2018

Image Series Prediction Via Convolutional Recurrent Neural Networks With Limited Training Data

Zao Zhang
Binghamton University--SUNY, zzhan126@binghamton.edu

IMAGE SERIES PREDICTION VIA CONVOLUTIONAL RECURRENT NEURAL NETWORKS WITH LIMITED TRAINING DATA

BY

ZAO ZHANG

BS, Yanshan University of China, 2015
MS, Bingamton University, 2018

THESIS

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in Electrical and Computer Engineering
in the Graduate School of Binghamton University
State University of New York
2018

Accepted in partial fulfillment of the requirements for
the degree of Master of Science in Electrical and Computer Engineering
in the Graduate School of
Binghamton University
State University of New York
2018

April 27, 2018

Professor. Xiaohua Li,
Department of Electrical and Computer Engineering, Binghamton University

Professor. Fowler Mark,
Department of Electrical and Computer Engineering, Binghamton University

Professor. Kirchner Matthias
Department of Electrical and Computer Engineering, Binghamton University

Abstract

Deep learning has become increasingly popular due to its wide applications in many areas such as computer vision, speech recognition, natural language processing, text processing, etc. Most of the deep learning studies are conducted over hand-labeled and well-designed large data sets, such as ImageNet, CIFAR, MNIST, Labeled Faces in the Wild, etc. One of the major challenges for the practical application of deep learning is that large-scale training data sets are very difficult to construct. It is extremely costly and timing consuming to collect and hand-label a lot of training data. There may not have such large-scale training data sets available in many practical situations. In these cases, it becomes important to investigate the performance of deep learning when only a limited amount of training data are available.

This thesis focuses on developing deep learning algorithms that can be used to forecast the image series under limited training data. Specifically, we study the problem of using a pine tree is existing appearance images to predict its future appearance images. Because it is not easy to obtain a sufficient number of photos of real pine trees with designated ages, we apply the software tool L-studio to generate images of pine trees. L-studio is a standard software that has been used widely in botanical research community to create plant simulation models. With L-studio we can conveniently generate various plant images. Based on the image series of simulated pine trees, we develop a convolutional recurrent neural network (CRNN) model to predict the images of pine trees at different ages. Especially, in order to study the performance of deep learning under limited training data, we intentionally limit the total number of image series in our artificially generated data set.

As major contributions of this thesis, first, we worked out a way to use MATLAB to control the operation of the legacy software L-studio, which makes it convenient to generate a lot of desired images automatically. It would be impossible to generate thousands of labeled pine tree images if operating L-studio and generating each image by hand. Second, we constructed convolutional recurrent neural network (CRNN) model to conduct image series forecasting, i.e., to predict and generate the images of pine trees in the future years according to their images in the past years. We implemented the model with the Keras Library in Python. Finally, we used extensive experiments to tune the CRNN so as to optimize its performance under our short training data set. Importantly, we studied thoroughly the impact of a lot of tunable CRNN parameters to the CRNN performance when the training data amount is not big enough.

At the end of this thesis, we give a conclusion and point out some potential future research directions.

Acknowledgement

I would like to thank many people, without whom I cannot finish this thesis. First and foremost, I would like to thank Professor Xiaohua Edward Li, whose guidance helps a lot in my entire thesis work. Second, I would like to thank Professor Kirchner Matthias and Professor Mark Fowler, not only for their service on my thesis committee, but also for their excellent teaching of my graduate courses. I learned a lot from their classes and used such knowledge in this thesis. Finally, I would like to thank my friends and my family. They have given me confidence and happiness during my graduate study.

Table of Contents

List of Tables

List of Figures

I. Introduction

1.1 Background of deep learning

The concept of deep learning is originated from the evolution of machine learning research, the availability of large-scale training data sets, as well as the increased computation power. Deep learning is a class of feature learning techniques that exploit multilayer artificial neural networks [1]. The major ways for human being to process information obtained from natural life are more or less feature learning as well. Human being also conducts learning via neural networks, albeit the biological neural network inside the brain. Brain scientists have been studying this kind of information processing in order to understand human being's capability of receiving and processing information. What deep learning investigators do is just like what brain scientists do. Their objective is to develop deep learning methods that are able to make computers process data and natural information much better.

Deep learning is mainly realized via deep neural networks (DNN). There are many different DNN architectures such as deep feedforward networks (DFN), convolutional neural networks (CNN), recurrent neural networks (RNN), deep belief networks (DBN), autoencoders, etc [1]. In this thesis, we will study a special architecture called convolutional recurrent neural network (CRNN), which is a combination of CNN and RNN [2][3].

Deep learning can also be classified according to whether they are trained by supervision or not. Supervised DNN requires labeled training data sets to train, while unsupervised DNN can exploit unlabeled data. The CRNN we used in this thesis is a supervised DNN. A brief introduction to supervised learning, convolutional neural networks and recurrent neural networks will be given in Section II.

Although the specific name of “deep learning” was coined just about a decade ago, deep learning has achieved profound progress and has found many very successful practical applications in computer version, speech recognition, natural language processing, and even entertainment fields such as arts, music, games, etc. There are many deep learning-based practical products emerging on the market, e.g., real-time speech recognition, target recognition, self-driving vehicles, language translation, etc. They are playing important roles to the living and working of human being. The quick and dramatic success of deep learning has motivated many more people to conduct further research in deep learning.

1.1.1 Supervised learning

Since deep learning is a subarea within the domain of machine learning, many concepts of machine learning can be applied to deep learning, such as the concept of supervised learning and unsupervised learning.

The learning is supervised or not means whether labeled training data are used in the training of the learning algorithm. Labeled training data means that the training data have been prepared and processed by human being, i.e., the meaning of the data is labeled, usually by hand. For example, if we want to make a classification between some animals, such as dog, cat, bird, fish, etc., we can give the learning algorithm a huge set of images of these animals. Each of these images has already been provided with a label indicating which animal it is. The labeling work usually can only be conducted by hand. Then learning algorithm will try to find some special features among those images and try to classify them so as to fit the labels in the training data set. This kind of deep learning is supervised learning. We are dealing with supervised learning only in this thesis.

If we just provide the learning algorithms with images without labels, then the learning algorithms have to conduct learning by looking for the difference among the images, which may be the inherent structural differences of the images. With such relatively coarse information, the learning algorithms may conduct learning tasks such as clustering, dimension reduction, etc. This is called unsupervised learning. Between supervised learning and unsupervised learning, there is semi-supervised learning. As the name suggests, semi-supervised learning means that some (usually a small set of) data come with labels and the others have no labels.

One of the most important supervised learning tasks is classification. There are many popular classification algorithms, such as neural network (NN) based algorithms, support vector machines (SVM), logistic regression, naïve Bayes, k-nearest neighbor algorithm, and so on. We will focus on the NN based supervised classification in this thesis.

1.1.2 Convolutional neural networks

Convolutional neural networks are special neural network architectures that are especially suitable for processing image data. Besides their superior performance in image processing, they can also be used to process other types of data, such as text, speech, audio, etc. For example, convolutional neural network is a building block of the Alpha Go artificial intelligence algorithm that has beat the most famous human Go players.

Convolutional neural network architectures are usually built with the following layers: convolution layer, rectified linear units (ReLU) layer, pooling layer, fully connected layer and loss layer [1].

1.1.2.1 Convolution layer

The convolution layer uses a batch of multi-dimensional filters to filter the input images so as to extract image features. The multi-dimensional filters process the images in blocks each time instead of processing each image pixel individually. This way of data processing not only can keep the continuity information of the pixels, but also can capture the embedded features in the images. The filter parameters (weights) and the abstract image features are learned automatically during training. As a result, time-consuming hand-based feature design is not needed.

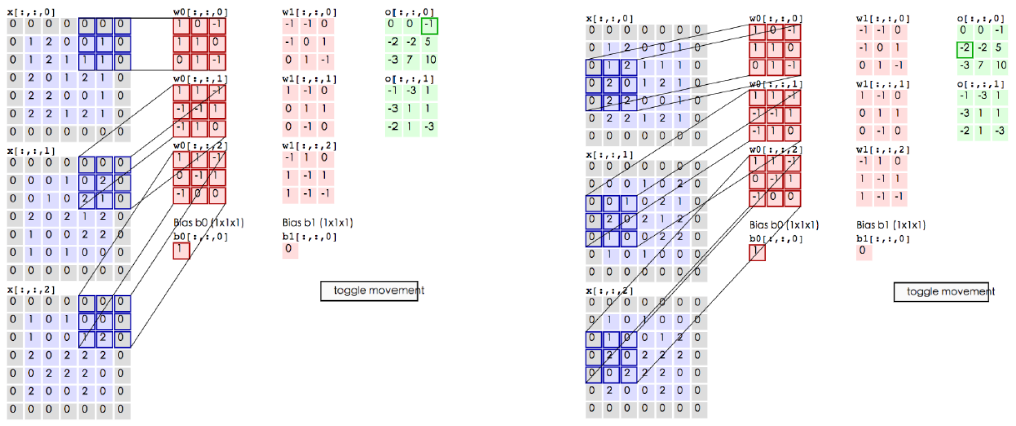


Figure 1. Illustration of the convolution operation of CNN [4].

Fig. 1 illustrates the filtering procedure of the convolution layer. Three dimensional filters with size $3 \times 3 \times 3$ are shown. Each filter is used to filter the whole image and to generate a feature map (another image) as the output. In practical applications, a convolutional neural network always has more than one convolution layers, and each convolution layer has a lot of such filters.

1.1.2.2 Rectified linear unit (ReLU) layer

This layer provides the necessary nonlinear activation function that is needed for neural network operations. The rectified linear unit (ReLU) function is a popular choice in convolutional neural networks. Sometimes this layer is modeled into the convolution layer as an integrated component rather than an individual layer. Appropriate nonlinear activation functions can greatly improve the performance of deep neural networks. Besides the ReLU function, commonly used activation functions include the hyperbolic tangent function (tanh), the sigmoid function, etc.

1.1.2.3 Pooling layer

Usually we will have a lot of convolution filters in each convolution layer because we want the convolutional neural networks to learn as many abstract image features as possible. Each filter will output a feature map which is just another image. This means that the output data amount will increase rapidly after each convolution layer. This also means that the output data are highly redundant. Obviously, we can apply some decimation so as to reduce redundancy and to reduce the amount of data, which is also good for computational complexity reduction. Pooling layers are designed specifically for this purpose in convolutional neural networks. The essence of pooling is just subsampling or decimation. There are many different image subsampling techniques developed for the pooling layer, among which max pooling is perhaps the most used one in convolutional neural networks. Because max pooling's performance is usually much better than other pooling methods such as average pooling, we will use max pooling as the pooling method in our design.

Max pooling subdivides an image into small blocks, and outputs the max value of each small block. We do not need to worry about the loss of accurate position of this max-value pixel during max pooling because in image processing, the connection between pixel values is much more important than the precise locations of these values in the image. The size of the small block is the pooling size, which also determines the decimation ratio. Since large pooling sizes will lose data too fast, we usually use small pooling sizes such as 2 to 4. When the amount of training data is limited, the effects of pooling should be studied more carefully, for which our results will be discussed in Section IV.

Fig. 2 illustrates the pooling of a three-dimensional data tensor, and the max pooling of a two-dimensional array.

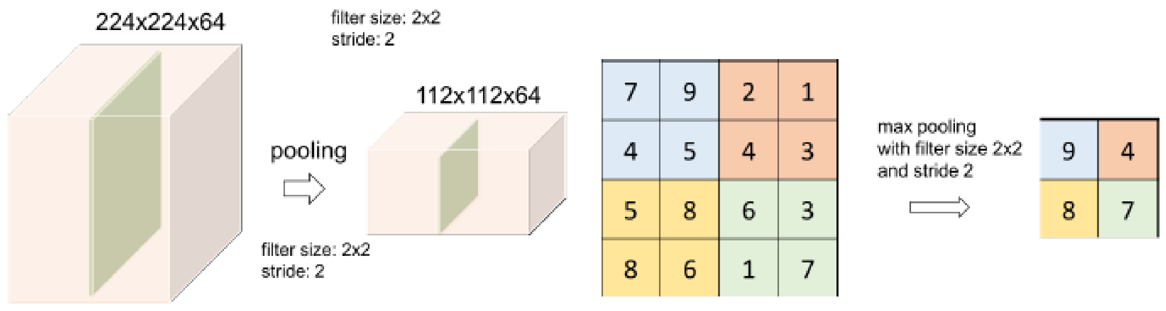


Figure 2. Pooling of data tensors. Pooling size (stride) 2 keeps just a quarter of the input data in both cases (Adapted from lecture notes of EECE680C Neural Networks & Deep Learning)

1.1.2.4 Fully connected layer

A typical convolutional neural network usually has multiple convolution layers and pooling layers. After convolution and pooling, it usually uses a fully connected layer to transform the feature map data into a data vector for subsequent processing such as information extraction and classification. This layer is implemented just as a single-layer fully connected neural network, where we vectorize the feature maps (or, reshape the multi-dimensional tensor into a one-dimensional vector), multiply the resulted vector with a weighting matrix, add bias and pass the result through a nonlinear activation function. One convolutional neural networks may have one or multiple fully connected layers. Because the size of the reshaped vector is usually big and the fully connected layer is a dense neural network, the total number of adaptable weighting parameters (i.e., weighting matrix and bias) is usually a huge number. As a matter of fact, for many convolutional neural networks, the majority of trainable weight variables are in the fully connected layers.

