OXFORD GENETICS

Deep Learning in Genomic Sequence Benchmarking: Implementing CNNs with a Focus in Reproducibility

Jordan Winemiller[1,†]

[1]Department of Electrical Engineering and Computer Science, Florida Atlantic Universtiy, Boca Raton, Florida, USA;

Abstract

As deep learning becomes increasingly prevalent in genomics, maintaining reproducibility in model development and evaluation is essential. In the biosciences, particularly for large genomic sequence datasets, benchmarking has become a valuable tool for enabling repeatable sequence classification experiments. This project uses publicly available genomic sequence benchmark datasets to implement convolutional neural networks *(CNNs)* for regulatory element classification and to examine the practical challenges of reproducing published workflows. We reimplement baseline CNN models using contemporary deep learning libraries, document the full training and evaluation pipeline, and compare our results with those reported in prior work. In doing so, we highlight common issues arising from unmaintained code bases, including dependency incompatibilities and undocumented preprocessing steps, and assess their impact on downstream performance and repeatability. The outcome is a transparent, reproducible set of scripts and guidelines intended to help students and researchers reliably apply deep learning to benchmark genomic sequence datasets.

Keywords: Deep Learning; Convolutional Neural Networks; Reproducibility; Benchmarking; Computational Genomic

Introduction

GENOMIC sequence classification has become a central task in computational biology, enabling researchers to characterize functional elements and regulatory patterns directly from DNA. Building on this momentum, the work presented here focuses on recreating previously proposed benchmarks from Genomics Benchmarks: A Collection of Datasets for Genomic Sequence Classification (Grešová *et al.* 2023), which was designed to lower the barrier for experimenting with genomic data. These benchmarks provide a structured setting for evaluating models while promoting transparency and comparability across studies.

This work is intended for students and researchers in computational genomics, biostatistics, and computer science who seek practical, reproducible examples of applying deep learning to standardized genomic sequence benchmarks.

Background

Motivated by a background in biostatistics and computer science, this project aims to deepen practical experience with genomic datasets and to lay the foundation for future collaborations with faculty specializing in genomics. Recent literature has demonstrated that convolutional neural networks and related deep learning architectures can effectively learn predictive representations of genomic sequences, capturing both local motifs and higher-order patterns. In this context, the present work implements deep learning models on the established benchmark datasets and examines the practical challenges of achieving robust, repeatable results. By emphasizing reproducibility and clear experimental design, this project seeks to provide a useful reference for researchers and students who are beginning to work with deep learning methods in genomics. Implementation details and examples are availabe on *GitHub* (**?**) and *Google Colab*.

Materials and Methods

Datasets

Genomic datasets are highly complex due to diverse data formats, pervasive missing values, variable sequence lengths, extreme scale, class imbalance in functional annotations, and batch effects from different sequencing technologies or experimental conditions. The data formats available in the materials are stored as genomic coordinates. Some sequences were previously mapped using the *seq2loc* Python package that specializes in mapping sequences for fasta files, human enhancers, and non-TATA promoters to genomic coordinates Browser Extensible Data *(BED)* format.

The selected datasets in table 1 for benchmarking consist of two (2) of non-uniform sequence and two (2) of uniform sequence length, with one containing unbalanced classes. The first non-uniform set *dummy_mouse_enhancers_ensembl* was designed as a small 'dummy' dataset for prototyping. The second non-uniform set *drosophila_enhancers_stark* consisting of the drosophila enhancer (Kvon *et al.* 2014). Both uniform datasets were previously converted using *seq2loc* to the BED format. The first set *human_enhancers_cohn* consists of human enhancers (Cohn *et al.* 2018). The second set *human_nontata_promoters* of non-TATA promoters (Umarov and Solovyev 2017) has a class imbalance.

The materials include additional datasets that are available through the *genomic_benchmarks* (ML-Bioinfo-CEITEC 2025) Python package by **ML-Bioinfo-CEITEC** , and Hugging Face.

Table 1 Description of selected datasets in genomic-benchmarks package

Name	Number of Sequences	Number of Classes	Class Ratio[a]	Median Length[b]	Sequence Length Range[c]
dummy_mouse_enhancers_ensembl	1210	2	1.0	2381	331 to 4476
drosophila_enhancers_stark	6914	2	1.0	2142	236 to 3237
human_enhancers_cohn	27791	2	1.0	500	-
human_nontata_promoters	36131	2	1.2	251	-

[a] Ratio of the largest class
[b] Median length for all sequences in the dataset
[c] Minimum and maximum length for all sequences in the dataset. *Note: if blank all sequence are the length

Statistical Methods

The classification metrics for predictions will look at *True Positive, True Negative, False Positive, False Negative, Precision,* and *Recall (eq. 1)*. These metrics will be used to measure the *Accuracy (eq. 2)* and F_1 *Score (eq. 3)* for benchmarking the models. The F_1 *Score* metric will be used for binary predictions. Otherwise, the F_β *Score (eq. 4)* metric will be used for any multi-class classification. In addition to these metrics, *Cost Functions* will be monitored during model training.

Calculations

$$\begin{aligned} TP &= True\ Positive \\ TN &= True\ Negative \\ FP &= False\ Positive \\ FN &= False\ Negative \\ Precision &= \frac{TP}{TP+FP} \\ Recall &= \frac{TN}{TN+FP} \end{aligned} \tag{1}$$

$$Accuracy = \frac{FP}{TP+FP} = 1 - \frac{TP}{TP+FP} \tag{2}$$

$$F_1\ Score = 2 * \frac{precision * recall}{precision + recall} \tag{3}$$

$$F_\beta\ Score = \frac{(1+\beta^2) * TP}{(TP+FN) * \beta^2 + (TP+FP)} \tag{4}$$

Model Architectures

Convolutional neural networks (CNNs) are powerful deep learning architectures that extract biologically meaningful features from genomic sequences with minimal hand-engineered input. These models excel at DNA feature extraction by automatically discovering regulatory elements such as the spacing patterns that distinguish promoters from enhancers, and demonstrate robustness across diverse species and sequence contexts (Zhang and Imoto 2024). The output of the CNN layer is fed to the Max pooling layer, which helps to reduce the spatial dimensions of the feature maps while retaining the most essential information by selecting the maximum value within each pooling window. This allows the network to focus on the most significant features and become less sensitive to small variations in the input sequence, making the model more robust. To normalize the inputs to the layers, a batch normalization layer is introduced before the next layer (K S and S. Nair 2024).

