transformation.
```

The model is now fixed. A GA can change a candidate sequence and query its predicted MRL, but it cannot improve the CNN or create new measured labels.

Turning predictions into fitness

A fixed predictor

The CNN was trained on standardized targets. We must undo that transformation to obtain predictions in **original MRL units** before combining predictions with a penalty. `model.predict` uses inference behavior: dropout is disabled and batch normalization uses its learned statistics.

```
model.trainable = False


def predict_mrl(sequences):
    """Return CNN predictions in original MRL units for a batch of strings."""
    encoded = one_hot_encode(sequences)
    scaled_predictions = model.predict(
        encoded, batch_size=BATCH_SIZE, verbose=0
    )
    return target_scaler.inverse_transform(scaled_predictions).ravel()
```

No gradients or weight updates are used by the GA. The prediction function accepts a list of strings and returns one score per string, so the search algorithm does not need to know the CNN's architecture.

A reference-guided objective

Choose a reference sequence s_0. For a candidate s, define

$$F(s) = \widehat{\mathrm{MRL}}(s) - \widehat{\mathrm{MRL}}(s_0) - \lambda\, d_H(s, s_0),$$

where $\widehat{\mathrm{MRL}}$ is the fixed CNN prediction and $d_H(s, s_0)$ is the **Hamming distance**: the number of positions at which the two length-50 sequences differ. We permit substitutions, without insertions or deletions. The coefficient $\lambda \geq 0$ has units of MRL per changed position.

- With $\lambda = 0$, maximizing fitness maximizes predicted MRL.
- With $\lambda > 0$, a predicted gain must compensate for the changes to the reference. The penalty encourages fewer changes; it imposes no hard maximum.

Subtracting the reference prediction is constant within a run, so it does not change candidate rankings. It gives the reference a fitness of zero and makes the predicted gain explicit. Both predictions come from the **same model**; subtracting a measured reference MRL from a generated prediction would mix prediction error with the effect of the search.

Predict before running: with $\lambda = 0.1$, which candidate has greater fitness?

Candidate	Predicted gain over reference	Changed positions	Fitness
A	0.8	4	0.4
B	0.6	1	0.5

Candidate B wins despite its smaller predicted MRL. Consequently, the history will record the prediction of the **fitness winner**, which need not be the largest predicted MRL in the population. The penalty does not establish that the CNN is reliable near the reference or that a generated sequence will improve experimental MRL.

Searching with a genetic algorithm

Representing candidate sequences

A chromosome is a length-50 array of categorical values: `0`, `1`, `2`, and `3` represent A, C, G, and T. Repeated nucleotides are valid. Unlike a TSP tour, a sequence is **not a permutation**: uniform crossover and nucleotide substitutions preserve its representation without a repair step.

The following conversion functions connect the GA's integer arrays to the strings accepted by the CNN prediction function.

```
GA_ALPHABET = np.array(list("ACGT"))


def encode_sequence(sequence):
    """Encode an unambiguous, length-50 reference as integers 0 through 3."""
    if len(sequence) != SEQUENCE_LENGTH or set(sequence) - set("ACGT"):
        raise ValueError("The reference must contain exactly 50 A/C/G/T bases.")
    return np.array(["ACGT".index(base) for base in sequence], dtype=np.uint8)


def decode_population(population):
```

```
    """Convert each integer chromosome to a nucleotide string."""
    return ["".join(GA_ALPHABET[row]) for row in population]
```

Scoring a population

The predictor returns MRL in its original units. For every candidate, we subtract the reference prediction and the penalty for its number of changed positions. The CNN prediction is a component of fitness; with a positive penalty, the candidate with the largest predicted MRL need not have the largest fitness.

