Deep Learning in Genomic Sequence Benchmarking

Implementing CNNs with a Focus in Reproducibility

Jordan Winemiller

Department of Electrical Engineering & Computer Science
Florida Atlanta Univeristy

February 9, 2026

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.

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 [2], 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.

Target Audience

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 available on *GitHub* [6] and *Google Colab*.

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.

Datasets

The selected datasets in table 8 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 [5]. Both uniform datasets were previously converted using *seq2loc* to the BED format. The first set *human_enhancers_cohn* consists of human enhancers [1]. The second set *human_nontata_promoters* of non-TATA promoters [7] has a class imbalance. [1]

[1] The materials include additional datasets that are available through the *genomic_benchmarks* [6] Python package by **ML-Bioinfo-CEITEC** , and Hugging Face

Selected Datasets

Name	Number of Sequences	Number of Classes	Class Ratio[2]	Median Length[3]	Sequence Length Range[4]
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	-

[2] Ratio of the largest class

[3] Median length for all sequences in the dataset

[4] 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.

Statistical Methods: 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 Architecture

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 [8]. 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 [4].

Model Architecture

The initial setup for this experiment is to implement two (2) deep convolutional neural networks with the **Keras** package.
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.

Model Architecture: Activation Functions

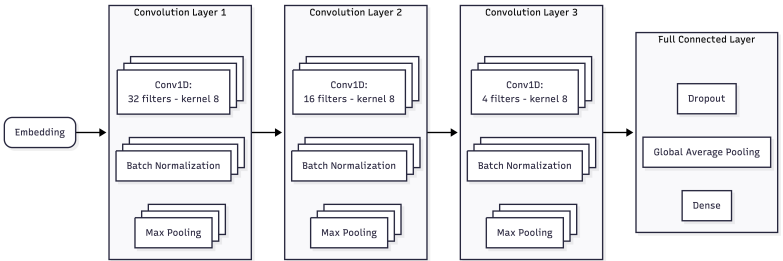
$$ReLU(x) = x^+ = max(0, x) = \frac{x + |x|}{2} = \begin{cases} x \ if \ x > 0, \\ 0 \ x \leq 0 \end{cases} \tag{5}$$

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

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

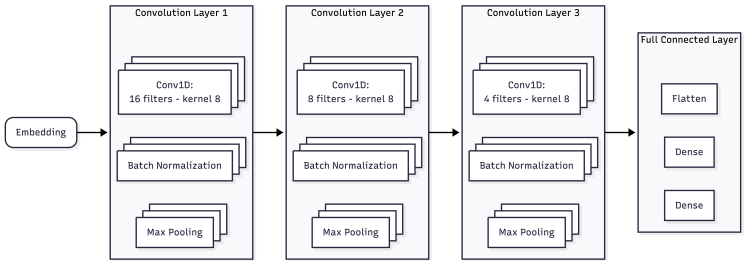
Model Architecture - Basic

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. The figure shows the details of the first CNN architecture.



Model Architecture - Final

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. The figure shows the details of the second CNN architecture.



Implementation

The **genomic-benchmarks** Python package from **ML-Bioinfo-CEITEC** is a GitHub repository[6] 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 1* shows the initial information prototype dataset. *Figure 2* shows the type of the training dataset.

Figure: Initial Prototype Dataset

Figure: Dataset Type

Impl: Dependencies

- PIP Install
  - genomic_benchmarks
  - mermaid-py
  - jupyter_capture_output
- Imports
  - pathlib
  - os
  - keras
    - backend
    - layers
    - losses
    - models
    - metrics
  - matplotlib
  - numpy
  - tensorflow

Impl: Dataset Download

Downloading the datasets from the *genomic-benchmarks* Python package creates a hidden folder in your working directory with the setup in code 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.

```
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")
```

Impl: Transforming the Data

The dataset needed to be transformed using a *one-hot* encoding for each class. This process vectorized the text. The vectorization process *codes 1* and *2* show the methods used to facilitate the data transformation.

