Cellular Automata with Neural Networks

Robbie Morris

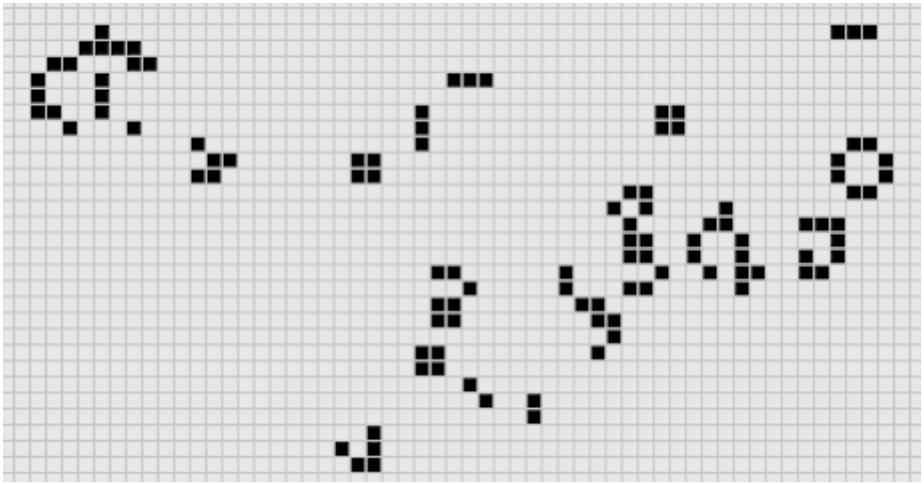
Follow along in the attached Google Colaboratory document.

1. Introduction

Simulating the real world is important, but impossibly difficult. There are often too many complex variables or unknowns to create accurate simulations, and as such sometimes it is better to embrace simplicity rather than attempt to replicate these complexities. Cellular Automata are one such method that primarily biologists have used to do so. Cellular Automata is essentially a set of rules through which a simple binary world operates. Each cell is either alive (in binary, 1) or dead (0), and depending on the surrounding cells, these states change over time.

Recently, researchers have outlined how cellular automata combined with neural networks could create self repairing images. Cellular automata have lots of potential to be useful in many ways which we have likely not discovered yet, and this article inspired me to approach cellular automata using machine learning from a different angle. In this paper, we will be using a machine learning model to learn rulesets of cellular automata. This provides us with an opportunity to observe and potentially draw conclusions on how computers can generate life on their own. If a computer can learn to run a virtual world entirely on its own, I think it reflects an interesting sentiment about what we might see in the future (only if Moore's law never fails).

In this project, I hope to create a model which, starting from scratch, can recreate Conway's game of life. A grid in Conway's game of life looks something like this.



The rules of the simulation are as follows: If a cell is alive, then it stays alive if it has either 2 or 3 live neighbors. If the cell is dead, then it springs to life only in the case that it has 3 live neighbors. There are many other simulations with slightly different rulesets, and I will be using a variation of the game of life with some small differences.

2. Methods

For this project, I will be using the Tensorflow/keras machine learning library, numpy, and matplotlib for visualization. My imports are pretty standard and don't require much explanation.

```
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

2i. **Implementing Cellular Automata**

Our first step is to implement the cellular automata without any machine learning. To do this, we first define 3x3 filters for each cell. The following code contains definitions of the filters which would create something similar to Conway's game of life.

```
radius = 3
pad = 1

neighbor_filter = np.ones((radius, radius))
neighbor_filter[1,1] = 0
middle_filter = np.zeros((radius, radius))
middle_filter[1,1] = 1
all_filters = np.dstack((middle_filter, neighbor_filter, neighbor_filter, neighbor_filter, neighbor_filter))
all_biases = np.array([0,  -1, -2, -3, -4])
total_filters = len(all_biases)
```

One step in cellular automata can be thought of as similar to one convolution of a neural network. The program looks at the 3x3 grid of cells surrounding a certain cell, and based on the rules of the automaton it assigns a new value to that cell. We can take a neural network approach to implementing the game, and use a singular convolution to calculate one step of the whole grid.

