Report on Exon Search with CNNs

Author: Kyrylo Stadniuk
Code can be found here.

Important Files

- `preprocess_data.py` - script for data preprocessing. It creates the dataset for all chromosomes for the given `window_size`.``
- `utils.py` - utility functions for working with data
- `train.py` - script for model training.
- `models.py` - file with model definitions

Data

I manually downloaded canonical chromosome annotations (`Homo_sapiens.GRCh38.113.chr.gtf.gz`) and chromosome sequences (`Homo_sapiens.GRCh38.dna.chromosome.*.fa.gz` for chromosomes 1–22, X, Y) from Ensembl, then unzipped them (automated in the provided bash script).

Dataset construction involved two stages — annotation parsing and sequence windowing — with examples grouped by gene and chromosome and saved to HDF5 for efficient loading. From the GTF, I extracted exon start positions, grouped them by gene_id, and split genes into disjoint 80/20 train/test sets (fixed seed), serialized with a pickle for reproducibility and zero gene overlap between splits.

For each window size (101 bp, 201 bp, 301 bp) and chromosome, I loaded the corresponding FASTA once, and used `preprocess_data()` to generate centered windows (one-hot encoded) around exon starts as positives and random intragenic positions as negatives. The resulting NumPy arrays — X_train, y_train, X_test, y_test — were saved to `train_test_data<chr>.h5`.

I first trained models on individual chromosomes. While valid (exon start patterns are similar), this was slow and may have introduced chromosome-specific noise. Switching to random chromosome batches sped things up significantly and improved accuracy from 0.745 to 0.76 with all else equal. (`class ChromosomeDataGenerator()` in `utils.py`)

Model Architectures

I have trained several models, so I will give them nicknames.

Simple model `build_simple_model()`

A simple CNN architecture consisting of two convolutional layers, max pooling, global max pooling and a dense classifier. It captures short motifs and then combines them via pooling. The lack of depth and attention layers have not allowed it to reach acceptable performance (it was about 0.69 on training data, if I am not mistaken).

Attention model `build_attention_model()`

Here I decided to try out the concept of attention. Multi-Head Attention allows model to focus on multiple parts of input sequence simultaneously. However, the lack of depth made this model to perform similary to the `Complex model`, which does not have any attention layers.

Complex Model `build_complex_model()`

This one is deeper than the previous one. It has three convolutional layers with increasing filter size and batch normalization after each to stabilize training. After extracting the features and downsampling them it summarizes them into a fixed-length representation. It ends with dense layer. Despite not using the attention it performed fairly well.

Enhanced model (the best) `enhanced_cnn()`

This one is based on the concept of Inception. The basic idea is that we do not commit to a single filter size, but apply two convolutional filters (3 and 5) and run them in parallel within the same layer. This allows us to capture both short and long motifs in the sequence. After each inception block, batch normalization is applied to stabilize and accelerate training, followed by max pooling to reduce dimensionality. The number of filters increases with depth to allow the network to learn more abstract features. Then I apply the attention gate to emphasize the most important positions in the sequence. The resulting vector is globally pooled and passed through dense classifier.

Additionally, in this model I decided to go with the Swish activation function instead of the standard ReLU used in other models. Swish is smoother and non-monotonic, which can lead to better performance in deep networks by preserving small negative activations and improving gradient flow.

An attention gate is used after the convolutional blocks to emphasize important sequence positions and suppress irrelevant ones. The attended feature maps are then aggregated using global average pooling, which is often more robust than max pooling — it reduces sensitivity to outliers by averaging across the entire sequence and tends to generalize better by smoothing over noise. The final dense layers perform classification based on this compact representation.

Training took 13 epochs for window of 201 and 15 epochs for window of 301. Optimal model was reached after 8th and 10th epoch respectively. Each epoch averaged out about 460 seconds (7 minutes 40 seconds) in the first case and 650 (10 minutes and 50 seconds) in the second one. In total: first - 99 minutes, second 162 minutes.

I also desided to try out Squeeze and excitation block instead of the attention block. It did improve the performance, but not very much. (this layer: `se_block()`)

Training `training.py`

To train the model, as decribed earlier, I implemented a chromosome-aware data loading strategy using custom `tf.keras.utils.Sequence` generators. The training data was stored in separate HDF5 files for each chromosome, and batches were randomly sampled across chromosomes during training. This stochastic sampling ensures exposure to a broad distribution of genomic contexts, improving generalization and preventing overfitting to any single chromosome. Each training batch was drawn from a randomly selected chromosome file, with its start index chosen at random to promote further variability.
For validation, a deterministic generator was used, loading entire test datasets per chromosome to provide consistent and complete evaluation across epochs.
`epochs` was set to 100, with early stopping triggered based on validation accuracy to avoid overfitting and save some time. Best performing model is then saved.

Window Sizes

In my experience, the choice of individual window size did not matter very much, the choice of model architecture was much more important. 101 and 201 seem to be optimal. Models with window 301 perform a bit worse. Although, I tested only window sizes of 101,201 and 301 and even the smallest one (101) might be just enough for our purposes.
Odd numbers were chosen because I chose to put the starts of exons in the center and that eased the work - the indices of exons starts were more intuitive (50, 100, 150).

Challenges

One of the main challenges was data preprocessing — even with 32GB of RAM on my working laptop, I had to carefully manage memory usage. Processing multiple large .h5 files and generating input windows from the full mouse genome led to several instances where the system nearly froze due to memory exhaustion. Another limitation was the absence of a GPU, which made training slower than it could have been. Nevertheless, the overall training time remained manageable, largely thanks to the use of early stopping to prevent unnecessary epochs.

Testing the best model on Mus Musculus

I tested my best model on 1-7 chromosomes of MusMusculus. Surprizingly, It has basically the same accuracy as in Humans - 0.7799. I would expect some difference, but probably we are indeed very similar. In other organisms (non-mammals) this number would probably be lower.

Discussion

Since the dataset was balanced with respect to classes, accuracy was used as the primary evaluation metric. This is reasonable given the 1:1 class ratio, but may not generalize perfectly to real-world genomic data where exon starts are relatively rare compared to non-coding regions.

Model	Window Size	Accuracy
Enhanced CNN	101	**0.7827**
Enhanced CNN	201	**0.7827**
Enhanced CNN	301	0.7584
Enhanced CNN SE	301	0.7620
Complex	201	0.7387
Attention	201	0.7121

The enhanced CNN architecture performed the best at smaller window sizes (101 and 201), achieving an accuracy of 0.7827. Interestingly, increasing the window size to 301 decreased performance slightly, possibly due to introducing redundant or noisy context that the model could not efficiently learn from. Adding a Squeeze-and-Excitation (SE) block modestly improved performance at window size 301 (from 0.7584 to 0.7620), suggesting that SE blocks in combination with smaller windows might be a promising direction to explore further.

What can be improved:

- **Model Architecture**: The current models are based on straightforward CNN and attention modules. More sophisticated architectures such as residual CNNs, dilated convolutions, or hybrid CNN-transformer models might capture long-range dependencies more effectively.
- **Hyperparameter Optimization**: Learning rate, number of filters, kernel sizes, and dropout rates were tuned manually. A more systematic search (e.g. Bayesian optimization) could yield further improvements.
- **Feature Engineering**: At present, only raw one-hot encoded sequences are used. Incorporating additional genomic signals such as conservation scores, GC content, or methylation data might improve the discriminative power.
- **SE Layer in 201-Window Model**: Given the performance gain from adding an SE block in the 301 model, applying the SE enhancement to the 201-window model might push the accuracy to 0.79 or even 0.80.