Sequential Images Prediction Using Convolutional LSTM with Application in Precipitation Nowcasting

Wu, Mingkuan

UNIVERSITY OF CALGARY

Sequential Images Prediction Using Convolutional LSTM with Application in Precipitation Nowcasting

by

Mingkuan Wu

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN MATHEMATICS AND STATISTICS

CALGARY, ALBERTA
AUGUST, 2019

Abstract

The precipitation nowcasting task is to predict the rainfall intensity in a local region in the short future. In this thesis, we formulate this task as a sequential images prediction problem in which both the input and output are temporal sequences of spatial images. The problem will be viewed from the machine learning perspective. Inspired by *fully connected long-short term memory* (FC-LSTM) which is good at capturing the temporal relationship, we build a *convolutional long-short term memory* (ConvLSTM) by adding convolutional operation in the input-to-state and state-to-state transitions. By incorporating an encoder-forecaster model structure, the experiments we have run show that stacked ConvLSTM is better to capture spatio-temporal relationships and outperforms FC-LSTM in the precipitation nowcasting task.

Acknowledgements

I would like to thank my supervisor Dr. Gemai Chen for his patience, guidance and support. I would also like to thank all faculties and people in the Department of Mathematics and Statistics for their kind help in the last two years. Most of all, I would like to thank my family for their constant encouragement and love.

Table of Contents

List of Figures and Illustrations

List of Tables

List of Symbols, Abbreviations and Nomenclature

Symbol or abbreviation	Definition
NN	Neural Network
NWP	Numerical Weather Prediction
RNN	Recurrent Neural Network
LSTM	Long-short Term Memory
FC-LSTM	Fully Connected LSTM
ConvLSTM	Convolutional LSTM
MLP	Multi-layer Perceptron
ReLU	Rectified Linear Units
Adam	Adaptive Moment Estimation
BPTT	Back-propagation Through Time
CNN	Convolutional Neural Network
ROVER	Real-time Optical Flow by Variational Methods for Echo of Radar

Chapter 1

Introduction

1.1 Introduction to Precipitation Nowcasting

Precipitation nowcasting has been an important task in weather forecasting field for a long time. The task is to provide a rapid prediction of the rainfall intensity in a local region from the present to a few hours ahead (0-6 hours) based on the radar echo maps, rainfall gauge and other observation data. Precipitation nowcasting has a significant impact on our daily lives and can play an important role in many real-world applications. For example, it provides the emergent rainfall alerts to airport and also facilitates the drivers by providing on-time road information under extreme fast-changing weather conditions. Unlike traditional weather forecasting tasks like daily average temperature prediction, precipitation nowcasting problems require higher resolution and time-accuracy which leads to a more challenging work.

Two categories of precipitation nowcasting methods are frequently used nowadays [17]: the traditional *numerical weather prediction* (NWP) based method and the radar echo extrapolation based method. The NWP method needs a complex and long simulation of the physical formulas in atmosphere model. It takes a long time to produce the results. This in itself is not a problem as you can use the model trained on the old data until the new model updated. But this leads to a major drawback of NWP. If something is happening that the model does not expect, the model cannot give a stationary prediction and needs time to re-run with the new data. By contrast, the radar echo extrapolation based method, which provides faster and more accurate predictions, is adopted as the current state-of-the-art[5, 14]. Some computer techniques, especially the optical flow based method, have been proved useful in making accurate extrapolation of radar echo maps [20]. Hong Kong Observatory has proposed the *Real-time Optical flow by Variational methods for Echo of Radar* (ROVER) algorithm [20] to operate rainfall nowcasting system. The ROVER algorithm computes the optical flow of the consecutive radar maps for generation of the motion fields. However, the ROVER method assumes invariability of the motion field which can lead to error in prediction when motion field can vary according to fast changes under synoptic situation. Hence, ROVER fails to predict the growth and decay of the radar echos which is a common limitation of the traditional radar-based method.

Since traditional radar echo extrapolation method has limitations, these issues may be addressed by viewing the prediction task from a machine learning perspective. The precipitation nowcasting task is, in fact, a spatio-temporal sequence prediction problem. The input is a sequence of radar echo maps in the past and the output target is a number of radar echo maps in the short future. However, it is quite challenging since we are dealing with high dimensional data (three-dimension image data) especially when we want to do the multi-step predictions (four-dimension output). Moreover, the complex nature of weather makes the problem more challenging.

The recurrent neural networks, especially *long short-term memory* (LSTM) networks, have been proved to tackle the temporal relationship with a strong ability [7, 6, 2, 18]. LSTM was first introduced by Hochreiter and Schmidhuber [7], and it was frequently applied into natural language process which involves sequential data such as words, sentences and sound spectrum and the real-life application examples include translation, sentiment analysis, text generation and more. If we want to solve the problem from the machine learning approach, we need an end-to-end model and sufficient data to train the model. The precipitation nowcasting problem satisfies these two requirements since we have a huge amount of radar echo maps data and the *convolutional LSTM* (ConvLSTM) encoder-forecaster network provides a complete end-to-end model to learn this prediction task by training the stacked ConvLSTM network in the encoder-forecaster model structure. In 2014, Ranzato et al. [12] proposed an model to be the first in predicting the future video images based on a recurrent neural network language model. Their model predicted the future images but in only one step ahead. In 2015, Srivastava et al. [16] pointed out the importance of multi-step prediction in sequential images prediction and proposed the *fully connected LSTM* (FC-LSTM) encoder-forecaster network to achieve the multi-step predictions. However, the FC-LSTM layer in the model does not capture the spatial relationship in the learning process. In 2015, a novel ConvLSTM network was proposed for spatio-temparal sequence prediction problem by Shi et al. [15]. Based on the FC-LSTM network, they added the convolutional operation in both the input-to-state and state-to-state transitions in the model which learned well on the spatial relationships inside each image. In this thesis, we are going to build a ConvLSTM and incorporate an encoder-forecaster structure into it to improve the precipitation nowcasting task. We use the FC-LSTM encoder-forecaster network as the baseline model and evaluate this novel ConvLSTM encoder-forecaster network on some real radar echo maps data and the synthetic moving MNIST data. The experiments we carried out show that our ConvLSTM encoder-forecaster network has a significant improvement over the FC-LSTM encoder-forecaster network on both data sets.