1.1.2.5 Loss function layer

This layer is used to calculate the distance between the predicted labels and true labels. According to different learning tasks, we may choose different loss functions. For example, Mean-squared-error loss can be used in regression. Softmax cross-entropy loss can be used in classification. Sigmoid cross-entropy loss is usually used in dichotomous task. The training task for convolutional neural networks is to optimize the weighting variables so as to minimize the loss function. In this thesis, because our objective is to generate new images, we use mean-squared-error loss between the predicted images and the true images as our loss function.

1.1.2.6 Classical CNN architectures

There are many classical convolutional neural network architectures that have been studied extensively and applied widely, such as the architectures shown in Fig. 3 and Fig. 4.

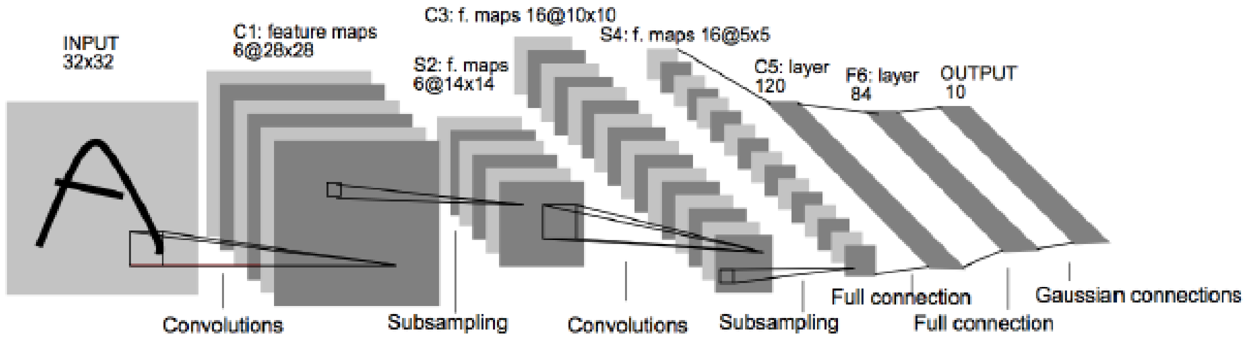


Figure 3. LeNet architecture [1]

Fig. 3 shows the LeNet architecture, which was developed in 1986 and was applied commercially in recognizing digits in bank notes and zip codes. As we can see, there are two convolution layers and two pooling layers applied sequentially. Then there are two fully connected layers and one Gaussian connections layer which is the loss layer.

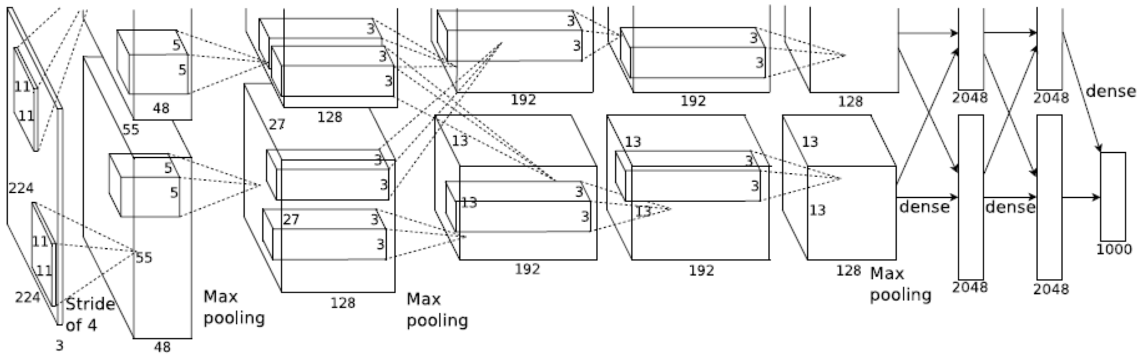


Figure 4. Alexnet architecture [1]

Fig. 4 shows the Alexnet architecture, which was published in a seminal deep learning paper in 2012. The authors used this CNN architecture to break the record of an ImageNet competition. They published a giant leap over other methods in classification accuracy and were the champion in the 2012 ImageNet competition. Alexnet is much more complex and more powerful than LeNet. Nevertheless, the major components are still convolution layer, pooling layer, ReLU layer, fully connected layer and loss layer. The development of Alexnet shows that convolutional neural networks have huge potential if increasing their complexity and depth. This success has motivated deep learning investigators to develop other and better convolutional neural network architectures.

1.1.3 Recurrent neural networks (RNN)

If the data that need processing are sequential data with correlations among data samples, such as text and speech signals, we need recurrent neural networks. Traditional shallow neural networks or convolutional neural networks may work very well because they have limited capability to model the long or short term correlations among the data. In contrast, recurrent neural networks are developed specifically for this purpose. They have the nice capability of processing sequential data, and can be designed to model both long and short term data correlations.

1.1.3.1 The architecture of RNN

The reason why recurrent neural networks can learn and memorize correlations within time sequences is that recurrent neural networks maintain certain states which depend on not only the current input data, but also the previous states. The schematic diagram of a recurrent neural network is shown in Fig. 5.

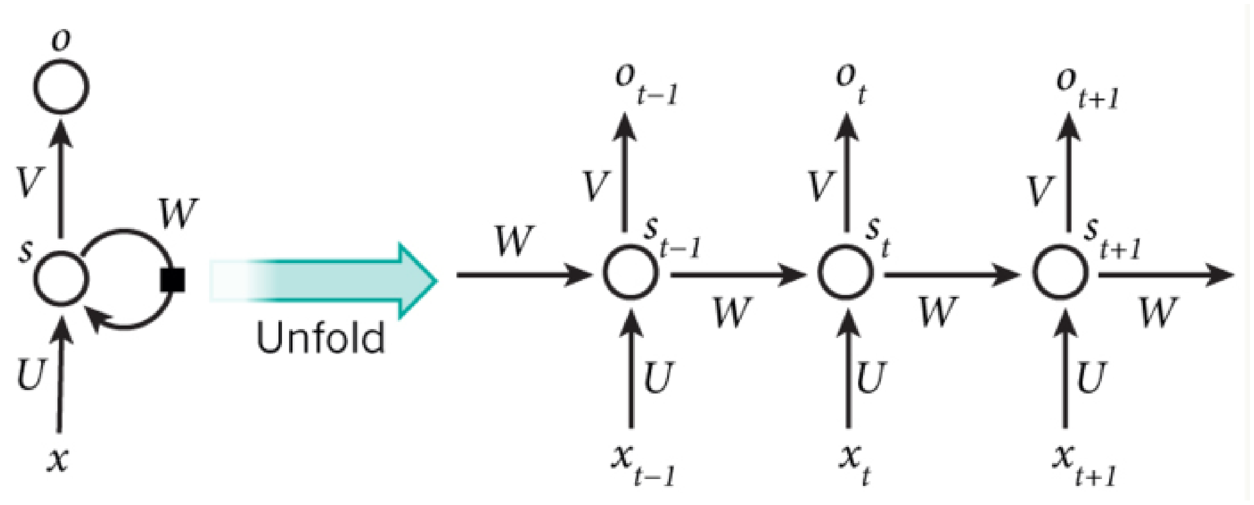


Figure 5. Schematic diagram of a recurrent neural network [1].

In the schematic diagram of Fig. 5, t is time, x_t is input at time t, s_t is the RNN state at time t, o_t is the output at time t. The matrices (or tensors) W, U and V are the weighting parameters used to calculate the states and the outputs from the inputs and the states, respectively. The training procedure will determine the values of these matrices or tensors. From this diagram, we can clearly see that the input data is a time series and the output data may also be a time series. Because W, U, V are shared among all the time steps, the RNN is able to learn and model the association among the input data.

1.1.3.2 Long short-term memory (LSTM)

One of the big problems for the vanilla RNN architecture shown in Fig. 5 is that it suffers from the vanishing or exploding gradient problem [5]. The cause of this problem is that gradients used in learning the weighting parameters W, U, V are obtained by multiplying the error gradients of each time step. Because the time step of RNN may be big, if the error gradients of each time step are mostly smaller than one, the overall multiplication results will approach zero. On the other hand, if the error gradients are mostly bigger than one, then the overall gradients may explode to infinity. Both vanishing gradients and exploding gradients will prevent the convergence of training.

While the exploding gradient problem can be mitigated by gradient clipping techniques, the vanishing gradient problem has to be addressed by novel RNN architectures. A special RNN architecture called long short-term memory (LSTM) has provided a popular and effective solution to this problem.

The concept of LSTM was first proposed in 1997 by Hochreiter and Schmidhuber in [6]. However, the original LSTM architecture had input gates and output gates only. In [7], Gers, etc., enhanced it with forget gates. Furthermore, in [8], Gers, etc., improved the performance of LSTM with Peephole Connection. After many investigators' study and improvement, the LSTM nowadays has superior performance in learning and modeling long and short term correlations among data sequences. LSTM uses smartly the concept of gates to resolve the vanishing gradient problem. Together with gradient clipping, they provide a nice solution to the vanishing and exploding gradient problems for RNNs.

An LSTM network is an RNN that consists of a sequence of LSTM cells. The structure of an LSTM cell is shown in Fig. 6, from which we can find that there are three gates: input gate, forget gate and output gate. If the correlation between the input data of the current time and the input data of previous times is not strong, these three gates may be weakened so as to reduce or even cancel completely the influence of the input data of previous times. Otherwise, the gates may become strong which makes all the input and state information travels freely down the time sequence. This is one of the reasons that LSTM can mitigate the vanishing gradient problem.

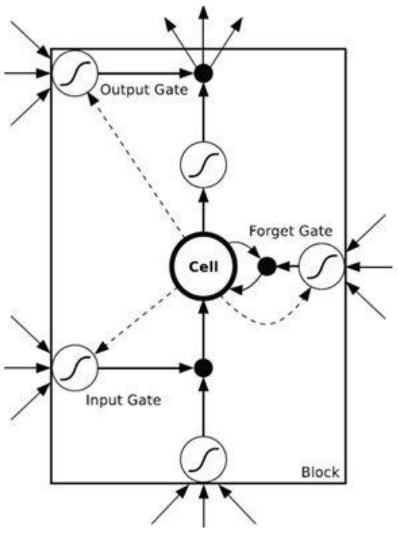


Figure 6. Schematic diagram of an LSTM cell [1].

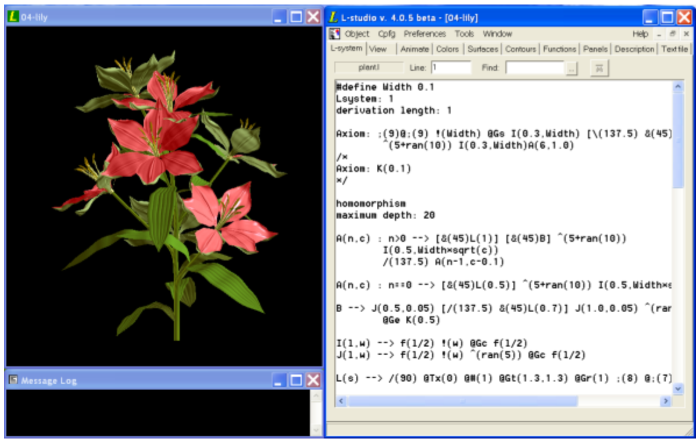
The great success of LSTM has also stimulated the development of other RNN architectures that can achieve similar performance but with simpler structures and lower computational complexity. A good example is the Gate Recurrent Unit (GRU), which was developed in 2014 in [9]. Because we study the image series prediction problem, we will surely use an LSTM network in this thesis.

1.2 Software used in study

Three major software tools are used to conduct the research in this thesis, which are L-studio, MATLAB and Python. L-studio and MATLAB are used for generating images of simulated pine trees with various years of age. Python is used to implement our deep learning model, to train the model, and to test the model. We will show that the model can produce satisfactory experiment results with low enough training/testing accuracy. Note that our model is developed to generate predicted images of pine trees at targeting ages. Because MATLAB and Python are well known and common software, we will just give them a very briefly introduction. In contrast, since L-studio is less known by the engineering people, we will give it a more detailed introduction.

1.2.1 L-studio

L-studio is a plant modeling and simulation software. With L-studio, we can write a program to set the parameters of a plant, and use this program to simulate the appearance of this plant. The result is an image of this plant. Various parameters, such as years of age, can be set freely in the program to simulate various plants or various appearance of the same plant. L-studio is a small software but used widely in the botanical research community. It provides an easy-to-use integrated user interface GUI (graphic user interface) for programming simulation models and running the simulations. With L-studio, it becomes extremely convenient to performing virtual plant experiments. A picture of the L-studio GUI is shown in Fig. 7.


04-lily
Message Log
L-studio v. 4.0.5 beta - [04-lily]
Object Cpfg Preferences Tools Window Help
L-system View Animate Colors Surfaces Contours Functions Panels Description Text file
plant.l Line: 1 Find:

```
#define Width 0.1
Lsystem: 1
derivation length: 1

Axiom: ;(9)@;(9) !(Width) @Gs I(0.3,Width) [\(137.5) &(45)
        ^(5+ran(10)) I(0.3,Width)A(6,1.0)
/*
Axiom: K(0.1)
*/

homomorphism
maximum depth: 20

A(n,c) : n>0 --> [&(45)L(1)] [&(45)B] ^(5+ran(10))
        I(0.5,Width*sqrt(c))
        /(137.5) A(n-1,c-0.1)

A(n,c) : n==0 --> [&(45)L(0.5)] ^(5+ran(10)) I(0.5,Width*s

B --> J(0.5,0.05) [/(137.5) &(45)L(0.7)] J(1.0,0.05) ^(rar
        @Ge K(0.5)

I(1,w) --> f(1/2) !(w) @Gc f(1/2)
J(1,w) --> f(1/2) !(w) ^(ran(5)) @Gc f(1/2)

L(s) --> /(90) @Tx(0) @H(1) @Gt(1.3,1.3) @Gr(1) ;(8) @;(7)
```

Figure 7. L-studio interface (http://algorithmicbotany.org/lstudio/whatis.html).

