Final Project of Statistical Signal Estimation

Using Adaptive Moment Estimation to Optimize CNN for Image Recognition

Zeyu Liu

Professor Milos

ECE 6865

Fall 2019

The George Washington University

Abstract

In the final project, I will use the Adam (Adaptive Moment Estimation) optimizer to optimize CNN (Convolutional Neural Network) for image recognition. Adam algorithm is an extension of the stochastic gradient descent algorithm and has recently been widely used in deep learning areas. I will explain and derive the algorithm process in the paper. The dataset I used is divided into two categories, one for airplane photos and one for lake photos, each with 700 photos. 80% of them are used as train set and 20% are used as test set. I will build a convolutional neural network model and train it by using Adam optimizer. Eventually achieve the correct prediction classification on the test set.

Keywords: Adam, CNN

Introduction

I will explain some important concepts in this part. They are CNN, Relu activation, Sigmoid activation, Adam algorithm, Binary cross-entropy loss, Accuracy. mathematical modeling and analysis are contained. The specific experimental model will be shown in the next part.

The dataset I used in the report named “Satellite identification”, This dataset contains 700 lake pictures and 700 airplane pictures. Some of them are showing in *Figure 1*.

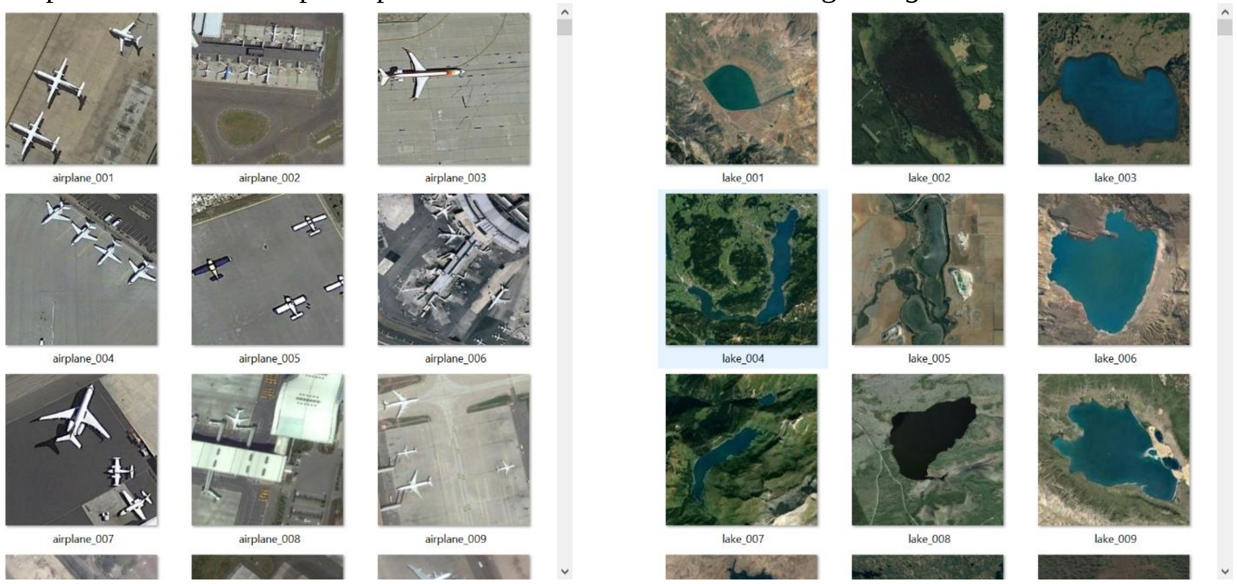


Figure 1. The dataset of airplane and lake.

The purpose of my project is to achieve classification and prediction of these two pictures through CNN which optimized by Adam algorithm.

CNN

Convolutional neural network (CNN) consists of input layer, convolution layer, activation function, pooling layer, and fully connected layer, that is, INPUT (input layer)-CONV2D (convolution layer)-RELU (activation function)-POOL (pooling layer)-FC (fully connected layer, dense layer). A standard CNN structure is shown in the *Figure 2*.

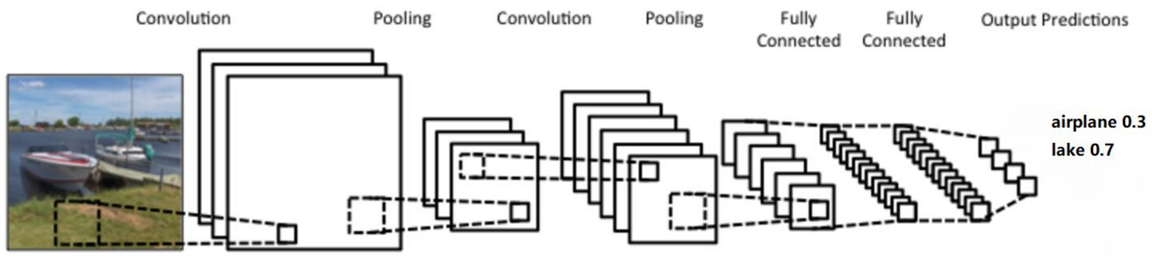


Figure 2. An example of a standard CNN structure.

There are three important layers in CNN model. Convolution layer, pooling layer, and dense layer.

The dense layer shows in *Figure 3*. We can find the output

$$y = (w_t x_t + b) \cdot activation\ function$$

in the figure. This conclusion can use in most neural network.

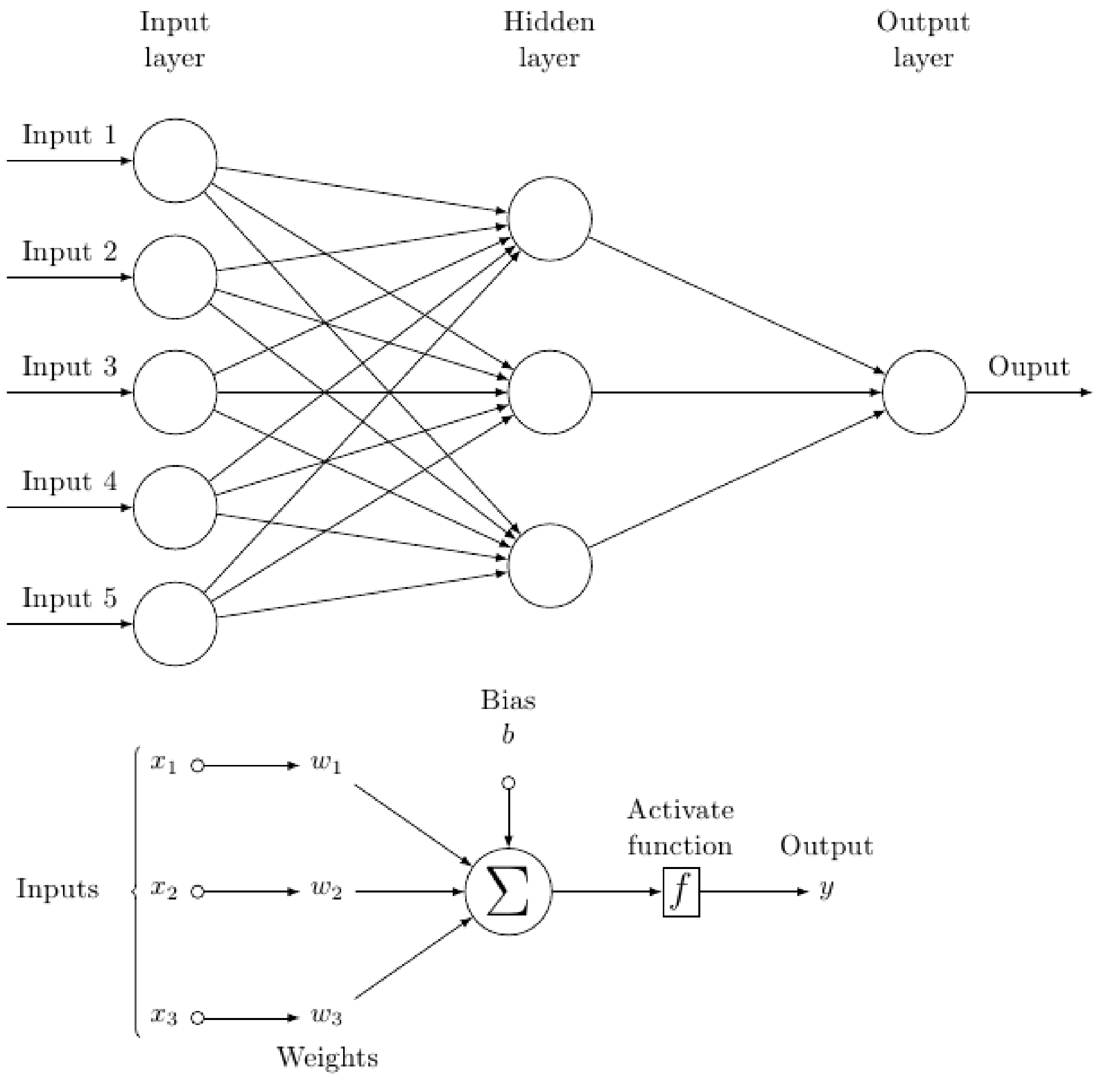


Figure 3. Dense layers in neural network.

