Ship Detection Report

whether there are ships in satellite images

Team member: Qingchun Xia [redacted]
Ruijia Yang [redacted]
Zhiyong Zhang [redacted]

Abstract

Research idea

Shipping traffic is growing fast. More ships increase the chances of infractions at sea like environmentally devastating ship accidents, piracy, illegal fishing, drug trafficking, and illegal cargo movement. This has compelled many organizations, from environmental protection agencies to insurance companies and national government authorities, to have a closer watch over the open seas. So we team want to research on whether there are ships in satellite images.

Method

Classification tasks need to be divided into two categories: ship and no ship. Our program designs a convolutional neural network for processing image classification tasks. Next, we will introduce a classification model based on convolutional neural networks, and focus on improving the classification accuracy step by step through different techniques.

Result

Through multiple steps, accuracy could be as high as 96.88%. Adjustments to the model cause different classification accuracies.

Introduction

The data set used in our project is a data set of satellite images with or without ships, which has 1,000 images, each of which is a color picture with a resolution of 768×768 (divided into 3 channels).

Since our image is actually composed of one pixel, each image can be regarded as a vector, which is to say, input tensors inside our convolutional neural network are 3D tensors - [height, width, channels].

About the preprocessing part, which is the main independent factor we team want to take a research on. We team have preprocessed the images into 200×200 in order to researching on whether the preprocessing will affect the result of our convolutional neural network.

Convolutional neural network can be regarded as a structured multiple layer perceptron network, combining three structural methods to achieve displacement, scaling and distortion invariance. These three structural methods are local receptive fields, shared weights, and sub-sampling in the spatial or temporal domain.

The local receptive field is that the neurons in each network layer are only connected to the neural units in a small neighborhood of the upper layer. Through local receptive fields, neurons can extract primary visual features such as direction segments, endpoints, corners, etc.

Weight sharing means that neurons in the same feature map share the same weight, making the convolutional neural network have fewer parameters. The local receptive field and weight sharing make the convolutional neural network have translation invariance, and each feature map extracts a feature that is insensitive to the location of the feature.

Subsampling reduces the resolution of the feature map, reducing sensitivity to displacement, scaling, and distortion.

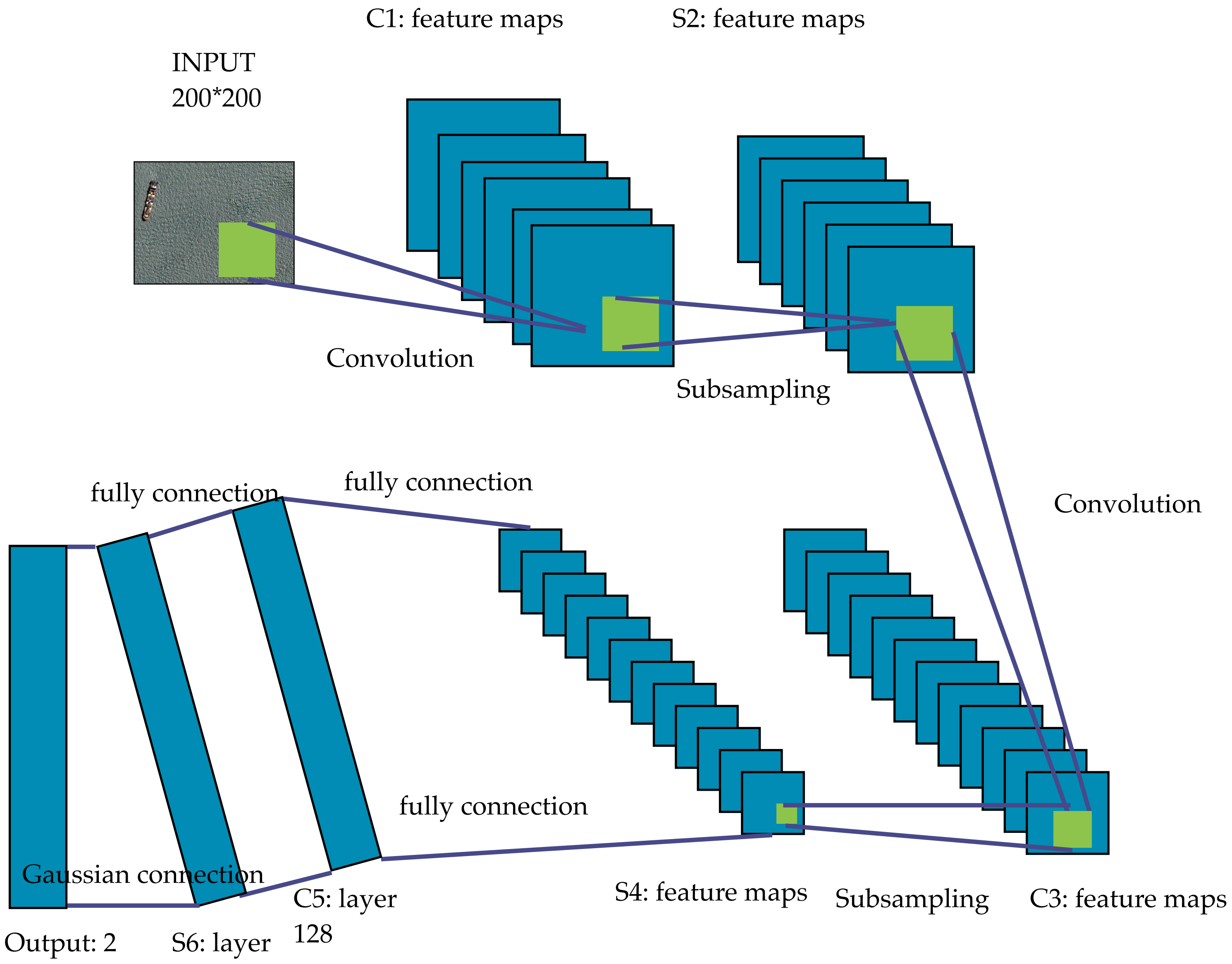
Through whole project, we applied at hyper-parameters with different value to take a research on how they would affect the accuracy.