L-studio consists of two simulators, which are CPFG and LPFG. It also has a set of sample models, primarily of plants, stored as L-studio objects. L-studio provides a user-friendly graphical browser for organizing and accessing objects on local and remote machines. It has also a series of editors and other modeling tools for creating and modifying objects, and a library of environmental programs which can be used to simulate environmental processes that affect plant's development.

To generate images of pine trees with different ages and appearance, we need compile a simulation program with parameters of the simulated plant, such as years, height, weight, blade angle, color of leaf and so on. A sample code is shown in text window in Fig. 7.

While L-studio is convenient use by hand operation, it cannot be iteratively called, run, and programmed automatically by another program. In other words, it is not difficult to use L-studio to generate some images, but it is almost impossible for us to use it to generate thousands of images. Because the source code of the L-studio simulation software is not available, we cannot recode and enhance it with the important loop-operation mode, i.e., creating a loop to generate images and store images automatically.

1.2.2 MATLAB

We need at least thousands of images in order to train a deep neural networks. It would be rather time-consuming to do it with hand-operated L-studio. We need a lot of clicking and input operations to generate each single image. In this thesis we will introduce a way to use a MATLAB program to control the operating of the L-studio software. Specifically, we use MATLAB to create a loop which controls L-studio to generate images. Through this way, we can easily generate many different pine tree images automatically. Details will be given in Chapter II.

1.2.3 Python

Python has become one of the most popular software for deep learning due to many nice advantages and properties. First, Python is an interpreted language which, unlike compiled languages, greatly reduces the workload of programmers. Second, there are a lot of third-party libraries for Python programmers to use. Such a convenience attracts even more users. Third, Python has many mature deep learning libraries. Most of these libraries are open source software that programmers can freely use to build their deep learning architectures. In this thesis, we use Python as the main software to implement our deep learning model.

1.3 Outline

The structure of this thesis is summarized as follows. In Chapter II, we will show some of the training images and explain how these images are generated, what the differences are among these images and why these images are correlated to each other. In Chapter III, we will describe our CRNN architecture. Chapter IV will provide a detailed description of our experiments and selection of CRNN parameters. Experiment results and their analysis will be given. In Chapter V, a conclusion is given, with a discussion over future works.

II. Generation of Training Images

In this chapter, we will introduce the controllable parameters in L-studio to generate different pine trees' images. We will also show how to use L-studio to generate images, and display some examples of the simulated images.

2.1 Controllable L-studio parameters for generating different images

To generate series of images of pine trees at various ages, the most important parameter in L-studio programming is the *years of tree*. We choose years of tree to be 10, 15, 20, 25, and 30, which means to model trees that are 10 years-old, 15 years-old, 20 years-old, 25 years-old, and 30 years-old, respectively. Each images of each individual pine tree thus form an image series. Each series consists of 5 images, which are images of a pine tree at 10, 15, 20, 25 and 30 years-old.

Besides the parameter of years of tree, there are many other parameters can be adjusted in order to model different pine trees and to generate different images. Some important parameters include the growth rate of tree's height, the growth rate of tree's diameter, the shrinkage rate of tree's growth, the slope of needle leaf of pine, the angle between branch, the slope of branch, the shrinkage rate of tree's height growth, the shrinkage rate of tree's diameter growth, the shrinkage rate of branch, the shrinkage rate of second-level branch, the diameter of branch, the diameter of crown. There are about fourteen parameters that can be used to generate different trees. Each of these parameters is adjustable within certain interval, such as [0, 1] or [0, 180] degree. Therefore, L-studio is very powerful for generating pine tree images with great difference in appearance.

After setting these parameters, we can use photos of real pine trees to determine the color of leaf, color of branch, color of trunk. In our experiments, in order to ensure that the species of pine trees are the same, we just used three parts of a single picture which has a lot of the real pine trees for this purpose.

2.2 Using L-studio to generate pine tree images

L-studio is a bionics software that doesn’t have built-in loop mode nor can be easily re-programmed to implement that mode. To solve this problem, we use MATLAB to call MS Windows VBS and JAVA scripts to simulate mouse and keyboard operations. By setting necessary parameters in a loop with MATLAB programming, we can make the computer to run L-studio, to set parameters, to generate various images, and to save the images into files automatically.

The detailed process is:

(1) Use MATLAB to randomly generate all the necessary parameters of simulated pine trees;

(2) Use MATLAB to launch L-studio;

(3) Call VBS and JAVA scripts in MATLAB to simulate mouse and keyboard moving, with which we can change the parameters inside the editor of L-studio;

(4) Set the growth interval of the simulated pine trees to years and save the generated pine tree images to a file folder;

(5) Repeating step (1) to step (4) to generate the needed number of images.

A problem is that some images may become same or too similar because their randomly-generated parameters may be similar to each other. To resolve this problem, we write another MATLAB program to test whether images are similar to each other. We delete those overly similar images to guarantee the diversity of training data.

Even though it is convenient for us to generate a huge number of pine tree images using this approach, because one of the goals of this research is study how to develop useful deep learning models when data set size is not very big, we generated only 3847 pine tree image series, where each image consists of a single tree, and each series consists of 5 images of a pine tree at the pre-set ages.

In fact, we generated 5000 image series. After the process of removing similar images, we have 3847 image series left. We believe that the increased diversity in training data can make our training process more reliable and more comprehensive, and make our training results much better, considering that we work with a limited amount of training data.

2.3 Example of simulated pine tree images

Some examples of the pine tree image series are shown in Fig. 8. In the figure, there are two image series. Each series consists of the images of a pine tree when it grows to the age 10, 15, 20, 25 and 30 years.

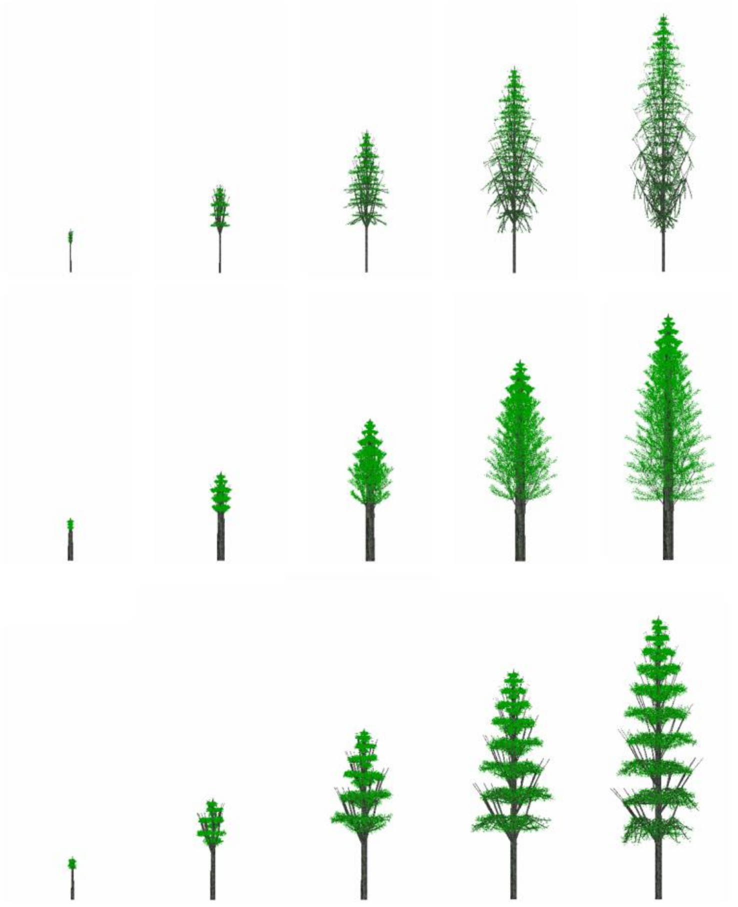


Figure 8. Examples of three pine tree image series.

We use such image series as training data to train our CRNN model. We use 3463 series (about nine tenth of the total data) as training data, and 384 series (about one tenth of the total data) as test data. Our CRNN is expected to predict the fifth image, which means the image of the 30 years-old pine tree, based on the first four images. This means we predict the future appearance images of the pine tree after several years of growth. We will compare the real fifth image with the predicted image to analyze the prediction performance.

One of the major issues is the size of the simulated images. The images shown in Fig. 8 has dimension 519×243×3, which is way too big as training image for our CRNN. Such a big size leads to very slow training speed and very high memory demand. In fact, it often exhausts all the resource of our computers because we just uses ordinary laptops for this work. To resolve this problem, we crop and downsample these images to a much smaller size 32×16×3, which means each image is reduced to a color image with 32 rows and 16 columns and 3 colors. Some examples of the decimated color images are shown in Fig. 9.

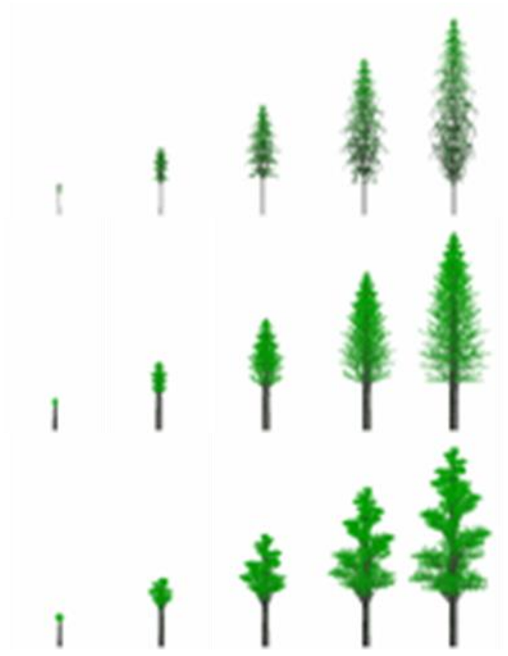


Figure 9. Pine tree images after decimation.

Fig. 10 show the photo of a real pine tree that we used to determine the color of leaf, color of branch, color of trunk of the simulated pine tree images.

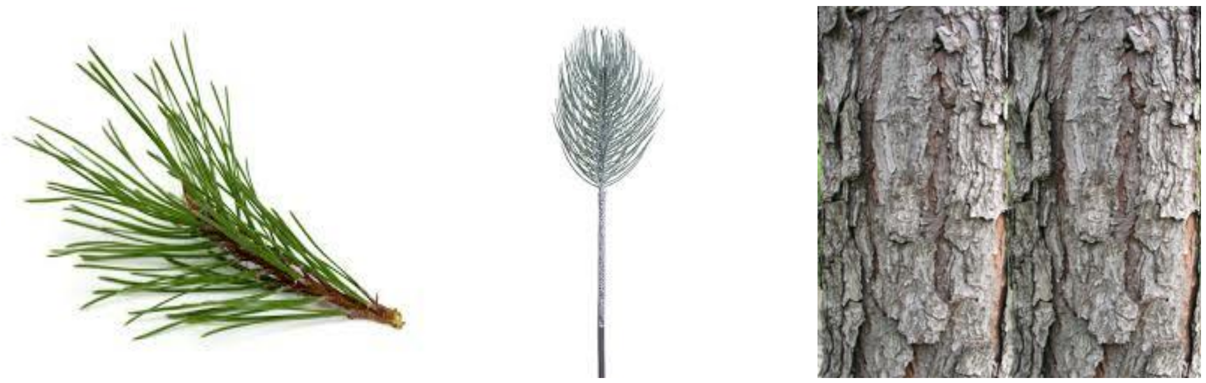


Figure 10. Real pine tree image

III. The CRNN Architecture

3.1 Description of CRNN

The architecture of the convolutional recurrent neural network (CRNN) model used in this research is shown in Fig. 11.

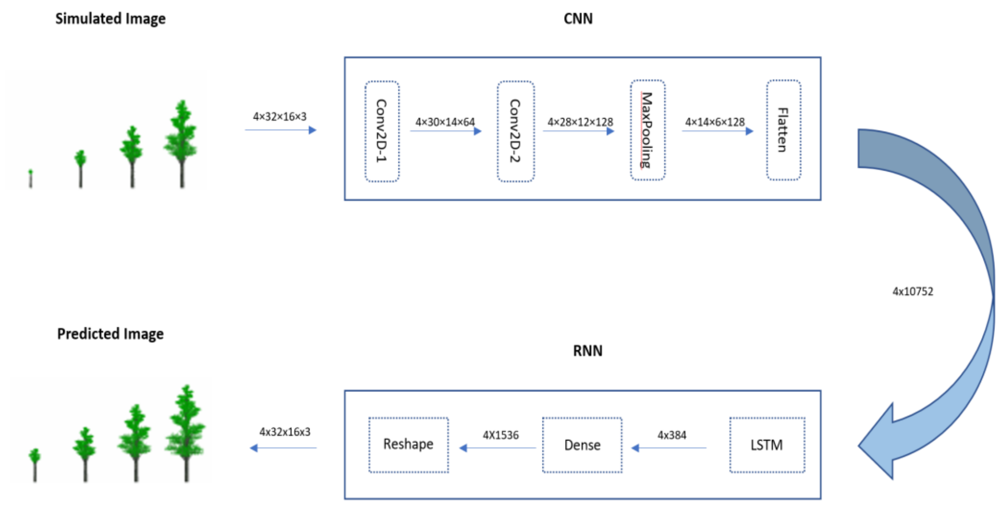


Figure 11. Outline of the CRNN model.