The initial setup for this experiment is to implement two (2) deep convolutional neural networks with the **Keras** package. The first network will consist of three (3) convolution units with a **Convolutional 1D Layer** for one (1) dimensional data, **Batch Normalization Layer**, and a **Max Pooling 1D Layer** with a max pooling size of two (2). The fully connected layer will consist of a **Dropout Layer** with **30%** of the input units to be removed, a **Global Average Pooling 1D Layer** and a **Dense Layer**. The three (3) convolutional layers will all have a **kernel size** of 8, **filters** of 32, 16, and 4 in descending order, and implements the *Rectified Linear Units (ReLU) (eq. 5)* activation function. *Figure 1* shows the details of the first CNN architecture.

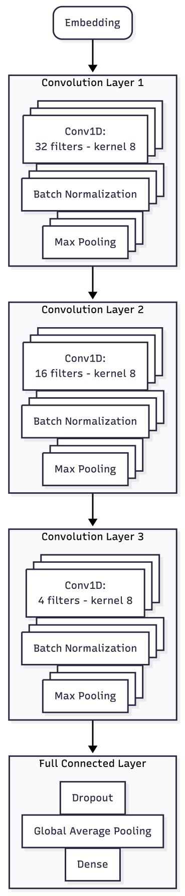


Figure 1 First Model Architecture

The second network will consist of three (3) convolution units with a **Convolutional 1D Layer** for the one (1) dimensional data, **Batch Normalization Layer**, and a **Max Pooling 1D Layer** with a max pooling size of two (2) as the first CNN architecture. The fully connected layer will consist of a **Flatten Layer** and two (2) **Dense Layers**. The three (3) convolutional layers will all have a **kernel size** of 8, **filters** of 16, 8, and 4 in descending order, and use the *ReLU* activation function. The first of the dense layers will have 512 units and use the *ReLU* activation function. *Figure 2* shows the details of the second CNN architecture.

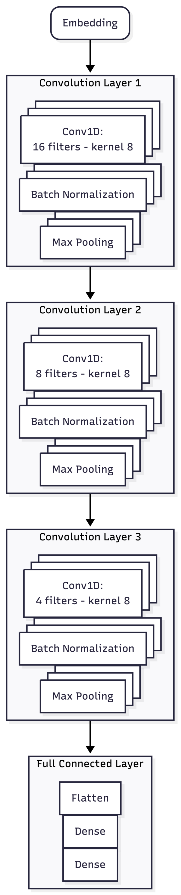


Figure 2 Second Model Architecture

The last dense layer in both networks will have one (1) unit for binary classification using the *Sigmoid (eq. 6)* activation function, and the number of classes for the units for multi-class classification using the *Softmax (eq. 7)* activation function.

Activation Functions

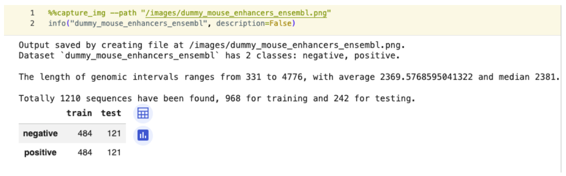
$$ReLU(x) = x^+ = max(0, x) = \frac{x + |x|}{2} = \begin{cases} x \; if \; x > 0, \\ 0 \; x \leq 0 \end{cases} \tag{5}$$

$$Sigmoid = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

$$Softmax = \sigma(z)_i = \frac{e^{z_i}}{\Sigma_{j=1}^{K} e^{z_j}} \tag{7}$$

Implementation

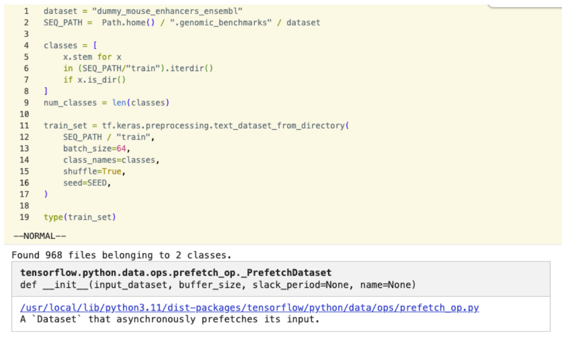
The **genomic-benchmarks** Python package from **ML-Bioinfo-CEITEC** is a GitHub repository(ML-Bioinfo-CEITEC 2025) that collects benchmarks for genomic sequence classification. The repository includes the datasets, source code, helper functions, and examples. The repository is also available via *PIP* installation. The original implementation of the datasets and example code do not use a validation set. *Figure 3* shows the initial information prototype dataset. *Figure 4* shows the type of the training dataset.



```
%%capture_img --path "/images/dummy_mouse_enhancers_ensembl.png"
info("dummy_mouse_enhancers_ensembl", description=False)
```

```
Output saved by creating file at /images/dummy_mouse_enhancers_ensembl.png.
Dataset `dummy_mouse_enhancers_ensembl` has 2 classes: negative, positive.

The length of genomic intervals ranges from 331 to 4776, with average 2369.5768595041322 and median 2381.

Totally 1210 sequences have been found, 968 for training and 242 for testing.
          train  test
negative    484   121
positive    484   121
```

Figure 3 Initial Prototype Dataset



```
dataset = "dummy_mouse_enhancers_ensembl"
SEQ_PATH =  Path.home() / ".genomic_benchmarks" / dataset

classes = [
    x.stem for x
    in (SEQ_PATH/"train").iterdir()
    if x.is_dir()
]
num_classes = len(classes)

train_set = tf.keras.preprocessing.text_dataset_from_directory(
    SEQ_PATH / "train",
    batch_size=64,
    class_names=classes,
    shuffle=True,
    seed=SEED,
)

type(train_set)
--NORMAL--
```

```
Found 968 files belonging to 2 classes.
tensorflow.python.data.ops.prefetch_op._PrefetchDataset
def __init__(input_dataset, buffer_size, slack_period=None, name=None)

/usr/local/lib/python3.11/dist-packages/tensorflow/python/data/ops/prefetch_op.py
A `Dataset` that asynchronously prefetches its input.
```

Figure 4 Dataset Type

The list of datasets available from *genomic-benchmarks* Python package can be found using *code 1*.

