Project 3: Neural Networks for Gene Regulation

Sydney Ballard, Qi Gao, Kush Gulati

The goal of this project was to develop neural network models to perform binary classification on 4 datasets to determine whether a given sequence is an enhancer or not. Below we will describe in detail (1) the models developed for each of the four data sets, (2) the training and testing procedures used, and (3) our results.

Developing the Models

Before we could develop any models, we needed to preprocess our data by 1) parsing the FASTA files into dictionaries with key as sequence and value as label (1 for enhancer and 0 for not), 2) converting the DNA sequences into one-hot-encodings, and 3) transforming the data and labels into a numpy array. Functions “parse_data”, “one_hot_encoding”, and “transform_data” do just that.

To build the models developed for each of the four data sets, we used the Tensorflow and Keras modules, specifically importing keras.layers to provide classes for different types of layers used in our neural network.

For each of our models, we first created a new sequential model, which is a linear stack of layers in Keras to which layers can be added using model.add(). Next, a **Convolution2D** layer was added, with 32 filters, each of size 6x4, and the ReLU activation function. We then needed to add a **Flatten** layer which reshapes the data from a multidimensional array to a 1D array, which is typically done before feeding the data into a fully connected layer. We then added a **Dense** layer, which is a fully connected layer that connects each neuron to every other neuron in the previous and subsequent layers, with 256 units and the ReLU activation function. Next came a **Dropout** layer to the model with a dropout rate of 0.5, which randomly sets 50% of the input units to 0 during training to prevent overfitting. After this layer, we added another fully connected **Dense** layer with 32 units and the ReLU activation function. We followed this with another **Dropout** layer with a dropout rate of 0.5. After the additional dropout layer, we added another **Dense** layer with 2 units, representing the two classes in the classification task, which uses the softmax activation function to produce probabilities for each class. Finally, we compiled the model by setting the loss function to “sparse_categorical_crossentropy” which is appropriate for multi-class classification tasks with integer labels, using the Adam optimizer for optimization and measuring the performance of our model during training using accuracy as our evaluation metric.

