Video Prediction From Single Image

Chaoran Zhang and Xueyi Zhao

Shanghai Jiao Tong University

1. INTRODUCTION

Visual prediction is one of the most fundamental and difficult tasks in computer vision. Existing video prediction methods mainly rely on observing multiple historical frames or focus on predicting the next one-frame. In this work, we study the problem of generating future frame by observing one single still image only.

Our project is creating a model that can use a single image, through a convolution network that can predict future optic flow, it can give back the next future frame that shows how the object in the origin image may move like. To accomplish this objective, we have tried LSTM (Long Short-Term Memory) Network and XGBoost method.

2. BACKGROUND

In a given scene, humans can often easily predict a set of immediate future events that might happen. However, it's not as easy as human for computers to predict the future. But it's still meaningful to let computer have the ability to predict future images as well. For example, this is a necessary ability for self-driving AI. It can also be used as an effective tool in video editing.

Our motivation to do this project is that we are interested in applying artificial intelligence and deep learning techniques onto image processing area and we want an opportunity to study the basics in both aspects.

In our work, we mainly focused on studying how to generate a future frame given one still image. As far as we concerned, using deep learning neural network is a good way to build a model that can learn the difference between future frame and previous one.

During our project process, we found that it's very difficult to create a universal method that is suitable for all categories of images in different contents and sizes. For our project, the input image size should be 320*180(can be edited in the code) and we suggest to use corresponding dataset for the target image to train before predicting.

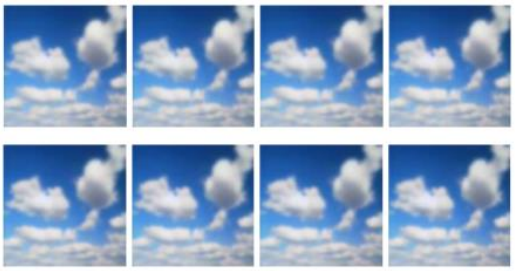


fig1-a series of frames predicted by Denton et al.

3. RELATED WORKS

Denton, E., Birodkar, V.: Unsupervised learning of disentangled representations from video. In: NIPS. (2017)

Xue, T., Wu, J., Bouman, K., Freeman, B.: Visual dynamics: Probabilistic future frame synthesis via cross convolutional networks. In: NIPS. (2016)

Yijun Li Flow-Grounded Spatial-Temporal Video. In: ECCV(2018)

Besides there are some methods using multiple frames to generate future image (or video). For fair, we haven't compared to them since the input can't be the same. The mentioned papers' method can receive one image as input as well as ours.

4. PROBLEM FORMULATION

We all know that every image is made of numerous pixels. Each pixel can be presented as a list with 3 elements representing R, G, B values of this point. Our basic idea is to use information in the neighbor pixels to predict the center pixel's next-frame image.

Image: a file containing 320*180*3 data for R, G, B separately

Input: a single image
Output: prediction of the future image
Training set: series of continuous images
Model: trained with a large number of training data, is able to predict a future frame of the input test data.
After our discussion, we divided the whole process into 4 steps: Data Preprocessing, Model Training, Predicting and Image Generation.

In the next part I'll explain each step's work more specifically.

5. Proposed Methods

In our experiments we have tried 2 kinds of model - LSTM and XGBoost. Their data processing method is same, however. The different part is the training and predicting steps.

5.1 Data Preprocessing

First, we use OpenCV to read the images which serve as dataset and store the information of the images (RGB) into a csv document line by line. Each line contains 3 float value representing a pixel's R, G, B values.

Second, we think the nine points in a square in the original image has a strong impact on the center point in the corresponding square in the next image. Under this thought, we use the information in the csv document and make a 2-dimension numpy array storing 9 points' R (or G or B) value line by line. This process can get the input dataset which will be input the neural network. Besides, the output should be one point, so we store the corresponding center point's value into another numpy array line by line.