```
#cast to tensor
kernel = tf.cast(tf.convert_to_tensor(all_filters), tf.float32)[:,:,tf.newaxis,:]
biases = tf.cast(tf.convert_to_tensor(all_biases), tf.float32)
```

After casting our kernel (synonym for filter) and biases to tensor, I wrote a function which convolves one time using some starting weights and biases I found on the internet.

```
initial_weights = np.array([[0, 0, 4/3, -8/3, -1/3], [3/2, 5/4, -5, -1/4, -1/4]]).T
initial_biases = np.array([-1/3,-7/4]).T

def true_ca(input_image):

    conv_image = tf.nn.conv2d(input_image[...,tf.newaxis], kernel, strides=[1,1,1,1], padding='SAME')

    activation_image = tf.nn.relu(conv_image + biases)
    activated_flat = tf.reshape(activation_image, [-1, total_filters])

    heuristic = tf.nn.relu(tf.matmul(activated_flat, initial_weights) + initial_biases)

    scores = tf.reduce_sum(heuristic, axis = -1)
    next_states = tf.reshape(scores, [*activation_image.shape[:3],1])

    return tf.squeeze(next_states)
```

Notice that we convolve the image, put it through the ReLu activation function, flatten it, and then reshape it to apply it back to change the original image into the next state.

This function works quite well to generate our data, as it takes one snapshot of the cellular automata and returns whatever the next iteration is. This is how we will train our model.

2ii. **Data**

Using the cellular automata function, we can generate as much data as we need. Data should be in the format of a 4th level tensor, with dimensions [dataset size, width, height, 1 (indicating grayscale)]. The following code creates one hundred 10 by 10 grids, and sets a random initial state (1 or 0) to each cell.

```
#### Make training data
train_size = 100
width = 10
height = 10

X_train = tf.convert_to_tensor(np.random.choice([0,1], (train_size, width, height), p=[.5,.5]), tf.float32)

Y_train = true_ca(tf.convert_to_tensor(X_train, tf.float32))

X_train = X_train[..., tf.newaxis]
Y_train = Y_train[..., tf.newaxis]
```

This array of random grids is converted to a tensor and assigned to X_train, and then passed through our true_ca function which gives us our expected value (one iteration forward in the automata).

Note that we don't need a train test split, because I can just create a whole new dataset which has no relation to the training set for the testing.

2ii. **Creating the Model**

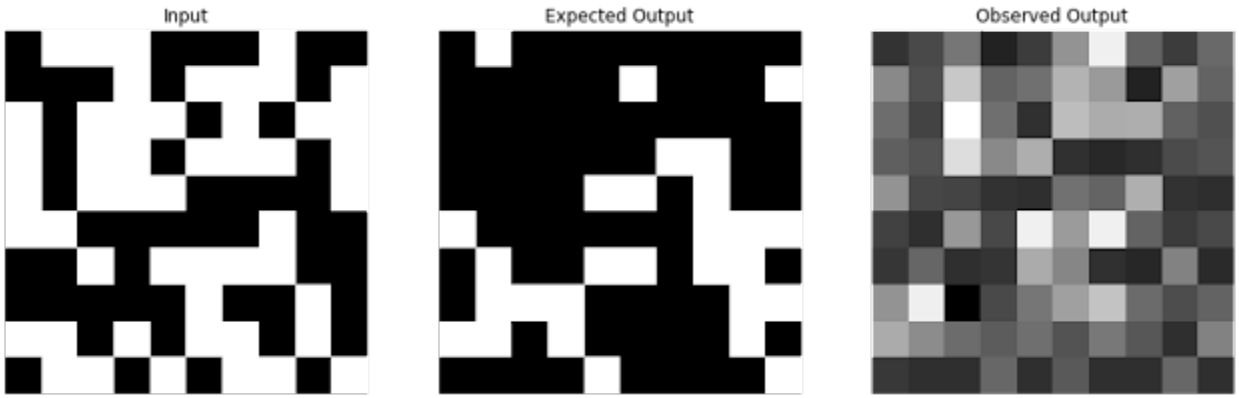
The general goal is to get the model to recognize which color each cell in the grid should be. This means that our model's decision should be binary. Hence, I will be using binary cross entropy as my loss function, and Rectified Linear Unit (ReLu) for my activation function.