Code 1 List the Available Datasets

```
from genomic_benchmarks.data_check import list_datasets

list_datasets()
```

There are other dependencies that are needed to use the helper functions and models (This is covered in the next section on *Complications*). The datasets are available as built-in data classes for *TensorFlow* and *Torch*. The text files were loaded and processed in a batch size of 64 into a TensorFlow PreFetchDataSet. The network architecture figures utilized the python package *mermaid-py*. The *code 2* shows the pip installation, and *code 3* shows the imports and setup.

Code 2 PIP installation

```
# PIP Package Installation
!pip install -q mermaid-py
!pip install -q genomic-benchmarks
!pip install -q jupyter_capture_output
```

Code 3 Setup and Imports

```
# Setup and Imports
from pathlib import Path
import random
import os
import warnings

from genomic_benchmarks.data_check import (
    info,
    is_downloaded,
    list_datasets,
)
from genomic_benchmarks.loc2seq import download_dataset
import jupyter_capture_output
import keras
from keras import backend as K
from keras.layers import (
    Activation,
    BatchNormalization,
    Conv1D,
    Dense,
    Dropout,
    Flatten,
    GlobalAveragePooling1D,
    Input,
    Lambda,
    MaxPooling1D,
    TextVectorization,
)
from keras.losses import (
    BinaryCrossentropy,
    CategoricalCrossentropy,
)
from keras.metrics import (
    BinaryAccuracy,
    CategoricalAccuracy,
)
from keras.models import Sequential
import keras.ops as ops
import matplotlib.pyplot as plt
from mermaid import Mermaid
import numpy as np
import pandas as pd
import tensorflow as tf


SEED = 1234


os.environ["PYTHONHASHSEED"] = str(SEED)
random.seed(SEED)
np.random.seed(SEED)
keras.utils.set_random_seed(SEED)
tf.random.set_seed(SEED)

# Suppress Keras warnings
warnings.filterwarnings("ignore")
```

Downloading the datasets from the *genomic-benchmarks* Python package creates a hidden folder in your working directory with the setup in *code 4* will only download a non-existent dataset. The info command will produce information about the dataset from the *metadata.yaml* file that accompanies each dataset.

Code 4 Dataset Download

```
selected_dataset_list = [
    "dummy_mouse_enhancers_ensembl",
    "drosophila_enhancers_stark",
    "human_enhancers_cohn",
    "human_nontata_promoters",
]

for dataset in selected_dataset_list:
    if not is_downloaded(dataset):
        download_dataset(dataset)
    print(info(dataset, description=True))
    print("\n")
```

The dataset needed to be transformed using a *one-hot* encoding for each class. This process vectorized the text. The vectorization process *codes 5* and *6* show the methods used to facilitate the data transformation.

Code 5 Vectorization Layer

```
char_split_fn = lambda x: tf.strings.unicode_split(
    x, input_encoding="UTF-8")
vectorize_layer = keras.layers.TextVectorization(
    output_mode="int",
    split=char_split_fn,
)

vectorize_layer.adapt(train_set.map(lambda x, y: x))
vocab_size = len(vectorize_layer.get_vocabulary())
vectorize_layer.get_vocabulary()
```

Code 6 Vectorize Text

```
def vectorize_text(text, label):
    """Returns a vector representation of the text.

    :param text: The text to vectorize
    :param label: The label of the text
    :return: A vector representation of the text
    """
    text = tf.expand_dims(text, axis=-1)
    return vectorize_layer(text)-2, label
```

Providing transparent evidence of the model's performance is the cornerstone of benchmarking. *Code 7* describes the implementation for capturing these important metrics and providing visual evidence of the CNN model for training and evaluation and new unseen test data. The graphs produced will contain the accuracy, cost functions *(i.e., learning curve)*, and F_1 *score* for training. The method also provides metrics for the models evaluation of the testing dataset.

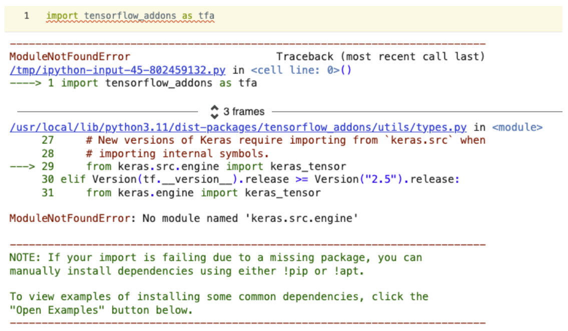
Code 7 Displaying Plots and Metrics

```
def plot_metrics(model_info_dict, set_name, horizontal=False):
    """Shows the plots for the training accuracy, f1 score, and loss.
    Then shows the scores for the model.

    :param model_info_dict: Dictionary of model information
    :param set_name: Name of dataset
    :param horizontal: Whether to show the plots horizontally
    """
    b_hist = model_info_dict["basic_history"]
    acc_1 = np.array(b_hist.history["binary_accuracy"])
    f1_1 = np.array(b_hist.history["f1_score"])
    loss_1 = np.array(b_hist.history["loss"])

    f_hist = model_info_dict["final_history"]
    acc_2 = np.array(f_hist.history["binary_accuracy"])
    f1_2 = np.array(f_hist.history["f1_score"])
    loss_2 = np.array(f_hist.history["loss"])

    # Shifted the starting index to start at 1 instead of 0
    epochs = np.arange(loss_1.shape[0]) + 1

    fig_size = (15, 5) if horizontal else (5, 15)
    plt.figure(figsize=fig_size)
    models = [model_info_dict["basic_name"], model_info_dict["final_name"]]

    rows = 1 if horizontal else 3
    cols = 3 if horizontal else 1

    plt.subplot(rows, cols, 1)
    plt.plot(epochs, acc_1, epochs, acc_2)
    plt.title("Training Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend(models, loc="lower right")

    plt.subplot(rows, cols, 2)
    plt.plot(epochs, f1_1, epochs, f1_2, linestyle="--")
    plt.title("Training F1 Score")
    plt.xlabel("Epochs")
    plt.ylabel
    plt.legend(models, loc="lower right")

    plt.subplot(rows, cols, 3)
    plt.plot(epochs, loss_1, epochs, loss_2)
    plt.title("Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Crossentropy Loss")
    plt.legend(models, loc="upper right")

    plt.show()

    for m in models:
        eval = model_info_dict[f"{m}_evaluation"]
        print(f"{m.title()} Model for Set: {set_name}:")
        print(f"Total Loss: {round(eval['loss'], 6)}")
        print(f"Accuracy: {round(eval['binary_accuracy'] * 100, 2)}%")
        print(f"F1 Score: {round(eval['f1_score'], 2)}")
        print()
```