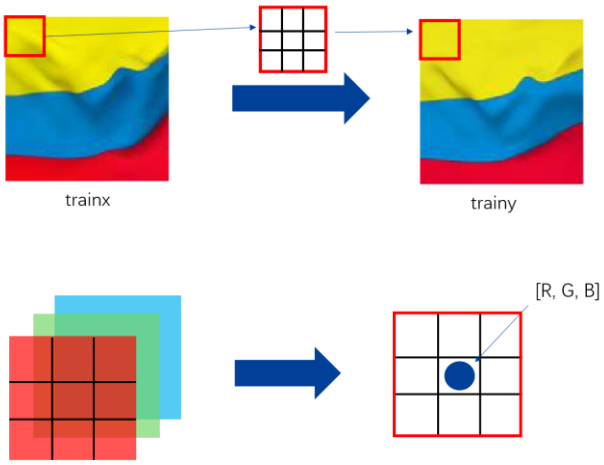


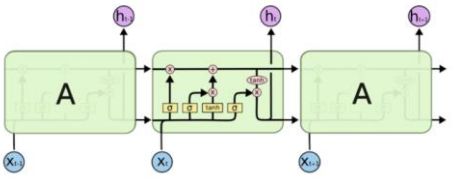
fig-2 Data Preprocessing

5.2 Model Training

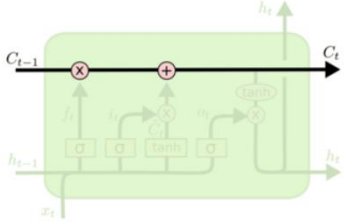
As mentioned before, we used LSTM and XGBoost models in our project. Each model is called 3 times for R&G&B each.

5.2.1 LSTM (Long short-term memory)

The long and short memory neural network, often called LSTM, is a special RNN that learns long dependencies. It works very well on a variety of issues and are now widely used.



The core idea of LSTM:



The key to LSTM is the state of the cell (the green image shows a cell) and the horizontal line through the cell. The cell state is similar

to a conveyor belt. Run directly on the entire chain with only a few linear interactions. It will be easy to keep the information flowing on it.

If only the above horizontal line is unable to add or delete information. It is implemented by a structure called gates. The gate can selectively pass information, mainly through a neural layer of sigmoid and a point-by-point multiplication operation.

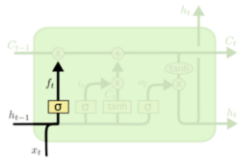
Each element of the sigmoid layer output (which is a vector) is a real number between 0 and 1, representing the weight (or proportion) through which the corresponding information passes. For example, 0 means "no information is passed" and 1 means "let all information pass".

Three doors of LSTM:

Forgotten Gate:

The first step in our LSTM is to decide what information we will discard from the cell state. This decision is made through a process called forgetting the door. The gate reads h_{t-1} and x_t and outputs a value between 0 and 1.

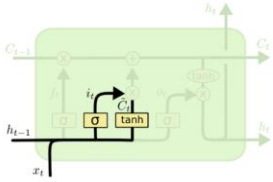
Each number in the cell state C_{t-1}. 1 represents "completely reserved" and 0 represents "completely discarded".



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Input gate:

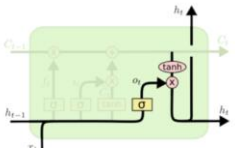
Decide how much new information to add to the cell state. There are two steps to accomplish this: First, a sigmoid layer called "input gate layer" determines which information needs to be updated; a tanh layer generates a vector, which is the alternative content for updating, C_t. In the next step, we combine the two parts to update the state of the cell.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Output gate:

We need to determine what value to output. This output will be based on our cell status, but it is also a filtered version. First, we run a sigmoid layer to determine which part of the cell state will be output. Next, we process the cell state through tanh (get a value between -1 and 1) and multiply it by the output of the sigmoid gate. Eventually we will only output the part of our output that we determined.



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

The realization of LSTM:

We make the input of the data preprocessing as the LSTM's input and the output of the data preprocessing as the LSTM output, then we use the LSTM model to train and test them. After these processes, we can get a trained model, then we put the test image, and get the result of the algorithm. The result will be put into the data post processing and get the image.

In keras, LSTM has been integrated into the library. In our project we directly used:

from keras.layers import LSTM

to add a lstm layer. Specified training process will be given in experiment part.

