

III. Model documentation and write-up

1. I am a computational biologist and machine learning enthusiast.
2. The challenge was about DNA sequence analysis, which is my main area of expertise. However, I never heard about the attribution problem before. It was interesting to see how far I can push the current methods.
3. I made two kinds of models. The first one is based on k-mer counts, because I know from experience that simple k-mer approaches are powerful for DNA sequence matching and comparison. The second is a convolutional neural network with capacity as large as I could make. The two kinds of models complement each other's strengths well: k-mers are good at precise sequence matching, and work well with low number of samples. CNNs are good at fuzzy motif finding and integrating weak evidence from multiple sites in the sequence.
Among the 7 CNNs in the ensemble, 5 are trained directly to predict the lab index, and 2 use self-supervised pretraining of the main CNN layer. The self-supervised target is to predict whether two sequences come from the same lab, using NT-Xent loss.
4. This (C++) efficiently extracts all k-mer (reverse-complement normalized) hashes from a string. A hash code for k-mer and its reverse complement is computed, and the minimum of the two is taken. This calculation is very fast and allows processing sequences at nearly disk-reading speeds on multicore machines.

```
-----
----
template<class Func>
void for_all_kmers(const string& s, const vector<int>& ks, Func func) const {
    if(ks.front() > (int)s.size())
        return;

    for(int p = 0; p < (int)s.size() - ks.front() + 1; ++p) {
        for(int size : ks) {
            if(p + size > (int)s.size())
                break;

            hash_t h = substr_hash(s, p, size);
            if(h != 0) {
                func(h, p, size);
                if(p == 0 || p + size == (int)s.size())
                    func(h ^ hbegin, p, size);
            }
        }
    }
}

hash_t substr_hash(const string& s, int p, int size) const {
    hash_t hf = 0, hrc = 0;
    for(int np = 0; np < size; ++np) {
        char l = s[p + np];
        if(l == 'N' && drop_n) {
            hf = 0;
            break;
        }

        hf ^= letter_hashes[np][l];
        hrc ^= letter_hashes_complement[size - np - 1][l];
    }

    return min(hf, hrc);
}
-----
----
```

This is minibatch generation with data augmentations for CNNs. I use 5% base pair dropout and reverse complement. Random subsequence of 8000 or 16000 base pairs is taken. If the original sequence is shorter, it is zero-padded. N is 0. This setup was discovered by trial and error, and deviation from it lowers the score. In particular, the long (8000) subsequence length is important.

```
-----
----
def seq_minibatch(seqs, train_sequence_length, cuda_dev, drop_prob=None,
random_rc=False, mut_prob=None):
    mb = torch.zeros((len(seqs), 4, train_sequence_length), dtype=torch.float32,
device=cuda_dev)
    for i, s in enumerate(seqs):
        s = s.to(cuda_dev)

        if mut_prob is not None:
            s = s.clone()
            to_mutate = int(len(s) * mut_prob)
            s[np.random.choice(len(s), to_mutate)] =
torch.LongTensor(np.random.randint(4, size=to_mutate))

        if len(s) <= train_sequence_length:
            content = torch.nn.functional.one_hot(s, 5)[: , :4].transpose(0, 1)
            mb[i, :, :len(s)] = content
        else:
            offset = np.random.randint(0, len(s) - train_sequence_length + 1)
            mb[i, ...] = torch.nn.functional.one_hot(s[offset:offset +
train_sequence_length], 5)[: , :4].transpose(0, 1)

        if drop_prob is not None:
            mb[i, :, np.random.choice(train_sequence_length,
int(train_sequence_length * drop_prob))] = 0

        if random_rc and np.random.randint(2) == 0:
            mb[i, ...] = torch.flip(mb[i, ...], [0, 1])

    return mb
-----
----
```

A basic CNN model configuration. 18 is the kernel width. It concatenates the sequence representation after max- and average-pooling with the processed binary features. Then a 2-layer MLP makes the final decision. Binary features give a minor boost in accuracy, but they are needed to be competitive.

```
-----
----
class JoinNet(torch.nn.Module):
    def __init__(self, net1, net2, head):
        super().__init__()
        self.net1 = net1
        self.net2 = net2
        self.head = head

    def forward(self, x1, x2):
        x1 = self.net1(x1)
        x2 = self.net2(x2)
        joined = torch.cat((x1, x2), axis=1)
        return self.head(joined)
```

```

def make_model1(num_aug_features):
    hidden_dim = 2000
    width = 2000
    seq_drop_prob = 0.05
    train_sequence_length = 8000
    weight_decay = 0.0001

    model_bin = MLP([num_aug_features, hidden_dim, hidden_dim, num_labs],
dropout=0.5)
    bin_headless = model_bin.mlp[:7]

    model_seq = SimpleCNN(18, hidden_dim, additional_layer=False)

    new_head = torch.nn.Sequential(
        torch.nn.Dropout(0.5),
        MLP([width * 2 + hidden_dim, num_labs])
    )

    joined_model = JoinNet(model_seq, bin_headless, new_head)

```

5. Specs.

CPU: AMD Ryzen 7 1700

GPU: 2x NVIDIA Titan RTX

Memory: 64 Gb

OS: Debian Linux

Train duration: ~40 hours for all models.

Inference duration: ~10 minutes.

6. I tried RNNs, but they were much too resource-intensive for the sequence lengths that I needed (~8000bp).

I spent much time trying to get a pseudolabeling pipeline to work. After it worked, the public leaderboard results were not great, and the approach was abandoned.

Later it turned out that it worked quite well on the private set. Still, the direct training gets almost the same score with much less resources and simpler pipeline, so in the end it's preferable.

Tried deeper ResNets, but the accuracy plateaued quickly. The final models have at most 2 CNN layers.

7. All main coding was done in Jupyter lab. Before submission, I transferred everything into .py files for easy packaging.

8. k-mer model can do leave-one-out cross-validation, which I used to tweak it. For CNNs, I only evaluated them on a single train-test split. Fitting multiple folds for cross-validation would take too much time. In addition, I looked at predicted lab frequency distributions on test set and compared it with train. There was a notable difference: lab 1008, the second most frequent in train, was nearly absent in test. I probed the public board to confirm that this was indeed the case, and not a problem with the model. Later this provided additional validation, showing that the model doesn't simply repeat the training distribution.

9. The CNNs are tailored to Titan RTX's 24 Gb memory. If a larger amount of GPU memory is available, it will likely be beneficial to increase the batch size and/or the sequence size. It can also be scaled back to 16 Gb, if desired, by reducing the sequence/batch sizes, with some loss of accuracy.

The CNNs don't use early stopping, but follow a fixed learning rate schedule with 100 epochs instead. If you wish to train them more, or less, this has to be

adjusted accordingly.

10. Not much. I've attached a picture showing the lab 1008 discrepancy mentioned above.

11. This year's hit would be transformers, of course. It would be interesting to see if I can make a large transformer to integrate data from different pieces of the sequences better than the linear CNN's way. However, the dataset may not be large enough for this.

A very promising direction will be to add a negative set of truly wildtype sequences, so that the model can learn to focus better on artificial-looking parts. A related thing to explore is a potential bias in the present models: some labs tend to work on the same set of proteins, so a certain protein sequence may end up being associated with some labs. But in reality, this is simply a historic feature. Any lab may introduce any protein in the future. So, focusing on proteins themselves is not productive; rather, the models should be looking at non-coding sequences (promoters, barcodes, etc.) and subtle features of proteins, like usage of synonymous codons.