Complications

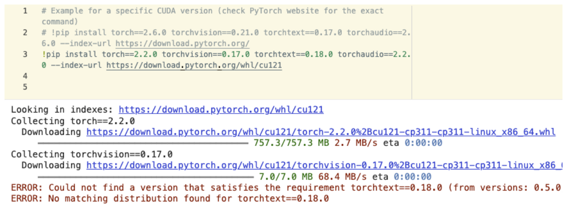
There were complications due to when *genomic-benchmarks* Python package was design and built. The authors built the source code with the helper functions to work with up to *python3.10*. There are a few dependencies in the source code that could not be used. Including *tensorflow-addons* for F_1 *score* due to the *Keras* package version (which is addressed in the next section), *torchtext* for tokenization methods in the source code, which does not have a compatible *torch* version that can be installed in *Google Colab*.

There were other complications due to the data being not uniform in length, adding complexity of using native *numpy* arrays. There was another issue with how the *one-hot* encoding source code was working and being utilized in the model (which is addressed in the next section). *Figures 5, 6, 7* show the complications.



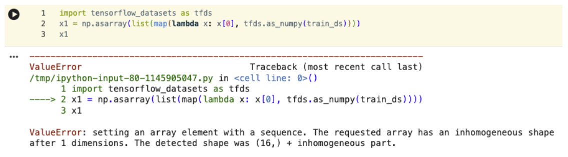
```
import tensorflow_addons as tfa

---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipython-input-45-802459132.py in <cell line: 0>()
----> 1 import tensorflow_addons as tfa

                                  3 frames
/usr/local/lib/python3.11/dist-packages/tensorflow_addons/utils/types.py in <module>
     27     # New versions of Keras require importing from `keras.src` when
     28     # importing internal symbols.
---> 29     from keras.src.engine import keras_tensor
     30 elif Version(tf.__version__).release >= Version("2.5").release:
     31     from keras.engine import keras_tensor

ModuleNotFoundError: No module named 'keras.src.engine'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

Figure 5 Tensorflow-addons Package Issue



```
# Example for a specific CUDA version (check PyTorch website for the exact
command)
# !pip install torch==2.6.0 torchvision==0.21.0 torchtext==0.17.0 torchaudio==2.
6.0 --index-url https://download.pytorch.org/
!pip install torch==2.2.0 torchvision==0.17.0 torchtext==0.18.0 torchaudio==2.2.
0 --index-url https://download.pytorch.org/whl/cu121


Looking in indexes: https://download.pytorch.org/whl/cu121
Collecting torch==2.2.0
  Downloading https://download.pytorch.org/whl/cu121/torch-2.2.0%2Bcu121-cp311-cp311-linux_x86_64.whl
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 757.3/757.3 MB 2.7 MB/s eta 0:00:00
Collecting torchvision==0.17.0
  Downloading https://download.pytorch.org/whl/cu121/torchvision-0.17.0%2Bcu121-cp311-cp311-linux_x86_
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.0/7.0 MB 68.4 MB/s eta 0:00:00
ERROR: Could not find a version that satisfies the requirement torchtext==0.18.0 (from versions: 0.5.0
ERROR: No matching distribution found for torchtext==0.18.0
```

Figure 6 Torchtext Package Issue



```
import tensorflow_datasets as tfds
x1 = np.asarray(list(map(lambda x: x[0], tfds.as_numpy(train_ds))))
x1

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-80-1145905047.py in <cell line: 0>()
      1 import tensorflow_datasets as tfds
----> 2 x1 = np.asarray(list(map(lambda x: x[0], tfds.as_numpy(train_ds))))
      3 x1

ValueError: setting an array element with a sequence. The requested array has an inhomogeneous shape
after 1 dimensions. The detected shape was (16,) + inhomogeneous part.
```

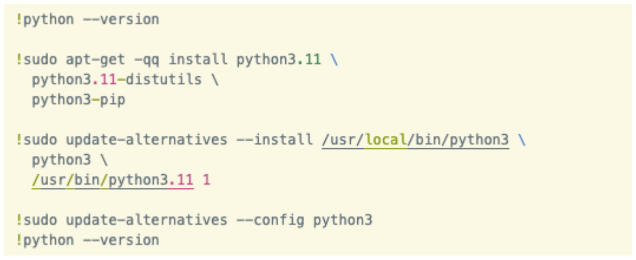
Figure 7 Numpy Data Shape Issue

Mitigation Attempts

Attempted: Working through the source code and complications, I decided to lower the python version to the lowest available in the google Colab environment. Initially, there was an attempt to switch the version shown in *figure 8*.

This was abandoned due to time constraints. After this change was made, the first model was able to produce results with minimal additional workarounds.

Fixed: The second model required additional new layers to be added to the architecture to handle the output dimension from the last **Convolutional Layer** to the **Flatten Layer**. A **Global Average Pooling Layer** was added to the CNN architecture. *Figure 9* shows the updated architecture for the second model.