```
def evaluate_sequence_population(population, reference, reference_prediction,
                                 predict_fn, fitness_lambda):
    """Return fitness, predicted MRL, and Hamming distance for each row."""
    predictions = np.asarray(predict_fn(decode_population(population)), dtype=float)
    if predictions.shape != (len(population),) or not np.isfinite(predictions).all():
        raise ValueError("predict_fn must return one finite MRL value per sequence.")
    edit_counts = np.count_nonzero(population != reference, axis=1)
    # Reuse the same baseline for the reference, avoiding batch roundoff differences.
    predictions[edit_counts == 0] = reference_prediction
    fitness = predictions - reference_prediction - fitness_lambda * edit_counts
    return fitness, predictions, edit_counts
```

Selection, crossover, mutation, and elitism

Predict before running: if the mutation probability is 0.02 per position, how many substitutions should we expect in one length-50 offspring? The expected number is $50 \times 0.02 = 1$; individual offspring can have zero, one, or several substitutions.

Tournament selection works with positive, zero, or negative fitness. Each tournament samples distinct participants and chooses its highest-scoring candidate. Uniform crossover exchanges corresponding positions in a selected pair. Mutation replaces a selected nucleotide with one of the other three nucleotides, so a mutation event always changes that position.

Elitism copies a small number of distinct, high-fitness chromosomes into the next population. Avoiding duplicate elites leaves more places for offspring; it does not guarantee a diverse population. When fitness values tie, we prefer fewer changed positions for elite ordering and the retained answer.

```
def select_sequence_parents(population, fitness, tournament_size, rng):
    """Select one parent per population member, with replacement across tournaments."""
    selected = []
```

```
    for _ in range(len(population)):
        participants = rng.choice(len(population), tournament_size, replace=False)
        winner = participants[np.argmax(fitness[participants])]
        selected.append(winner)
    return population[selected].copy()


def uniform_sequence_crossover(parents, crossover_rate, rng):
    """Apply crossover with the given probability per pair of parents."""
    shuffled = parents[rng.permutation(len(parents))]
    offspring = shuffled.copy()
    for index in range(0, len(parents) - 1, 2):
        if rng.random() < crossover_rate:
            mask = rng.random(parents.shape[1]) < 0.5
            offspring[index, mask] = shuffled[index + 1, mask]
            offspring[index + 1, mask] = shuffled[index, mask]
    return offspring  # An unpaired final parent, if present, is copied unchanged.


def mutate_sequences(offspring, mutation_rate, rng):
    """Substitute each nucleotide independently with the given probability."""
    mutated = offspring.copy()
    mask = rng.random(offspring.shape) < mutation_rate
    offsets = rng.integers(1, 4, size=offspring.shape, dtype=np.uint8)
    mutated[mask] = (mutated[mask] + offsets[mask]) % 4
    return mutated


def unique_sequence_elites(population, fitness, edit_counts, elite_size):
    """Copy up to elite_size distinct chromosomes, ordered by fitness."""
    ranked = np.lexsort((edit_counts, -fitness))
    elites = []
    seen = set()
    for index in ranked:
        chromosome = population[index]
        key = chromosome.tobytes()
        if key not in seen:
            elites.append(chromosome.copy())
            seen.add(key)
        if len(elites) == elite_size:
            break
    return np.asarray(elites, dtype=np.uint8)
```

Running the genetic algorithm

The initial population is random, with the reference inserted as its first member. We retain the best candidate encountered separately from the evolving population. The reference starts with fitness zero, so the returned answer cannot have worse fitness than the reference under this fixed predictor.

Generation 0 is the initialized population. With population size P and G replacement generations, the algorithm evaluates $P(G+1)$ candidates, including the final population, plus one prediction of the reference. Thus the budget is

$$1 + P(G+1)$$

individual sequence evaluations. One batched CNN call evaluates several sequences; we count the sequences, not the calls. Repeated chromosomes are evaluated again in this simple implementation. Deduplicating the final results table does not reduce that budget.

The history records the **best-so-far fitness**, the predicted MRL and number of edits of that retained winner, and the current population's mean fitness. The winning prediction need not increase monotonically when the edit penalty is positive.

```
def run_sequence_ga(predict_fn, reference_sequence, *, rng,
                    population_size=64, generations=20, fitness_lambda=0.1,
                    tournament_size=3, crossover_rate=0.8, mutation_rate=0.02,
                    elite_fraction=0.05):
    """Maximize predicted improvement minus a per-substitution penalty.

    predict_fn accepts a list of strings and returns a 1D array of raw MRL
    predictions. Its model and preprocessing must remain fixed during search.
    Returns a unique candidate table and a generation-0-through-G history.
    """
    reference = encode_sequence(reference_sequence)
    if population_size < 2 or not 1 <= tournament_size <= population_size:
        raise ValueError("Use at least two individuals and a valid tournament size.")
    if generations < 0 or not np.isfinite(fitness_lambda) or fitness_lambda < 0:
        raise ValueError("Generations and the finite edit penalty must be nonnegative.")
    if not all(0 <= rate <= 1 for rate in (crossover_rate, mutation_rate)):
        raise ValueError("Crossover and mutation probabilities must lie in [0, 1].")
    if not 0 < elite_fraction < 1:
        raise ValueError("The elite fraction must lie strictly between 0 and 1.")