Methods

Convolutional Neural Network is the main method we used to detect ship. A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, RELU layer i.e. activation function, pooling layers, fully connected layers and normalization layers.

Description of the process as a convolution in neural networks is by convention. Mathematically it is a cross-correlation rather than a convolution (although cross-correlation is a related operation). This only has significance for the indices in the matrix, and thus which weights are placed at which index.

Our team's convolutional neural network is made of 2 convolutional layers, 2 pooling layer, 2 normalization layers and 2 fully connected layers. As shown below:

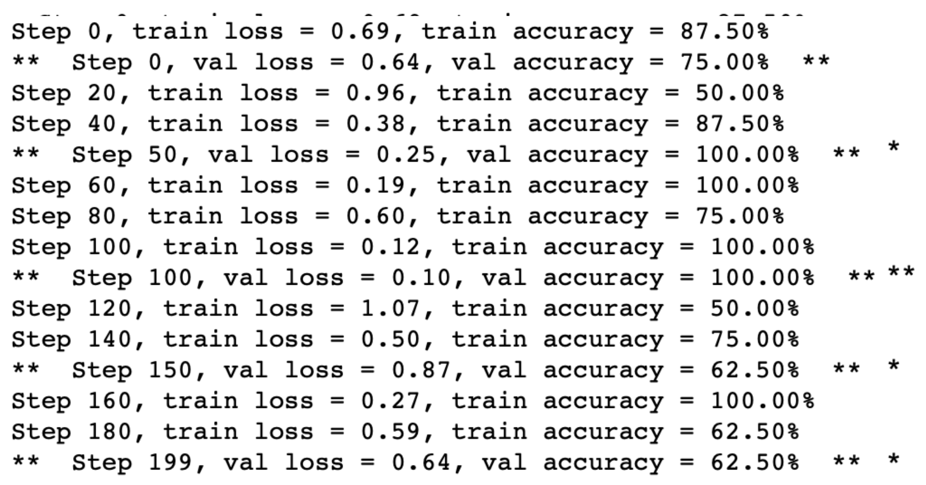


Besides, we use variable-controlling approach to test each hyper-parameter. The main hyper-parameter is the size of images. The original csv file this dataset provided contains 2 columns: the image's id and the detailed information about the ship's pixels which are encoded by run-length encoding. So there is one python file which set forth how we interpret run-length encoding clearly and we decoded run-length encoding. By decoding, we were able to accomplish the task of cropping 768*768 images into 200*200 images containing ships.

We also controlled hyper-parameters like activation function, epochs, loss function, etc.

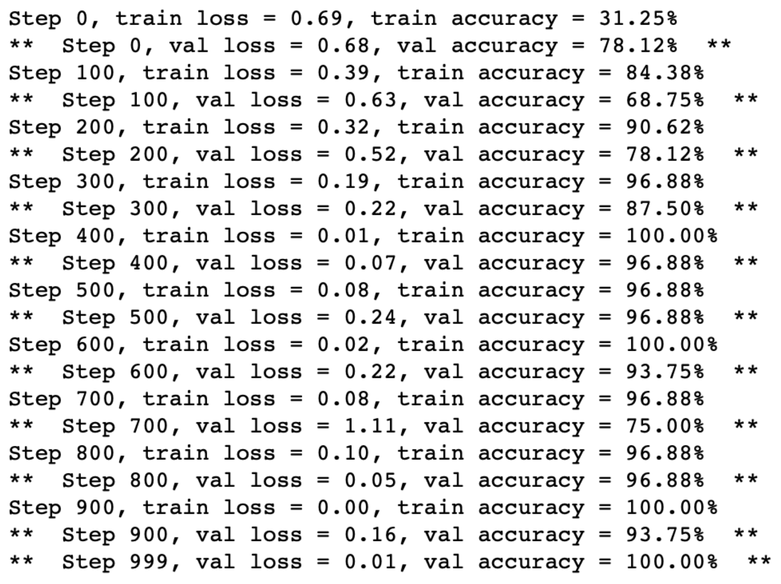
Results

Results obtained by 768*768 images is shown below:



