DeepSudoKu: inferencing combinatorial pooling for whole-genome knockout collection through self-supervised learning

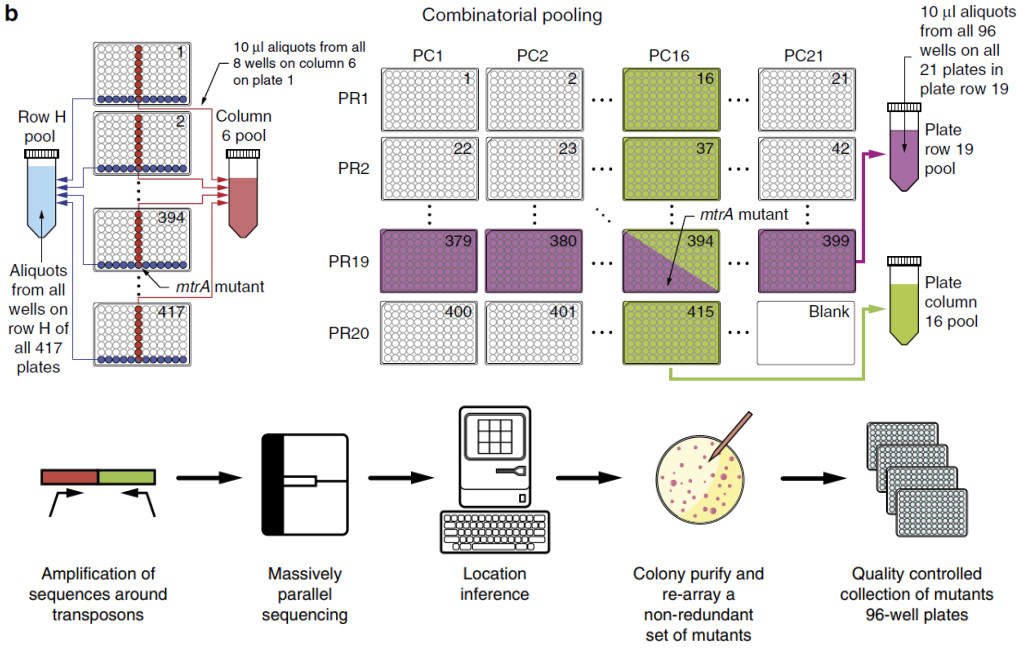
Canal Li

0. Abstract

A whole-genome knockout collection is a powerful genetic tool that enables gene functional discovery that includes a single knock-out mutant for each gene in the genome. But not until recently, creating such a collection has been an enormous financial and technical undertaking. To solve this problem, the Barstow lab invented Knockout Sudoku, a technology that drastically decreases the cost and human labor for the knockout collection construction. Here we propose KOsudoku2, an upgraded inference algorithm for Knockout Sudoku that uses deep learning to decide the locations of mutants in the collection. We train a deep neural network with unambitious mutant-well mappings, and it achieves 93% accuracy on hold-out mappings. The network is then used to infer the ambitious mappings. Compared with the original inference algorithm, this method gives better predictions.

1. Background

Here is a short recap of the original Knockout Sudoku method. The conventional way to create a collection would be to design a specific primer for each of the thousands of genes in the genome, then try to knock out each gene through traditional cloning methods. However, an enormous financial cost would be required to even manufacture all the primers in thousands.



We focus our attention on the *Shewanella oneidensis* knockout collection, which is published with the

original Knockout Sudoku method. Knockout Sudoku functions by using transposons to randomly knock out genes and therefore is able to produce tens of thousands of single knockout mutants simultaneously, a number sufficiently large to cover essentially all non-essential mutants. However, the identity of each mutant is still unclear, while it is extremely costly to sequence each mutant one by one, sequencing batches of them would be inexpensive. After the mutants are picked to place into hundreds of 96-well plates, the mutants are pooled in specific manners: for example, all the rows in all the 96-well plates go into one pool (see figure above). This forms a collection of less than a hundred pools and is much less expensive to sequence with Next-Generation Sequencing. With the NGS results, then we can decide the whereabouts of each of the mutants by overlapping the pools corresponding to the well location. This is where the inference algorithm comes into play. For example, the mutant inside well A3 in plate 1 would appear in four pools: Row A pool, Column 3 pool, PC1 pool, and PR1 pool. All these four pools should share one gene coordinate, and that is the identity of the mutant inside well A3, plate 1. Unfortunately, because we are sequencing thousands of mutants in one pool, we would have many ambitious