```
#define loss function (categorical crossentropy)
loss_fn = tf.keras.losses.CategoricalCrossentropy(from_logits=True)

model.compile(optimizer=tf.keras.optimizers.Adam(lr=1e-2), loss=loss_fn, metrics = ['accuracy'])
```

Below is an example model (number 1). Between different model architectures, the input and output layers always stay the same. The input layer takes a tensor containing a black and white 10x10 image, and the output is always a [None, 10, 10, 2] tensor, which matches up with the output from my true_ca function. There are also always 10 filters and a filter size of 3, because these are properties of a cellular automata (vision only one cell away, and each cell needs to be checked every iteration).

```
num_filters = 10
filter_size = 3

try:
    del model
except:
    pass

#input dimensions 10x10, and black and white (1)
input = tf.keras.Input(shape = (10, 10, 1))
#10 filters, size 3x3
output = tf.keras.layers.Conv2D(filters=num_filters, kernel_size=filter_size, padding='same')(input)
output = tf.keras.layers.Dense(10, activation='relu')(output)

output = tf.keras.layers.Dense(10, activation='relu')(output)
output = tf.keras.layers.Dense(2)(output)

model = tf.keras.Model(input, output)
```

Here is an example of an untrained model. The output is quite staticy, and shows no understanding of the ruleset.



We can evaluate this model by creating another dataset the same way as before. Note that we manipulate Y_test and remove one axis to match X_test, because we do not care about which iteration of the game we are in, and keeping that information creates errors.

```
test_size = 100
X_test = tf.convert_to_tensor(np.random.choice([0,1], (test_size, width, height), p=[.5,.5]), tf.float32)

Y_test = true_ca(tf.convert_to_tensor(X_test, tf.float32))

X_test = X_test[..., tf.newaxis]
# X_test = tf.repeat(X_test, 2, 3)
Y_test = Y_test[..., tf.newaxis]
Y_test_output = tf.repeat(Y_test, 2, 3)
```

You can see from both the loss and accuracy that this model sucks.