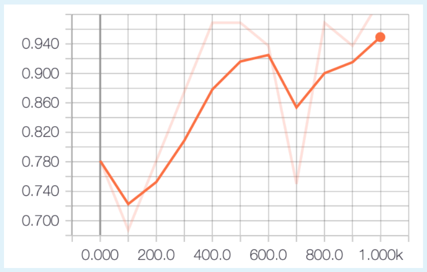
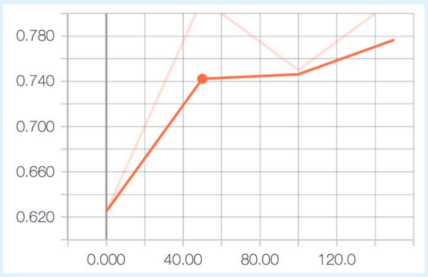
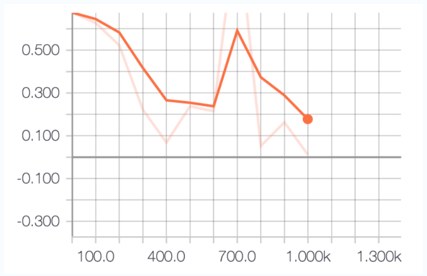
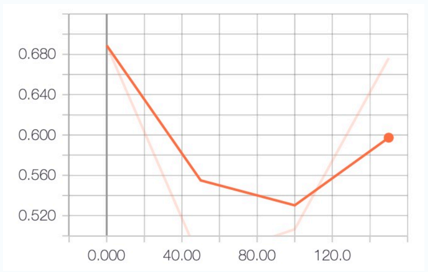
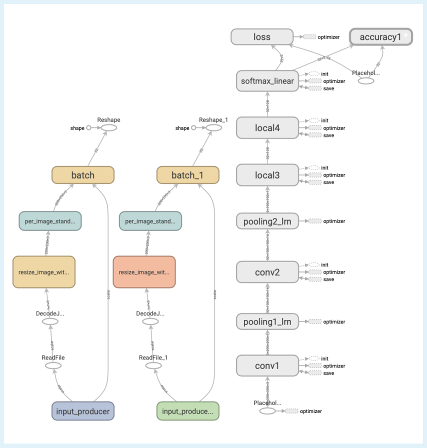
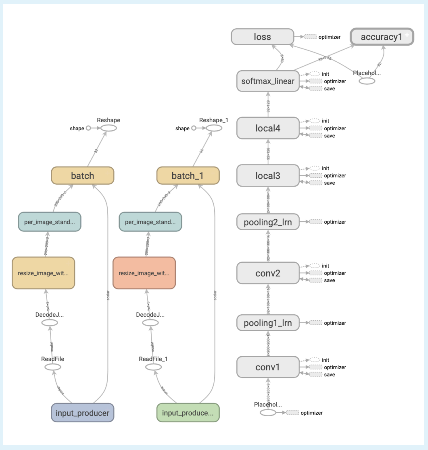
```
Step 0, train loss = 0.69, train accuracy = 87.50%
**  Step 0, val loss = 0.64, val accuracy = 75.00%  **
Step 20, train loss = 0.96, train accuracy = 50.00%
Step 40, train loss = 0.38, train accuracy = 87.50%
**  Step 50, val loss = 0.25, val accuracy = 100.00%  **  *
Step 60, train loss = 0.19, train accuracy = 100.00%
Step 80, train loss = 0.60, train accuracy = 75.00%
Step 100, train loss = 0.12, train accuracy = 100.00%
**  Step 100, val loss = 0.10, val accuracy = 100.00%  ** **
Step 120, train loss = 1.07, train accuracy = 50.00%
Step 140, train loss = 0.50, train accuracy = 75.00%
**  Step 150, val loss = 0.87, val accuracy = 62.50%  **  *
Step 160, train loss = 0.27, train accuracy = 100.00%
Step 180, train loss = 0.59, train accuracy = 62.50%
**  Step 199, val loss = 0.64, val accuracy = 62.50%  **  *
```