As shown clearly in Fig. 11, our training data are pine tree image series with series length 4, which includes images of a pine tree at 10, 15, 20 and 25 years old. Then we apply a CNN to process each image. The CNN consists of four layers: two convolution layers, one pooling layer and one flatten layer. After the CNN, there is an LSTM-based RNN, which consists of two layers: the LSTM layer and the Dense layer. The output will be a generated image series of length 4. This will be compared with the label, which is also an image series with series length 4. The label is actually the images of this pine tree at ages 15,20, 25 and 30 years. We focus on analyzing the performance of predicting the pine tree at 30 years of age from this pine tree's images at 10, 15, 20 and 25 years of ages. After training, we can use this CRNN model to predict future appearance of pine trees from their past appearances.

3.2 Data process, tensor dimensions, and model parameters

In order to understand better the CRNN model, it is helpful to describe the procedure of data processing in details, including the dimensions and values of important parameters and tensors. The dimensions of tensors are selected carefully so that our laptop computer can finish the training within reasonable time duration. The values of the CRNN parameters are also selected carefully with many repeated experiments. Details about parameter selection will be explained in the Chapter IV.

From Fig. 11, we can see that the dimension of the input tensor is $4 \times 32 \times 16 \times 3$, which means the input data is a series of color images with series length 4 and the size of each image is 32 rows and 16 columns. Because the kernel size of the first convolution layer is 3×3 and the number of filters is 64, the output of the first convolution layer is a tensor of dimension $4 \times 30 \times 14 \times 64$ (The reason why we have 30 and 14 can be seen from the convolution procedure shown in Fig. 1).

The output of the first convolution layer becomes the input of the second convolution layer. In the second convolution layer, the number of filters is 128 and the kernel size is still 3×3. Therefore, the dimension of the output tensor of the second convolution layer is $4 \times 28 \times 12 \times 128$. After the two convolution layers, a pooling layer is applied, which reduces the dimension of the data tensor to $4 \times 14 \times 6 \times 128$ with a stride (or decimation ratio) 2 (The pooling process has been explained in Fig. 2).

Then, a Flatten layer is used in order to connect the CNN with the RNN. As the layer name suggests, the function of this layer is to flatten each $4 \times 14 \times 6 \times 128$ data tensor into a two-dimensional data array with size $4 \times (14 \times 6 \times 128) = 4 \times 10752$. This finishes the CNN portion of the CRNN model.

Note that the CNN portion processes each image individually. Next, we apply RNN to learn the information embedded in the image series. The first layer of the RNN portion is an LSTM layer. The LSTM layer has 4 time steps, which consists of 4 LSTM cells. We set the dimensions of both the LSTM states and outputs to be 384. Therefore, the output of the LSTM layer is a data array with dimension 4×384.

To generate the predicted images, we use a Dense layer to generate output data tensors with the same dimension as the targeting pine tree image series. Specifically, the dimension is 4×1536. Note that 1536 equals to $32 \times 16 \times 3$, the size of a color image. We apply a reshape step at the end to obtain 4 predicted images with size 32×16×3. This will be compared to the label image series for loss function calculation during training.

IV. CRNN Implementation and Parameter Optimization

We implement the CRNN model shown in Fig. 11 by Keras with the Tensorflow backend [10]. Similar to most of the typical deep learning projects, there are a lot of parameters, including many hyper-parameters, that need to be selected and optimized in the CRNN model. Some parameters can be determined according to existing theory and practice of deep learning. Some other parameters have to be tested by trial-and-error experiments in order to find the optimal values. In this section, we will explain our procedure of adjusting these parameters according to the order of layers in Fig. 11.

4.1 Parameters of the convolution layers

4.1.1 Number of filters

The number of filters influences the number of image features that can be learned by the convolution layers. Intuitively, more filters will lead to better results. However, more filters also means increased amount of data and higher computation load. There is no accurate theory to determine what is the best number of filters. In this thesis, we conduct extensive experiments to look for the desirable number of filters.

Fig. 12 and Fig. 13 show the experiment results when different numbers of filters are used. The results are obtained with two connected convolution layers (the reason will be explained in Section 4.1.2). The second layer has twice number of filters as the first layer. The reason we use “twice” is that the input of the second layer is the output of the first layer where the information is already the image features. So we need more filters in the second layer in order to represent more abstract features.

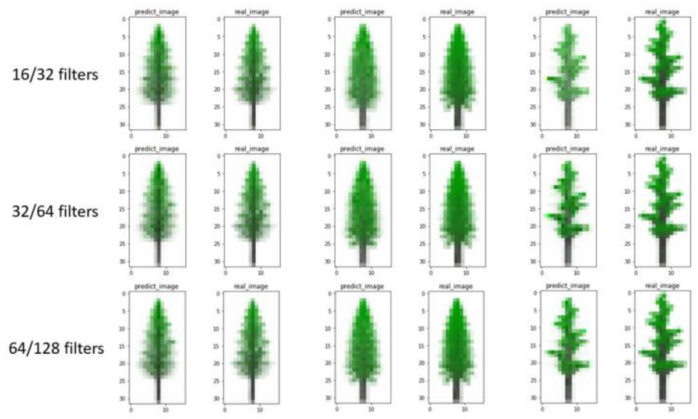


Figure 12. Real pine tree images and predicted images generated with different number of filters. Note that real pine tree images are actually the images generated by L-studio.

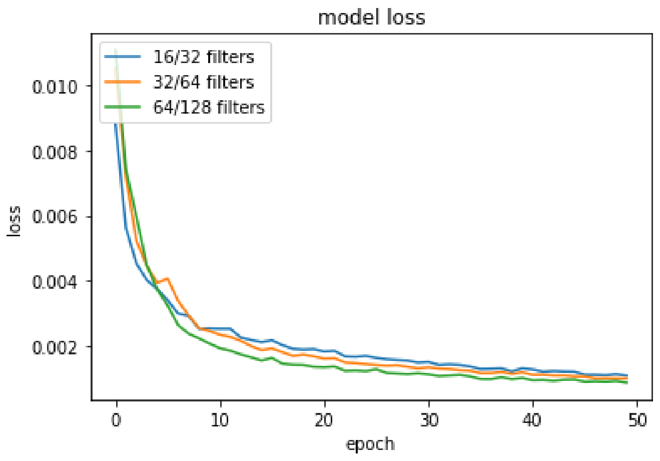


Figure 13. Learning curves under different number of filters.

From Fig. 12, we can see that for the third pine tree, the image predicted with 16/32 filters is a little shallow than others. From Fig. 13, we find that the loss becomes smaller when the number of filters increases. This indicates that more filters lead to better performance. However, more filters also lead to higher computational complexity and longer training time. The time spent per epoch in our case is 11 seconds with 16/32 filters, 17 seconds with 32/64 filters, and 30 seconds with 64/128 filters. Considering that the loss reduces at a slower pace but the time needed for training increases rapidly with more filters, we set the number of filters to be 64/128 in the subsequent experiments.

4.1.2 Size of kernel

Before testing different kernel sizes, let us first explain an interconnection between kernel size and number of convolution layers. With respect to the convolution results, a single convolution layer with 7×7 kernel size equals to two convolution layers with 5×5 kernel size, which again equals to three convolution layers with 3×3 kernel size, as illustrated in Fig. 14.

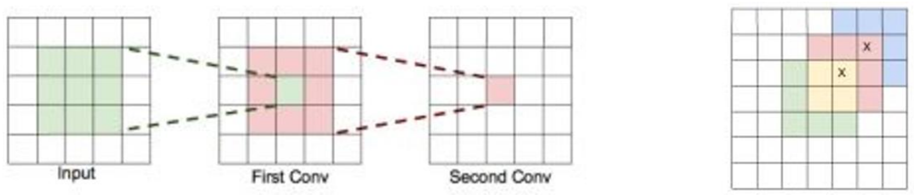


Figure 14. Schematic of kernel scan process (Adapted from lecture of HDL Prof. Ommer).

On the other hand, although three convolution layers with kernel size 3×3 equals to one convolution layer with kernel size 7×7 in terms of convolution results, the computational complexity of the former is much less than the latter. The case of three convolution layers with

3×3 kernel size has about 27 million operations, while the one convolution layer case has about 49 million operations. Considering this difference, we decide to use the fixed kernel size 3×3 and compare the performance of CRNNs with various number of convolution layers (one to three). The comparison results are shown in Fig. 15 and Fig. 16.

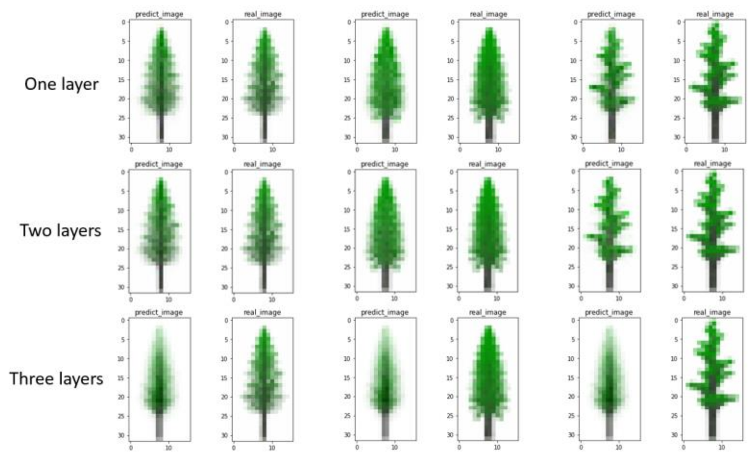


Figure 15. Real pine tree images and predicted pine tree images with one to three convolution layers.

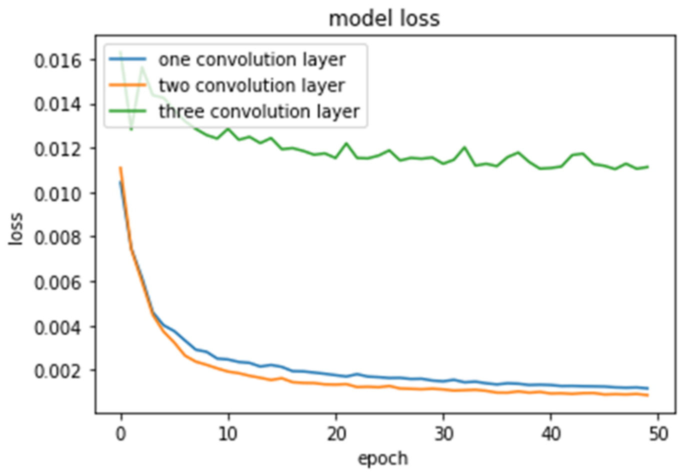


Figure 16. Learning curves of CRNNs under different number of convolution layers.

From Fig. 15, we find that the performances of one layer and two layers cases are similar, but the performance of the three layers case is much worse. Fig. 16 shows similar results in term of loss. The training loss of the two convolution layers case is smaller than that of the one convolution layer case. The reason is that our experiments are conducted under a premise that the training data set is small. Too many convolution layers can easily lead to neuron "dead", which happens when a neuron has a gradient that is too large to update its weights. If all the other gradients are too small compared to this large gradient, then the weights of this neuron may not get updated. In our experiment setting, we should avoid the CRNN models with three or more convolution layers. Therefore, we finally choose two convolution layers with kernel size 3×3 in our CRNN model.

4.1.3 Strides of convolution layers

Stride is a convolution layer parameter that results in decimation. Note that this is difference from the stride of the pooling layer. Because our input data are downsampled from the original size $519 \times 243 \times 3$ to a very small size $32 \times 16 \times 3$, we have to choose as small stride values as possible to preserve the information of the input data. Therefore, we set the strides to 1.

4.1.4 Padding of convolution layers

There are mainly three padding methods implemented in Keras for deep learning: Valid padding, Same padding and Causal padding. "Causal padding" works in time sequence deep learning models, which looks suitable to our model. However, it will generate dilation convolution result, which is not very suitable to our training data. More specific reasons will be explained again in Section 4.1.6. Considering the other two padding choices, we compare their performance with experiments, and the results are shown in Fig. 17 and Fig. 18.

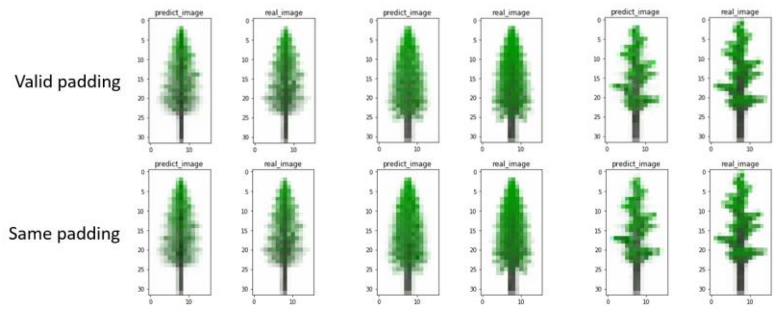


Figure 17. Real pine tree images and predicted pine tree images with different padding methods.

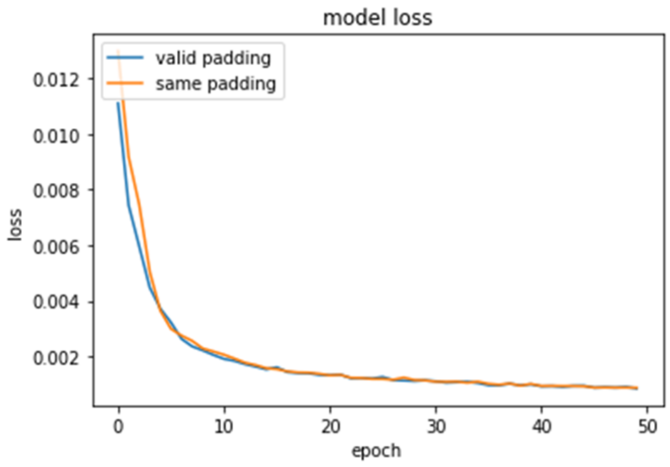


Figure 18. Learning curves under different padding methods.