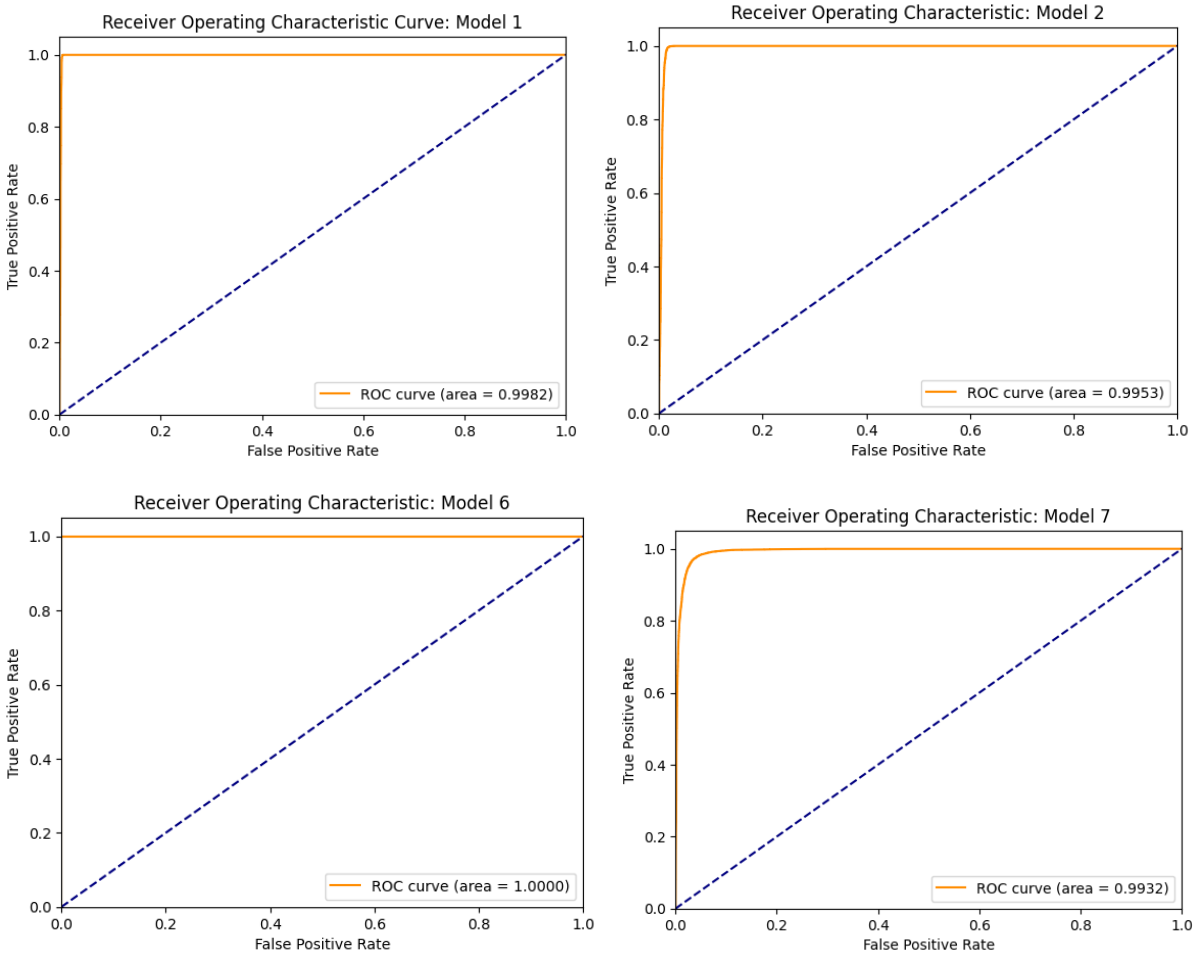
Training and Testing Procedures

For each of the models, we trained the model using X_train and y_train as the training set and X_valid and y_valid as the testing set. The “batch_size” parameter, which determines the number of samples to use in each update of the model weights during training, was set to 128. The “epochs” parameter, which specifies the number of times to iterate over the entire training set, was set to 100. Two callbacks were used during the model training: EarlyStopping

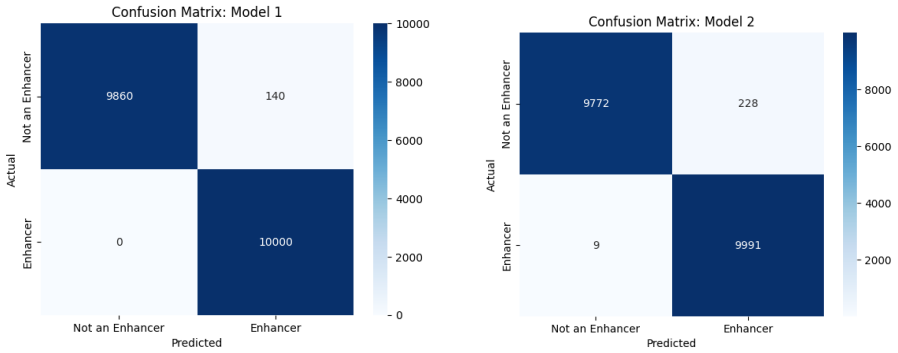
and History. The EarlyStopping callback was used to monitor the validation loss (set by the “monitor” parameter) during training and stop the training process early if the validation does not improve for 10 consecutive epochs, as indicated by the “patience” parameter. Finally, the “restore_best_weights” parameter is set to True, which restores the weights of the best model observed during training, as determined by validation loss, before training ends. This helps to prevent overfitting and ensures that the model with the best performance on the validation set is used. The History callback is used to record various training metrics, such as loss and accuracy, during training for later visualization and analysis.

Testing our model is relatively straightforward, as the predict() method is called on the trained model object to generate predictions for X_test, with a batch size set to 128. The pred variable contains the predicted probabilities for each class for each sample in the test data. Finally, the count of correct predictions (given by pred.argmax == Y_test) is divided by the length of Y_test to calculate the accuracy of each model.

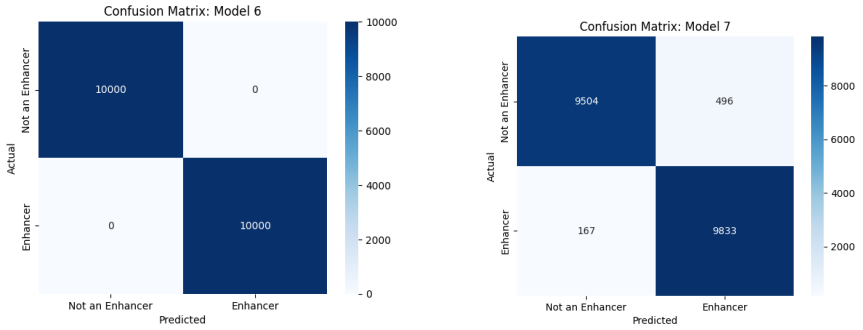
Results



Model	Maximum Validation Accuracy	Test Accuracy	AUC-ROC
Model 1	0.9927	0.993	0.9982
Model 2	0.9888	0.98815	0.9953
Model 6	0.9999	1.000	1.0000
Model 7	0.9973	0.96685	0.9932



CM Model 1: Only 140 false positives identifying enhancers
CM Model 2: Only 228 false positives identifying enhancers, 9 false negatives



CM Model 3: All labels correct in testing
CM Model 4: 496 false positives, only 167 false negatives

The confusion matrix results above showcase high-performing models for all four train/test attempts as detailed above. Model 3 specifically highlights perfect performance, with 100% of test labels being correct. Model 4 is arguably the “worst” although most labels are correct, the most false positives/negatives in labeling were experienced in this model relative to the other three. Frankly, though, as shown in the accuracy table results and in the clear color indications of true positive and true negative labeling in the confusion matrices, all models performed quite well.