1: 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()
```

2: 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
```

Impl: Metrics

Providing transparent evidence of the model's performance is the cornerstone of benchmarking. 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.

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*.

Complications Cont.

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 3, 4, 5* show the complications.

Figure: Tensorflow-addons Package Issue

Figure: Torchtext Package Issue

Figure: 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 6*.

```
!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: Python3.11 Installation

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

Mitigation Attempts Cont.

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 7* shows the updated architecture for the second model.

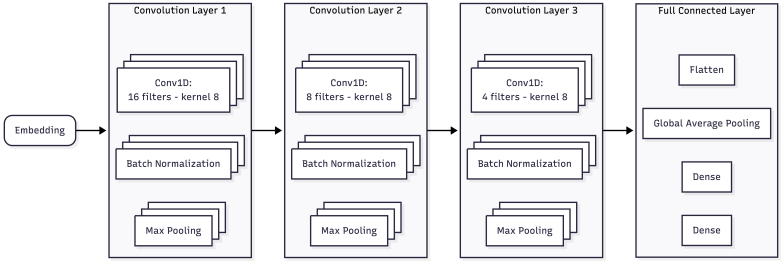


Figure: Second Model Architecture Update

Mitigation Attempts Cont.

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 was written for the implementation of the F_1 *Score* metric for the evaluation of the model.

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. This allows for an update for training and test datasets in the notebook. This was required because the variables existing in a global scope.
The models were trained and evaluated using the method to populate a dictionary with model metrics after the models were compiled, trained and evaluated.
Repeating the setup with the above utilities for training and evaluation of the models on each dataset was reduced to the *code 3*.

3: Example Training and Evaluation Run.

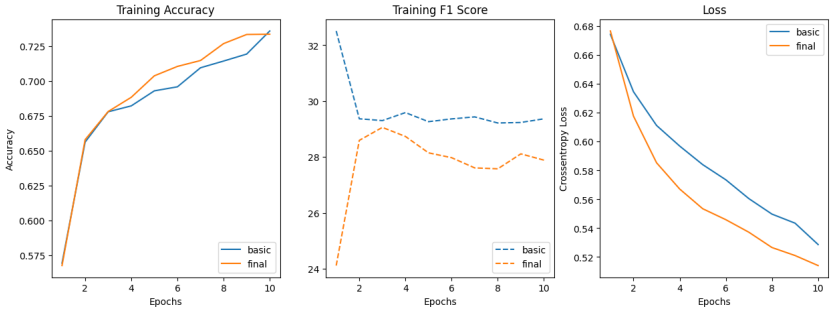
```
dataset_number = 0
dataset = update_train_test_sets(0)
model_0 = train_and_evaluate_models(dataset, epochs)
```

Results: 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 and final model were built using training and evaluation method in the paper.

Results: Models - Dummy Mouse Enhancers Enseml

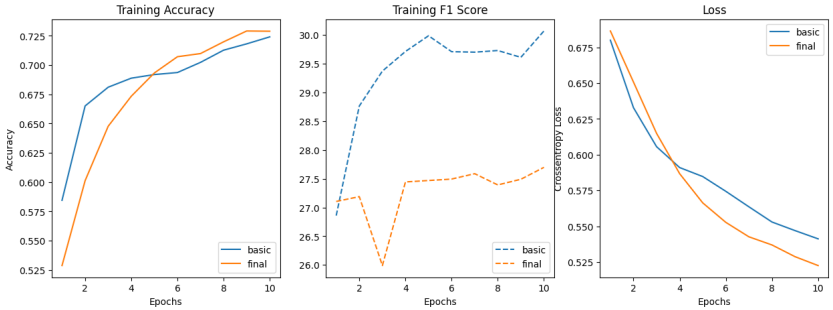
The models were trained on 968 sequences for 10 epochs. Then, tested on 242 sequences. The results of the training are shown figure below.
The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table. *Note: this is dataset index 0.*



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

Results: Models - Drosophila Enhancers Stark

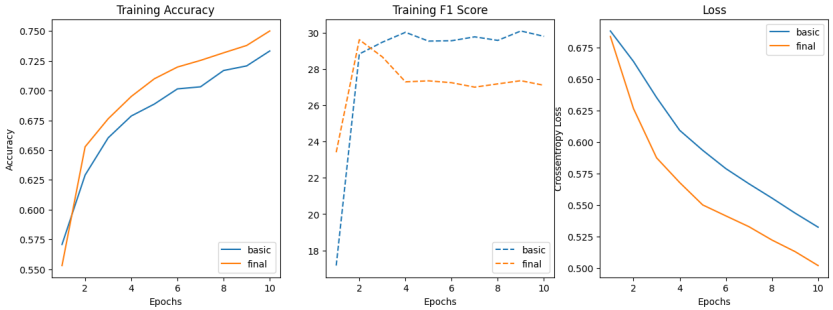