Results obtained by 200*200 images is shown below:



```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

By controlling activation function: relu $\longrightarrow$ leaky_relu

	Relu	leaky_relu
Accuracy		
Loss		
Structure		

Output

Relu:

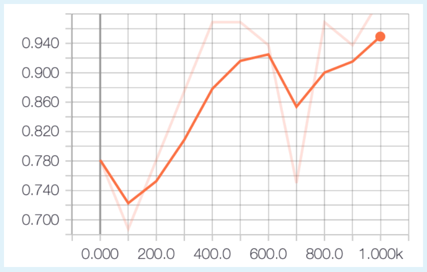
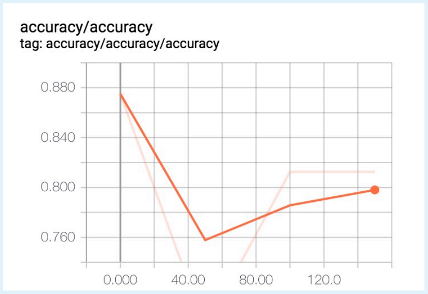
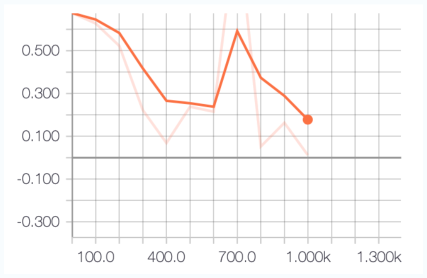
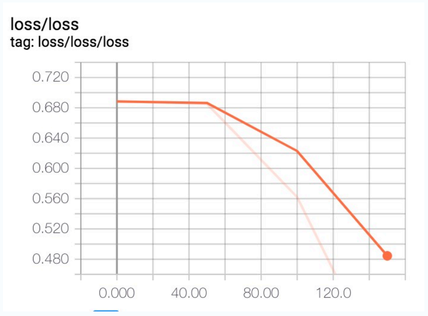
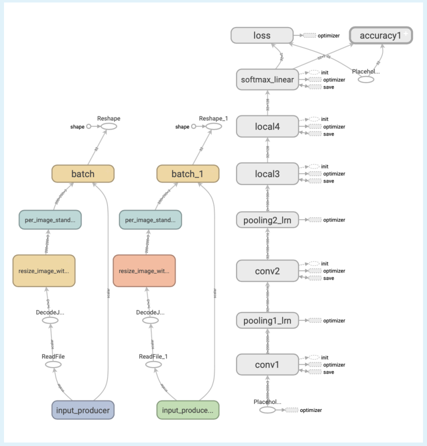
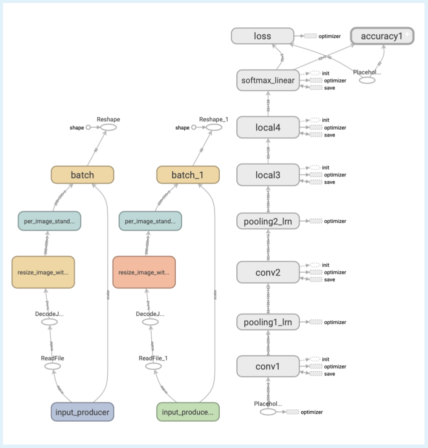
```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

leaky_relu:

```
Step 0, train loss = 0.69, train accuracy = 56.25%
**  Step 0, val loss = 0.69, val accuracy = 62.50%  **
Step 20, train loss = 0.70, train accuracy = 68.75%
Step 40, train loss = 0.66, train accuracy = 50.00%
**  Step 50, val loss = 0.47, val accuracy = 81.25%  **
Step 60, train loss = 0.33, train accuracy = 87.50%
Step 80, train loss = 0.48, train accuracy = 75.00%
Step 100, train loss = 0.36, train accuracy = 81.25%
**  Step 100, val loss = 0.51, val accuracy = 75.00%  **
Step 120, train loss = 0.18, train accuracy = 93.75%
Step 140, train loss = 0.50, train accuracy = 75.00%
**  Step 150, val loss = 0.68, val accuracy = 81.25%  **
Step 160, train loss = 0.39, train accuracy = 81.25%
Step 180, train loss = 0.12, train accuracy = 100.00%
**  Step 199, val loss = 0.60, val accuracy = 62.50%  **
```

By controlling cost function:

sparse_cross_entropy_with_logits $\longrightarrow$ sparse_cross_entropy

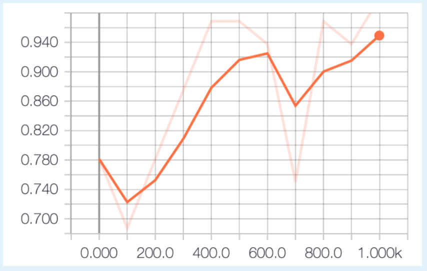
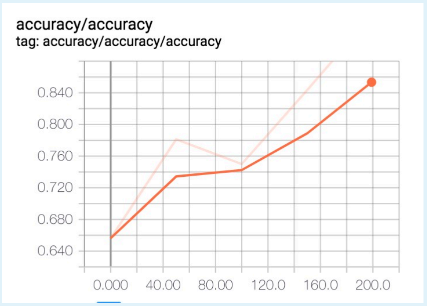
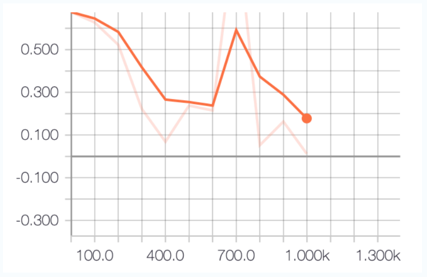
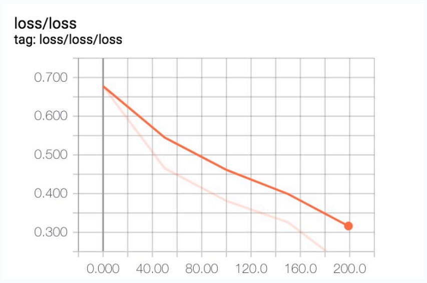
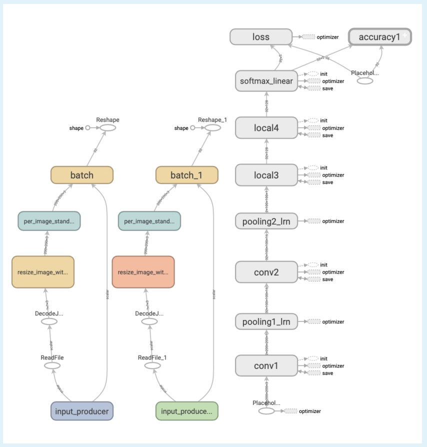
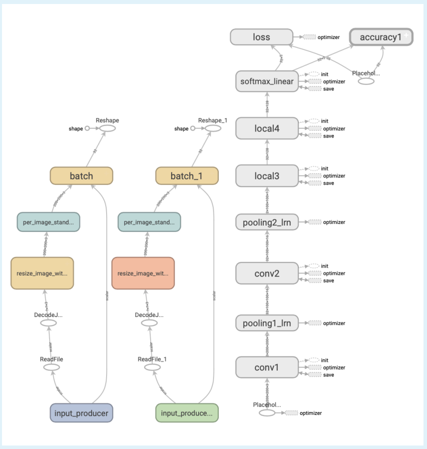
	sparse_cross_entropy_with_logits	sparse_cross_entropy
Accuracy		
Loss		
Structure	 loss accuracy1 softmax_linear local4 local3 pooling2_lrn conv2 pooling1_lrn conv1 batch batch_1 per_image_stand... resize_image_wit... DecodeJ... ReadFile input_producer input_produce... Reshape Reshape_1 shape Placehol... init optimizer save	 loss accuracy1 softmax_linear local4 local3 pooling2_lrn conv2 pooling1_lrn conv1 batch batch_1 per_image_stand... resize_image_wit... DecodeJ... ReadFile input_producer input_produce... Reshape Reshape_1 shape Placehol... init optimizer save