5.2.2 XGBoost

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost

provides a parallel tree boosting (also known as GBDT) that solve many data science problems in a fast and accurate way.

XGBoost implements a general Tree Boosting algorithm. One representative of this algorithm is the Gradient Boosting Decision Tree (GBDT), also known as the Multiple Additive Regression Tree (MART).

The principle of GBDT is to first train a tree using the training set and the sample true value (that is, the standard answer), and then use this tree to predict the training set, and obtain the predicted value of each sample. Because the predicted value has a deviation from the true value, the subtraction of the two can get the "residual". Next, train the second tree. Instead of using the true value, it uses the residual as the standard answer. After the two trees are trained, the residuals of each sample can be obtained again, and then the third tree can be further trained, and so on. The total number of trees in the tree can be specified manually, and some indicators (such as errors on the validation set) can be monitored to stop training.

When predicting a new sample, each tree will have an output value, and the output values will be added to obtain the final predicted value of the sample.

5.3 Predicting

We can store the data of the test image into a .csv file in the same way as data preprocessing. Each line represents a pixel's RGB values. Let the model use this file as input, we can get the outputs: predicted RGB values in 3 numpy arrays.

5.4 Image Generation

We get 3 arrays (RGB) from the neural networks. And we should put them into one file and use pandas to read them and transfer them into the image format. Finally, we can use OpenCV to write the picture into .png form. Since our algorithm use the 9 to 1 strategy, so we get the picture will be reduced to a 318*178 image. This reduction can be eliminated by add a padding while doing the prediction.

6. EXPERIMENTAL RESULTS

We've used datasets of flags, clouds and some other objects to train and test our model. Every dataset is a series of continuous images or a short video (cut into images). Using our getrain.py can transfer pictures into .csv files.

6.1 LSTM

The model is construct as follow:
LSTM layer (unit=128, input shape=8*10, return_sequences=True)
Dropout layer (dropout rate=0.5)
LSTM layer (return_sequences=True)
LSTM layer (return_sequences=False)
Dense layer (dense=1)
Activation=relu, loss=mse, optimizer=adam metrics=mse

Besides, I used a callback EarlyStopping monitor to avoid overfitting. This monitor can automatically stop training when loss is not reducing.

Other variables: epochs=5, validation set=last 0.01 of input data, batch_size=128, shuffle=True.

The mean-square-error of the final epoch is 2.5007e-6, and here is a predicting result of a rotating plate:

6.2 XGBoost

To use XGB we need to install and import xgboost library. I installed it by pip.

The input of xgbooster must be 2-dimensional matrix. For x I used ndim=2 method in np.array. For y I put each RGB

value into a list to construct a 2-dim matrix. Here are some parameters' meaning:

1. max_depth: The default value is set to 6. We need to specify the maximum depth of a tree. The parameter range is from 1 to ∞.

2. eta: The default value is set to 0.3. We need to specify a step size contraction to prevent overfitting. After each promotion step, we can directly get the weight of the new feature. In fact, the process of increasing the weight of the eta contraction feature is more conservative. The range is 0 to 1. A low η value means that the model overfitting is more robust.

3. evals: This is a list that is used to evaluate the elements in the list during the training process. The form is evals = [(dtrain, 'train'), (dval, 'val')] or evals = [(dtrain, 'train')], it allows us to observe during training Verify the effect of the set.

After we finished training the booster, we can use it on the test data to predict the result. The model's train-rmse is 15.3181, 15.4047, 16.1403(for R, G, B)

Result:

Notice that since we chose to use 9 surrounding points to generate a point, and for some test image, only a small amount of the pixels has been changed, so the model may learn from the unchanged points, which is the main part of the image, and give out an output image that doesn't change much. The shown case suggests this problem. So the algorithm may perform better on images that the whole image is changing, like cloud images.

7. CONCLUSION

In this work, we make a neural network to predict the next image according to the its previous images' information. We think the previous images' changing rule can fit the next image by training. We use several deep learning models to test the results of predicting by compare original image and output result. Finally, we find that XGBoost has the best result. Though our work is not as good as those paper's, by doing this experiment we have been more familiar with the deep learning and image process tools, which may be useful to us in the future study.