From Fig. 17 and Fig. 18, we find that these two different padding methods produce almost the same performance. One of the reasons is that the difference between these two padding methods is not critical to our training data. “Valid padding” means that the convolution is not conducted at the edge of the images, while “Same padding” means that the convolution is preserved at the edge via zero-padding. In our training images, the edge part is almost all white. So it does not make any difference whether or not doing convolution at the edge of the images. Considering that we need to reduce computational complexity, we choose “Valid padding” as our padding method in convolution layers.

4.1.5 Input data format

Because we use Keras with Tensorflow backend to implement our CRNN model, the input data must be transformed to a tensor format “channel last”, which means the dimension of the input tensor is (batch size, image height, image width, image channels).

4.1.6 Dilation rate

Dilation rate is an important parameter for dilated convolution. The method of dilated convolution aims to mitigate the defects of pooling layer. Note that the pooling layer conducts image decimation, which may lose some image information. Because we have set the pooling size to be as small as possible, we do not need to conduct dilated convolution. In other words, we just set the dilation rate to 1, which means no dilation.

4.1.7 Activation functions

There are many activation functions to use, such as sigmoid, relu, selu, tanh, linear, etc. Linear activation function is seldomly used except at the output layer. Some nonlinear activation functions are needed in our CRNN convolution layers. We have tested the performance of the following activation functions.

4.1.7.1 Sigmoid activation function

The math expression of the sigmoid function is

$$\mathrm{S(x)} = \frac{1}{1+e^{-x}} = \frac{e^{x}}{e^{x}+1} \tag{1}$$

and Fig. 19 shows its input-output response.

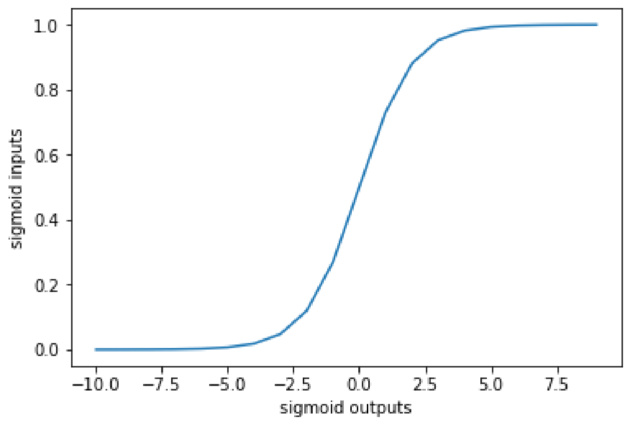


Figure 19. Graph of the sigmoid function.

The sigmoid function is a special case of the logistic function. Its output has values inside (0,1). While sigmoid function has been a popular activation function for neural networks, it has some disadvantages that make it not as popular as many other activation functions for deep learning. The first and the most important disadvantage is that this function can easily lead neurons to saturate. Because the gradients in the saturation state is near zero, the neurons can hardly adapt. This problem has happened clearly in our experiments, as the predicted images are very worse when the sigmoid function is used, see Fig. 20.

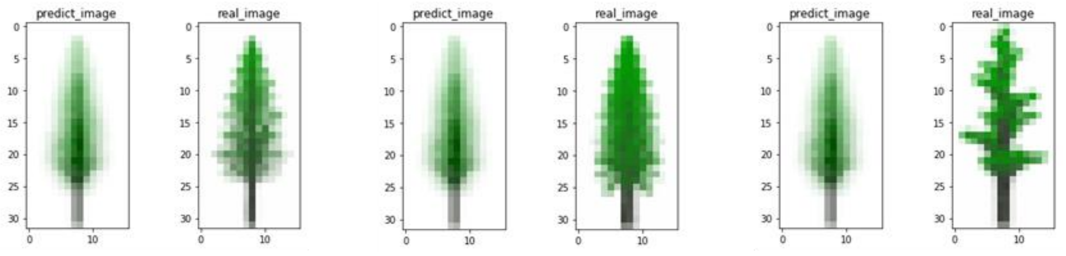


Figure 20. Real pine tree images and predicted pine tree images with sigmoid function.

The second disadvantage of the sigmoid function is that the output of the sigmoid function is not zero-centered. This makes the input of next layer not zero-centered, which may produce worse training results. Considering these issues, we do not use sigmoid function as the activation function of our convolution layers.

4.1.7.2 Selu activation function

Selu means scaled exponential linear unit. It was proposed by G. Klambauer in [11]. The math expression of selu is

$$\mathrm{selu(x)} = \lambda \begin{cases} x & if\ x > 0 \\ \alpha e^{x} - \alpha & if\ x \le 0 \end{cases} \tag{2}$$

$$\text{where} \quad \lambda \approx 1.0507 \quad \alpha \approx 1.6733$$

Although this activation function is shown to have superior performance in many applications, in our experiments, however, it leads to over-fitting. Note that it does work very well at the beginning. The over-fitting problem may be due to the limited training data set in our case. Its best performance appears at the 28th iteration. The results are shown in Fig. 21.

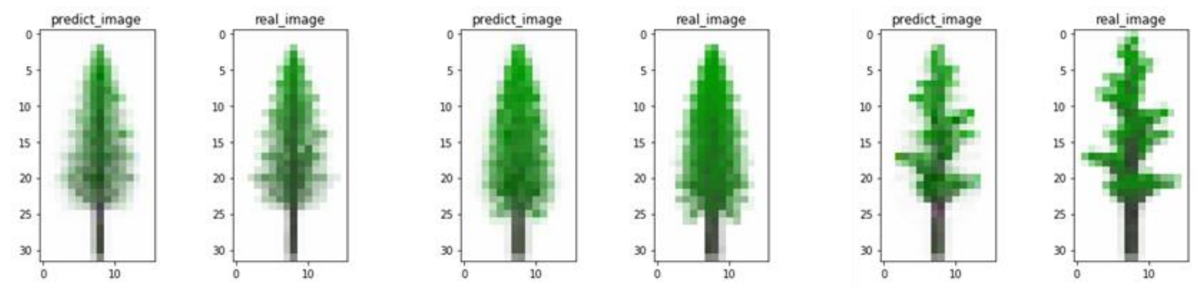


Figure 21. Real pine tree images and predicted pine tree images with the selu active function.

4.1.7.3 ReLU

ReLU means rectified linear unit. It has become one of the most popular activation functions in deep learning nowadays. The math expression of ReLU is

$$\mathrm{relu}(x) = \max(0, x) \tag{3}$$

and the input-output graph is shown in Fig. 22.

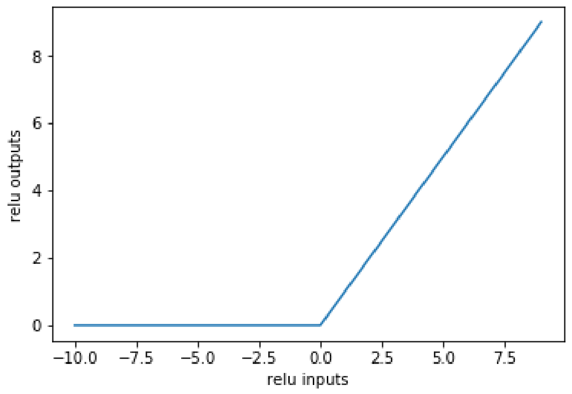


Figure 22. ReLU activation function.

ReLU has been shown to have many distinct advantages. One of the advantages is that it can improve the convergence speed of training deep neural networks. The convergence speed has been shown to be much faster than when other activation functions such as sigmoid or tanh are used. On the other hand, there are also some weaknesses for the ReLU function. One of weaknesses is that the ReLU function has zero-gradient for non-positive inputs, which may make many neurons unadaptable and thus become “dead” neurons. Although this problem can be mitigated by appropriate initialization and by using small learning rates, we do not use this activation function in our CRNN model because we have observed that it may lead to instability. Some experiment results with the ReLU function are shown in Fig. 23.

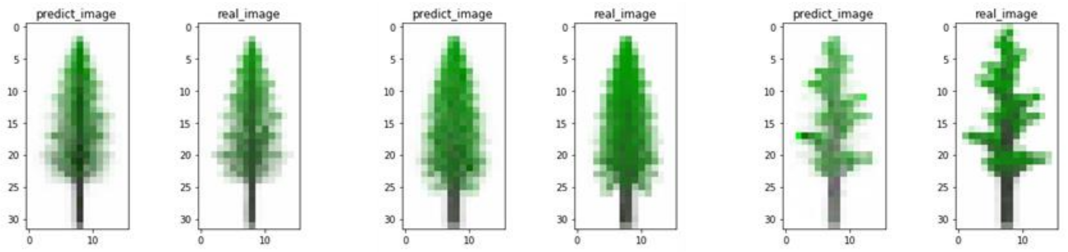


Figure 23. Real pine tree images and predicted pine tree images with the ReLU function.

4.1.7.4 Tanh activation function

The tanh (hyperbolic tangent) activation function is derived as a simple linear transform of the sigmoid function, i.e.,

$$\tanh(\mathrm{x}) = 2\mathrm{sigmoid}(\mathrm{x}) - 1 = \frac{2e^x}{e^x+1} \quad (4)$$

The tanh function is a zero-centered function. The input-output response is shown in Fig. 24.

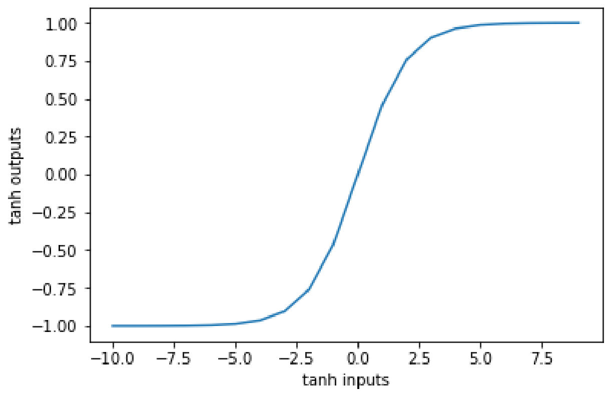


Figure 24. tanh activation function.

Our experiments indicate that the tanh function can produce better results than many other activation functions we have tested. Some experiment results are shown in Fig. 25.

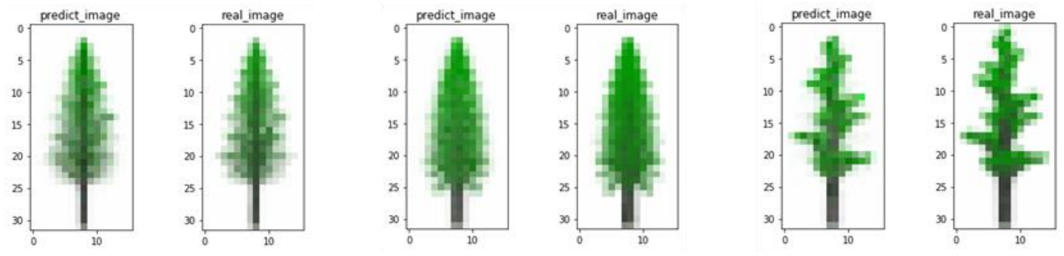


Figure 25. Real pine tree images and predicted pine tree images with tanh function.

We have conducted an extensive list of experiments to compare the performance of the above three activation functions. From the learning curves (i.e., training loss as function of number of epochs) of these activation functions shown in Fig. 26, we can see that tanh and relu has better performance than selu. The performances of tanh and relu are almost similar. However, because of the instability issue we encountered with the relu function, we adopt tanh as activation function in the convolution layers of our CRNN model.

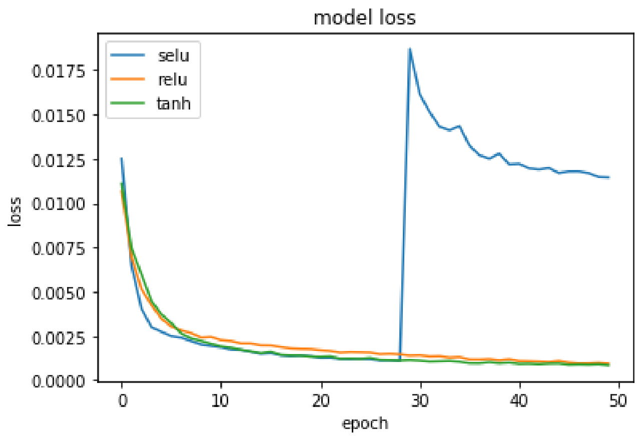


Figure 26. Learning curves of different activation functions.

4.1.8 Initialization of weights

In general, neural network operation is conducted in the form of $y = f(xW + b)$, where W contains the weights and b is the bias. The variables in $W$and b are the parameters to learn during training. The initialization of the weights means how to set the values of W initially to start the training procedure.

One of the straightforward methods of weight initialization is zero-initialization, i.e., let the weights to be zero in the beginning. However, this is usually bad for deep learning. The reason is that with zero-initialization all neurons will have the same outputs, which leads to the same gradients. This finally makes all the neurons to become identical, and the neural networks will lose asymmetry.