1.2 Formulation of Precipitation Nowcasting Problem

The goal of precipitation nowcasting problem is to predict a fixed length of future radar echo maps based on a sequence of past radar echo maps. In real applications, we take the radar echo maps every 6 to 10 minutes. We will do the nowcasting for the next 0 to 6 hours, i.e., to predict the next 1 to 60 frames of future radar echo maps. This can be viewed as sequential images prediction problem from the machine learning perspective.

Suppose the radar echo map of a local region is treated as an image which has $N_1 \times N_2$ grids. On each grid, we can have H measurements about the rainfall information which will vary over time. So an observation at one time step can be viewed as a 3-dimension tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times N_2 \times H}$ where $\mathbb{R}$ denotes the real domain of the observed features. The past J observed radar echo maps are then a sequence of tensors $\mathcal{X}_{t-J+1}, \mathcal{X}_{t-J+2}, \ldots, \mathcal{X}_t$ and we want to predict the most likely sequence of length K in the future based on the previous J observations. The mathematical expression is:

$$\tilde{\mathcal{X}}_{t+1}, \ldots, \tilde{\mathcal{X}}_{t+K} = \underset{\mathcal{X}_{t+1}, \ldots, \mathcal{X}_{t+K}}{\operatorname{argmax}} P(\mathcal{X}_{t+1}, \ldots, \mathcal{X}_{t+K} | \mathcal{X}_{t-J+1}, \mathcal{X}_{t-J+2}, \ldots, \mathcal{X}_t),$$

where P stands for probability.

For the precipitation nowcasting problem, the radar echo maps are usually pre-processed into grey scale with pixel values of integers between 0 (dark) and 255 (light) on a single channel. As we view each radar echo map as a 3D tensor with dimension N_1 row $\times$ N_2 column $\times$ 1 channel, the problem is now transformed into a sequential images prediction problem as in Figure 1.1.

The rest of the thesis is organized as below. In chapter 2, we introduce the background information and the various neural networks we need to know and use later. In chapter 3, we build a ConvLSTM network based on the FC-LSTM network and incorporate an encoder-forecaster structure into it to solve the sequential images prediction task. In chapter 4, we first train our models on the moving MNIST data to understand the behavior of the ConvLSTM network. Then we will train our models on the more complex radar echo maps data to accomplish the precipitation nowcasting task. In chapter 5, we draw some conclusions.

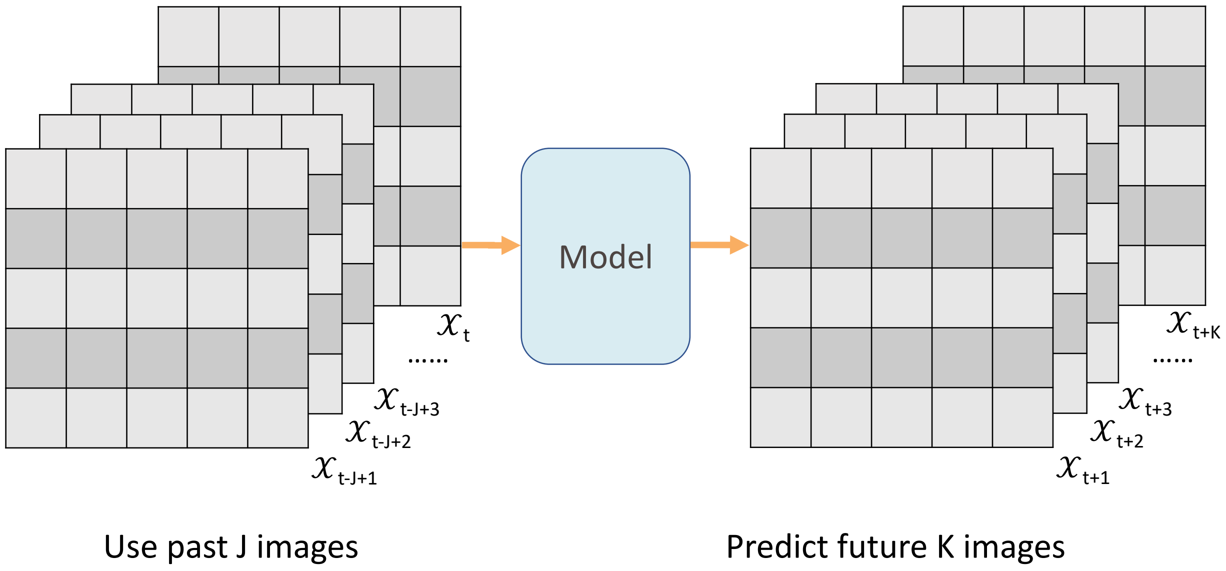


Figure 1.1: Sequential images prediction task formulation

Chapter 2

Neural Networks

This chapter is going to provide necessary background information about neural networks. We start from the very basic multi-layer perceptron model to simple recurrent neural network which is aimed at solving sequential data problems. A type of advanced recurrent neural network called long-short term memory neural network is further introduced followed by convolution neural network which is used especially for image type data. We will introduce both the forward pass and backward pass of all these networks as well as the optimization algorithm to train the neural networks.