```
!python --version

!sudo apt-get -qq install python3.11 \
  python3.11-distutils \
  python3-pip

!sudo update-alternatives --install /usr/local/bin/python3 \
  python3 \
  /usr/bin/python3.11 1

!sudo update-alternatives --config python3
!python --version
```

Figure 8 Python3.11 Installation

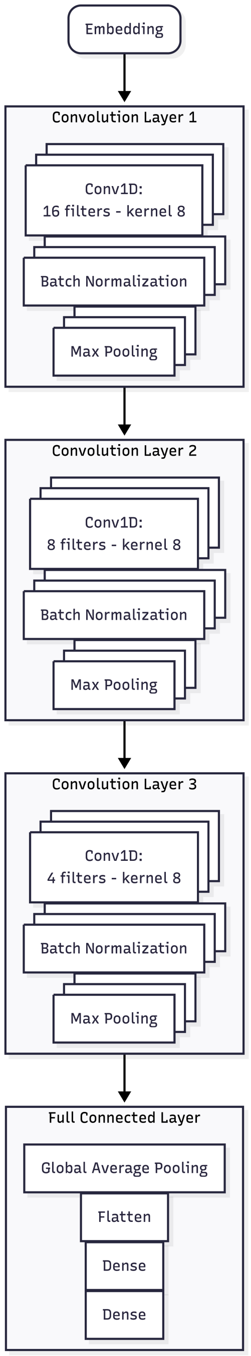


Figure 9 Second Model Architecture Update

Fixed: The F_1 *Score* that is implemented in the *genomic-benchmarks* Python package requires a version of *Keras* that is not supported in the available version of *Google Colab*. A Python function *code 8* was written for the implementation of the F_1 *Score* metric for the evaluation of the model.

Code 8 F_1 *Score* Calculation

```
def f1_score(y_true, y_pred):
    """Returns the F1 score.

    :param y_true: The true labels
    :param y_pred: The predicted labels
    :return: The F1 score
    """
    def precision(y_true, y_pred):
        """Returns the precision.

        :param y_true: The true labels
        :param y_pred: The predicted labels
        :return: The precision
        """
        true_positives = ops.sum(
            ops.round(ops.clip(tf.cast(y_true, tf.float32) * y_pred, 0, 1)))
        predicted_positives = ops.sum(ops.round(ops.clip(y_pred, 0, 1)))
        precision = true_positives / (predicted_positives + K.epsilon())
        return precision

    def recall(y_true, y_pred):
        """Returns the recall.

        :param y_true: The true labels
        :param y_pred: The predicted labels
        :return: The recall
        """
        true_positives = ops.sum(
            ops.round(ops.clip(tf.cast(y_true, tf.float32) * y_pred, 0, 1)))
        possible_positives = tf.cast(
            ops.sum(ops.round(ops.clip(y_true, 0, 1))), tf.float32)
        recall = (
            tf.cast(true_positives, tf.float32)
            / (possible_positives + K.epsilon())
        )
        return recall

    precision = precision(y_true, y_pred)
    recall = recall(y_true, y_pred)
    return 2 * ((precision * recall) / (precision + recall + K.epsilon()))
```

Results

Utility Code

To facilitate the execution for training and testing the models on multiple datasets; utility code was built to allow for a repeatable setup. *Code 9* allows for an update for training and test datasets in the notebook. This was required because the variables existing in a global scope.

Code 9 Update the Dataset Selection.

```
def update_train_test_sets(id_num, batch_size=64):
    """Updates the training and testing sets, and returns the name of the
    dataset.

    :param id_num: The ID number of the dataset
    :param batch_size: The batch size
    """
    batch_size = batch_size
    selected_dataset = selected_dataset_list[id_num]
    SEQ_PATH = Path.home() / ".genomic_benchmarks" / selected_dataset

    classes = [
        x.stem for x
        in (SEQ_PATH/"train").iterdir()
        if x.is_dir()
    ]
    num_classes = len(classes)

    train_set = tf.keras.preprocessing.text_dataset_from_directory(
        SEQ_PATH / "train",
        batch_size=batch_size,
        class_names=classes,
        shuffle=True,
        seed=SEED,
    )

    test_set = tf.keras.preprocessing.text_dataset_from_directory(
        SEQ_PATH / "test",
        batch_size=batch_size,
        class_names=classes,
    )

    if num_classes > 2:
        train_set = train_set.map(
            lambda x, y: (x, tf.one_hot(y, depth=num_classes)))

    if num_classes > 2:
        test_set = test_set.map(
            lambda x, y: (x, tf.one_hot(y, depth=num_classes)))
    vectorize_layer.adapt(train_set.map(lambda x, y: x))
    vocab_size = len(vectorize_layer.get_vocabulary())
    vectorize_layer.get_vocabulary()

    train_ds = train_set.map(vectorize_text)
    test_ds = test_set.map(vectorize_text)

    return selected_dataset
```

The models were trained and evaluated using the method *code 10* to populate a dictionary with model metrics after the models were compiled, trained and evaluated.

Code 10 Train and Evaluate Models.

```
def train_and_evaluate_models(set_name, epochs):
    """Returns the training and evaluation the metrics for the models.

    :param set_name: Name of dataset
    :param epochs: Number of epochs to train
    :return: Dictionary of model information
    """
    model_info = {
        "basic_name": None,
        "basic_history": None,
        "basic_evaluation": None,
        "final_name": None,
        "final_history": None,
        "final_evaluation": None,
    }

    print(f"Training Models for: {set_name}")
    basic_name, basic_model = create_basic_cnn_model(num_classes, vocab_size)
    model_info["basic_name"] = basic_name
    basic_history = basic_model.fit(
        train_ds,
        epochs=epochs,
        verbose=0,
    )
    model_info["basic_history"] = basic_history
    model_info["basic_evaluation"] = basic_model.evaluate(
        test_ds, verbose=0, return_dict=True)

    final_name, final_model = create_final_cnn_model(num_classes, vocab_size)
    model_info["final_name"] = final_name
    final_history = final_model.fit(
        train_ds,
        epochs=epochs,
        verbose=0,
    )
    model_info["final_history"] = final_history
    model_info["final_evaluation"] = final_model.evaluate(
        test_ds, verbose=0, return_dict=True)

    return model_info
```

Repeating the setup with the above utilities for training and evaluation of the models on each dataset was reduced to the *code 11*.

Code 11 Example Training and Evaluation Run.

```
dataset_number = 0
dataset = update_train_test_sets(0)
model_0 = train_and_evaluate_models(dataset, epochs)
```

Models

The models for these experiments were built with the *Keras* Python package using the architectures mentioned in the model architecture . All of the experiments ran for 10 epochs. The baseline model was built with *code 12*, and the final model was compiled with the *code 13*.

Code 12 Baseline Model.

```
def create_basic_cnn_model(num_classes, vocab_size):
    """Returns a basic CNN model and model name.