In convolution layer, here is a concept of local perception: In the process of the human brain's recognition of a picture, not the whole picture is recognized at the same time. Even each feature in the picture is first perceived locally, and then the local operation is integrated at a higher level to obtain more information (Nielsen, 2015). The input picture is a color image (rgb three channels). After four-layer convolution kernel convolution operation, each layer of convolution kernel has its own biase value. The convolution kernel of each layer first calculates the r, g, and b channels separately, and then accumulates the calculated values of the r, g, and b channels respectively, and merges them into the final convolution layer at the corresponding position. So, the four convolutional layers finally get the four convolved layers. (Karpathy, 2015)

The principle of the convolutional layer is showing in the figure below.

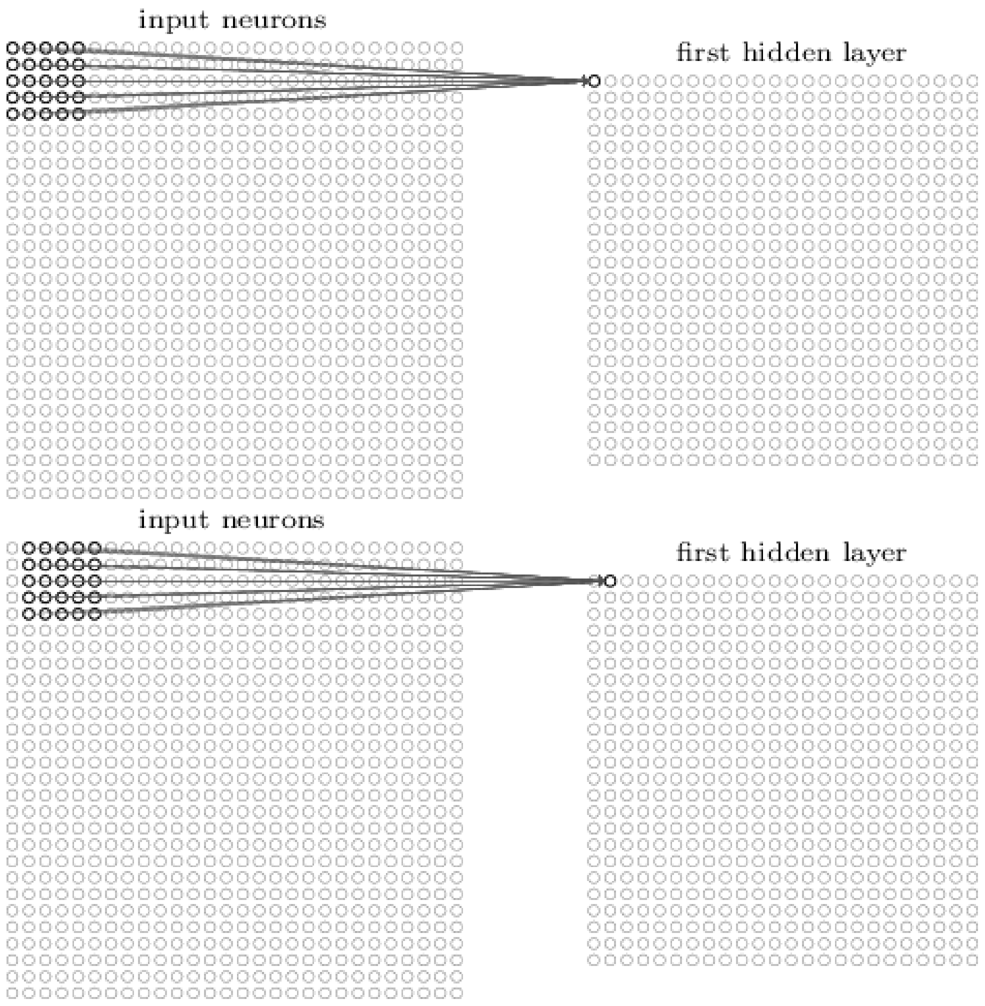


Figure 4. The first and second step of a 5 × 5 convolutional kernel working.

The pooling layer just performs simple operations on neurons in the window range, such as summing, maximizing, and using the obtained value as the input value of the neurons in the pooling layer. It was introduced to simplify the output of the convolutional layer.

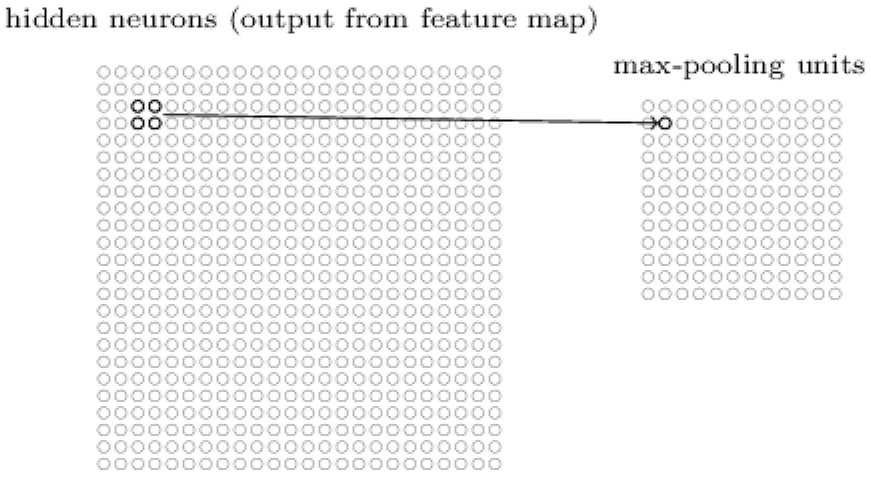


Figure 5. The pooling layer working method.

I use max pooling method in this project. The figure of pooling is showing in *Figure 5* and *Figure 6.*

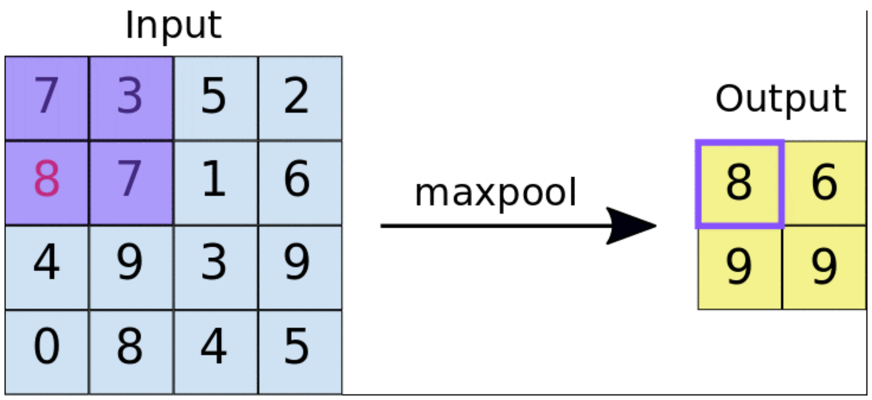


Figure 6. The max pooling layer working method.

The convolutional neural network continuously calculates forward and backward, and has been updating the appropriate weights, that is, it has been updating the convolution kernels. The convolution kernel is updated, and the learned features are also updated. A complete step of convolutional neural networks is updating the convolution kernel parameters (weights) is equivalent to constantly updating the extracted image features to obtain the most suitable features that can correctly classify the images.

Relu activation

Through the theory of biological neuron signal transmission, we learned that not every neuron will be activated in a signal transmission processing. Only activated neurons can transmit signals. So, this transmission process is not linear.

Consider this reason, after we built the CNN model, we need an activation function. If the activation function is not used, in this case each layer output is a linear function of the upper layer input. It is easy to verify that no matter how many layers the neural network has, the output is a linear combination of inputs. Therefore, a non-linear function is introduced as the activation function, so that the deep neural network makes sense, it is no longer a linear combination of inputs but can also approximate any function. In this way, neural networks can be applied to many non-linear models.

So, I use the Relu (rectified linear unit) activation as the activation function in this program. Relu activation can be defined as:

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

It's the same as $f(x) = max(0, x)$. I use MATLAB to draw the Relu activation function shows in the *Figure 1*.

To confirm the equation, use backward derivation process: Let the *l* layer output is z^l, and the output after the activation function is z^{l+1}; record the output of the loss function L on the *l* layer of the partial derivative of z^l is $\delta^l = \frac{\partial L}{\partial z^l}$ .The partial derivative of the loss function L with respect to the *l* layer is:

$$\delta^l = \frac{\partial L}{\partial z^{l+1}} \frac{\partial z^{l+1}}{\partial z^l} = \partial^{l+1} \frac{\partial Relu(z^l)}{\partial z^l} = \partial^{l+1} \begin{cases} 1 & z^l > 0 \\ 0 & z^l \leq 0 \end{cases}$$

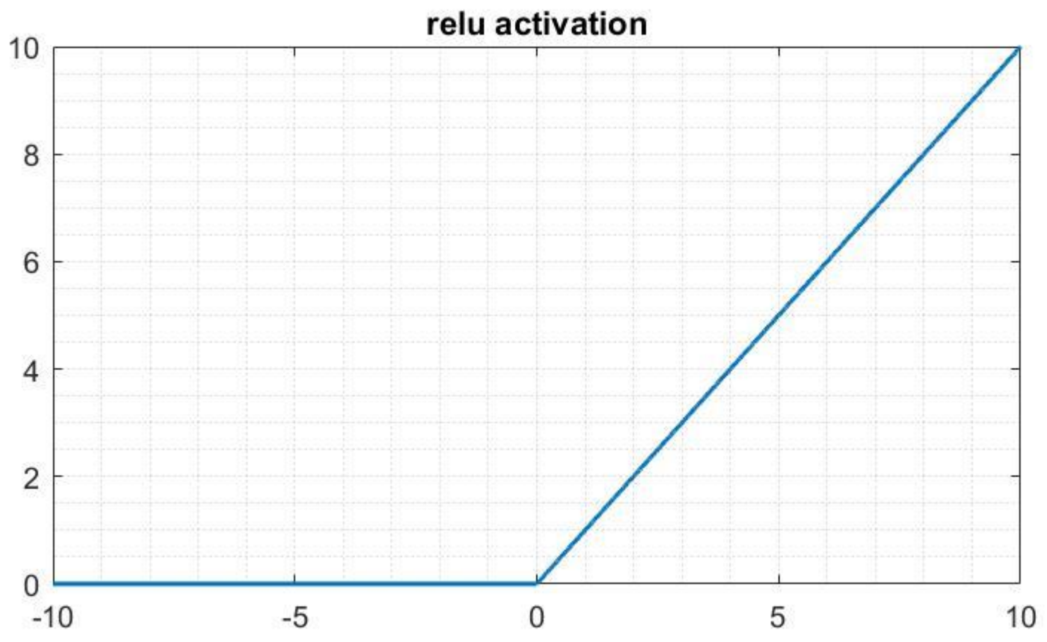


Figure 1. The Relu activation function.