How about initialize the weights to be some identical but non-zero values? It will also lead to the same symmetry problem. A better and widely used method of weight initialization is to initialize the weights based on some random distributions with zero mean and small variances. In [5], Glorot, etc., recommend the variance to be

$$\mathrm{Var}(\omega) = \frac{2}{(n_{in}+n_{out})} \tag{5}$$

where n_{in} and n_{out} refer to the numbers of input and output neurons in this layer, respectively. This method is shown to have better performance in practice. So we adopt this method for weight initialization.

4.1.9 Initialization of bias

As to the initialization of the bias b, the most common way is simply let it be zero. This will not cause the problem of losing asymmetry when the weight W is initialized properly [5]. In our experiments, we just use zero-initialization for all the biases.

4.2 Pooling layer

For the pooling layers, we need to determine the pooling size, strides, padding method and data format. With the similar reasons as in the convolution layers, we choose the value of strides to be as small as possible. We adopt the "Valid padding" method because the edge of our images does not have very important information. We choose "channel last" as data format because we use Tensorflow as backend of Keras. Then the only parameter that needs to be set by experiments is the pooling size.

The experiment results of using different pooling sizes are shown in Fig. 27 and Fig. 28.

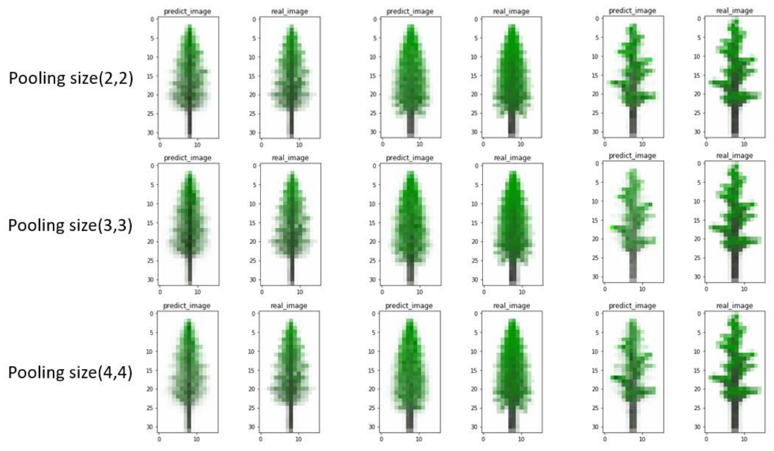


Figure 27. Real pine tree images and predicted pine tree images with different pooling sizes.

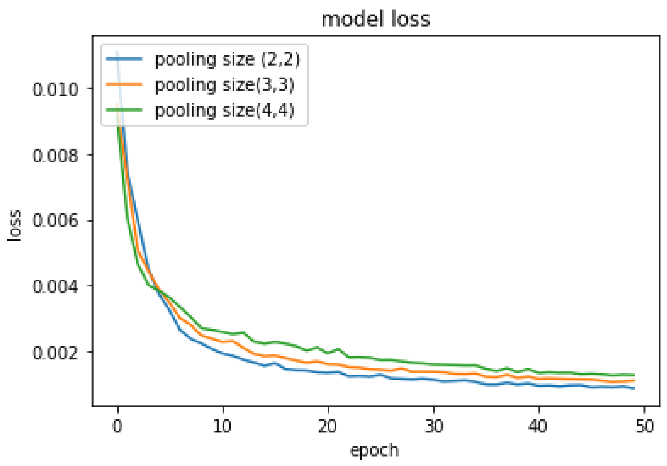


Figure 28. Learning curves under various pooling sizes.

From Fig. 27, we can find that the performance decreases with the increase of pooling size. Fig. 28 indicates that with a higher pooling size, the training loss is lower at the beginning of training, but becomes higher at the end. We believe that higher pooling size leads to the loss of more information of the training images. Therefore, we set the pooling size to a small value, which is (2,2), in our CRNN model.

4.3 Flatten layer

This layer is just for transforming multi-dimensional tensors to two-dimensional arrays so as to feed the output of the CNN to the LSTM layer. No adjustable parameters are needed.

4.4 LSTM layer

The LSTM layer has many adjustable parameters. Fortunately, not all of these parameters need to be set via time-consuming experiments. Instead, some parameters can be determined according to existing theory or practice. For example, because we need bias in our model, we choose "bias =

True”. The initialization of the weights and biases is similar to that discussed in Sections 4.1.8 and 4.1.9. Therefore, we initialize all the weights with Glorot uniform, and initialize all the biases to zeros [12]. In addition, because we need the forget gate to improve the performance of LSTM layer, we set “unit forget bias = True”.

Since we want to generate a full image series to test the performance of our model, not just the last output image, we set “return sequence = Ture” which will output a full sequence [13]. Because we do not need the last state at the output, we set “return state =False” [14]. Since the method of unroll is not suitable for short sequences, we set “unroll = False”.

Besides the above easily determinable parameters, there are some parameters that have to be determined via experiments. The determination of these parameters is described below.

4.4.1 The number of neurons in LSTM cell

The number of neurons in each LSTM cell is the dimension of the cell state vector. It influences the complexity and the performance of the CRNN model. Our objective is to use less number of neurons to achieve desirable performance.

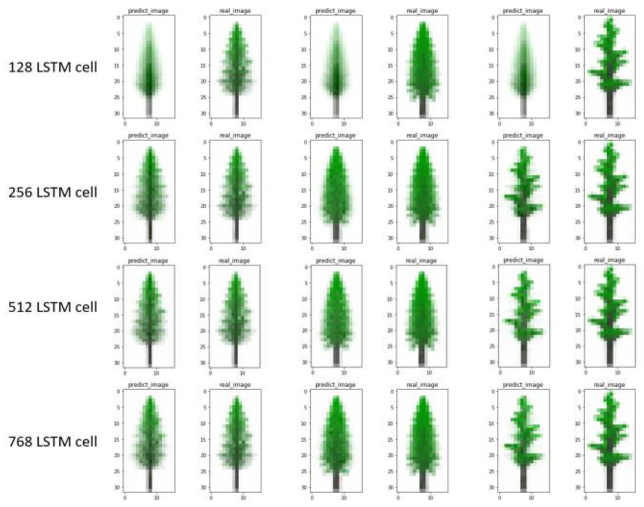


Figure 29. Real pine tree images and predicted pine tree images with various number of neurons in each LSTM cell.

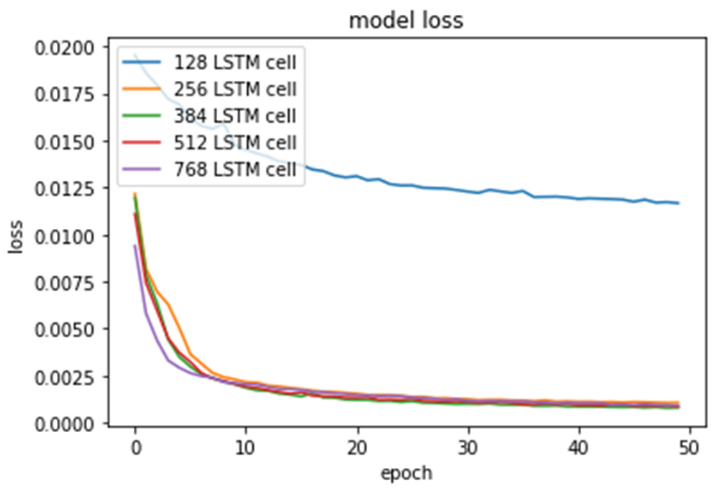


Figure 30. Learning curves under various number of neurons in each LSTM cell.

From our experiment results shown in Fig. 29 and Fig. 30, we can find that using 128 neurons is not enough to learn the input information. The performances of using 256, 384, 512, or 768 neurons are much better. In particular, the performance of using 384 neurons is the best, with the minimum loss 8.32e-04. By comparison, the minimum loss with 512 neurons is 8.58e-04, the minimum loss with 256 neurons is 1.05e-3, and the minimum loss with 768 neurons is 9.05e-04. Considering also the computational complexity, we choose 384 as the number of neurons in each LSTM cell.

4.4.2 The initializer for recurrent kernel

As we all know, one of the major weaknesses of RNN is gradient vanishing and gradient exploding [15]. Jointly using LSTM and gradient clipping can mitigate this weakness. Another way to mitigate this weakness is to initialize the RNN with an orthogonal recurrent kernel [16].

The reason why RNN can easily lead to gradient vanishing or gradient exploding is that the gradients of RNN involves the multiplication of many weight matrices. Such a repeated matrix multiplication will surely cause the gradient values to increase or decrease exponentially. Because orthogonal matrices have a nice property, i.e., their absolute eigenvalues are 1, the multiplication of many orthogonal matrices will not lead to values that increase or decrease exponentially. If all the weight matrices are orthogonal matrices, the gradients of RNN may not vanish or explode anymore. Therefore, we initialize the weight matrices of the LSTM cells as orthogonal matrices, which means orthogonal recurrent kernel.

4.4.3 Dropout value

Dropout is a very important technique in deep learning. It can effectively suppress overfitting. On the other hand, excessive dropout may reduce training efficiency. We adopt two dropout values in the LSTM layer: one for the inputs, and the other for the recurrent state. The dropout values can be determined via trial-and-error, beginning from a small value [16].

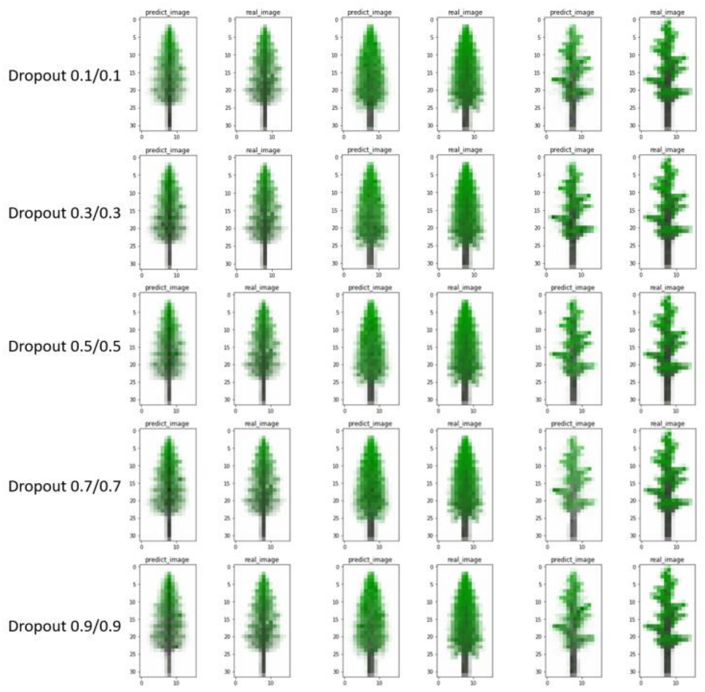


Figure 31. Real pine tree images and predicted pine tree images with different dropout values.

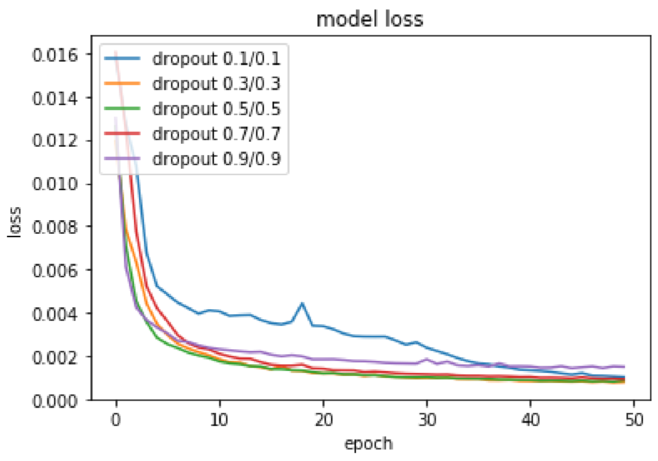


Figure 32. Learning curves under different dropout values.

From our experiment results in Fig. 31, it is not easy to distinguish the performance of different dropout values. However, from Fig. 32, we find that the dropout value 0.3/0.3 has the best performance. The minimum loss with dropout value 0.3/0.3 is 7.84e-04. Therefore, we adopt 0.3/0.3 as our dropout values.

4.4.4 Implementation mode

There are two implementation modes in LSTM networks. Mode 1 arranges the operations into a larger number of smaller dot products and additions, while Mode 2 arranges the operations into fewer operations with large dot products and additions. The performance of these two modes is hard to be distinguished in our case, as indicated in the experiment results in Fig. 33 and Fig. 34. In Fig. 33, the performance of two implementations seems almost same. In Fig. 34, the minimum loss of the two modes are also almost identical. We choose the implementation Mode 1 because it has slightly better performance at the beginning of the training.

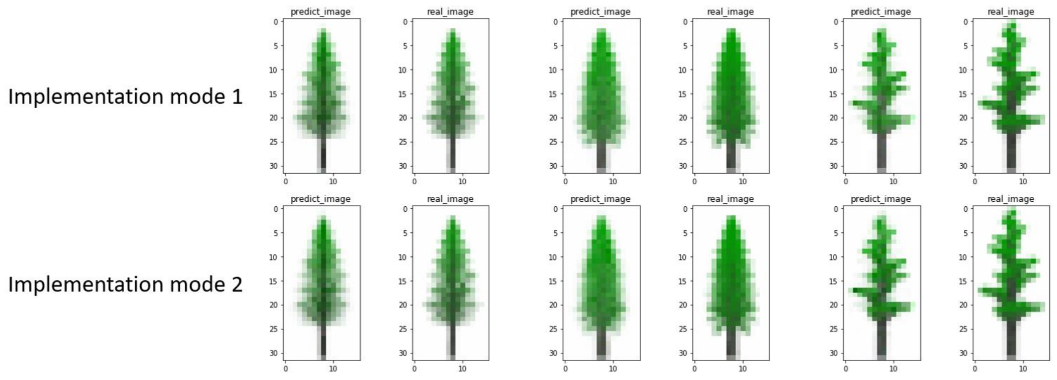


Figure 33. Real pine tree images and predicted pine tree images with different implementation modes.