    reference_prediction = float(np.asarray(predict_fn([reference_sequence]))[0])
    if not np.isfinite(reference_prediction):
```

```
        raise ValueError("The reference prediction must be finite.")
    population = rng.integers(0, 4, size=(population_size, len(reference)),
                              dtype=np.uint8)
    population[0] = reference
    elite_size = min(population_size - 1,
                     max(1, int(population_size * elite_fraction)))

    best_sequence = reference_sequence
    best_prediction = reference_prediction
    best_fitness = 0.0
    best_edits = 0
    history = []

    for generation in range(generations + 1):
        fitness, predictions, edit_counts = evaluate_sequence_population(
            population, reference, reference_prediction, predict_fn, fitness_lambda
        )
        winner = np.lexsort((edit_counts, -fitness))[0]
        # Equal fitness favors fewer substitutions; an identical tie keeps the answer.
        if (fitness[winner], -edit_counts[winner]) > (best_fitness, -best_edits):
            best_sequence = decode_population(population[winner:winner + 1])[0]
            best_prediction = float(predictions[winner])
            best_fitness = float(fitness[winner])
            best_edits = int(edit_counts[winner])

        history.append({
            "generation": generation,
            "best_fitness": best_fitness,
            "winner_predicted_mrl": best_prediction,
            "winner_edits": best_edits,
            "mean_fitness": float(fitness.mean()),
            "sequence_evaluations": 1 + population_size * (generation + 1),
        })
        if generation == generations:
            break  # The final population has now been evaluated and recorded.

        elites = unique_sequence_elites(population, fitness, edit_counts, elite_size)
        parents = select_sequence_parents(population, fitness, tournament_size, rng)
        offspring = uniform_sequence_crossover(parents, crossover_rate, rng)
        offspring = mutate_sequences(offspring, mutation_rate, rng)
        population = np.vstack((elites, offspring[:population_size - len(elites)]))
```

```
    results = pd.DataFrame({
        "sequence": decode_population(population),
        "predicted_mrl": predictions,
        "predicted_gain": predictions - reference_prediction,
        "edit_count": edit_counts,
        "fitness": fitness,
    })
    # Keep the reference and retained answer even if they leave the population.
    retained = pd.DataFrame([
        {"sequence": reference_sequence, "predicted_mrl": reference_prediction,
         "predicted_gain": 0.0, "edit_count": 0, "fitness": 0.0},
        {"sequence": best_sequence, "predicted_mrl": best_prediction,
         "predicted_gain": best_prediction - reference_prediction,
         "edit_count": best_edits, "fitness": best_fitness},
    ])
    results = pd.concat((retained, results), ignore_index=True)
    results = results.drop_duplicates("sequence").sort_values(
        ["fitness", "edit_count"], ascending=[False, True], kind="stable"
    ).reset_index(drop=True)
    return results, pd.DataFrame(history)
```

A small reproducible experiment

Choose a reference and hold the search budget fixed

We take the first A/C/G/T-only sequence in the training partition. This rule chooses a starting point without searching for a favorable measured or predicted value. Its use is appropriate for demonstrating optimization; it is not an independent test of the CNN's predictive accuracy. The test partition remains reserved and is not used to select the reference or tune the GA.

The data use `T` in their sequence strings, and we retain that convention; RNA's corresponding nucleotide is conventionally written `U`. Although the training encoder also accepts the unknown symbol `N`, the GA uses only the four unambiguous symbols A/C/G/T.

We compare $\lambda = 0$ with $\lambda = 0.10$. The positive value is an illustrative choice, not a calibrated biological threshold. Both runs use the same population size, number of generations, and starting random seed. This gives the same initial population; selection can cause the subsequent populations to diverge.