b Pool presence table compilation

Line No.	Genomic coordinate	Feature	Row pool reads	Col. pool reads	Plate-row pool reads	Plate-col. pool reads	Address inference method	Most likely address(es)
1	1858875	*mtrA*	H: 24	6: 60	19: 93	16: 124	Direct	H_6_PR19_PC16 (H6, plate 394)
2	2190757	*hypF*	A: 1, B: 1, D: 122	2: 1, 6: 1, 8: 107	PR04: 1, PR05: 1, PR08: 691	PC06: 448, PC07: 2, PC08: 1	Direct	D_8_PR08_PC06 (D8, plate 154)
3	2931139	*SO_2808*	E: 55, G: 19, H: 1	2: 135, 3: 1, 4: 1, 12: 23	PR13: 22, PR14: 523, PR15: 1, PR18: 1, PR20: 1	PC13: 2, PC15: 27, PC19: 1, PC20: 1, PC21: 933	Probabilistic	E_2_PR14_PC21 (E2, plate 294), G_12_PR13_PC15 (G12, plate 267)

mutant-well mappings (line No.3).

The original KOsudoku algorithm solves this problem by first fitting the read counts of possible locations to the fitted Voigt distribution, then selecting the most probable locations to distribute the mutant to these locations. However, because the Voigt distribution is only a gross proxy for the true read counts distribution, the old method predicts ~10000 wells out of a total of 40032 wells to reside more than one mutant, when in fact almost all of them only have one mutant. KOsudoku2 aims to solve this problem by utilizing all the available information with a deep neural network, training it with known-for-certain mutant-well mappings, and finally predicting the ambiguous mappings with the learned network.

2. Motivations behind KOsudoku2

The original KOsudoku algorithm gives counter-intuitive results: the algorithm inferences that about 25% of all wells have more than one mutant, and about 20% of wells do not have any mutant. Whereas in reality, scanned plate images show only ~5% of wells are empty. And base on the sequencing verification of the previous research, the predicted to be multiply-occupied wells only give back 1 transposon coordinate and therefore have one mutant.

```
Sudoku Taxonomy Post-Grouping
totalMutants: 31835
mutantsInSinglyOccupiedWells: 19064
mutantsInMultiplyOccupiedWells: 20093
emptyWells: 7870
singlyOccupiedWells: 20548
multiplyOccupiedWells: 11902
```

3. Materials and Methods

Here are the four steps in the KOsudoku2 algorithm:

1. *Generate <knockout_collection>, mark scanned empty wells inaccessible*
2. *Fill in wells with unambiguous transposon coordinates*
3. *Fill in wells with first-choice ambiguous transposon (determined by **Voigt score**)*
4. *Fill in unfilled wells through a trained deep-learning algorithm*

Step 1: Mark empty wells inaccessible.

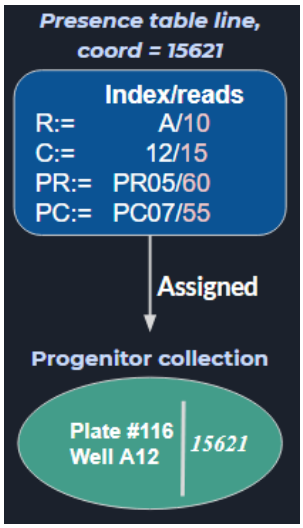
We used OpenCV to distinguish between occupied and unoccupied wells, labeling each well with its occupancy using the barcode provided in the original image. By doing so, the downstream inferencing procedure can rule out these well locations, making more reasonable predictions. After scanning all the plates, we found that only 2448/40032 wells are actually empty, in contrast to the 7870 wells predicted by KOsudoku.

```
import numpy as np
occupancy = np.load('./voigt_data/occupancy.npy')  #(417, 8, 12)
occupancy.shape[0]*occupancy.shape[1]*occupancy.shape[2] - np.count_nonzero(occupancy)
```

```
2448
```

Step 2: Assign unambiguous coordinates.

This step is almost identical to the original algorithm since it assigns the known-for-certain mutant-well mappings. In the original algorithm, if there are multiple reads on only one axis, the mutant with the transposon coordinate is still unambiguous and can be mapped to multiple locations. For example, for transposon coordinate 52231, if there are reads on both the PR05 pool and the PR22 pool, it is assigned to the two locations corresponding to PR05 and PR22. However, this is a physically very unlikely event. In KOsudoku2, we fix this by setting a thresholding parameter ratio_threshold. ratio_threshold is applied to the largest read so that only reads with read counts greater than largest_read/ratio_threshold can be considered also unambitious. This removes unlikely events while still keeping rare assignments. After this step, 25173/40032 wells are filled, with 23837/40032 being singly occupied and 1336/40032 being multiplely occupied.