As can be seen from the Relu function image, it is a piecewise linear function. All negative values are 0, and all positive values are unchanged. This operation is called unilateral suppression. Relu function will make the output of some neurons equals 0, which will cause the network to be sparse, and reduce the interdependence of parameters, and alleviate the problem of overfitting. Also, more satisfied with the interpretation of biological neurons. Varying the number of active neurons allows a model to control the effective dimensionality of the representation for a given input and the required precision (Glorot, Bordes, Bengio, 2011).

Sigmoid activation

The sigmoid function can be written as:

$$f(x) = \frac{1}{1 + e^{-wx+b}}$$

x is the input, w is the weight, and b is the offset(bias). When we change the weight w and offset b, we can construct multiple output possibilities for neurons. This is just one neuron. In a neural network, the combination of thousands of neurons can produce complex outputs mode. Because the domain of the sigmoid function is $(-\infty, +\infty)$, the range is (0, 1). Therefore, the most basic LR classifier is suitable for classifying two classification (class 0, class 1) targets. I draw the sigmoid function in MATLAB in *Figure 1*.

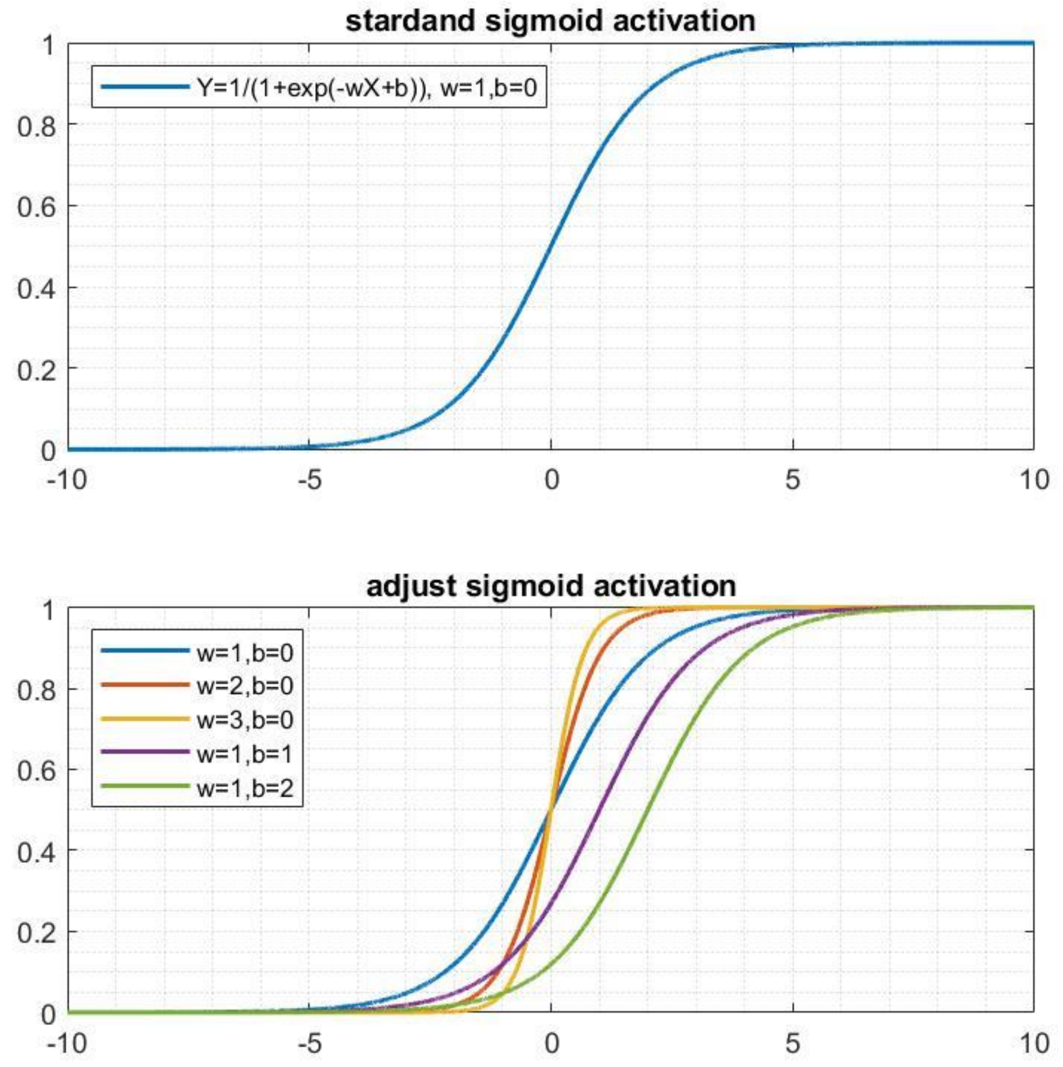
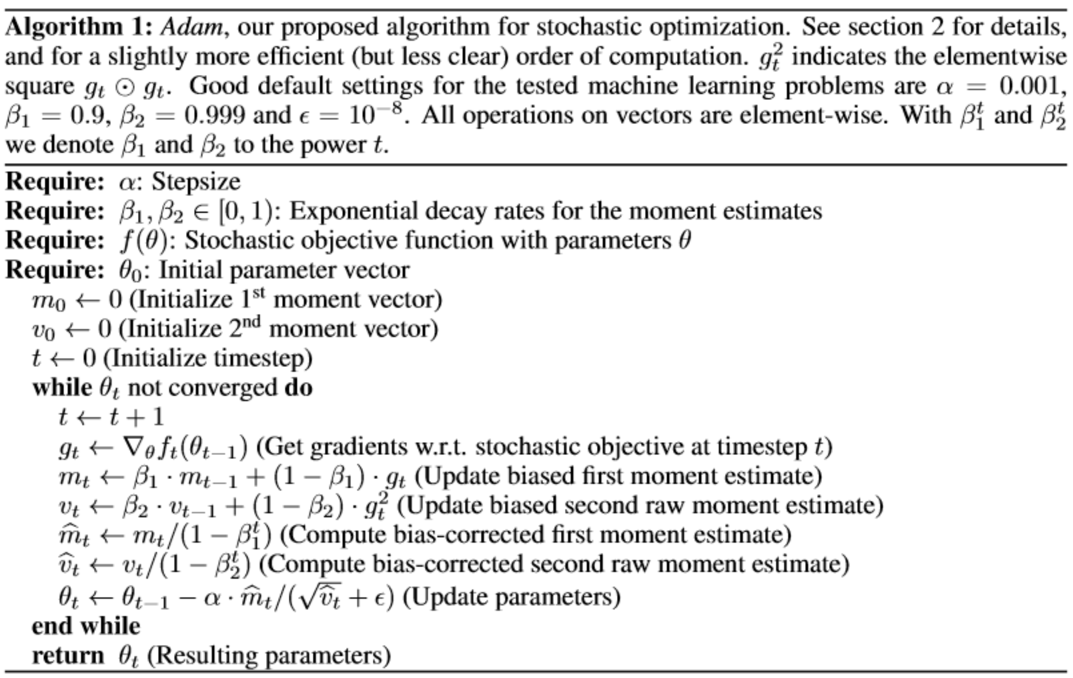


Figure 1. The sigmoid activation functions.

Adam algorithm

Adam algorithm which for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients (Diederik, Ba, 2014).

The Algorithm is showing below.



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Figure 1. The Adam algorithm.

In addition to storing an exponentially decaying average of past squared gradients v_t like AdaGrad (Duchi et al., 2011) and RMSprop (Tieleman & Hinton, 2012), Adam also keeps an exponentially decaying average of past gradients mt, like momentum (Ruder, 2016):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method.