Output

```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

```
Step 0, train loss = 0.69, train accuracy = 50.00%
**  Step 0, val loss = 0.69, val accuracy = 87.50%  **
Step 20, train loss = 0.69, train accuracy = 62.50%
Step 40, train loss = 0.69, train accuracy = 75.00%
**  Step 50, val loss = 0.69, val accuracy = 68.75%  **
Step 60, train loss = 0.68, train accuracy = 68.75%
Step 80, train loss = 0.66, train accuracy = 75.00%
Step 100, train loss = 0.61, train accuracy = 75.00%
**  Step 100, val loss = 0.56, val accuracy = 81.25%  **
Step 120, train loss = 0.56, train accuracy = 75.00%
Step 140, train loss = 0.55, train accuracy = 68.75%
**  Step 150, val loss = 0.32, val accuracy = 81.25%  **
Step 160, train loss = 0.44, train accuracy = 81.25%
Step 180, train loss = 0.51, train accuracy = 62.50%
**  Step 199, val loss = 0.31, val accuracy = 81.25%  **
```

By controlling epochs:
batch_size=16 → batch_size=32

	batch_size=16	batch_size=32
Accuracy		
Loss		
Structure		

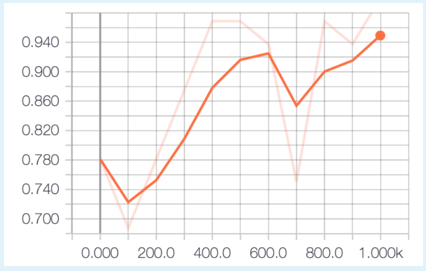
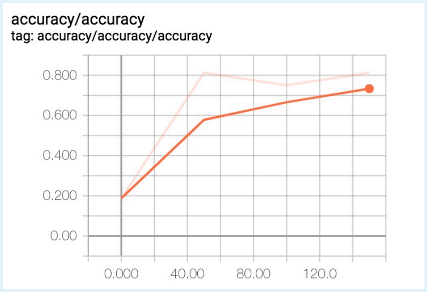
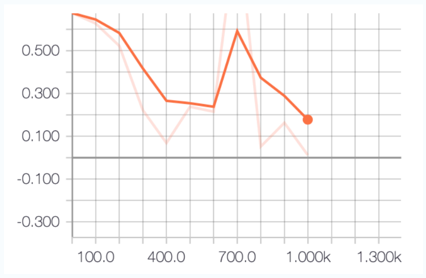
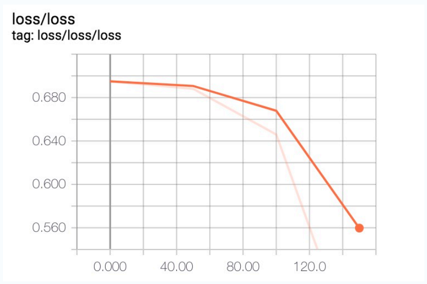
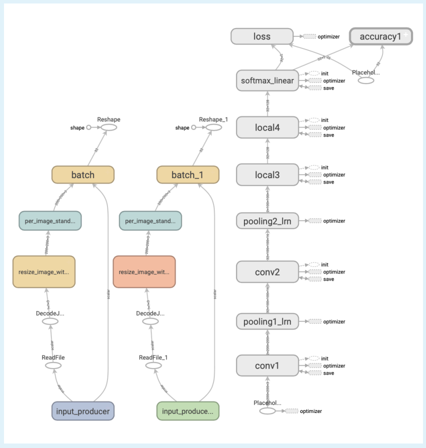
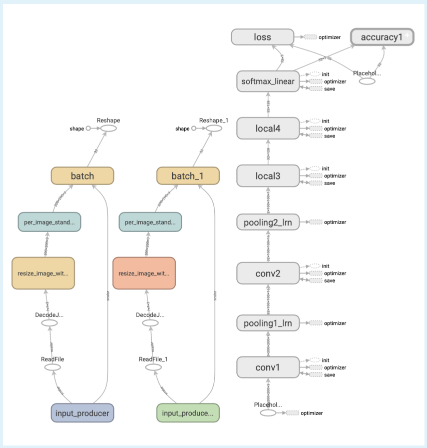
Output