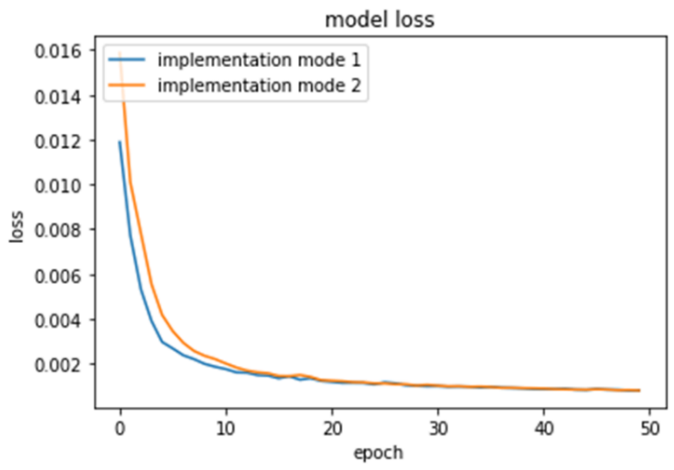


Figure 34. Learning curves under different implementation modes.

4.4.5 Activation function in LSTM

In LSTM, there are two activation functions, one for initializing the LSTM and the other for the recurrent steps in the LSTM. We have tried several pairs of activation functions, such as “tanh + hard sigmoid”, “selu + tanh”, and etc. Considering that the relu function may lead to too big outputs in RNN [17] and tanh or hard sigmoid is improvement over sigmoid, we try to choose the initializing activation function from tanh and selu, and to choose the recurrent activation function from tanh and hard-sigmoid. Therefore, we need to compare four combinations in total.

Let us first introduce the hard sigmoid activation function. The math expression is

$$\mathrm{hs}(\mathrm{x}) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right) \qquad (6)$$

and the input-output response is shown in Fig. 35.

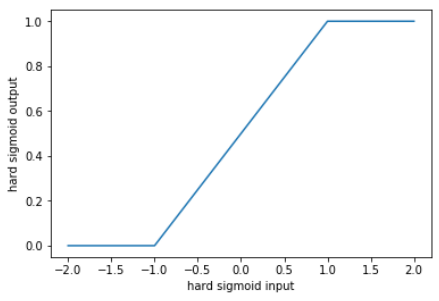


Figure 35. Input-output of hard sigmoid function.

The hard sigmoid function is a hard saturation expression of the sigmoid activation function [18]. This activation function has some linear characteristics, which leads to bigger gradient when the cell is not in saturation and leads to constant gradient values during saturation. Unfortunately, this activation function may sometimes lead to unexpected results, such as gradient vanishing.

We have conducted a list of experiments to compare the performance of the four activation function combinations. The results are shown in Fig. 36 and Fig. 37.

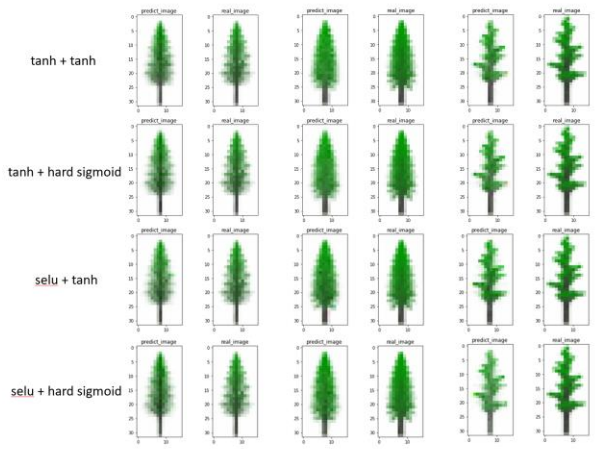


Figure 36. Real pine tree images and predicted pine tree images with different activation functions in LSTM.

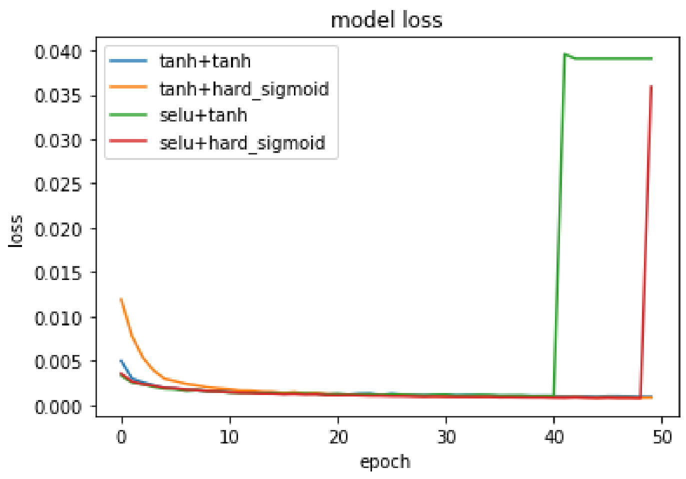


Figure 37. Learning curves under different activation functions in LSTM.

From Fig. 36 and Fig. 37, we can see that the selu activation function will saturate much earlier than tanh, and the hard sigmoid has better performance in the recurrent step. Therefore, we choose tanh as the initial activation function and hard sigmoid function as the recurrent step activation function.

4.5 Dense layer

In the dense layer, some parameters are determined by the input and output data dimensions. For example, the number of output units should equal to the image size $32 \times 16 \times 3$. We need bias, so we set "bias = True". The kernel weight is initialized with "Glorot uniform" [5], and the bias is initialized to be zero. The only issue left is which activation function we should use. We have compared the performance of three common activation functions in this layer: relu, sigmoid and tanh. The results are shown in Fig. 38 and Fig. 39.

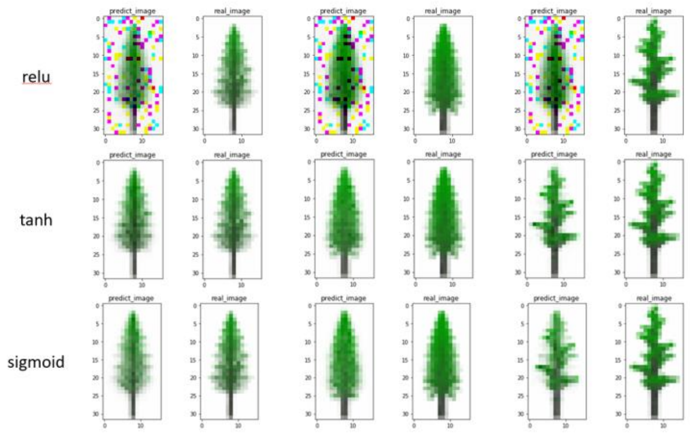


Figure 38. Real pine tree images and predicted pine tree images with different activation function in the dense layer.

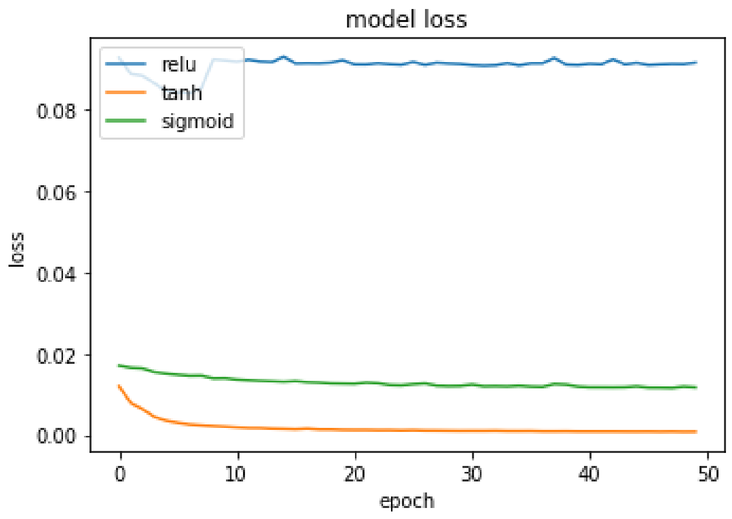


Figure 39. Learning curves under different activation function in the dense layer.

From Fig. 38, we can find that the result of using relu activation has led to heavy noise. The results of tanh and sigmoid are similar and are both much better than relu. From Fig. 39, we can easily see that tanh is a better choice than the other functions in the dense layer. Therefore, we use tanh as the activation function.

4.6 Loss function and optimizer

Before training the CRNN model, we need choose our loss function and optimizer. Because the output data and label data are both images, the loss function can be chosen from mean squared error (MSE), mean absolute error (MAE), mean absolute percentage error (MAPE) or mean squared logarithmic error (MSLE). The candidates of optimizer can be SGD, RMSprop, Adam or Nadam. In order to select the best one, we check their performance one by one with experiments.

4.6.1 Loss function

Because different loss functions calculate different loss values, we cannot just use the loss values to compare their performance. Instead, we introduce four criteria to compare their performance: visual performance check with human eyes, MSE between the predicted images and the label images, MAE between the predicted images and the label images, SNR between the predicted images and the label images. They are described in the following subsections.

4.6.1.1 Mean squared error (MSE)

Mean square error is a concept from statistics. It measures the average of deviation. The math expression in our model is

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\widehat{Y_i} - Y_i)^2 \qquad (7)$$

where n is the number of pixels of the images, Y_i is the pixel value of the label images, and $\widehat{Y_i}$ is the pixel value of the predicted images.

4.6.1.2 Mean absolute error (MAE)

Mean absolute error is also a concept from statistics. It measures the difference between the predicted values and the real values. The math expression is

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |\widehat{Y_i} - Y_i|}{n} \tag{8}$$

where n is the number of pixels of the images, Y_i is the pixel value of the label images, and $\widehat{Y_i}$ is the pixel value of the predicted images.

4.6.1.3 Mean squared logarithmic error (MSLE)

Mean squared logarithmic error is applied to avoid two problems of mean squared error. One problem is that using mean squared error may lead to very slow learning speed. The other problem is that mean squared error works not well when data value is too big. The math expression of mean squared logarithmic error is

$$\mathrm{MSLE} = \frac{1}{n} \sum_{i=1}^{n} (\log \widehat{Y_i} - \log Y_i)^2 \tag{9}$$

where n, Y_i and $\widehat{Y_i}$ are defined similarly as last subsections.

With these different loss functions, we have the experiment results shown in Fig. 40. It can be seen that there exist noise points in the predicted images when using MAE and MSLE as loss function. Therefore, with just visual performance check, we need to choose MSE as loss function.

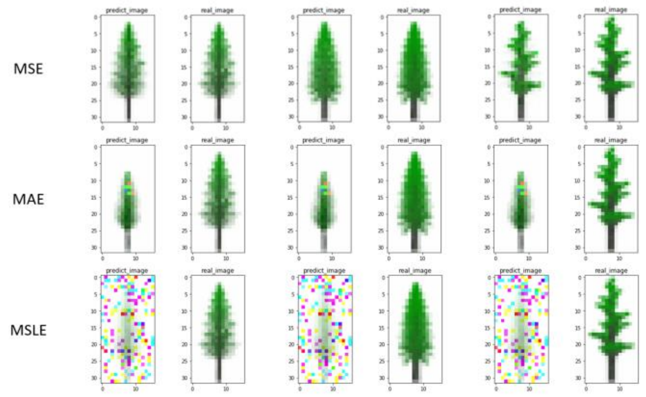


Figure 40. Real pine tree images and predicted pine tree images with different loss functions.

4.6.2 Optimizer

Different optimizers have different characteristics. In this subsection, we first make a brief introduction to the optimizers that we tested, and then compare their performance in our experiments.

4.6.2.1 SGD (stochastic gradient descent)

SGD is implemented as a batch gradient descent or mini-batch gradient descent algorithm that iteratively calculates the gradient and updates the weights. The weights are updated by adding

$$\Delta\theta_t = -\eta g_t \tag{10}$$

to the existing weights, where η is the learning rate and g_t is the gradient. This optimized method is one of the classical methods. The weakness of SGD is that it is hard to find the optimal learning rate due to the fact that the gradients g_t are highly random.

4.6.2.2 RMSprop

RMSprop means root mean square propagation. Different from SGD that suffers heavily from the randomness of gradients, RMSprop applies the root mean square of the recent gradients to mitigate randomness. Specifically, the gradient is averaged as

$$\mathrm{RMS}|g|_t = \sqrt{E|g^2|_t + \epsilon} \tag{11}$$

where E means expectation, ϵ is a small non-zero constant. Nevertheless, the performance of RMSprop may not be desirable when the gradients are not smooth.

4.6.2.3 Adam

Adam means adaptive moment estimation. It is a very new gradient search method developed in 2015 [18]. It combines both momentum and RMSprop, and has the capability to dynamic adjust its learning rate [18]. One of the advantages of Adam is that the learning rate automatically determined by Adam is within a small interval, which leads to smooth weights update. The gradient used for weight updating is calculated as

$$\Delta\theta_t = -\eta \frac{\frac{m_t}{1-\mu^t}}{\sqrt{\frac{n_t}{1-\nu^t}+\epsilon}} \tag{12}$$

$$m_t = \mu m_{t-1} + (1-\mu) g_t, \quad n_t = \nu n_{t-1} + (1-\nu) {g_t}^2$$

where μ, ν are momentum factors, $and\ n_t$ is a regularizer.