As $m_t$and v_t are initialized as vectors of 0's

$$m_t = 0, \; v_t = 0$$

The authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

$$\beta_1 \to 1, \; \beta_2 \to 1$$

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\widehat{v}_t} + \epsilon} \widehat{m}_t$$

The efficiency of the algorithm can be improved by changing the calculation order, such as replacing the last three lines of the loop code in the pseudo code with the following two:

$$\alpha_t = \alpha \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha_t}{\sqrt{v_t} + \hat{\epsilon}} m_t$$

The code is tf.keras.optimizers.Adam(lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08)

The optimizer's default values are taken from references (Diederik, Ba, 2014).

lr: floating point number greater than or equal to 0, learning rate. It controls the update rate of weights (such as 0.001). Larger values will have faster initial learning before the learning rate is updated, while smaller values will allow training to converge to better performance.

beta1/beta2: floating numbers, 0 <beta <1, usually very close to 1.

beta1: exponential decay rate for first-order moment estimation (such as 0.9).

beta2: exponential decay rate for second-order moment estimation (such as 0.999). This hyperparameter should be set to a number close to 1 in sparse gradients.

epsilon: small floating number greater than or equal to 0 to prevent division by 0 errors.

Adam's update rule

An important feature of the Adam algorithm's update rule is that it carefully chooses the step size. Assuming $\epsilon = 0$, the effective descending step size in time step *t* and parameter space is:

$$\Delta_t = \alpha \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}}$$

There are two upper bounds on the effective descent step size, when

$$1 - \beta_1 > \sqrt{1 - \beta_2}$$

In this case, the upper bound of the effective step size satisfies:

$$|\Delta_t| \leq \alpha \frac{1 - \beta_1}{1 - \beta_2}$$

Satisfy in other cases:

$$|\Delta_t| \leq \alpha$$

The first case occurs only when it is extremely sparse: the gradient is zero except for the current time step which is not zero. In less sparse cases, the effective step size will become smaller. When:

$$1 - \beta_1 = \sqrt{1 - \beta_2}$$

We have:

$$\left|\frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}}\right| < 1$$

So, we can get the upper bound:

$$|\Delta_t| \leq \alpha$$

In more general scenarios, because:

$$\left|\frac{E[g]}{\sqrt{E[g]^2}}\right| \leq 1$$

We have:

$$\frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}} \approx \pm 1$$

The effective step size of each time step in the parameter space is approximately limited by the step size factor α, that is:

$$|\Delta_t| \approx< \alpha$$

This can be understood as determining a confidence region under the current parameter values, so it is better than current gradient estimates that do not provide enough information. This makes it relatively easy to know the correct range of α in advance.

For many machine learning models, for instance, we often know in advance that good optima are with high probability within some set region in parameter space; it is not uncommon, for example, to have a prior distribution over the parameters. Since α sets (an upper bound of) the magnitude of steps in parameter space, we can often deduce the right order of magnitude of α such that optima can be reached from θ_0 within some number of iterations. With a slight abuse of terminology, we will call the ratio:

$$\frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}}$$

signal-to-noise ratio/SNR (Diederik, Ba, 2014). If the SNR value is small, the effective step size Δ_t will be close to 0, and the objective function will also converge to the extreme value. This is a very satisfying property, because a smaller SNR means that the algorithm has greater uncertainty as to whether the $\widehat{m}_t$ direction matches the true gradient direction. For example, the SNR value tends to 0 near the optimal solution, so it will also have a smaller effective step size in the parameter space: a form of automatic annealing. The effective step size Δ_t is still invariant for gradient scaling. If we rescale the gradient g with the factor c, that is equivalent to rescaling $\widehat{m}_t$. With the factor c and $\widehat{v}_t$. With the factor c^2, and the scaling factor will be offset when calculating the SNR:

$$\frac{c \cdot \widehat{m}_t}{\sqrt{c^2 \cdot \widehat{v}_t}} = \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}}$$

Initialization bias correction

Adam utilizes initialization bias correction terms. We will here derive the term for the second moment estimate; the derivation for the first moment estimate is completely analogous.

Let g be the gradient of the stochastic objective *f*, and we wish to estimate its second raw moment (uncentered variance) using an exponential moving average of the squared gradient, with decay rate β_2. Let g_1,..., g_T be the gradients at subsequent timesteps, each a draw from an underlying gradient distribution $g_t \sim p(g_t)$. After we initialize the exponential moving average $v_0 = 0$ (zero vector), the update of the exponential moving average at time step *t* can be expressed as (Diederik, Ba, 2014):

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

g_t^2 indicates the elementwise square $g_t \odot g_t$, can be written as a function of the gradients at all previous timesteps:

$$v_t = (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2$$

Taking expectations of the left-hand and right-hand sides of the function above:

$$E[v_t] = E\left[(1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2\right] = E[g_t^2] \cdot (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} + \zeta = E[g_t^2] \cdot (1-\beta_2) + \zeta$$

If the true second moment $E[g_t^2]$ is static, then $\zeta = 0$. Otherwise, ζ can keep a small value, because we should choose the exponential decay rate β_1 to assign a small weight to the gradient of the exponential moving average. So, initializing the zero vector means that only (β_2^t) terms remain. We therefore divided the *z* term in Algorithm 1 to correct the initialization bias.

In case of sparse gradients, for a reliable estimate of the second moment one needs to average over many gradients by choosing a small value of β_2; however it is exactly this case of small β_2 where a lack of initialization bias correction would lead to initial steps that are much larger (Diederik, Ba, 2014).

Loss (Binary cross-entropy)

Cross-entropy is to determine how close the actual output is to the expected output. Because the project is a binary classifier, so we use binary cross-entropy as the loss function, the function of the binary cross-entropy is showing below. *N* =2.

$$J(w,b) = -\frac{1}{N}\sum_{i=1}^{N}[y_i\, log(\hat{y_i}) + (1 - y_i)log(1 - \hat{y_i})]$$

Where y is the label (1 for lake and 0 for airplane) and p(y) is the predicted probability of the point being lake for all *N* pictures. For each lake (y=1), it adds *-log* $(\hat{y_i})$ to the loss, the $\hat{y_i}$ grows large, that is, the log probability of it being lake. Conversely, it adds *-log (1-*y_i*)*, the $\hat{y_i}$ become smaller, that is, the log probability of it being airplane for each airplane (y=0).

Maximizing the log-likelihood function is equivalent to minimizing the negative log-likelihood function. Referring to the definition of cross-entropy, for a distributed random variable *X ~ p(x),* there is a model q(x) for approximating the probability distribution of p(x). Then the cross-entropy between the distribution X and the model q will be:

$$H(X,q) =-(xlog(q) + (1- x)log(1 - q))$$

It can be said that binary cross-entropy is a direct way to measure the difference between two distributions, or two models. The likelihood function is to explain the dataset by a distribution model with the output parameter. Therefore, it can be said that the two functions are "different explanation of the same appearance", but "the same way." Then we can get the cost function by using the binary cross-entropy. Binary cross-entropy measures how far away from the true value (which is either 0 or 1) the prediction is for each of the classes and then averages these class-wise errors to obtain the final loss. It is also known as log loss (Kevin, 2012).

Acc (Accuracy)

The definition of accuracy is for a given test data set, the ratio of the number of samples correctly classified by the classifier to the total number of samples.

It can be explained by confusion matrix below.

		Actual class	
		positive class(lake)	negative class(airplane)
Predicted class	positive class (lake)	True Positive (TP)	False Positive (FP)
	negative class (airplane)	False Negative (FN)	True Negative (TN)

Table 1. Standard confusion matrix.

We can calculate the accuracy of classification by using the function:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{\text{right data}}{\text{All data}}$$

Approach Elaboration

The steps of the approach elaboration are showing in *Figure 1*.

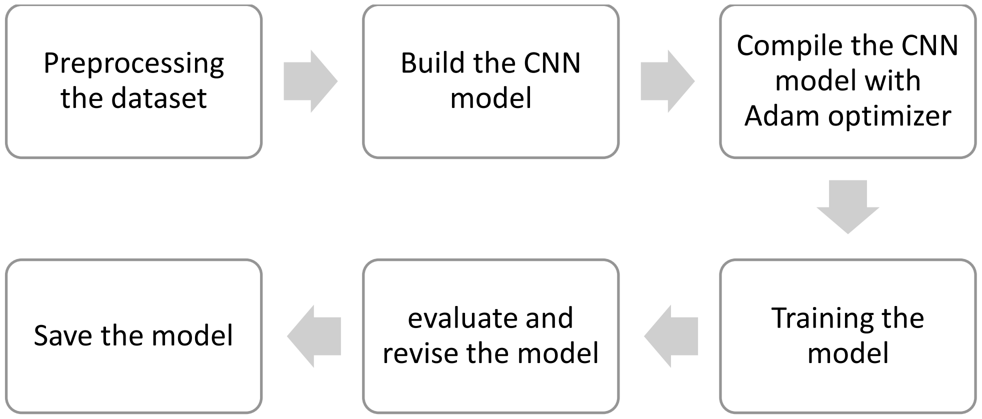
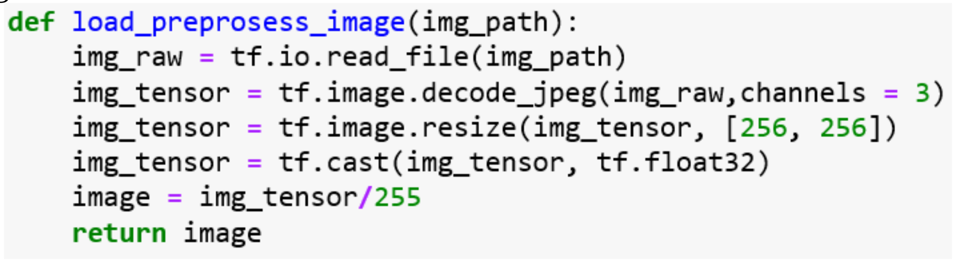

Preprocessing the dataset
Build the CNN model
Compile the CNN model with Adam optimizer
Training the model
evaluate and revise the model
Save the model

Figure 1. The steps of the elaboration.