The models were trained on 5,184 sequences for 10 epochs. Then, tested on 1,730 sequences. The results of the training are shown figure below.
The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table. *Note: this is dataset index 1.*



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

Results: Models - Human Enhancers Cohn

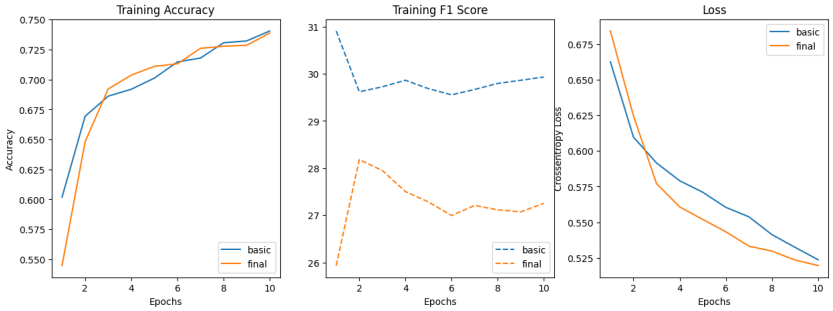
The models were trained on 20,843 sequences for 10 epochs. Then, tested on 6,948 sequences. The results of the training are shown figure below.
The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table. *Note: this is dataset index 2.*



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

Results: Models - 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 figure below.
The **total loss, accuracy, and F_1 score** evaluation metrics are shown in table. *Note: this is dataset index 3.*



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

Results: 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.

Future 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 used to create the report is in GitHub [3] The diagrams use *mermaid-py* for this project

Data Availability

All datasets are available for GitHub [6], and all code covered in this article can be found at [3] or at Google Colab.

References I

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

[2] K. Grešová, V. Martinek, D. Čechák, P. Šimeček, and P. Alexiou. Genomic benchmarks: a collection of datasets for genomic sequence classification. *BMC Genomic Data*, 24(1):25, 2023.

[3] jmwinemiller. cap-6619-project. `https://github.com/jmwinemiller/cap-6619-project`, 2026.

[4] S. K S and M. S. Nair. Dnacoder: a cnn-lstm attention-based network for genomic sequence data compression. *Neural Computing and Applications*, 36:18363–18376, 07 2024.

[5] E. Kvon, T. Kazmar, G. Stampfel, J. Yáñez-Cuna, M. Pagani, K. Schernhuber, B. Dickson, and A. Stark. Genome-scale functional characterization of drosophila developmental enhancers in vivo. *Nature*, 512, 06 2014.

References II

[6] ML-Bioinfo-CEITEC. genomic_benchmarks. `https://github.com/ML-Bioinfo-CEITEC/genomic_benchmarks`, 2025.

[7] R. Umarov and V. Solovyev. Recognition of prokaryotic and eukaryotic promoters using convolutional deep learning neural networks. *PLOS ONE*, 12, 02 2017.

[8] Y.-z. Zhang and S. Imoto. Genome analysis through image processing with deep learning models. *Journal of Human Genetics*, 69, 07 2024.