```
eval = model.evaluate(X_test, Y_test_output)
print(eval)
```

```
4/4 [==============================] - 0s 5ms/step - loss: 0.4704 - accuracy: 0.3124
[0.4704433083534241, 0.3124000132083893]
```

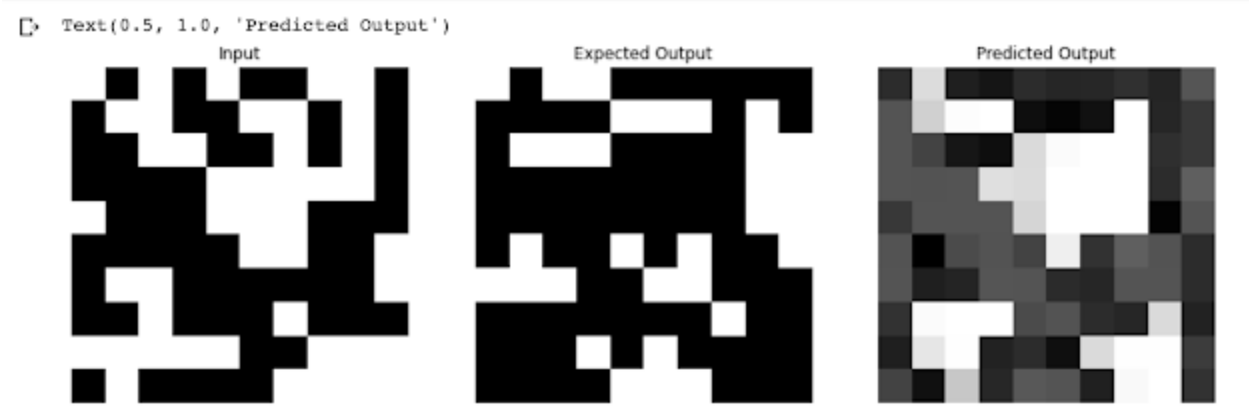
Our goal is to see some sort of correlation in the expected output area. It is unlikely that the model will be able to predict every cell with exact certainty, and so my expected outputs will all look quite staticy. If we wanted, we could simply convert values above 0.5 to fully black and vice versa, however I've left it like this to maintain more information.

2iii. **Training Different Model Architectures**

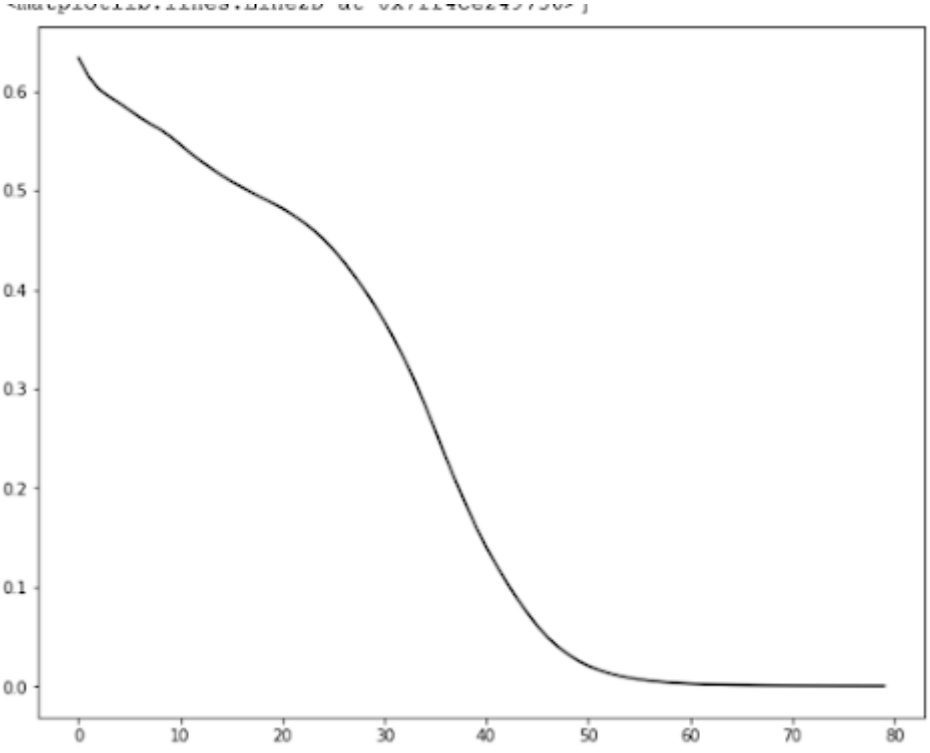
I read somewhere that if you don't know what type of model architecture should work for your situation, start small and work your way up. So I started first with a singular convolutional layer, and a couple of dense layers to follow. The learning rate for this model is 1e-2, the batch size is 10, and there are 80 epochs. These are just base values which I had real intention.

```
Model: "model_70"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_73 (InputLayer)       [(None, 10, 10, 1)]       0