1. Preprocessing the dataset.

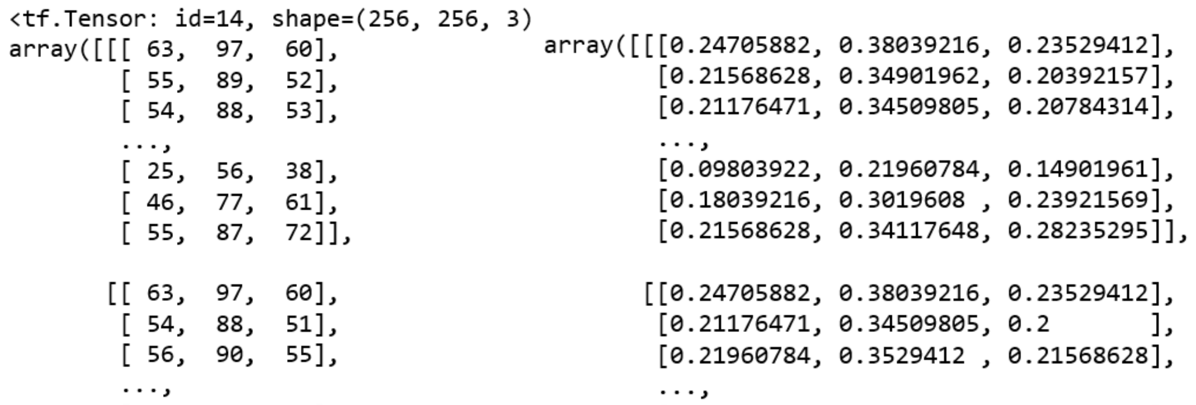
There are some key libraries I used in the Jupyter Notebook. The code is python 3.7: tensorflow, matplotlib.pyplot, numpy, pathlib, keras, random, IPython.display.

From the path of the pictures, I have constructed a preprocessing function, read the picture binary number, decode, convert the data type, and normalize to [0,1] range. The code and result are showing below.



```python
def load_preprosess_image(img_path):
    img_raw = tf.io.read_file(img_path)
    img_tensor = tf.image.decode_jpeg(img_raw,channels = 3)
    img_tensor = tf.image.resize(img_tensor, [256, 256])
    img_tensor = tf.cast(img_tensor, tf.float32)
    image = img_tensor/255
    return image
```

Figure 1. Preprocessing the image



```
<tf.Tensor: id=14, shape=(256, 256, 3)
array([[[ 63,  97,  60],
        [ 55,  89,  52],
        [ 54,  88,  53],
        ...,
        [ 25,  56,  38],
        [ 46,  77,  61],
        [ 55,  87,  72]],

       [[ 63,  97,  60],
        [ 54,  88,  51],
        [ 56,  90,  55],
        ...,
```

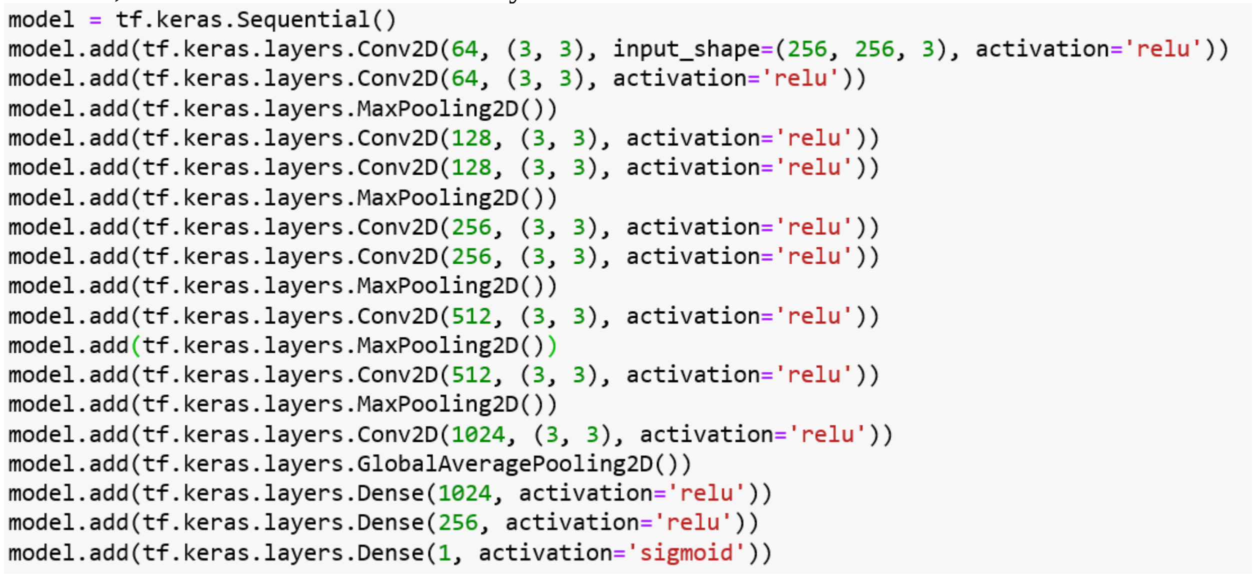
```
array([[[0.24705882, 0.38039216, 0.23529412],
        [0.21568628, 0.34901962, 0.20392157],
        [0.21176471, 0.34509805, 0.20784314],
        ...,
        [0.09803922, 0.21960784, 0.14901961],
        [0.18039216, 0.3019608 , 0.23921569],
        [0.21568628, 0.34117648, 0.28235295]],

       [[0.24705882, 0.38039216, 0.23529412],
        [0.21176471, 0.34509805, 0.2       ],
        [0.21960784, 0.3529412 , 0.21568628],
        ...,
```

Figure 1. Left: before preprocessing; Right: after preprocessing.

We can find that the images were normalized to [0,1] range. After that we zip image data and label data together and divide it into train dataset of 80% samples and test dataset of 20% samples.