```
reference_sequence = next(
    sequence
```

```
    for sequence in train_data["utr"]
    if set(sequence) <= set("ACGT")
)

GA_SEED = 4106
POPULATION_SIZE = 64
GENERATIONS = 20
PENALTIES = (0.0, 0.10)

ga_runs = {}
for penalty in PENALTIES:
    candidates, history = run_sequence_ga(
        predict_mrl,
        reference_sequence,
        rng=np.random.default_rng(GA_SEED),
        population_size=POPULATION_SIZE,
        generations=GENERATIONS,
        fitness_lambda=penalty,
        tournament_size=3,
        crossover_rate=0.8,
        mutation_rate=0.02,
        elite_fraction=0.05,
    )
    ga_runs[penalty] = {"candidates": candidates, "history": history}

evaluations_per_run = 1 + POPULATION_SIZE * (GENERATIONS + 1)
print(f"Individual sequence evaluations per run: {evaluations_per_run:,}")
print(f"Total across {len(PENALTIES)} runs: "
      f"{len(PENALTIES) * evaluations_per_run:,}")
```

```
Individual sequence evaluations per run: 1,345
Total across 2 runs: 2,690
```

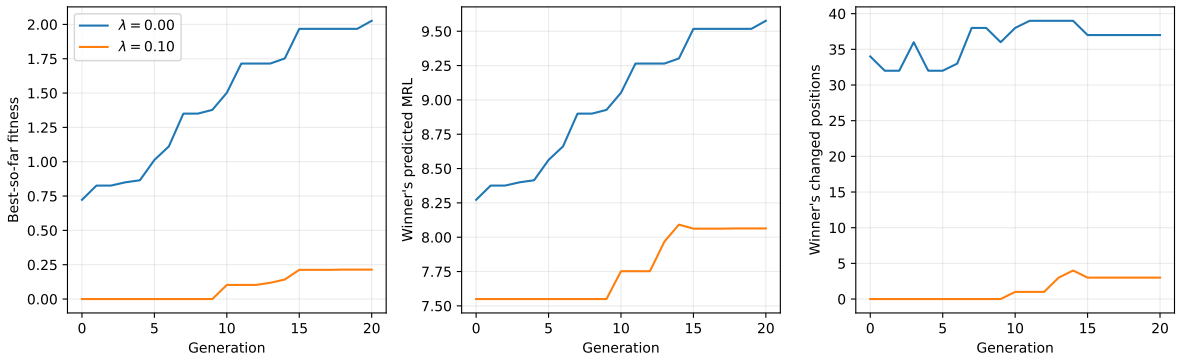
These are **individual sequence evaluations**, rather than the number of batched CNN calls or the number of distinct sequences. Each run includes one reference prediction and the populations at generations 0 through 20: $1 + 64 \times 21 = 1{,}345$ evaluations. The implementation deliberately has no prediction cache, so repeated candidates count again. CNN training is separate from this search budget.

Inspect the trajectories

Before displaying the result, predict which of the three curves below must be nondecreasing. The best-so-far fitness cannot decrease. Its associated MRL prediction and number of edits can move in either direction when a different candidate becomes the fitness winner.

```
fig, axes = plt.subplots(1, 3, figsize=(12, 3.8))
panels = [
    ("best_fitness", "Best-so-far fitness"),
    ("winner_predicted_mrl", "Winner's predicted MRL"),
    ("winner_edits", "Winner's changed positions"),
]
for penalty, run in ga_runs.items():
    history = run["history"]
    for axis, (column, title) in zip(axes, panels):
        axis.plot(history["generation"], history[column],
                  label=fr"$\lambda={penalty:.2f}$")
        axis.set(xlabel="Generation", ylabel=title)
        axis.grid(alpha=0.25)

axes[0].legend()
fig.tight_layout()
plt.show()
```



For $\lambda = 0$, predicted MRL and fitness differ by a constant, so the winning prediction is also nondecreasing. For positive λ, improved fitness may instead come from reducing the number of changed positions. Fitness values from runs with different penalties use different objectives; compare their predicted gains and edit counts as well as their scores.

Inspect the selected sequences

```
summary_rows = []
for penalty, run in ga_runs.items():
    winner = run["candidates"].iloc[0]
    reference = run["candidates"].loc[
        run["candidates"]["sequence"] == reference_sequence
    ].iloc[0]
    summary_rows.append({
        "lambda": penalty,
        "reference_predicted_mrl": reference["predicted_mrl"],
        "winner_predicted_mrl": winner["predicted_mrl"],
        "predicted_gain": winner["predicted_gain"],
        "changed_positions": int(winner["edit_count"]),
        "fitness": winner["fitness"],
        "sequence_evaluations": int(run["history"].iloc[-1][
            "sequence_evaluations"
        ]),
    })

summary = pd.DataFrame(summary_rows)
summary.round(3)
```

	lambda	reference_predicted_mrl	winner_predicted_mrl	predicted_gain	changed_positions	fitness
0	0.0	7.55	9.576	2.026	37	2.026
1	0.1	7.55	8.064	0.514	3	0.214

The table compares each winner with the reference prediction, without using experimental measurements as if they were generated-sequence outcomes. The best-so-far result includes the reference, so a run can return the reference when it finds no candidate with a better score.