 conv2d_94 (Conv2D)          (None, 10, 10, 10)        100

 dense_200 (Dense)           (None, 10, 10, 10)        110

 dense_201 (Dense)           (None, 10, 10, 10)        110

 dense_202 (Dense)           (None, 10, 10, 2)         22

=================================================================
Total params: 342
Trainable params: 342
Non-trainable params: 0
_________________________________________________________________
```

This model learned to somewhat effectively replicate the input image, but had no concept of the changes that were supposed to be made to the cells.



The loss function isn't exactly exponential, and I find it likely that the model was overfitting at the end. At this point, I was also using padding in my true_ca function, which meant that I was attempting (foolishly) to remove the bias that the model will have on the edges of the model.



The accuracy of this model was 0.7332. Better than guessing randomly, but not much. This is probably because of how the game of life works–the cells which were previously alive are more likely to be alive in the succeeding iteration that the cells which were not.

The second model which I trained, I added another convolutional layer in between the two dense layers. The learning rate is 1e-3, the batch size is 10, and there are 100 epochs

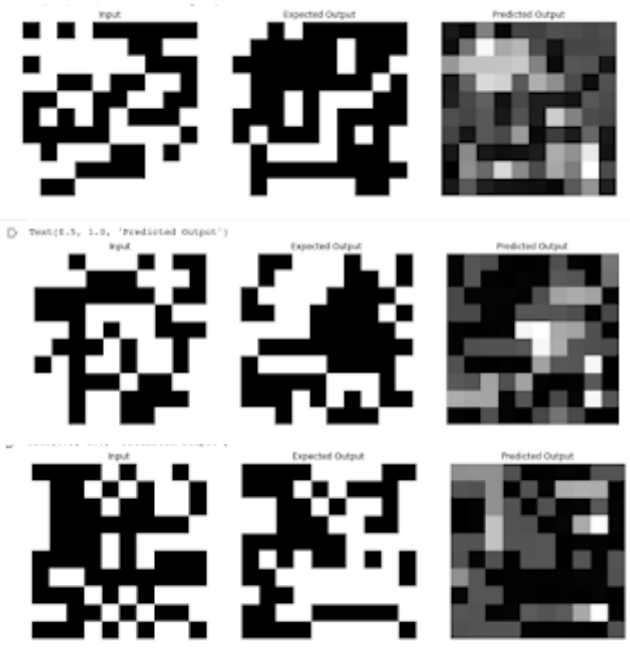
Layer (type)	Output Shape	Param #
input_78 (InputLayer)	[(None, 10, 10, 1)]	0
conv2d_102 (Conv2D)	(None, 10, 10, 10)	100
dense_214 (Dense)	(None, 10, 10, 10)	110
conv2d_103 (Conv2D)	(None, 10, 10, 10)	910
dense_215 (Dense)	(None, 10, 10, 2)	22

Total params: 1,142
Trainable params: 1,142
Non-trainable params: 0

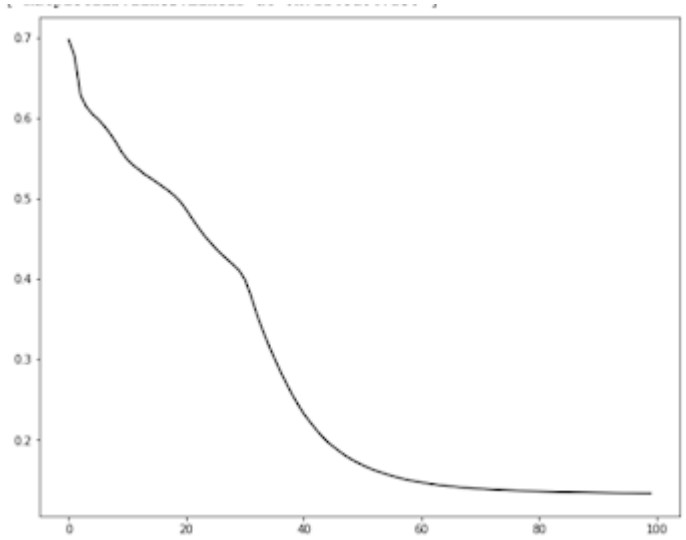
At first glance, this model looks terrible.