2. Build the CNN model

Now, we will build a CNN model by the code below:



```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(64, (3, 3), input_shape=(256, 256, 3), activation='relu'))
model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(tf.keras.layers.Conv2D(128, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Conv2D(256, (3, 3), activation='relu'))
model.add(tf.keras.layers.Conv2D(256, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Conv2D(512, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Conv2D(512, (3, 3), activation='relu'))
model.add(tf.keras.layers.MaxPooling2D())
model.add(tf.keras.layers.Conv2D(1024, (3, 3), activation='relu'))
model.add(tf.keras.layers.GlobalAveragePooling2D())
model.add(tf.keras.layers.Dense(1024, activation='relu'))
model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

Figure 1. The code CNN model for classify.

This CNN model is a sequential model, the detailed structure can be obtained using model.summary(), the model is showing in *Figure 1*. From that, we can get the sequential of the CNN model. Through the neural network, we extract feature parameters and reduce the number of dimensions. Finally, we will train 10,717,249 parameters in this neural network.

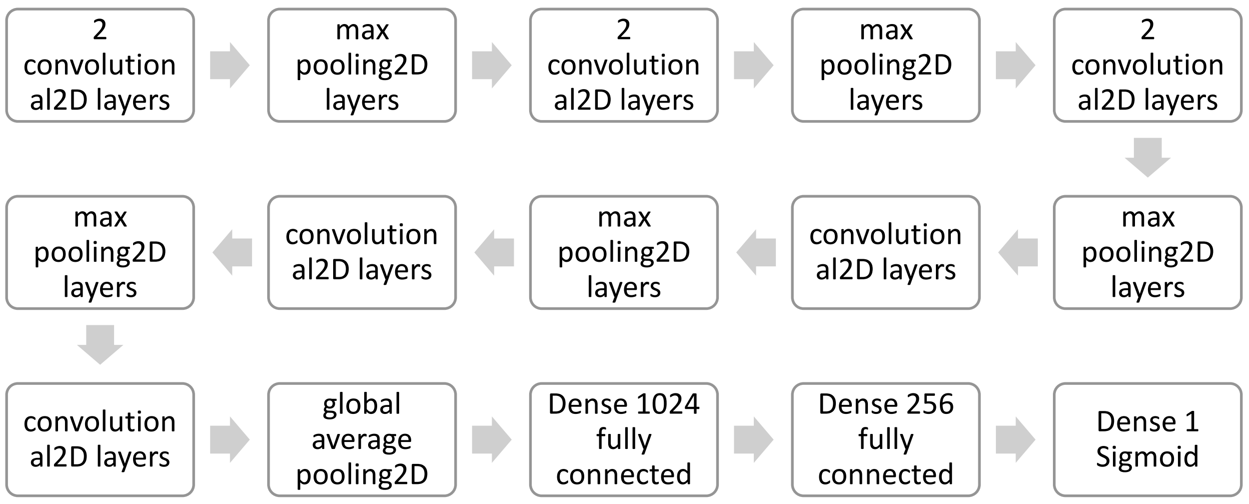
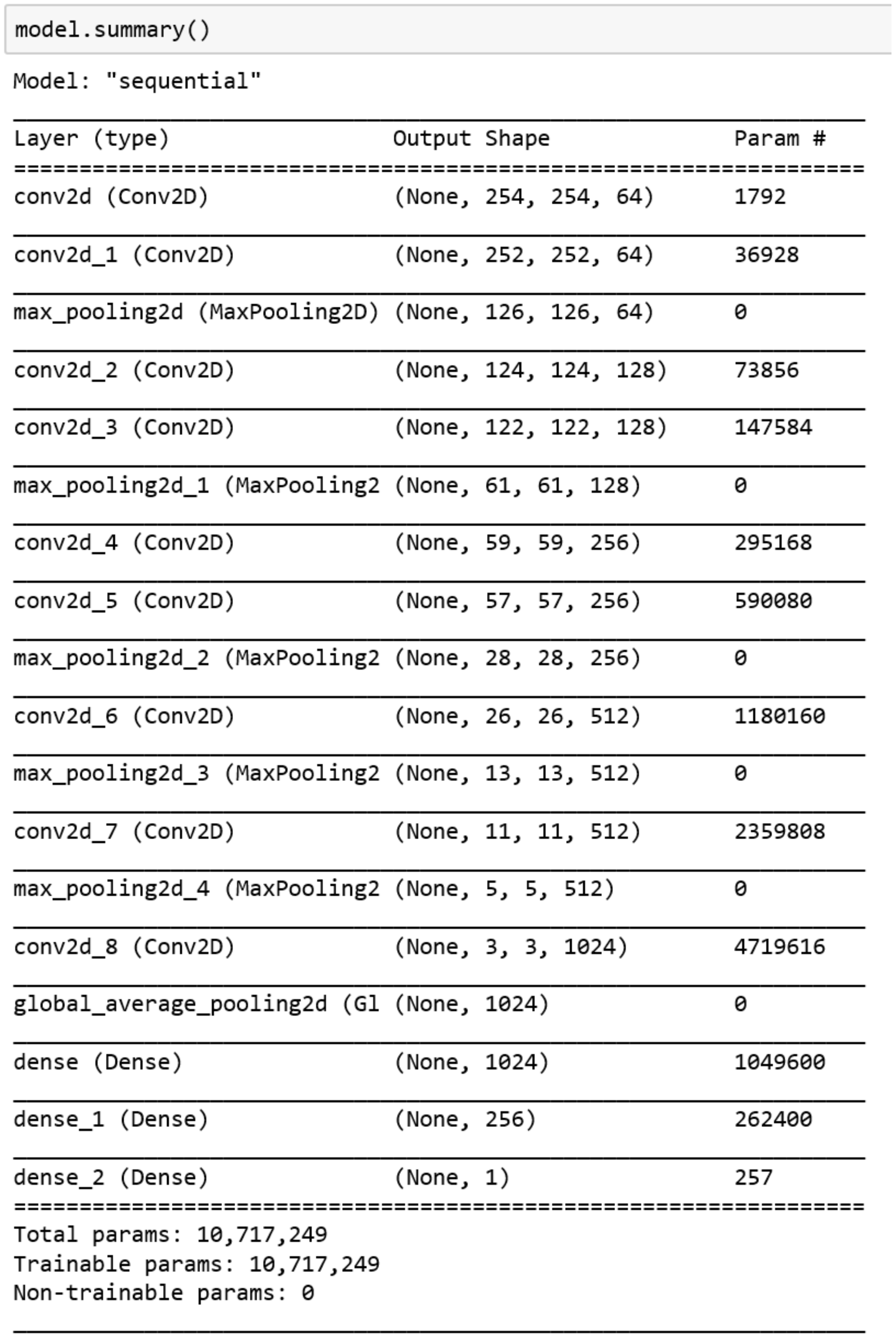


Figure 1. The sequential of the CNN model.



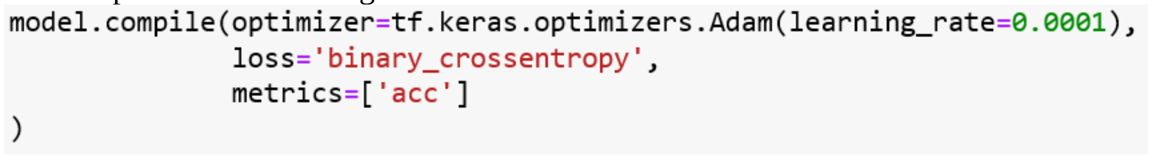
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 254, 254, 64)      1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 252, 252, 64)      36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 126, 126, 64)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 124, 124, 128)     73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 122, 122, 128)     147584
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 61, 61, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 59, 59, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 57, 57, 256)       590080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 26, 26, 512)       1180160
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 512)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 11, 11, 512)       2359808
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 5, 5, 512)         0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 3, 1024)        4719616
_________________________________________________________________
global_average_pooling2d (Gl (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              1049600
_________________________________________________________________
dense_1 (Dense)              (None, 256)               262400
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 10,717,249
Trainable params: 10,717,249
Non-trainable params: 0
_________________________________________________________________
```

Figure 1. The structure of project CNN model.

3. Compile the CNN model with Adam optimizer

We just built a CNN model structure; we need compile it to make it work.
The compile code is showing below.



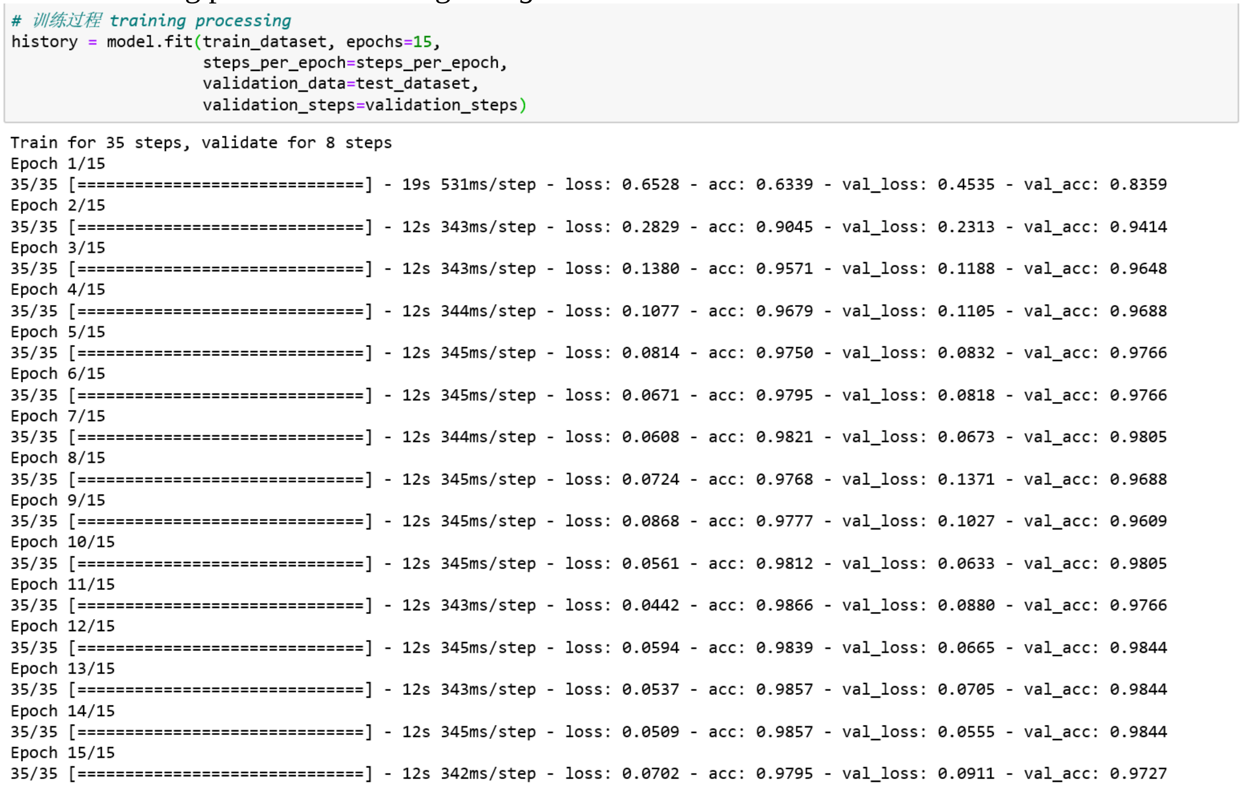
```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
              loss='binary_crossentropy',
              metrics=['acc']
)
```

Figure 15. The code of compile the CNN model.

From the introduction part, we use a small learning rate of 0.0001 here because the neural network is very large. A small learning rate can make sure we don’t miss any local minima; but it could also mean that it will take us a long time to converge. However, the dataset is not big, and I have a good Graphics card to train the model, so the time cost is not so high.