Step 3: Assign first-choice ambiguous coordinates.

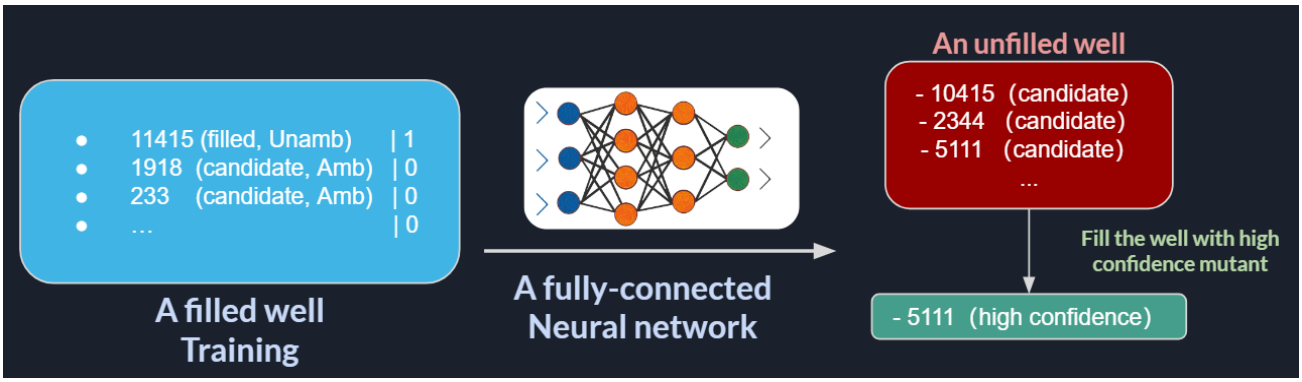
There could be an exponential number of possible addresses for a mutant with multiple reads in each axis. For example, below, the mutant with transposon coordinate 2931139 can be assigned to 2*2*2*2=8 locations if we ignore reads less than 5. However, we only assign the most likely one for each ambiguous coordinate and assume it to be correct (since empirical results agree with this assumption). Of course, since we are picking much more mutants than the actual number of non-essential genes, it is impossible that only one well has mutant 2931139. We resolve the other possible wells also containing mutant 2931139 at the later step.

3	2931139	*SO_2808*	E: 55, G: 19, H: 1	2: 135, 3: 1, 4: 1, 12: 23	PR13: 22, PR14: 523, PR15: 1, PR18: 1, PR20: 1	PC13: 2, PC15: 27, PC19: 1, PC20: 1, PC21: 933	Probabilistic	E_2_PR14_PC21 (E2, plate 294), G_12_PR13_PC15 (G12, plate 267)

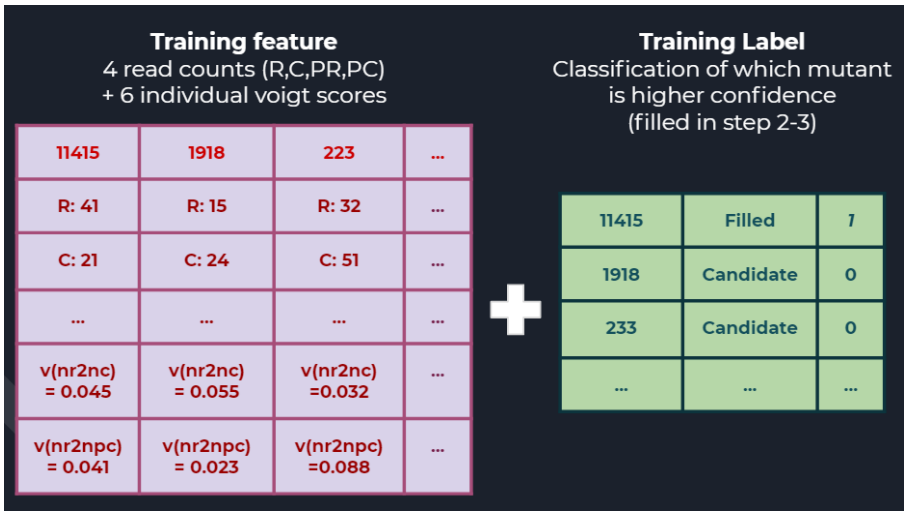
In order to measure what is the likelihood of a well containing the mutant of interest, the original method uses the Voigt score. Before calculating the score for an ambiguous mutant, 6 Voigt distributions are fitted for the 6 read count ratios: #RowTo#Column, #RowTo#PlateColumn,..., #PlateRowTo#PlateColumn. The ratios from the unambiguous mappings in step 2 are used to fit the distributions. Then the Voigt Score is obtained by multiplying the probability densities of an ambiguous mapping's ratios. Empirically, mapping with the highest Voigt score is almost always correct, while the 2nd and 3rd lead to incorrect mappings (the source of error in the original algorithm). To re-emphasize, KOsudoku2 only assigns the mapping with the highest Voigt score.