2.1 Basic Neural Networks: MLP

A *neural network* (NN) is a machine learning computing model inspired by the biological neural networks in animal brain. The basic structure of a NN is a network of small nodes, or units, which are joined with each other by weighted connections. In a biological model, the nodes refer to neurons and the weighted connections represent the synapses between neurons. NN is activated by passing the input to the nodes and then the information is spread out through the weighted connections.

The very basic and widely used form of NN is *multi-layer perceptron* (MLP) [10]. MLP arranges units inside each layer. Units in one layer are connected by a forward feeding from units in the previous layer as in Figure 2.1. We always define layers with units from input information and outputs as input layer and output layer, respectively, and all other layers between input and output layer are just hidden layers. This process is known as *forward pass* of the network. MLP can be viewed as a function parameterized by the connection weights, matching the input information to the output as we expect.

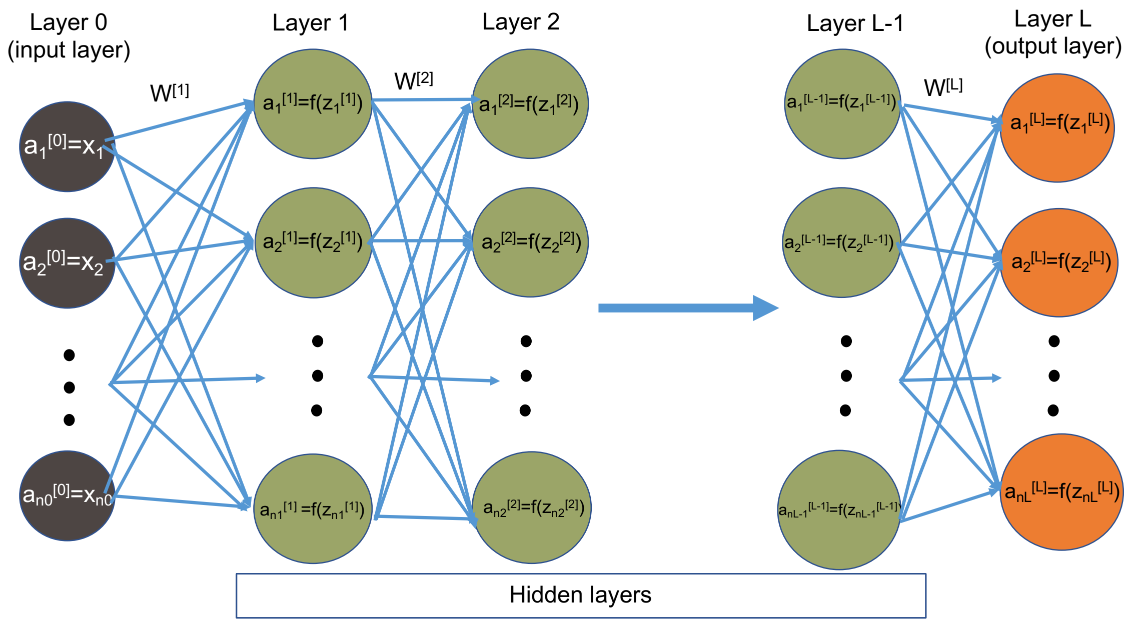


Figure 2.1: An example of L-layer MLP

2.1.1 Forward Pass

A node, which represents a scalar value, in a hidden layer will receive the information from all nodes in the previous layer first through a linear transformation which is multiplying all nodes in the previous layer by corresponding weights and adding a bias, and then applying a non-linear transformation via activation function.

Let

$$\mathbf{a}^{[k]} = \begin{bmatrix} a_1^{[k]} \\ a_2^{[k]} \\ \vdots \\ a_{n_k}^{[k]} \end{bmatrix}_{n_k \times 1}$$

denote a vector of all n_k nodes in the k-th layer where $a_i^{[k]}$ is the i-th node in that layer, $i = 1, ..., n_k$ and $k = 0, 1, 2, ..., L$ in MLP with L layers. The input information is passed to MLP in a vector form as $\mathbf{a}^{[0]} = [x_0, x_1, ..., x_{n_0}]^T$.

Let an $n_k \times n_{k-1}$ matrix

$$\mathbf{W}^{[k]} = \begin{bmatrix} \mathbf{W}_1^{[k]T} \\ \mathbf{W}_2^{[k]T} \\ \vdots \\ \mathbf{W}_{n_k}^{[k]T} \end{bmatrix}_{n_k \times n_{k-1}}$$

denote the weight of the k-th layer where $\mathbf{W}_i^{[k]T}$ is a row vector of length n_{k-1} corresponding to weights for the i-th node in the k-th layer and let

$$\mathbf{b}^{[k]} = \begin{bmatrix} b_1^{[k]} \\ b_2^{[k]} \\ \vdots \\ b_{n_k}^{[k]} \end{bmatrix}_{n_k \times 1}$$

be the bias term with respect to the k-th layer.