```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

```
Step 0, train loss = 0.69, train accuracy = 75.00%
**  Step 0, val loss = 0.68, val accuracy = 65.62%  **
Step 20, train loss = 0.61, train accuracy = 68.75%
Step 40, train loss = 0.66, train accuracy = 62.50%
**  Step 50, val loss = 0.47, val accuracy = 78.12%  **
Step 60, train loss = 0.63, train accuracy = 62.50%
Step 80, train loss = 0.37, train accuracy = 75.00%
Step 100, train loss = 0.50, train accuracy = 75.00%
**  Step 100, val loss = 0.38, val accuracy = 75.00%  **
Step 120, train loss = 0.42, train accuracy = 78.12%
Step 140, train loss = 0.48, train accuracy = 78.12%
**  Step 150, val loss = 0.33, val accuracy = 84.38%  **
Step 160, train loss = 0.27, train accuracy = 90.62%
Step 180, train loss = 0.29, train accuracy = 87.50%
**  Step 199, val loss = 0.21, val accuracy = 93.75%  **
```

By controlling gradient estimation:

ADAM $\longrightarrow$ RMSProp

	ADAM	RMSProp
Accuracy		
Loss		
Structure		

Output

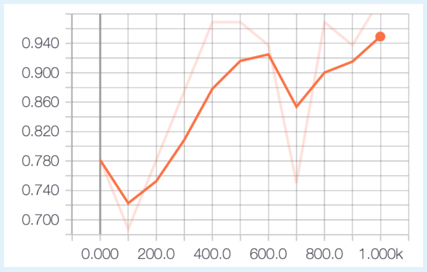
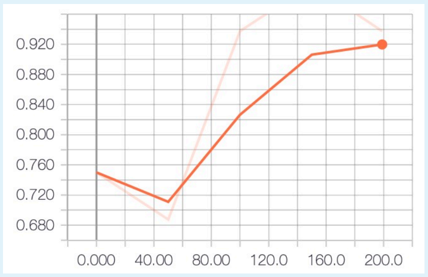
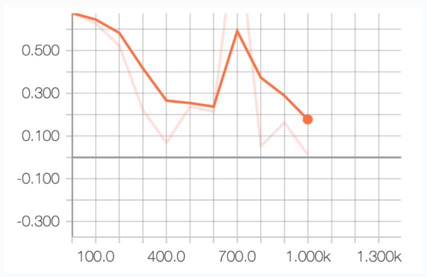
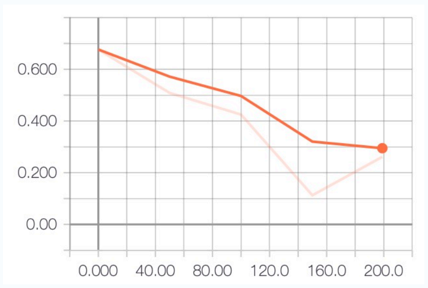
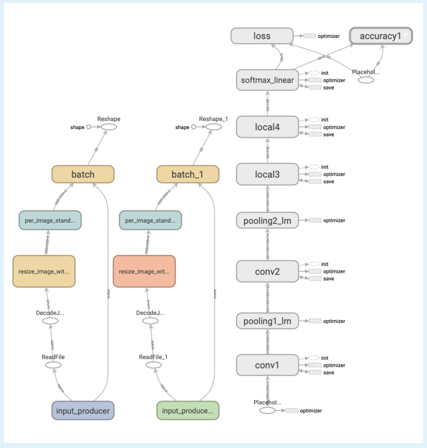
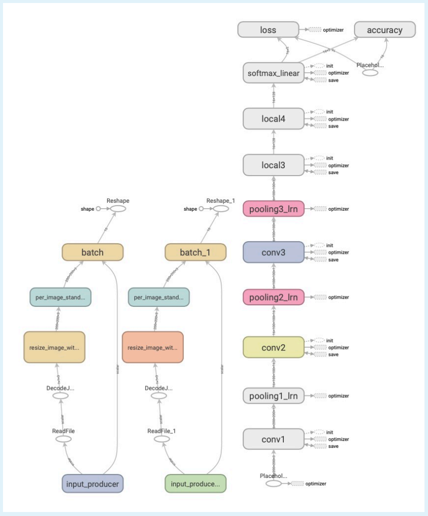
ADAM:

```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

RMSProp:

```
Step 0, train loss = 0.70, train accuracy = 18.75%
**  Step 0, val loss = 0.70, val accuracy = 18.75%  **
Step 20, train loss = 0.69, train accuracy = 31.25%
Step 40, train loss = 0.69, train accuracy = 81.25%
**  Step 50, val loss = 0.69, val accuracy = 81.25%  **
Step 60, train loss = 0.68, train accuracy = 87.50%
Step 80, train loss = 0.68, train accuracy = 75.00%
Step 100, train loss = 0.63, train accuracy = 87.50%
**  Step 100, val loss = 0.65, val accuracy = 75.00%  **
Step 120, train loss = 0.56, train accuracy = 75.00%
Step 140, train loss = 0.57, train accuracy = 75.00%
**  Step 150, val loss = 0.43, val accuracy = 81.25%  **
Step 160, train loss = 0.71, train accuracy = 56.25%
Step 180, train loss = 0.53, train accuracy = 75.00%
**  Step 199, val loss = 0.56, val accuracy = 68.75%  **
```

By controlling network structure: adding one convolutional layer

	Two convolutional layers	Three convolutional layers
Accuracy		
Loss		
Structure		
Output	see below	see below

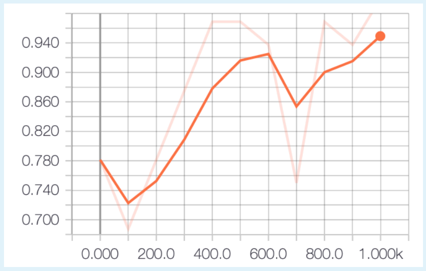
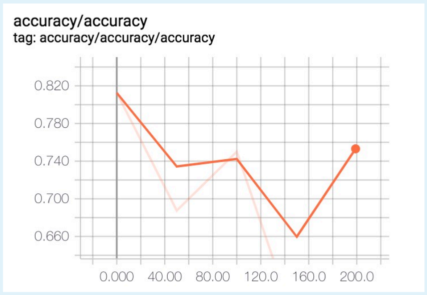
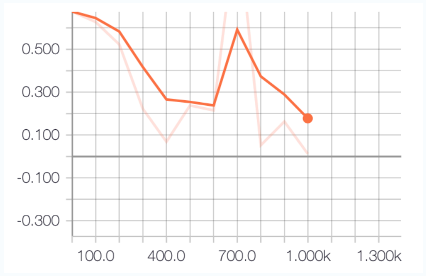
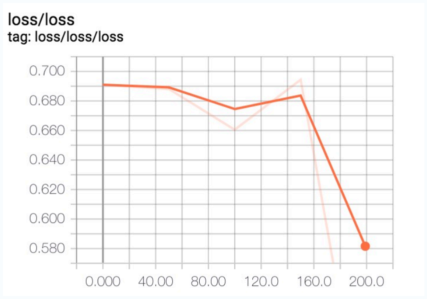
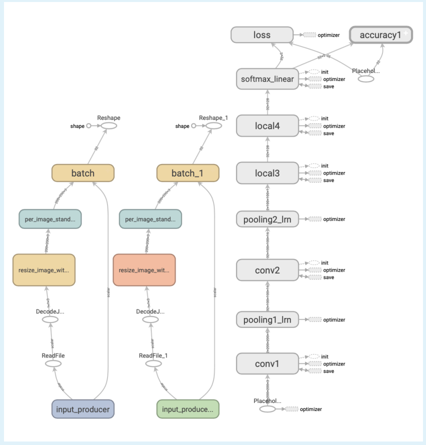
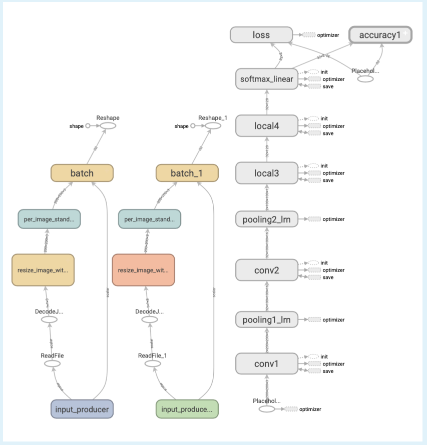
Two convolutional layers:

```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

Three convolutional layers:

```
Step 0, train loss = 0.70, train accuracy = 25.00%
**  Step 0, val loss = 0.68, val accuracy = 75.00%  **
Step 20, train loss = 0.65, train accuracy = 62.50%
Step 40, train loss = 0.54, train accuracy = 81.25%
**  Step 50, val loss = 0.51, val accuracy = 68.75%  **
Step 60, train loss = 0.63, train accuracy = 43.75%
Step 80, train loss = 0.35, train accuracy = 81.25%
Step 100, train loss = 0.41, train accuracy = 75.00%
**  Step 100, val loss = 0.42, val accuracy = 93.75%  **
Step 120, train loss = 0.46, train accuracy = 81.25%
Step 140, train loss = 0.50, train accuracy = 75.00%
**  Step 150, val loss = 0.11, val accuracy = 100.00%  **
Step 160, train loss = 0.45, train accuracy = 81.25%
Step 180, train loss = 0.39, train accuracy = 75.00%
**  Step 199, val loss = 0.26, val accuracy = 93.75%  **
```