4. Training the model

The training process is showing in *Figure 15.*



```
# 训练过程 training processing
history = model.fit(train_dataset, epochs=15,
                    steps_per_epoch=steps_per_epoch,
                    validation_data=test_dataset,
                    validation_steps=validation_steps)
```

```
Train for 35 steps, validate for 8 steps
Epoch 1/15
35/35 [==============================] - 19s 531ms/step - loss: 0.6528 - acc: 0.6339 - val_loss: 0.4535 - val_acc: 0.8359
Epoch 2/15
35/35 [==============================] - 12s 343ms/step - loss: 0.2829 - acc: 0.9045 - val_loss: 0.2313 - val_acc: 0.9414
Epoch 3/15
35/35 [==============================] - 12s 343ms/step - loss: 0.1380 - acc: 0.9571 - val_loss: 0.1188 - val_acc: 0.9648
Epoch 4/15
35/35 [==============================] - 12s 344ms/step - loss: 0.1077 - acc: 0.9679 - val_loss: 0.1105 - val_acc: 0.9688
Epoch 5/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0814 - acc: 0.9750 - val_loss: 0.0832 - val_acc: 0.9766
Epoch 6/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0671 - acc: 0.9795 - val_loss: 0.0818 - val_acc: 0.9766
Epoch 7/15
35/35 [==============================] - 12s 344ms/step - loss: 0.0608 - acc: 0.9821 - val_loss: 0.0673 - val_acc: 0.9805
Epoch 8/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0724 - acc: 0.9768 - val_loss: 0.1371 - val_acc: 0.9688
Epoch 9/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0868 - acc: 0.9777 - val_loss: 0.1027 - val_acc: 0.9609
Epoch 10/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0561 - acc: 0.9812 - val_loss: 0.0633 - val_acc: 0.9805
Epoch 11/15
35/35 [==============================] - 12s 343ms/step - loss: 0.0442 - acc: 0.9866 - val_loss: 0.0880 - val_acc: 0.9766
Epoch 12/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0594 - acc: 0.9839 - val_loss: 0.0665 - val_acc: 0.9844
Epoch 13/15
35/35 [==============================] - 12s 343ms/step - loss: 0.0537 - acc: 0.9857 - val_loss: 0.0705 - val_acc: 0.9844
Epoch 14/15
35/35 [==============================] - 12s 345ms/step - loss: 0.0509 - acc: 0.9857 - val_loss: 0.0555 - val_acc: 0.9844
Epoch 15/15
35/35 [==============================] - 12s 342ms/step - loss: 0.0702 - acc: 0.9795 - val_loss: 0.0911 - val_acc: 0.9727
```

Figure 15. The training process of CNN models. The best result with 15 epochs.

I trained the model for 15 epochs to get the best result. It takes about 3 minutes time for training by GPU. However, if I use CPU for training, it will take 260 second per epoch. The whole training time will be more than one hour. I have trained the model for 30 epochs, and 20 epochs either. It spent more times for training.

5. Evaluate and revise the model

For the training model. I have drawn the accuracy, validation accuracy, loss and validation loss in different training epochs. The 30, 20 and 15 epochs' figures are comparing below.

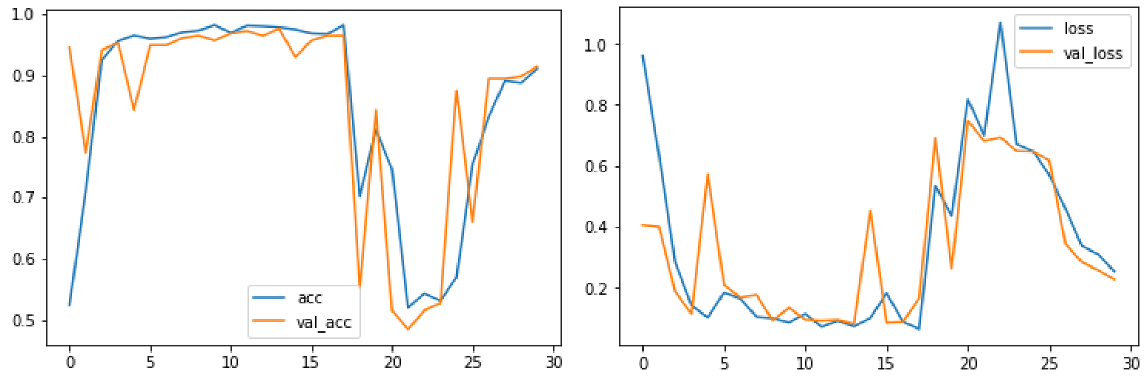


Figure 15. 30 epoch accuracy, validation accuracy, loss and validation loss, learning rate=0.001

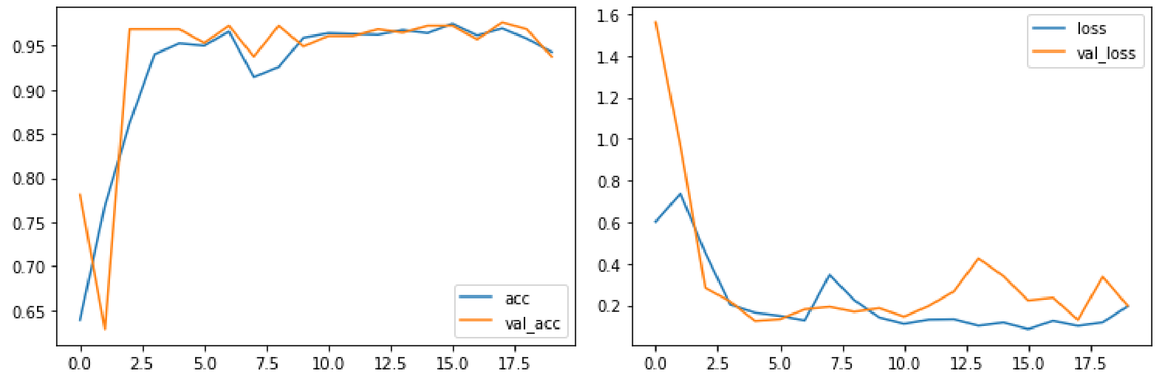


Figure 15. 20 epoch accuracy, validation accuracy, loss and validation loss, learning rate=0.001

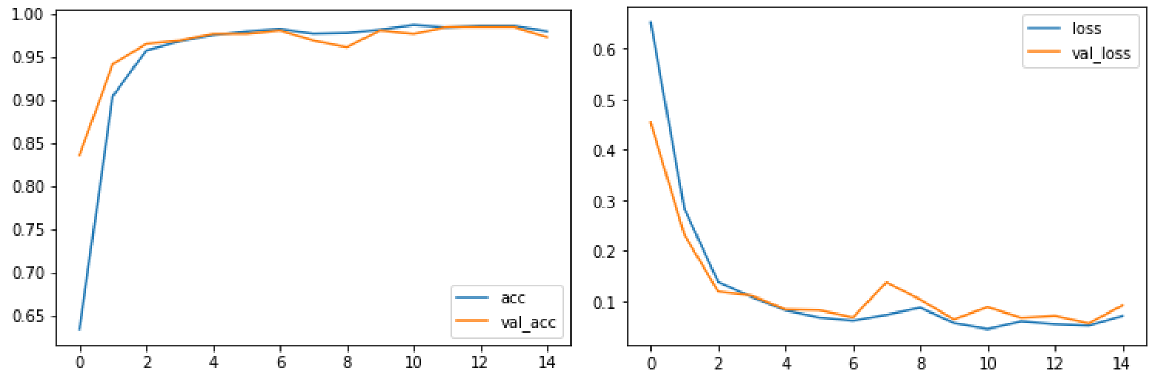
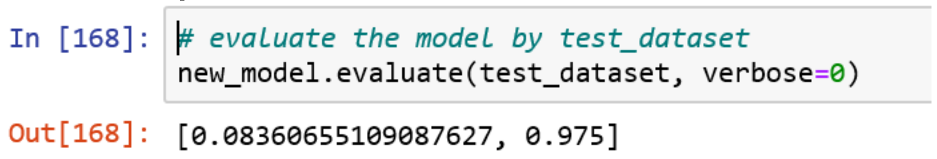


Figure 15. 15 epoch accuracy, validation accuracy, loss and validation loss, learning rate=0.0001

At first, I trained the model by 30 epochs, and in the *Figure 15*, it shows that overfitting after about 20 epochs. So, I reduce the training epoch to 20 and try again. This time the result in *Figure 15* is better, but there also have a little overfitting at the last, and the learning curve is not stable. It has large fluctuations. This should be caused by the high learning rate of 0.001. Thus, I lowered the learning rate and reduced the epoch again for training. Currently, I get the result in *Figure 15*. This time, the validation accuracy and validation loss become smoother and no overfitting occurred.

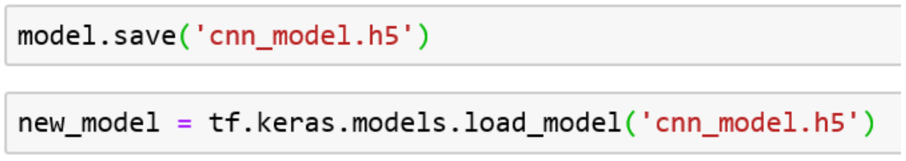
It looks like we got a very nice model. Finally, I used the evaluate command to perform an overall evaluation of the model in test dataset. In the *Figure 15*, the model has a 97.5% validation accuracy and a 0.0836 loss.



```
In [168]: # evaluate the model by test_dataset
          new_model.evaluate(test_dataset, verbose=0)

Out[168]: [0.08360655109087627, 0.975]
```

Figure 15. Evaluate the model.