Let function $f^{[k]}$ be the activation function for the k-th layer. Typical activation functions operating non-linear transformations include: *sigmoid* σ (2.1), *tanh* (2.2) and *rectified linear* *units (ReLU)* (2.3) [10] given below.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.1}$$

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.2}$$

$$ReLU(z) = max(0, z) \tag{2.3}$$

$$a(z) = z \tag{2.4}$$

Linear activation (2.4) is also used when non-linear activation is not needed.

The forward pass from all nodes $\mathbf{a}^{[k-1]}$ in layer $k-1$ to $\mathbf{a}^{[k]}$ in layer k is operated first by a linear transformation:

$$\begin{aligned}
\mathbf{z}^{[k]} &= \mathbf{W}^{[k]} \cdot \mathbf{a}^{[k-1]} + \mathbf{b}^{[k]} \\
&= \begin{bmatrix} \mathbf{W}_1^{[k]T} \\ \mathbf{W}_2^{[k]T} \\ \vdots \\ \mathbf{W}_{n_k}^{[k]T} \end{bmatrix} \cdot \begin{bmatrix} a_1^{[k]} \\ a_2^{[k]} \\ \vdots \\ a_{n_{k-1}}^{[k]} \end{bmatrix} + \begin{bmatrix} b_1^{[k]} \\ b_2^{[k]} \\ \vdots \\ b_{n_k}^{[k]} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{W}_1^{[k]T} \cdot \mathbf{a}^{[k-1]} + b_1^{[k]} \\ \mathbf{W}_2^{[k]T} \cdot \mathbf{a}^{[k-1]} + b_2^{[k]} \\ \vdots \\ \mathbf{W}_{n_k}^{[k]T} \cdot \mathbf{a}^{[k-1]} + b_{n_k}^{[k]} \end{bmatrix} \\
&= \begin{bmatrix} z_1^{[k]} \\ z_2^{[k]} \\ \vdots \\ z_{n_k}^{[k]} \end{bmatrix}.
\end{aligned}$$

Then followed by a non-linear activation:

$$\begin{aligned}
\mathbf{a}^{[k]} &= f^{[k]}(\mathbf{z}^{[k]}) \\
&= f^{[k]}\left(\begin{bmatrix} z_1^{[k]} \\ z_2^{[k]} \\ \vdots \\ z_{n_k}^{[k]} \end{bmatrix}\right) \\
&= \begin{bmatrix} f^{[k]}(z_1^{[k]}) \\ f^{[k]}(z_2^{[k]}) \\ \vdots \\ f^{[k]}(z_{n_k}^{[k]}) \end{bmatrix}.
\end{aligned}$$

A complete forward pass of one example will repeat these operations form $k = 1$ to $k = L$ and $\mathbf{a}^{[L]}$ at layer L is the final output of a MLP.

2.1.2 Loss Function and Cost Function

The output vector $\hat{\mathbf{y}} = \mathbf{a}^{[L]}$ of an MLP is given by all nodes in the output layer. A loss function $L(\mathbf{y}, \hat{\mathbf{y}})$ measures the distance between network output and ground truth inside one sample. Some commonly used loss functions include L_2 norm (equation (2.5)), mean absolute error within sample (equation (2.6)) and mean squared error within sample (equation (2.7)).

$$\begin{aligned}
L(\mathbf{y}, \hat{\mathbf{y}}) &= L_2(\mathbf{y}, \hat{\mathbf{y}}) \\
&= \sqrt{\sum_{i=1}^{n_L} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}
\end{aligned} \tag{2.5}$$

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n_L} \sum_{i=1}^{n_L} |\mathbf{y}_i - \hat{\mathbf{y}}_i| \tag{2.6}$$

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n_L} \sum_{i=1}^{n_L} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{2.7}$$

A cost function J (equation (2.8)) is the mean value of the loss functions among a group of samples. Cost function is constructed to facilitate the learning process of network. We want to minimize the cost function via backward pass (described below) to make the MLP learn the prediction task successfully.

$$J = \frac{1}{M} \sum_{m=1}^{M} L(\mathbf{y}^{(m)}, \hat{\mathbf{y}}^{(m)}) \tag{2.8}$$

In equation (2.8), $\mathbf{y}^{(m)}$ and $\hat{\mathbf{y}}^{(m)}$ are the ground truth and prediction for the m-th sample inside a group of M samples.

2.1.3 Backward Pass

MLP is trained by minimizing the differential cost function using gradient descent. The basic idea of gradient descent is to find the derivatives of the loss function with respect to weights and biases, then adjust them in the opposite direction of the slope [10].

Let $\delta \mathbf{a}^{[k](m)}$ denote the derivative of the cost function with respect to the vector of all nodes in k-th layer on the m-th sample. Given cost function J, we first compute the derivative of the output layer L:

$$\begin{aligned} \delta \mathbf{a}^{[L](m)} &= \frac{\partial J}{\partial \mathbf{a}^{[L](m)}} \\ &= \begin{bmatrix} \frac{\partial J}{\partial a_1^{[L](m)}} \\ \frac{\partial J}{\partial a_2^{[L](m)}} \\ \vdots \\ \frac{\partial J}{\partial a_{n_L}^{[L](m)}} \end{bmatrix}. \end{aligned}$$

Let $\delta \mathbf{W}^{[k](m)}$, $\delta \mathbf{b}^{[k](m)}$ denote the partial derivatives of the cost function with respect to weight and bias respectively in layer k in the m-th sample. The backward pass for a sample is then operated as finding $\delta\mathbf{W}^{[k](m)}$, $\delta\mathbf{b}^{[k](m)}$ as well as $\delta\mathbf{a}^{[k-1](m)}$, given the derivative of loss function with respect to activation nodes $\delta\mathbf{a}^{[k](m)}$ in the current layer k, from $k = L$ to $k = 1$.