Step 4: Fill in unfilled wells with deep learning

From steps 2 and 3, we filled and labeled transposon coordinates in high-confidence addresses (unambiguous and ambiguous mappings with the highest Voigt score). In step 3, we did not assign 2nd, 3rd… addresses for ambiguous transposons, instead, we labeled them as low-confidence. We then used these higher/lower confidence pairs to train a neural network for distinguishing between higher/lower confidence pairs in a well. For all the wells left unfilled from Step 1~3, we then use the network to pick the mutant with the highest confidence and fill it in.



For each mapping, there are ten features: the four individual read counts, and the six individual probability densities from the six fitted Voigt distributions, which gives a feature vector of dimension 10. Each higher/lower confidence pairs are concatenated together randomly 15 times to give a combined feature vector of dimension 150, with the higher confidence mapping's place in the 15 mappings labeled as the correct label for classification.



We used a fully connected neural network with BatchNorm layers and ReLu activation to determine the best match between leftover empty wells and all possible coordinates. The network is implemented in PyTorch, with the network architecture shown as below:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 128)               19456
_________________________________________________________________
batch_normalization (BatchNo (None, 128)               512
_________________________________________________________________
activation (Activation)      (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
batch_normalization_1 (Batch (None, 128)               512
_________________________________________________________________
activation_1 (Activation)    (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
activation_2 (Activation)    (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 15)                1935
=================================================================
Total params: 55,951
Trainable params: 55,183
Non-trainable params: 768
_________________________________________________________________
```

4. Results

We used Adam as our optimizing method and set the batch size for training to be 256. We perturbed the ordering of higher/lower confidence pairs 30 times for data augmentation, and use 80% of all pairs for training and 20% for validation. For all the network architectures that we experimented with, most converge to about 92% validation accuracy after 100 passes through the entire training set, and the best network converges to 93%.

We kept the best-performing predictions from the original algorithm (unambiguous and 1st choice ambiguous). We reduced redundant unambiguous and ambiguous mappings (2nd, 3rd … choice ones),

significantly reducing the number of multiplely occupied wells. A large number of wells that the original algorithm believes to be empty are now filled, coherent with what the photos of the plates suggest. Read counts and spatial information are taken into account, leading to more informed mapping decisions. See below for a straight comparison of results from both algorithms.

5. Discussion

Although from the verification sequencing in the original, our method predicts almost as well as the original algorithm in the ones that the original algorithm correctly predicted, because the actual transposon coordinates of the wells that got predicted wrong are not available, we can not compare directly whether our method is actually better. Therefore we plan to sequence the following number of wells:

About 96 wells for the following cases:

1. ***[Old: multiply-occupied, New: multiply-occupied]*** The original sudoku algorithm makes a large number of predictions with over-populated wells, which are verified to be not the case. Our new algorithm gives much fewer multiply-occupied wells, and we would like to confirm our predictions of such wells are of higher accuracy.
2. ***[Old: multiply-occupied, New: singly-occupied]*** The original sudoku algorithm makes a large number of predictions with over-populated wells, which are verified to be not the case. Our new algorithm predicts a lot of the original overpopulated wells to be singly occupied. We aim to verify that these wells are indeed singly occupied.

About 192 wells for the following cases:

3. ***[Old: empty, New: occupied]*** The original sudoku algorithm left out a large number of empty wells, which by observation are occupied. Our new algorithm is able to predict such wells.
4. ***[Old: ambiguous, New: ambiguous]*** The original sudoku algorithm makes ambiguous predictions on some wells, and based on the design of the new algorithm, these wells will probably be predicted ambiguous by the new algorithm. We want to show that the new algorithm achieves similar or better results.

6. Conclusion

This method is the first machine learning-based combinatorial pooling technique for knockout collections. Knock Sudoku was and still is the state-of-the-art method for combinatorial pooling, and has already been helpful in discovering new gene functions in our lab. However, the large number of multiplely occupied wells are the artifact of the original KOsudoku inference algorithm. KOsudoku2 makes no modifications to the wet lab procedure, keeps the best of Knock Sudoku, and recovers more mutants. If the sequencing results come back promising, we are in a decent position to publish one solid paper.