    :param num_classes: The number of classes to classify
    :param vocab_size: The size of the vocabulary
    :return: Model name and CNN model
    """
    name = "basic"

    if num_classes == 2:
        last_layer = Dense(1, activation="sigmoid")
        loss = BinaryCrossentropy(from_logits=True)
        acc = BinaryAccuracy()

    else:
        last_layer = Dense(num_classes, activation="softmax")
        loss = "categorical_crossentropy"
        acc = CategoricalAccuracy()

    onehot_layer = Lambda(
        lambda x: tf.one_hot(tf.cast(x, "int64"), depth=vocab_size))

    print(onehot_layer)
    model = Sequential(
        [
            onehot_layer,
            Conv1D(
                filters=32,
                kernel_size=8,
                data_format="channels_last",
                activation="relu",
            ),
            BatchNormalization(),
            MaxPooling1D(),
            Conv1D(
                filters=16,
                kernel_size=8,
                data_format="channels_last",
                activation="relu",
            ),
            BatchNormalization(),
            MaxPooling1D(),
            Conv1D(
                filters=4,
                kernel_size=8,
                data_format="channels_last",
                activation="relu",
            ),
            BatchNormalization(),
            MaxPooling1D(),
            Dropout(0.3),
            GlobalAveragePooling1D(),
            last_layer,
        ]
    )

    model.compile(
        optimizer="adam",
        loss=loss,
        metrics=[acc, f1_score],
    )

    return name, model
```

Code 13 Final Model.

```
def create_final_cnn_model(num_classes, vocab_size):
    """Returns a CNN model and model name.

    :param num_classes: The number of classes to classify
    :param vocab_size: The size of the vocabulary
    :return: Model name and CNN model
    """
    name = "final"

    if num_classes == 2:
        last_layer = Dense(1, activation="sigmoid")
        loss = BinaryCrossentropy(from_logits=True)
        acc = BinaryAccuracy(threshold=0.5)
    else:
        last_layer = Dense(num_classes, activation="softmax")
        loss = "categorical_crossentropy"
        acc = CategoricalAccuracy()

    onehot_layer = Lambda(
        lambda x: tf.one_hot(tf.cast(x, "int64"), depth=vocab_size))

    model = Sequential(
        [
            onehot_layer,

            Conv1D(
                filters=16,
                kernel_size=8,
                data_format="channels_last",
            ),
            BatchNormalization(),
            Activation("relu"),
            MaxPooling1D(),

            Conv1D(
                filters=8,
                kernel_size=8,
                data_format="channels_last",
                activation="relu",
            ),
            BatchNormalization(),
            MaxPooling1D(),

            Conv1D(
                filters=4,
                kernel_size=8,
                data_format="channels_last",
                activation="relu",
            ),
            BatchNormalization(),
            MaxPooling1D(),

            GlobalAveragePooling1D(),
            Flatten(),
            Dense(units=512, activation="relu"),
            last_layer,
        ]
    )

    model.compile(
        optimizer="adam",
        loss=loss,
        metrics=[acc, f1_score],
    )