The first step is to find the useful cache $\delta\mathbf{z}^{[k](m)}$ which is the derivative with respect to the linear transformation item $\mathbf{z}^{[k](m)}$ before non-linear transformation in layer k:

$$\begin{aligned}
\delta\mathbf{z}^{[k](m)} &= \delta\mathbf{a}^{[k](m)} \circ f'^{[k]}(\mathbf{z}^{[k](m)}) \\
&= \begin{bmatrix} \frac{\partial J}{\partial a_1^{[k](m)}} \\ \frac{\partial J}{\partial a_2^{[k](m)}} \\ \vdots \\ \frac{\partial J}{\partial a_{n_k}^{[k](m)}} \end{bmatrix} \circ \begin{bmatrix} f'^{[k]}(z_1^{[k](m)}) \\ f'^{[k]}(z_2^{[k](m)}) \\ \vdots \\ f'^{[k]}(z_{n_k}^{[k](m)}) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial J}{\partial a_1^{[k](m)}} \times f'^{[k]}(z_1^{[k](m)}) \\ \frac{\partial J}{\partial a_2^{[k](m)}} \times f'^{[k]}(z_2^{[k](m)}) \\ \vdots \\ \frac{\partial J}{\partial a_{n_k}^{[k](m)}} \times f'^{[k]}(z_{n_k}^{[k](m)}) \end{bmatrix}_{n_k \times 1} ,
\end{aligned}$$

where '$\circ$' denotes the Hadamard product which is a element-wise multiplication product of two equal size matrices. The derivatives f' of possible activation functions f listed before are:

$$\begin{aligned}
\sigma'(z) &= \sigma(z)(1 - \sigma(z)) \\
tanh'(z) &= 1 - tanh^2(z) \\
ReLU'(z) &= \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases} .
\end{aligned}$$

Then we apply the chain rule to find

$$\begin{aligned}\delta\mathbf{W}^{[k](m)} &= \delta\mathbf{z}^{[k](m)} \cdot \mathbf{a}^{[k-1](m)T},\\ \delta\mathbf{b}^{[k](m)} &= \delta\mathbf{z}^{[k](m)}\end{aligned}$$

as well as

$$\delta\mathbf{a}^{[k-1](m)} = \mathbf{W}^{[k](m)T} \cdot \delta\mathbf{z}^{[k](m)}$$

which is needed for backward pass in the previous layer $k-1$. We will keep doing this from $k = L$ until $k = 1$ to get all necessary derivatives in one backward pass of the m-th sample. After repeating the backward pass for all M samples, we take the average of derivatives among all samples

$$\begin{aligned}\delta\mathbf{W}^{[k]} &= \frac{1}{M}\sum_{m=1}^{M}\delta\mathbf{W}^{[k](m)},\\ \delta\mathbf{b}^{[k]} &= \frac{1}{M}\sum_{m=1}^{M}\delta\mathbf{b}^{[k](m)},\end{aligned}$$

and use $\delta\mathbf{W}^{[k]}$ and $\delta\mathbf{b}^{[k]}$ to train the model.

2.1.4 Train the MLP

The learning problem is then converted to an optimization problem and our goal is to minimize the cost function by training the parameters of MLP. *Adaptive Moment Estimation* (Adam) [8] is the optimization algorithm we will use to minimize the cost function J. Adam combines and takes the advantage of Mommentums and RMSprops optimization algorithms which are two basic optimization algorithms in training neural network model [8]. We will update all weights and biases by using the derivatives we have computed in the backward pass.

To implement Adam optimization algorithm, we first initialize Momentums $V_{\delta\mathbf{W}^{[k]}}$, $V_{\delta\mathbf{b}^{[k]}}$ and RMSprops $S_{\delta\mathbf{W}^{[k]}}$, $S_{\delta\mathbf{b}^{[k]}}$ for $\mathbf{W}^{[k]}$ and $\mathbf{b}^{[k]}$ as

$$V_{\delta\mathbf{W}^{[k]}} = 0,\ V_{\delta\mathbf{b}^{[k]}} = 0,\ S_{\delta\mathbf{W}^{[k]}} = 0,\ S_{\delta\mathbf{b}^{[k]}} = 0,$$

for $k = 1, 2, ..., L$.

Suppose the model is updated for a total of T iterations. On the t-th iteration , we use $\delta\mathbf{W}^{[k]}$ and $\delta\mathbf{b}^{[k]}$ computed from the backward pass to update the Momentums and RMSprops in layer k:

$$\begin{aligned}
V_{\delta\mathbf{W}^{[k]}} &= \beta_1 \cdot V_{\delta\mathbf{W}^{[k]}} + (1-\beta_1)\cdot \delta\mathbf{W}^{[k]},\\
V_{\delta\mathbf{b}^{[k]}} &= \beta_1 \cdot V_{\delta\mathbf{b}^{[k]}} + (1-\beta_1)\cdot \delta\mathbf{b}^{[k]},\\
S_{\delta\mathbf{W}^{[k]}} &= \beta_2 \cdot S_{\delta\mathbf{W}^{[k]}} + (1-\beta_2)\cdot (\delta\mathbf{W}^{[k]})^2,\\
S_{\delta\mathbf{b}^{[k]}} &= \beta_2 \cdot S_{\delta\mathbf{b}^{[k]}} + (1-\beta_2)\cdot (\delta\mathbf{b}^{[k]})^2.
\end{aligned}$$