4.6.2.4 Nadam

Nadam is a combination of the Adam algorithm and the Nesterov momentum algorithm. The regularizer of Nadam will lead the learning rate into a smaller interval than that of Adam [19]. The gradient is

$$\Delta\theta_t = -\eta \frac{(1-\mu_t)\widehat{g_t} + \mu_{t+1}\frac{m_t}{1-\prod_{i=1}^{t+1}\mu_i}}{\sqrt{\frac{n_t}{1-\nu^t}}+\epsilon} \tag{13}$$

$$m_t = \mu m_{t-1} + (1-\mu)g_t, \quad n_t = \nu n_{t-1} + (1-\nu){g_t}^2$$

We have experimented our CRNN training with the above optimizers, and the results are shown in Figs. 41 to 43.

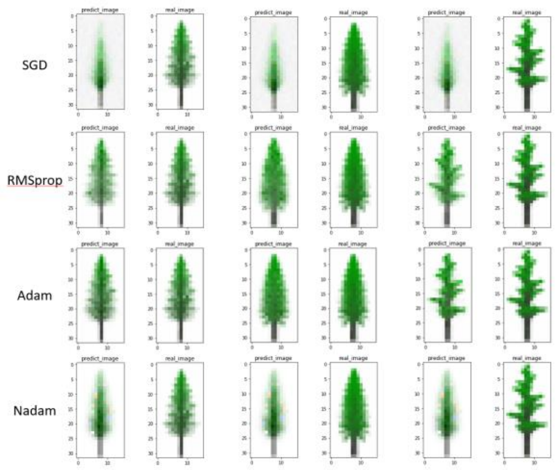


Figure 41. Real pine tree images and predicted pine tree images obtained with different optimizers.

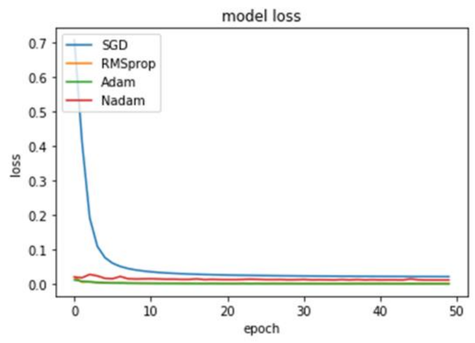


Figure 42. Learning curves of 4 different optimizers.

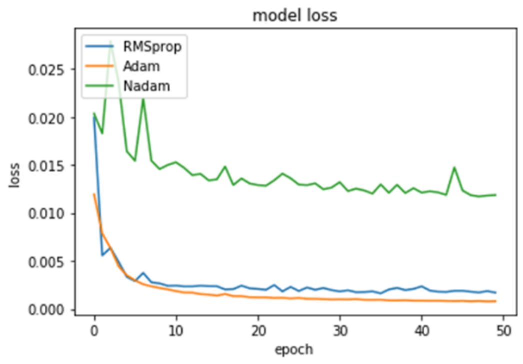


Figure 43. Learning curves of 3 different optimizer.

From Fig. 41, we can find that the performance of RMSprop and Adam looks good, while the performance of SGD and Nadam is worse. From Fig. 42, we can find the reason why the performance of SGD is bad. The reason is neuron saturation. Because the loss of SGD is too high compared with the other three optimizers, to examine more clearly the difference between the other three optimizers, we display their performance specifically in Fig. 43. From this figure, we can see that the loss of Nadam is not stable and not smooth. This means that Nadam may lead to learning too fast and finally lead to over-fitting. In contrast, the performance of Adam is stable and outstanding. Therefore, we use Adam as the optimizer in our CRNN training.

4.7 CRNN model training

We use 3463 image series as training data set to train our CRNN model, and use the reserved 384 image series as test data set to test the performance of our model. Before training, we need to determine the appropriate batch size and the appropriate number of iterations. These two hyper-parameters are not independent from each other because smaller batch sizes always need more iterations to converge [17]. Again, we use experiments to determine these parameters.

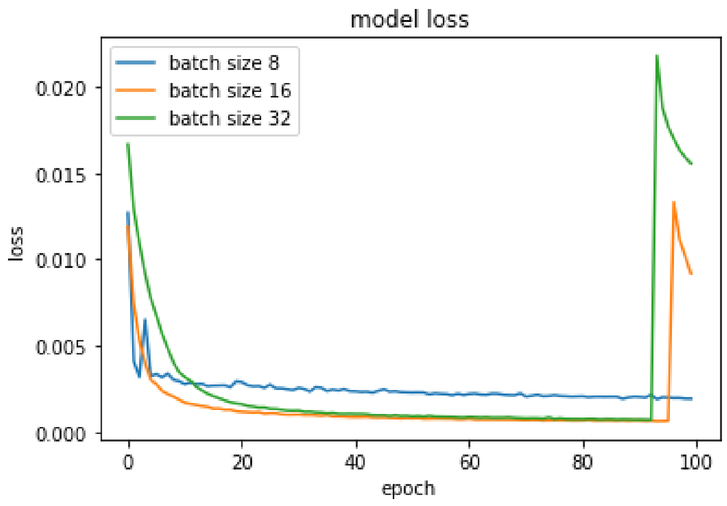


Figure 44. Learning curve with different batch sizes.

From the experiment results shown in Fig. 44, we can find that higher batch sizes will lead to over-fitting earlier. However, when the batch size decreases to 8, the loss suffers from a higher error floor compared with batch size 16. This might be due to gradient vanishing. As a result, we set batch size as 16 and set the number of epochs to 95 because this gives the lowest loss in Fig. 44.

4.8 CRNN performance

After the CRNN is designed and the parameters are tuned as discussed in the previous sections, we train it and use it process the test data. Over the test data set, the performance of our CRNN for pine tree image prediction is listed in Table 1.

Criterion	Value
MAE	8.57
MSE	377.35
SNR	322.55
Min Val-Loss	6.24e-04

Table 1. Performance of image prediction measured by 4 different performance criteria.

Some typical examples of the real pine tree images (which are generated by L-studio) and predicted pine tree images (which are the output of CRNN) are shown in Fig. 45. As it can be seen, our CRNN model can successfully predict the pine tree images.

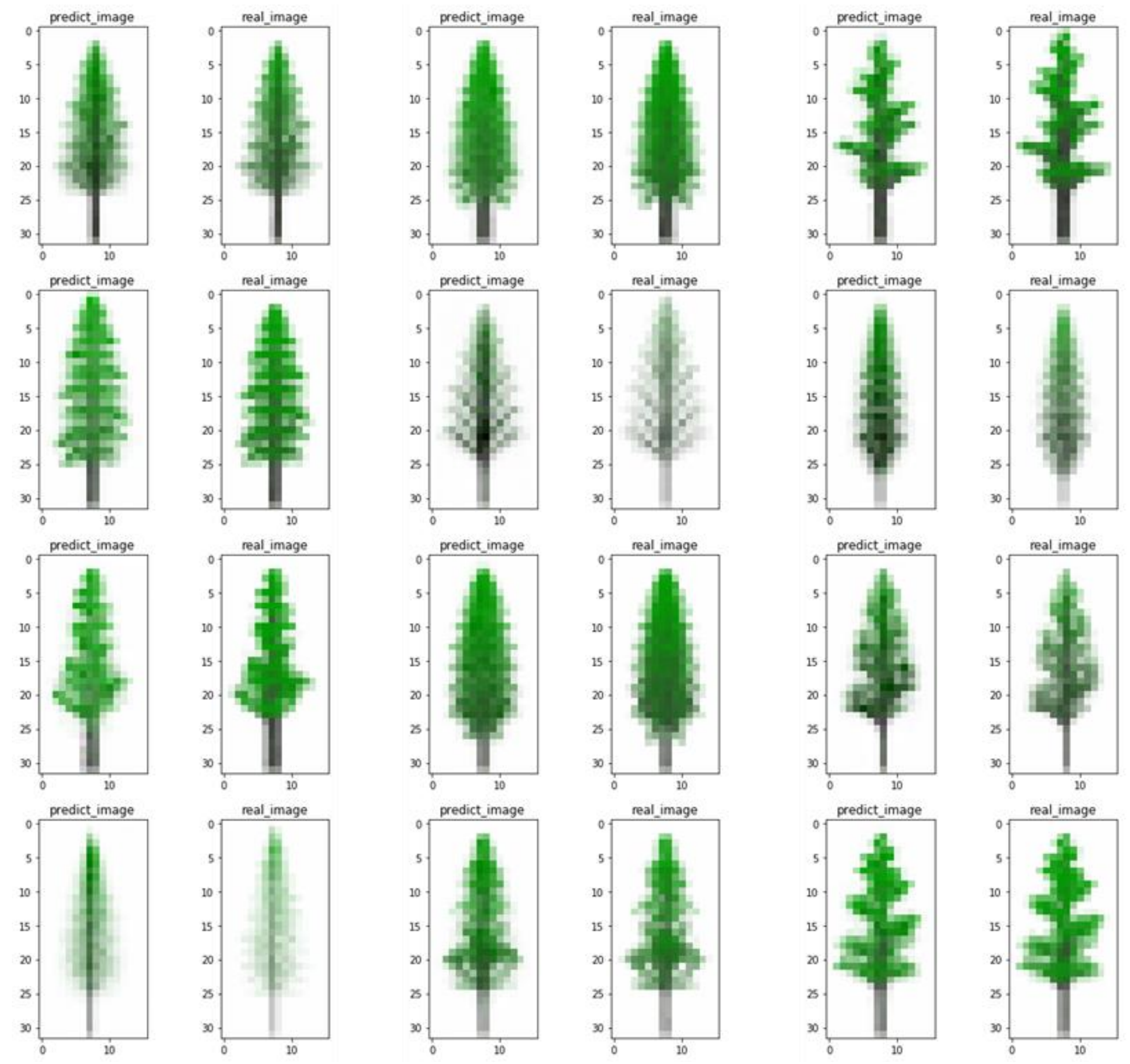


Figure 45. Real pine tree images and predicted pine tree images.

V. Conclusion and Future Work

In this thesis, we have developed a deep learning model that uses convolutional recurrent neural network (CRNN) for image prediction with limited training data. Specifically, we train the CRNN model with a pine tree image data set generated by L-studio, and demonstrate that this model can predict successfully the future appearance of pine trees according to their past appearances.

In general, the predict result of the developed and optimized CRNN is good. However, there still be some problems that can be addressed to further improve the performance.

First, one of the biggest problems in our CRNN model is over-fitting due to the small data set. To avoid over-fitting, we have already tried to use shallower structure, smoother activation function, smaller learning rate and less iteration epochs. Such limitations may have reduced the performance of our model. Potentially, we can develop or exploit other over-fitting mitigation techniques, such as batch normalization, to further enhance performance.

Second, due to the limitation on the graphics memory size and running speed of our computer (we use the graphics processor GTX 960M with 4GB RAM for most of the experiment work), we could not experiment with bigger image sizes. Experimenting with bigger image sizes may potentially lead to predicted images with much better quality.

Finally, the data set is generated by the simulation software L-studio, which is still far away from real pine tree photos. Although L-studio has many adjustable parameters for us to generate many different images, it still cannot be compared with the diversity of natural images. As future work, we should collect enough images of natural pine tree or other objects and use them to test the prediction capability of the proposed CRNN model.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, The MIT Press, 2016. http://www.deeplearningbook.org/.

[2] P. H. O. Pinheiro and R. Collobert, “Recurrent Convolutional Neural Networks for Scene Labeling,” *ICML* (4), pp. 82-90, 2014.

[3] M. Zihlmann, D. Perekrestenko, and M. Tschannen, “Convolutional Recurrent Neural Networks for Electrocardiogram Classification,” Computing in Cardiology(CinC) 2017, PhysioNet/CinC Challenge 2017.

[4] M. Nielsen, *Neural Networks and Deep Learning*, Online book, Dec 2017.

[5] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep forward neural networks,” *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, PMLR 9:249-256,2010.

[6] S. Hochreiter and J. Schimidhuber, “Long short-term memory,” *Neural Computation* 9(8):1735-1780, 1997.

[7] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to Forget: Continual Prediction with LSTM”, *Neural Computation* 12(10), October 2000.

[8] F. A. Gers and J. Schmidhuber, “LSTM Recurrent Networks Learn Simple Context Free and Context Sensitive Languages,” *IDSIA*, Galleria 2, 6928Manno, Switzerland 2001.

[9] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”, arXiv:1406.1078, 2014.

[10] Keras documentation online: https://keras.io/

[11] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, “Self-Normalizing Neural Networks,” *Advances in Neural Information Processing System* 30, NIPS 2017.

[12] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” *Proceedings of the 30th International Conference on Machine Learning(ICML-13)*, 2013.

[13] I. Sutskever, O. Vinyals, and Q. Le, “Sequence to Sequence learning with neural networks,” *Advances in Neural Information Processing Systems,* 2014.

[14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv: 1412.3555,* 2014.

[15] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks,* 5(2):157-166, 1994.

[16] E. Vorontsov, C. Trabelsi, S. Kadoury, and C. Pal, “On orthogonality and learning recurrent networks with long term dependencies,” *arXiv:1702.00071v4* [cs:LG] 12 Oct 2017.

[17] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, “End-To-End Memory Networks,” *Advances in Neural Information Processing System 28,* NIPS 2015.

[18] D. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *Third International Conference for Learning Representations,* San Diego, 2015.

[19] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” *ICML’13 Proceedings of the 30th International Conference on International Conference on Machine Learning – Volume 28,* Atlanta, GA, USA, June 16-21, 2013