By controlling network initialization: Truncated Normal to tf.initializers.zeros

	Truncated Normal	tf.initializers.zeros
Accuracy		
Loss		
Structure		

Output

Truncated Normal:

```
Step 0, train loss = 0.69, train accuracy = 31.25%
**  Step 0, val loss = 0.68, val accuracy = 78.12%  **
Step 100, train loss = 0.39, train accuracy = 84.38%
**  Step 100, val loss = 0.63, val accuracy = 68.75%  **
Step 200, train loss = 0.32, train accuracy = 90.62%
**  Step 200, val loss = 0.52, val accuracy = 78.12%  **
Step 300, train loss = 0.19, train accuracy = 96.88%
**  Step 300, val loss = 0.22, val accuracy = 87.50%  **
Step 400, train loss = 0.01, train accuracy = 100.00%
**  Step 400, val loss = 0.07, val accuracy = 96.88%  **
Step 500, train loss = 0.08, train accuracy = 96.88%
**  Step 500, val loss = 0.24, val accuracy = 96.88%  **
Step 600, train loss = 0.02, train accuracy = 100.00%
**  Step 600, val loss = 0.22, val accuracy = 93.75%  **
Step 700, train loss = 0.08, train accuracy = 96.88%
**  Step 700, val loss = 1.11, val accuracy = 75.00%  **
Step 800, train loss = 0.10, train accuracy = 96.88%
**  Step 800, val loss = 0.05, val accuracy = 96.88%  **
Step 900, train loss = 0.00, train accuracy = 100.00%
**  Step 900, val loss = 0.16, val accuracy = 93.75%  **
**  Step 999, val loss = 0.01, val accuracy = 100.00%  **
```

tf.initializers.zeros:

```
Step 0, train loss = 0.69, train accuracy = 68.75%
**  Step 0, val loss = 0.69, val accuracy = 81.25%  **
Step 20, train loss = 0.69, train accuracy = 68.75%
Step 40, train loss = 0.69, train accuracy = 68.75%
**  Step 50, val loss = 0.69, val accuracy = 68.75%  **
Step 60, train loss = 0.69, train accuracy = 62.50%
Step 80, train loss = 0.68, train accuracy = 68.75%
Step 100, train loss = 0.66, train accuracy = 75.00%
**  Step 100, val loss = 0.66, val accuracy = 75.00%  **
Step 120, train loss = 0.69, train accuracy = 56.25%
Step 140, train loss = 0.68, train accuracy = 62.50%
**  Step 150, val loss = 0.69, val accuracy = 56.25%  **
Step 160, train loss = 0.74, train accuracy = 56.25%
Step 180, train loss = 0.55, train accuracy = 75.00%
**  Step 199, val loss = 0.45, val accuracy = 87.50%  **
```

Discussion

For our project, it's apparent that preprocessing could not only help us to accelerate the speed of running our model, but also raises the accuracy of our model.

	768*768 images	200*200 images
running time	1.5 h	10 mins
accuracy	62.5%	100%

What's more, controlling each hyper-parameter causes different results as above results part.

So there are two conclusions:

1. Preprocessing images is a very important part could not only improve running efficiency but also improve accuracy.
2. Activation function, cost function, epochs, gradient estimation, network architecture and network initialization matter in a convolutional neural network.

References

Scope

Our group has three members, and our group's project is image recognition. Image recognition requires a complete CNN model, including many parameters. For the whole model, we need to config the number of convolution layer, the number of max pooling layer, the number of full connection layer and the order of these layers. For convolution layer, we need to config filter length, filter width, filter number, filter stride. For max pooling layer, we need to config pool length, pool width, pool stride. For full connection layer, we need to config the number of neurons. Also, we need to config parameters like activation function for each layer, loss function, the initialization type of weights and biases. For input data, we have to make sure that the data can be feed to the model. We have to transform the data type, the array dimension, how to manage the input data structure. Batch size for train data, batch size for test data, epochs, step numbers, queue size, etc.

Since the origin picture is too big, it's 768*768, which may take too much time to train and test. So, we resize the picture based on the run length code provided, which indicates which pixel is ship, which is not. After many attempts, we decided to resize to 200*200.

All of the parameters we have mentioned has to be tried many times, and except the basic model api, we code all the preprocessing progress by ourselves. We try to figure out the best optimized model so it took really a lot of time of all of us to do this project.

Context

Basically, our code can be divided into four parts:

1. Preprocessing

The first part is the pre-processing and in there we cut the 768 * 768 pixels picture into smaller one with the size of 200 * 200 pixels. we are doing it by decoding the initial pixel values of the original picture so our smaller pictures will also include the ship images inside of that particular picture. Here is the technology of "Run Length Decoding" in which we can locate the exact location of ship pixels on the pictures and cut it out.

We have written our own code to generate 200*200 pixel pictures from our original images.

2. Generating Batch

Our images are relatively large and therefore we need an input queue and batches to load the pictures.

We were writing this part of the code under the help of Tensorflow functions as it offers functions for the batch generation. First, we will go over the image directory, loading the name of images and labels into a queue, and then generating image content batch and label batch.

3. Model

We referenced on the CIFAR-10 model from the official GitHub(https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10) of Tensorflow as the base code for Convolutional Neuron Network(CNN) part.

CIFAR-10 is a commonly used model in the area of image recognition. For our project, we slightly change the model part as CNN's major techniques are convolution, pooling, and fully connected layer.

4. Training

And then we will train our CNN model to see the different results on original 768*768 pictures and 200*200 pictures to compare the impact of the pre-processing. We are doing it using placeholders to feed training and validation data into the placeholder to see the accuracy and loss variation and result.