Then Momentums and RMSprops are then corrected by β_1 and β_2 via

$$\begin{aligned}
V^{corrected}_{\delta\mathbf{W}^{[k]}} &= \frac{V_{\delta\mathbf{W}^{[k]}}}{1-\beta_1^t},\\
V^{corrected}_{\delta\mathbf{b}^{[k]}} &= \frac{V_{\delta\mathbf{b}^{[k]}}}{1-\beta_1^t},\\
S^{corrected}_{\delta\mathbf{W}^{[k]}} &= \frac{S_{\delta\mathbf{W}^{[k]}}}{1-\beta_2^t},\\
S^{corrected}_{\delta\mathbf{b}^{[k]}} &= \frac{S_{\delta\mathbf{b}^{[k]}}}{1-\beta_2^t}.
\end{aligned}$$

The last step is to use the corrected Momentums and RMSprops to update weights and biases:

$$
\begin{aligned}
\mathbf{W}^{[k]} &= \mathbf{W}^{[k]} - \alpha \cdot \frac{V^{corrected}_{\delta \mathbf{W}^{[k]}}}{\sqrt{S^{corrected}_{\delta \mathbf{W}^{[k]}}} + \epsilon}, \\
\mathbf{b}^{[k]} &= \mathbf{b}^{[k]} - \alpha \cdot \frac{V^{corrected}_{\delta \mathbf{b}^{[k]}}}{\sqrt{S^{corrected}_{\delta \mathbf{b}^{[k]}}} + \epsilon}.
\end{aligned}
$$

The scalar learning rate α is used to determine how fast we want to update every time and we usually set the hyper-parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. Under Adam optimization algorithm, the learning rate adapts itself during the training steps so the parameters in MLP will be updated more slightly when we are getting close to the target value [8]. By such a unique algorithm, we are usually faster to achieve the optimization task.

The process is done in iteration by making several passes over all the training samples. A pass over all training samples is called an epoch [8] and after every epoch the parameters are closer to their optimized values which minimizes the cost function. The model is always trained until the cost function cannot be minimized anymore on a validation set. When the data set is very large, calculating the cost and gradient over the whole training samples will be computationally slow. Therefore, a transformation of Adam called mini-batched Adam is implemented. The equal amount of subsets of the divided training samples are called mini-batches and the cost and parameters are updated over every batch.

2.2 Recurrent Neural Networks

2.2.1 Need for Recurrent Neural Networks for Sequential Data

It would be necessary and worthwhile to understand why there is a need for *recurrent neural network* (RNN) and the shortage of MLP in modelling sequential data.

Sequential data basically are just ordered data, where related things follow each other. Examples include financial data, the DNA sequence or the speech record. The most popular type of sequential data is perhaps time series, which is a series of data points indexed in a time order. There are many different tasks and applications when dealing with sequential data. For example, speech recognition is a typical task where we want the output to be a sequence of text when we receive an audio clip which is another sequence of data. Another example is machine translation, the computer software needs to translate a sequence of text in one language to a new sequence of text in another language.

To perform tasks with sequential data like above examples, we could use a regular MLP and feed it the entire sequence input. The information will be taken all at once at the input layer and move straight in one direction to the output layer through the hidden layers. This causes the first problem when training the model. One can definitely define a MLP having input layer with the number of nodes matching the length of input sequence and output layer with the number of nodes matching the length of expected output sequence, but input size and output size would be fixed which is quite limiting. Like the machine translation application mentioned above, you may have many samples of input text sequences and expect output sequences to vary in length. One mechanism to account for sequential dependency is to move a fixed size window over the sequence data [4]. This technique is often used when we apply classic time series model in auto-regressive data. Finding the optimal window size is very important to achieve the task: a small window size does not capture enough information whereas large window size just brings unnecessary noise. More importantly, if there is a long-range dependency in sequential data, a window-based method will not work.

The second problem with the regular NN is that it does not share features learned across different positions of the sequence. Since we input entire sequence at once, each piece of information has its own corresponding weights and biases throughout the MLP. Some identical features repeated in a sequence will be treated independently but in fact we want them to be learned together by the model.

2.2.2 Forward Pass of RNN

A RNN is a special type of neural network suitable for processing sequential data. The main feature of RNN is a state vector $\mathbf{h}_t$ (the hidden units) which maintains a memory of all previous elements of the sequence [7]. The most simple RNN is shown in Figure 2.2. As can be seen, RNN has a recurrent connection which connects the hidden states through time.

At time t, the RNN will receive the input of the current element, a vector $\mathbf{x}_t$, in the sequence and hidden state from the previous time step $\mathbf{h}_{t-1}$. Hidden state $\mathbf{h}_t$ is then updated and an output $\mathbf{y}_t$ corresponding to current time step is calculated if it's needed. In this way, the hidden state $\mathbf{h}_t$ involves information depending on $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_t$. $\mathbf{W}_{hx}$ is the weight matrix between the input and hidden states. $\mathbf{W}_{hh}$ is the weight matrix for the recurrent transition between one hidden state to the next. $\mathbf{W}_{yh}$ is the weight matrix for new updated hidden state to output transition. Equation (2.9) shows how these operations work at time step t.

$$\begin{aligned} \mathbf{h}_t &= tanh(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h), \\ \mathbf{y}_t &= g(\mathbf{W}_{yh}\mathbf{h}_t + \mathbf{b}_y). \end{aligned} \tag{2.9}$$

The input $\mathbf{x}_t$ is an $n \times 1$ vector if your input data has dimension n at every time step or we simply want to use n possibly correlated single variable sequential data to do prediction. The weights $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ are then $n_{state} \times n$ square matrices and bias $\mathbf{b}_h$ is $n_{state} \times 1$ vector where n_{state} is the number of nodes in hidden states where the model need to pass the information. The choice of weight $\mathbf{W}_{yh}$, bias $\mathbf{b}_y$ and function g depend on the optional output y_t and we sometimes may not need them based on the specific tasks.