```
print("Reference:", reference_sequence)
for penalty, run in ga_runs.items():
    winner_sequence = run["candidates"].iloc[0]["sequence"]
    changes = "".join(
        "^" if candidate != original else " "
        for candidate, original in zip(winner_sequence, reference_sequence)
    )
    print(f"\nlambda = {penalty:.2f}")
```

```
    print("Generated:", winner_sequence)
    print("Changes:  ", changes)
```

```
Reference: GAGGCTAAAGGGAAACCAGCCGCAAGTGACACGCTCAACAAGACACATAC

lambda = 0.00
Generated: TTGCTAAGATTTAGATGATGATCATTAATTTCAATAGAATGAAAGTAATG
Changes:   ^^ ^^^ ^ ^^^ ^ ^^ ^^^^  ^^^^^^^ ^^ ^^ ^^^^ ^^^ ^^^

lambda = 0.10
Generated: GAGGATAAAGGGAAACCAGTCGCAAGTGACACGCTCAACAAGACACAGAC
Changes:       ^               ^                             ^
```

The returned candidate table removes duplicates for inspection. The GA population may contain duplicates: avoiding duplicate elites does not make all offspring distinct.

What has the experiment established?

The GA searches a space of 4^{50} possible sequences using a limited budget. Its progress shows how selection and variation can improve a **specified learned fitness function**. Neither the best observed score nor a stable trajectory proves that the GA has found a global optimum.

More fundamentally, the GA sees predictions, not experimental MRL. It can select sequences for which the CNN is overconfident or inaccurate. A predictor that works well on held-out data from the original sampling distribution can behave differently on candidates deliberately selected for extreme scores. Keeping a candidate close to the reference may help express a design preference; it supplies no accuracy guarantee.

Discuss: what additional evidence would support a claim of improved translation efficiency? Independent predictive checks can reveal some problems, but establishing improvement in experimental MRL requires appropriate measurements. Agreement with a second learned model would still be agreement between surrogates.

The main connection is now complete: a CNN can supply the numerical objective for a GA, while the representation and operators determine which candidate sequences the search explores. The quality of the resulting design depends on both the search and the validity of the surrogate in the region it explores.

Appendix

Further experiments

These extensions use the existing trained CNN; they do not require another fit.

- **Penalty:** try a few predeclared values of λ. Compare predicted gain and changed positions at equal evaluation budgets. Does a larger penalty always produce fewer edits in a finite stochastic run? Explain why the penalty expresses a preference rather than a hard constraint.
- **Randomness:** repeat the two configurations with a short paired seed list. Keep the reference and all other settings fixed. Summarize variation instead of drawing a general conclusion from one pair of trajectories.
- **A baseline:** compare the GA with a random search that includes the reference and uses the same number of individual sequence evaluations. What would this comparison tell us about the benefit of selection and variation?

Do not tune these choices using the reserved test labels. An evaluation of surrogate accuracy on the original data is a different experiment from testing the generated sequences.

Sources and attribution

- Lecture 15 CNN tutorial: data preparation, random partitioning, target standardization, and the Keras architecture.
- Khalaj, S., and Turcotte, M. *AUGMENT: A Graph-Based Reinforcement Learning Framework for the Targeted Optimization of 5 UTR Translation Efficiency*. Manuscript under review. This notebook adapts its reference-guided GA baseline for teaching; it does not reproduce the full research study.
- Sample, P. J., et al. (2019). Human 5 UTR design and variant effect prediction from a massively parallel translation assay. *Nature Biotechnology*, **37**, 803–809. Source of the experimental dataset.
- Tang, X., et al. (2024). A novel deep generative model for mRNA vaccine development: Designing 5 UTRs with N1-methyl-pseudouridine modification. *Acta Pharmaceutica Sinica B*, **14**(4), 1814–1826. Architectural context for the Lecture 15 tutorial.