    return name, model
```

Dataset: Dummy Mouse Enhancers Ensembl

The models were trained on 968 sequences for 10 epochs. Then, tested on 242 sequences. The results of the training are shown in *figure 10*. The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table 2. *Note: this is dataset index 0.*

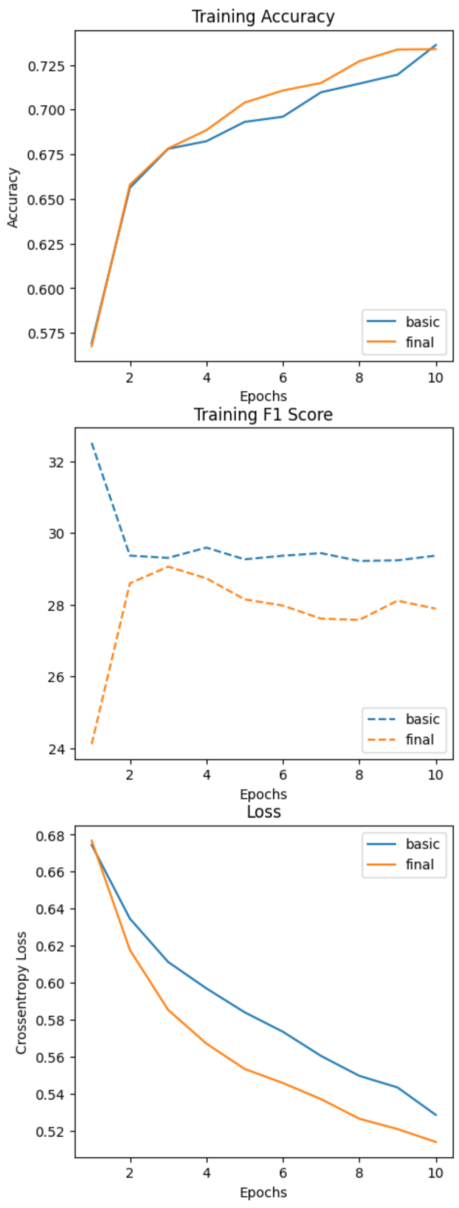


Figure 10 Model Training for dummy_mouse_enhancers_ensembl

Table 2 Evaluation Metrics for dummy_mouse_enhancers_ensembl Dataset

Model Name	Total Loss	Accuracy	F_1 Score
Basic	0.5797	66.82%	33.13
Final	0.9998	50.64%	0.76

Dataset: Drosophila Enhancers Stark

The models were trained on 5,184 sequences for 10 epochs. Then, tested on 1,730 sequences. The results of the training are shown in *figure 11*. The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table 3. *Note: this is dataset index 1.*

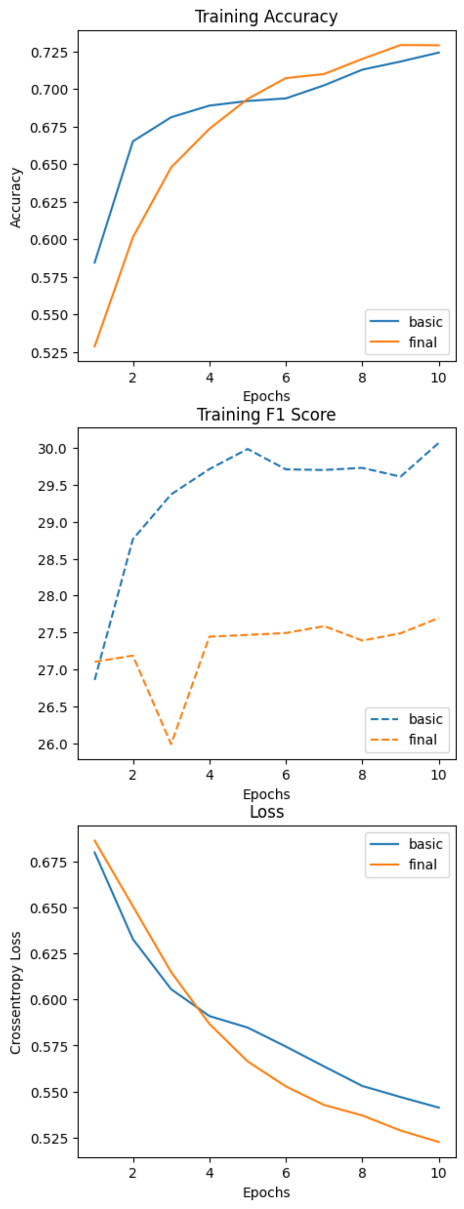


Figure 11 Model Training for drosophila_enhancers_stark Dataset

Table 3 Evaluation Metrics for drosophila_enhancers_stark

Model Name	Total Loss	Accuracy	F_1 Score
Basic	1.3692	49.94%	40.93
Final	0.8922	50.46%	0.82

Dataset: Human Enhancers Cohn

The models were trained on 20,843 sequences for 10 epochs. Then, tested on 6,948 sequences. The results of the training are shown in *figure 12*. The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table 4. *Note: this is dataset index 2.*

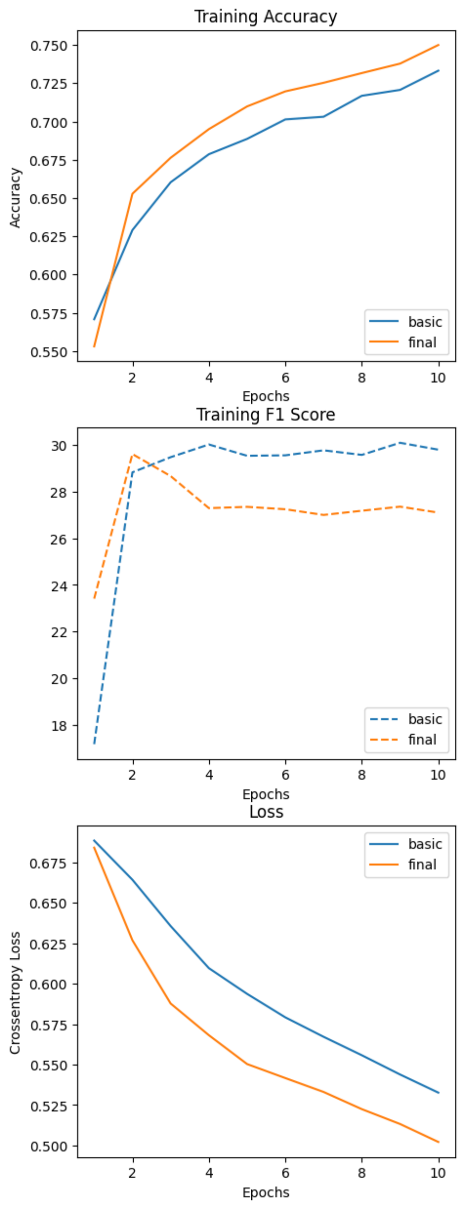


Figure 12 Model Training for human_enhancers_cohn Dataset

Table 4 Evaluation Metrics for human_enhancers_cohn

Model Name	Total Loss	Accuracy	F_1 Score
Basic	0.7929	52.60%	40.45
Final	0.6263	65.09%	14.97

Dataset: Human Non-TATA Promoters

The models were trained on 27,097 sequences for 10 epochs. Then, tested on 9,034 sequences. The results of the training are shown in *figure 13*. The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table 5. *Note: this is dataset index 3.*

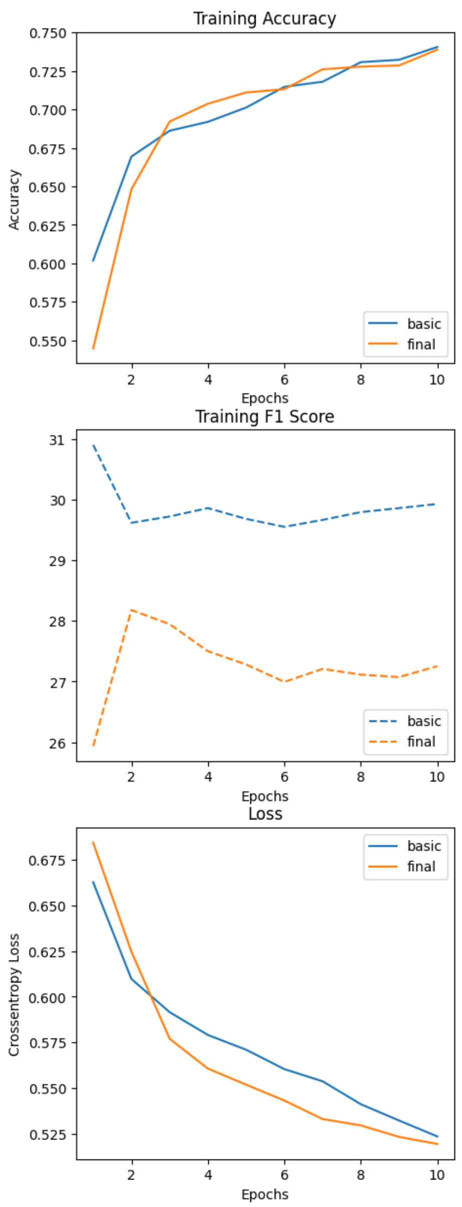


Figure 13 Model Training for human_nontata_promoters Dataset

Table 5 Evaluation Metrics for human_nontata_promoters

Model Name	Total Loss	Accuracy	F_1 Score
Basic	0.5817	71.56%	23.30
Final	0.6068	66.36%	20.33

Benchmark Analysis

In benchmarking the models on the selected datasets, the models showed difficulties in obtaining any significant training loss or accuracy. Upon evaluation of the models, the accuracy appears to predict classes at random expected with the *human non-TATA promoters*, but that dataset had imbalance favoring the predicted class. The F_1 scores for evaluation performed better on the baseline model, which could be due in part to the additional *global average pooling* layer that had to be added to the model architecture. The original final model architecture was compiled using *Pytorch*, but that should not account for the amount of lack of performance.

Improvements

There are a few improvements beyond the updates required for newer versions of the packages used in the experiments. Future experiments would benefit from the addition of validation sets to test for overfitting during training. Moving the activation layer for the CNN units to after the batch normalization layer. Increasing the number of epochs for training iterations. Using built-in optimizers to allow for control of the learning rates and built-in data encoding for input layers to understand the number of parameters in the models.

Other improvements consists of testing different model architectures, using data augmentation, and handling of sparse data.

Conclusions

Although initial attempts to reproduce the previous results with the deep learning models did not produce the expected results, mainly due to the implementation obstacles. The undertaking of a project to become familiar with genomic data and developing Convolutional Neural Networks through updating and working through the proposed architectures; there is an excitement to continue exploring more deep learning opportunities in the field of genomics.

Miscellaneous Code

Additional code used to create the report:

The diagrams use *mermaid-py* and *code 14* shows an example use case for this project.

Code 14 Mermaid Diagram Example