A standard RNN as shown in Figure 2.2 is also a regular MLP if one considers how it behaves during operation. As shown on the right side of the figure, once the network is unfolded in time, it can be considered as a regular MLP with the number of layers the same as the number of time steps. Since the same weights and biases are used at different layers,

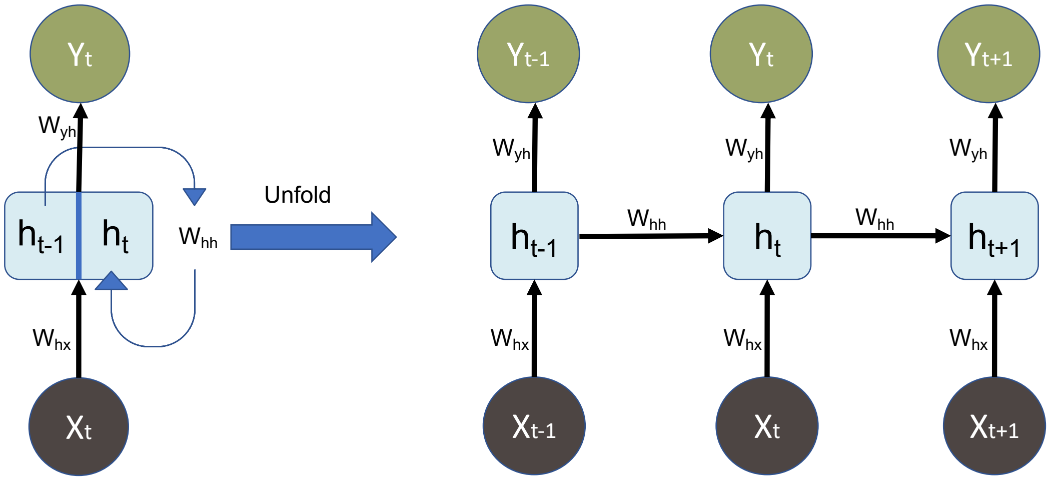


Figure 2.2: A standard RNN

an RNN can take the input of sequences with various lengths. At each time step, a new input is received and the hidden state is updated, so the information can flow in the RNN for an arbitrary number of time steps, allowing the RNN to maintain the memory of all the past information. In Figure 2.2, the left-hand side illustrates the folded state of the RNN and the right-hand side illustrates the unfolded state. The folded state simply shows the network of a complete sequence.

A major difference in RNN as compared to MLP is that, in RNN, all the weights and biases at any step in RNN will have the same value. However, these weights will be different at different layers in MLP. This is because the same task is performed at each step with different inputs. This reduces the total number of parameters an RNN will need to learn.

The structure of RNN could vary according to different tasks. Except the unfolded cases illustrated in Figure 2.2 where input and output sequences have the same length and we are giving prediction at each time step, the RNN could vary due to the different purposes in the task. Different RNN structures are illustrated in Figure 2.3. The 'one to many' could be

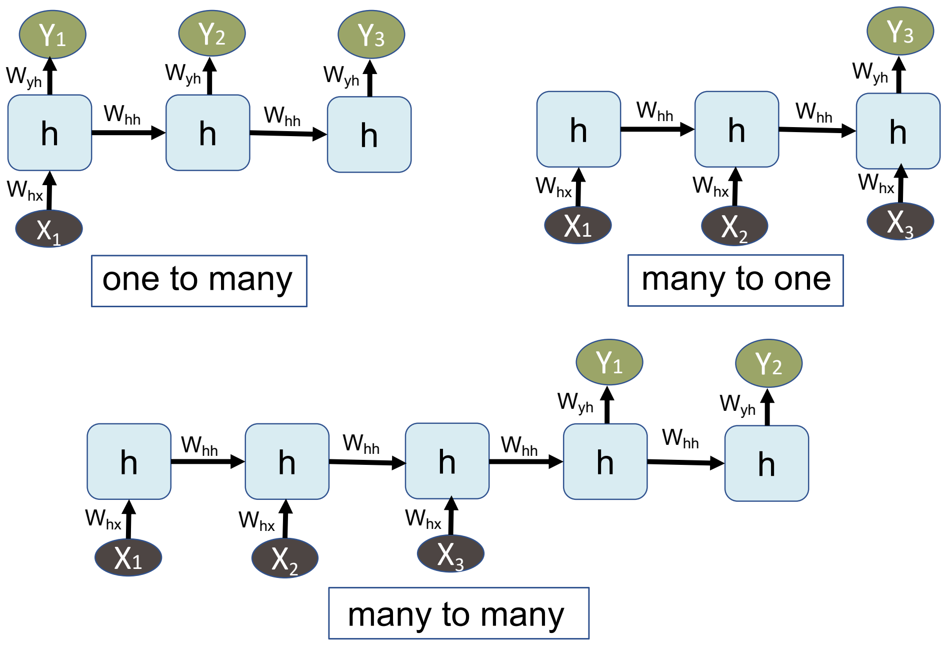


Figure 2.3: Different RNN Structures

applied when we try to generalize a piece of music when it is provided with only the started note. ‘Many to one’ structure could be applied to emotion classification problem where we want a positive or negative judgment when a sentence of statement is provided. ’Many to many’ structure is more mentioned in machine translation task. For example, the length of input French text will be different from that of the translated English text. The model would read the input French text word by word at first and store all information in the hidden state. Then it will read the hidden state and start translating it into English text word by word until a complete sentence is made.