```
4/4 [==============================] - 0s 5ms/step - loss: 1.9960 - accuracy: 0.6579
[1.9959723949432373, 0.6578999757766724]
```

After looking at the visualizations though, the model actually seems to have trained effectively. As you can see, it accurately predicts the shapes of the expected output each time, but for some reason inverts every cell. After simply inverting the 1s and 0s, we get what seems to be an accurate model.



Here is the loss history. From this graph, it looks like the model looks like it should perform quite well. It's clear that this variation of the model had some programming kinks which I needed to fix, but it showed lots of promise.



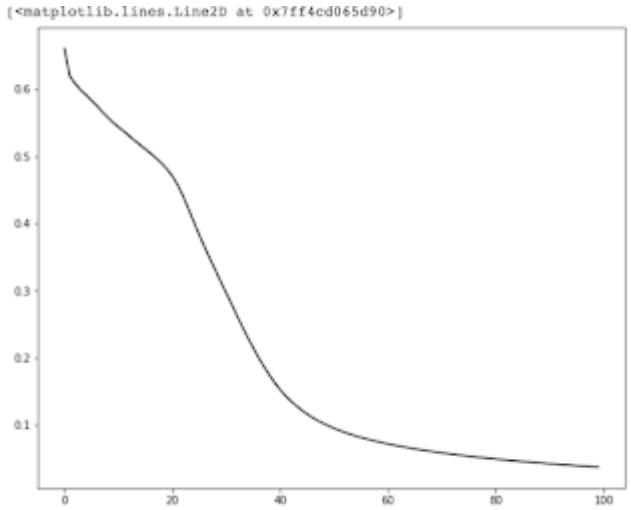
The next iteration of my model, I made a few small structural changes and did a lot of bug-fixing. I added another dense layer after the second convolutional layer, and I changed the number of filters in the second layer to 5 (making it more general).

Layer (type)	Output Shape	Param #
input_89 (InputLayer)	[(None, 10, 10, 1)]	0
conv2d_124 (Conv2D)	(None, 10, 10, 10)	100
dense_236 (Dense)	(None, 10, 10, 10)	110
conv2d_125 (Conv2D)	(None, 10, 10, 5)	455
dense_237 (Dense)	(None, 10, 10, 10)	60
dense_238 (Dense)	(None, 10, 10, 2)	22

Total params: 747
Trainable params: 747
Non-trainable params: 0

The model was trained with 100 epochs, a batch size of 10, and a learning rate of 1e-3. As you can see, after re-running the analysis code, we now no longer have inverted outputs.

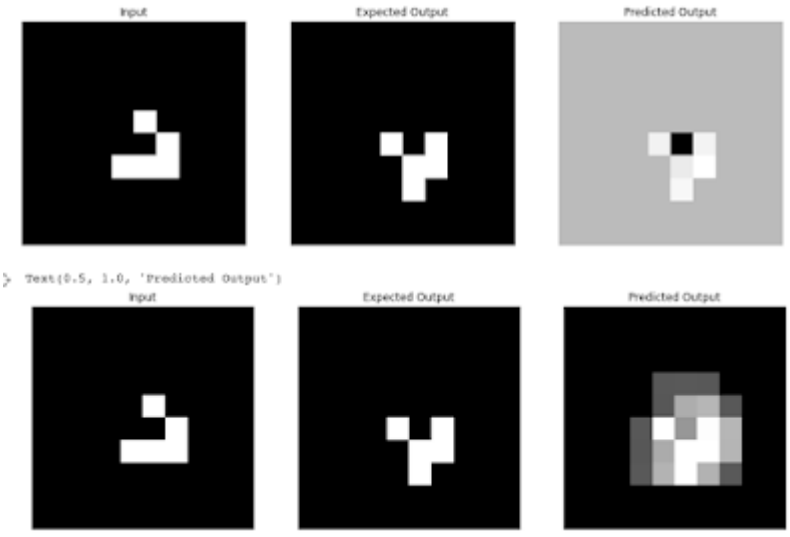
You can see that because I have no padding, the model is still a bit unsure about the darker areas near the edges. As you get closer to the middle, typically the true predictions are more confident (darker shades). Padding is something I could add if I ever got time to work more on this model, and will remain a limit to how well it can perform until I fix it. This is because the cells on the very outside don't understand that they are on the edge, and see a lot more dead cells surrounding them than actually exist.



For this model, our loss is approximately 0.02, and our accuracy is 0.9545. As you can see from the training history, our inverted loss approaches a very small number, but doesn't seem to be overfitting like previous models. If it was overfitting, the change in the loss would be close to zero for an extended period of time

Conclusions

I also considered applying the model to different famous configurations in cellular automata. I've recreated (with help from the internet) one of the most famous Game of Life configurations, the glider. As you can see, a few different versions of the final model can recreate the glider's second iteration pretty closely.



I found this to be a very cool demonstration of the working model and the differences between the variations. Some will be confident about the dark space, but not super precise, while the others will be very precise but unsure about other things which should be obvious. While it struggles with extremely complex configurations like I was training it on, it can effectively get through simpler ones. This leads me to believe that maybe I used training data which was

unnecessarily random, overcomplicating the training process. Perhaps a better approach would be to use a variety of simple configurations and augment the data such that the dataset is large enough for training.

Neural networks are certainly capable of learning the rulesets for a simple world of cells. If taken further, this technology could be very useful to biologists who, for example, don't understand what rules a new cellular creature which they have discovered follows. This project is unique in that it combines cellular automata with neural networks. Both technologies have strong applications in the real world, but are rarely combined. As a student and not a PHD professor, this project is clearly limited in both scope and dedication. There's a lot more to explore, and a lot of directions which could lead to even more meaningful conclusions. While I'm obviously not the first person to make the connection between cellular automata and neural networks, this technology is relatively unexplored and has a lot of potential for growth. I believe that in the near future, there will be more advanced combinations of the two and I can't wait to see what comes from it.