6. Save the model



```
model.save('cnn_model.h5')
```

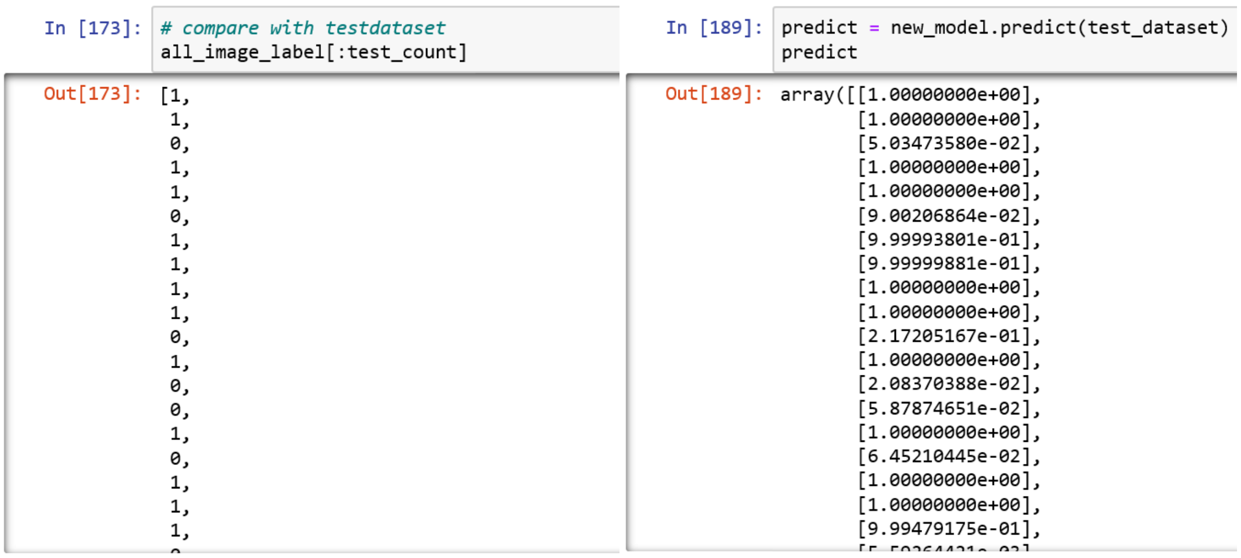
```
new_model = tf.keras.models.load_model('cnn_model.h5')
```

Figure 15. Save and load model.

By using the code in Figure 15. We can get our trained CNN model. It can be used for test and predict which class a picture will be. The test result is showing in next part.

Result and Conclusions

From the trained CNN model. We can get the predict class for all test data showing below. And we can compare with the true class showing in the *Figure 15.*



```
In [173]: # compare with testdataset
          all_image_label[:test_count]

Out[173]: [1,
           1,
           0,
           1,
           1,
           0,
           1,
           1,
           1,
           1,
           0,
           1,
           0,
           0,
           1,
           0,
           1,
           1,
           1,
```

```
In [189]: predict = new_model.predict(test_dataset)
          predict

Out[189]: array([[1.00000000e+00],
                 [1.00000000e+00],
                 [5.03473580e-02],
                 [1.00000000e+00],
                 [1.00000000e+00],
                 [9.00206864e-02],
                 [9.99993801e-01],
                 [9.99999881e-01],
                 [1.00000000e+00],
                 [1.00000000e+00],
                 [2.17205167e-01],
                 [1.00000000e+00],
                 [2.08370388e-02],
                 [5.87874651e-02],
                 [1.00000000e+00],
                 [6.45210445e-02],
                 [1.00000000e+00],
                 [1.00000000e+00],
                 [9.99479175e-01],
```

Figure 15. Compare with true class and predict class. The left is true class, the right is the predict class.

The value in the predict class can be divided in two class from 0.5. Under the 0.5 it will be 0, represents the airplane. Over the 0.5 it will be 1, represents the lake.

We choose some of pictures to validate the model. The result shows in *Figure 15*, *Figure 15*, and *Figure 15.*

From the result in the three-test example, we can find they all be classified in the right class. Because the model we trained has a very high accuracy at 97.5%. Thus, the test result is justified. The Adam optimizer that optimizes this model performs well. It works for this CNN model.

Although we get this amazing accuracy rate from the Adam optimizer, it looks perfect. This result requires more data for validation. In the future, we can further verify its performance by using the Adam optimizer for more neural networks

```
In [190]: predict[10],all_image_label[10]

Out[190]: (array([0.21720517], dtype=float32), 0)

In [192]: image_path1 = all_image_path[10]
          plt.imshow(load_preprosess_image(image_path1))

Out[192]: <matplotlib.image.AxesImage at 0x249f538ae88>
```

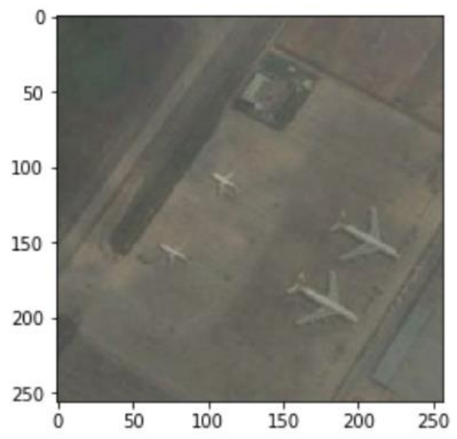


Figure 15. The No.10 pictures in test set. It has a predict value of 0.21720517, below the 0.5, the predict result is airplane.

```
In [193]: predict[100],all_image_label[100]

Out[193]: (array([0.06937873], dtype=float32), 0)

In [194]: image_path1 = all_image_path[100]
          plt.imshow(load_preprosess_image(image_path1))

Out[194]: <matplotlib.image.AxesImage at 0x249f53f5788>
```

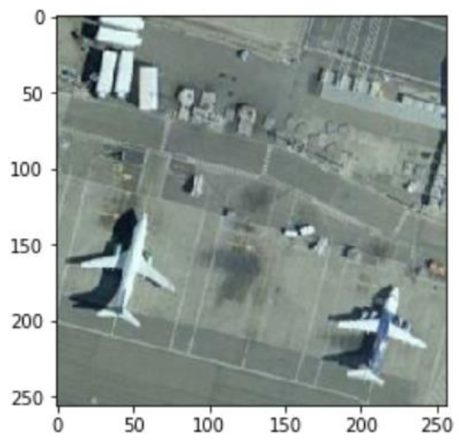
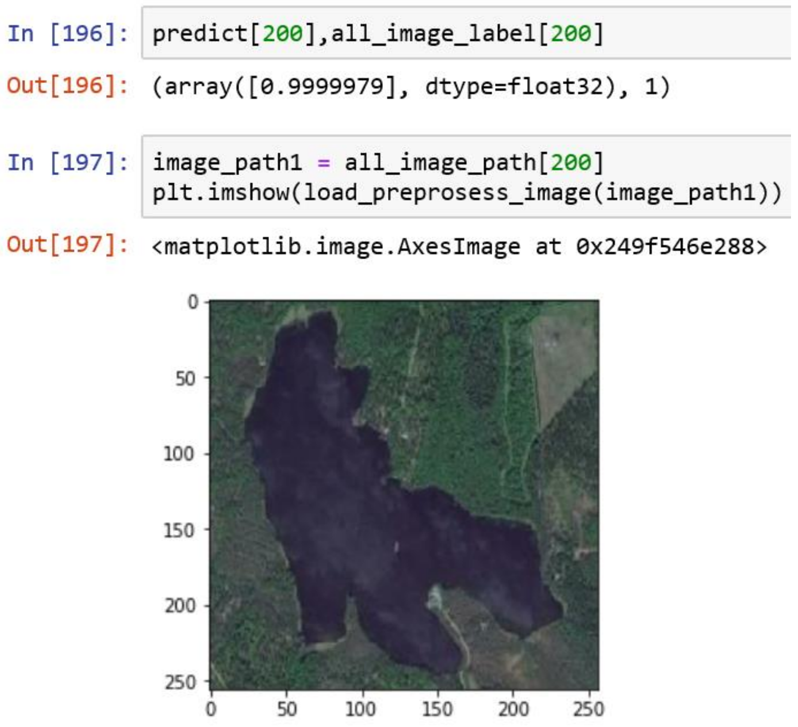


Figure 15. The No.100 pictures in test set. It has a predict value of 0.06937873, below the 0.5, the predict result is airplane.



```
In [196]: predict[200],all_image_label[200]

Out[196]: (array([0.9999979], dtype=float32), 1)

In [197]: image_path1 = all_image_path[200]
          plt.imshow(load_preprosess_image(image_path1))

Out[197]: <matplotlib.image.AxesImage at 0x249f546e288>
```

Figure 15. The No.200 pictures in test set. It has a predict value of 0.9999979, over the 0.5, the predict result is lake.

Acknowledgement

ECE6865 is a great course that tells us the theory and method of statistical signal estimation. Such as minimum variance unbiased estimation, Cramer-Rao bound, Bayesian estimators, maximum likelihood estimation. Professor Milos is a very academically rigorous person. He made meticulous corrections for each of our homework. I'm so appreciate for his patience even though I did not perform well in the course. Moreover, I also would like to thank my classmates for the help from them.

References

Diederik, Kingma; Ba, Jimmy (2014). "Adam: A method for stochastic optimization". arXiv:1412.6980

Duchi,John,Hazan,Elad,and Singer,Yoram (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Karpathy, Andrej. 2015. “Neural Networks Part 1: Setting Up the Architecture.” Notes for CS231n Convolutional Neural Networks for Visual Recognition, Stanford University. http://cs231n.github.io/neural-networks-1/

Murphy, Kevin (2012). *Machine Learning: A Probabilistic Perspective*. MIT. ISBN 978-0262018029.

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). San Francisco, CA, USA:: Determination press.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Tieleman, T. and Hinton, G. Lecture 6.5 - RMSProp, COURSERA (2012): Neural Networks for Machine Learning. *Technical report*.

Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). Deep sparse rectifier neural networks. *AISTATS*.

Appendix

Get the full code on my GitHub:
https://github.com/jerrylance/Statistical-Signal-Estimation-/tree/master/final%20project
File **cnn satellite image recognition.ipynb** write by python for CNN.
File **generate_finalproject_figures.m** write by MATLAB for some concept's figures.