```
final_cnn_graph_update = """
%%{init:{'flowchart':{'nodeSpacing':1, 'rankSpacing':10}}}%%

flowchart TD
    classDef withMargines fill-opacity:0.0,color:#FFFFFF,stroke-width:0px;
    i(Embedding)

    subgraph conv1[Convolution Layer 1]
        space1["<p style='width:100px;height:0px;margin:0'>Space</p>"]:::withMargines;
        c1@{ shape: st-rect, label: "Conv1D:\n16 filters - kernel 8" }
        b1@{ shape: st-rect, label: "Batch Normalization" }
        a1@{ shape: st-rect, label: "Activation ReLU" }
        p1@{ shape: st-rect, label: "Max Pooling" }
    end
    %% Define a class to make the padding subgraph invisible
    classDef padding stroke:none,fill:none

    subgraph conv2[Convolution Layer 2]
        space2["<p style='width:100px;height:0px;margin:0'>Space</p>"]:::withMargines;
        c2@{ shape: st-rect, label: "Conv1D:\n8 filters - kernel 8" }
        b2@{ shape: st-rect, label: "Batch Normalization" }
        p2@{ shape: st-rect, label: "Max Pooling" }
    end

    subgraph conv3[Convolution Layer 3]
        space3["<p style='width:100px;height:0px;margin:0'>Space</p>"]:::withMargines;
        c3@{ shape: st-rect, label: "Conv1D:\n4 filters - kernel 8" }
        b3@{ shape: st-rect, label: "Batch Normalization" }
        p3@{ shape: st-rect, label: "Max Pooling" }
    end
    subgraph dense[Full Connected Layer]
        space4["<p style='width:100px;height:0px;margin:0'>Space</p>"]:::withMargines;
        p4[Global Average Pooling]
        f[Flatten]
        d1[Dense]
        d2[Dense]
    end

     i ------> conv1
    conv1 ------> conv2
    conv2 ------> conv3
    conv3 ------> dense
"""

Mermaid(final_cnn_graph_update)
```

Verification of the length of the dataset during exploratory data analysis and data cleansing is critical in work with genomic data. *Code 15* shows the implementation of this method.

Code 15 Checking the Length Method

```
def check_seq_lengths(dataset, use_padding):
    """Returns the maximum sequence length and the length of the
     sequence with tokens.

    :param dataset: List of sequences
    :param use_padding: Padding
    :return: Maximum sequence length and length of sequence
        with tokens
    """
    max_seq_len = max(
        [len(dataset[i][0]) for i in range(len(dataset))])
    print(f"Max Sequence Length: {max_seq_len}")
    same_length = [
        len(dataset[i][0]) == max_seq_len for i in range(len(dataset))]
    if not all(same_length):
        print("not all sequences are of the same length")

    if use_padding:
        len_with_tokens = max_seq_len + 3
    else:
        len_with_tokens = max_seq_len + 2

    return max_seq_len, len_with_tokens
```

Also, the package *listings* package was added and renamed to display **Code** instead of **Listing** using the *renewcommand* in the class file.

Data availability

All datasets are available for GitHub (ML-Bioinfo-CEITEC 2025), and all code covered in this article can be found at (jmwinemiller 2026) or at Google Colab.

Acknowledgments

Not applicable.

Conflicts of interest

There are no conflicts of interest.

Literature cited

Cohn D, Zuk O, Kaplan T. 2018. Enhancer identification using transfer and adversarial deep learning of dna sequences. bioRxiv. .

Grešová K, Martinek V, Čechák D, Šimeček P, Alexiou P. 2023. Genomic benchmarks: a collection of datasets for genomic sequence classification. BMC Genomic Data. 24:25.

jmwinemiller. 2026. cap-6619-project. https://github.com/jmwinemiller/cap-6619-project.

K S S, S. Nair M. 2024. Dnacoder: a cnn-lstm attention-based network for genomic sequence data compression. Neural Computing and Applications. 36:18363–18376.

Kvon E, Kazmar T, Stampfel G, Yáñez-Cuna J, Pagani M, Schernhuber K, Dickson B, Stark A. 2014. Genome-scale functional characterization of drosophila developmental enhancers in vivo. Nature. 512.

ML-Bioinfo-CEITEC. 2025. genomic_benchmarks. https://github.com/ML-Bioinfo-CEITEC/genomic_benchmarks.

Umarov R, Solovyev V. 2017. Recognition of prokaryotic and eukaryotic promoters using convolutional deep learning neural networks. PLOS ONE. 12.

Zhang Yz, Imoto S. 2024. Genome analysis through image processing with deep learning models. Journal of Human Genetics. 69.