2.2.3 Back-Propagation Through Time

RNN is trained by unfolding RNN and copying the weights for each time step. RNN can be treated as a MLP and can be trained in a similar way through backward pass in MLP. This approach to train RNN is called *back-propagation through time* (BPTT) [19]. We will use many to one RNN structure as example to show BPTT. Let

$$\begin{aligned}
\delta \mathbf{h}_t &= \frac{\partial J}{\partial \mathbf{h}_t}, \\
\delta \mathbf{W}_{hx}^t &= \frac{\partial J}{\partial \mathbf{W}_{hx}}, \\
\delta \mathbf{W}_{hh}^t &= \frac{\partial J}{\partial \mathbf{W}_{hh}}, \\
\delta \mathbf{b}_h^t &= \frac{\partial J}{\partial \mathbf{b}_h}
\end{aligned}$$

be the derivatives of cost function with respect to $\mathbf{h}_t, \mathbf{W}_{hh}$, $\mathbf{W}_{hx}$ and $\mathbf{b}_y$ at time step t and J is the cost function based on the output at time step T. We first compute $\delta \mathbf{h}_T$ which is a $n_{state} \times 1$ vector.

For the back-propagation process, given $\delta \mathbf{h}_t$ at time step t, we apply chain rule to find

$$\begin{aligned}
\delta \mathbf{W}_{hx}^t &= \frac{\partial J}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}} \\
&= \frac{\partial J}{\partial \mathbf{h}_t} \circ [1 - tanh^2(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)] \cdot \mathbf{x}_t^T,
\end{aligned}$$

$$\begin{aligned}
\delta \mathbf{W}_{hh}^t &= \frac{\partial J}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} \\
&= \frac{\partial J}{\partial \mathbf{h}_t} \circ [1 - tanh^2(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)] \cdot \mathbf{h}_{t-1}^T,
\end{aligned}$$

$$\begin{aligned}
\delta \mathbf{b}_h^t &= \frac{\partial J}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_h} \\
&= \frac{\partial J}{\partial \mathbf{h}_t} \circ [1 - tanh^2(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)]
\end{aligned}$$

as well as

$$\begin{aligned}\delta \mathbf{h}_{t-1} &= \frac{\partial J}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \\ &= \mathbf{W}_{hh}^T \cdot \frac{\partial J}{\partial \mathbf{h}_t} \circ [1 - tanh^2(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)],\end{aligned}$$

where

$$[1 - tanh^2(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)] = \begin{bmatrix} 1 - tanh^2(\mathbf{W}_{hx(1)}\mathbf{x}_t + \mathbf{W}_{hh(1)}\mathbf{h}_{t-1} + b_{h(1)}) \\ 1 - tanh^2(\mathbf{W}_{hx(2)}\mathbf{x}_t + \mathbf{W}_{hh(2)}\mathbf{h}_{t-1} + b_{h(2)}) \\ \vdots \\ 1 - tanh^2(\mathbf{W}_{hx(n)}\mathbf{x}_t + \mathbf{W}_{hh(n)}\mathbf{h}_{t-1} + b_{h(n)}) \end{bmatrix}$$

is an $n_{state} \times 1$ vector. $\mathbf{W}_{(i)}$ is the i-th row of the weight matrix $\mathbf{W}$ and $b_{h(i)}$ means the i-th element in bias $\mathbf{b}_h$.

By repeating this process from $t = T$ to $t = 1$, we finish BPTT in one iteration for one sample. At the end of one iteration, we take the average of derivatives with respect to weights and biases from all time steps:

$$\begin{aligned}\delta \mathbf{W}_{hh} &= \frac{1}{T}\sum_{t=1}^{T} \delta \mathbf{W}_{hh}^t, \\ \delta \mathbf{W}_{hx} &= \frac{1}{T}\sum_{t=1}^{T} \delta \mathbf{W}_{hx}^t, \\ \delta \mathbf{b}_h &= \frac{1}{T}\sum_{t=1}^{T} \delta \mathbf{b}_h^t.\end{aligned}$$

Then apply $\delta \mathbf{W}_{hh}$, $\delta \mathbf{W}_{hx}$ and $\delta \mathbf{b}_h$ in Adam optimization algorithm to update $\mathbf{W}_{hh}$, $\mathbf{W}_{hx}$ and $\mathbf{b}_h$.

2.2.4 RNN Training and Challenges

Unfortunately, a simple RNN can be difficult to train. This difficulty arises because the back-propagation of gradients within a neural network is a recursive multiplication process. In RNN, the gradients can accumulate during an update and result in very large gradients. These in turn result in a large update to the network weights, therefore, a very unstable network. If the gradients are very small they will shrink exponentially, making the network hard to be improved. These problems are called gradient exploding and gradient vanishing respectively.

2.3 Long Short-Term Memory

Long Short-Term Memory network, usually called 'LSTM', is an advanced type of recurrent neural network. It was introduced by Hochhreiter and Schmidhhuber (1997) and designed for solving the vanishing gradient problem in the simple RNN [7]. LSTM has been proved to be stable and powerful for learning long-range dependence in various studies [13, 3].

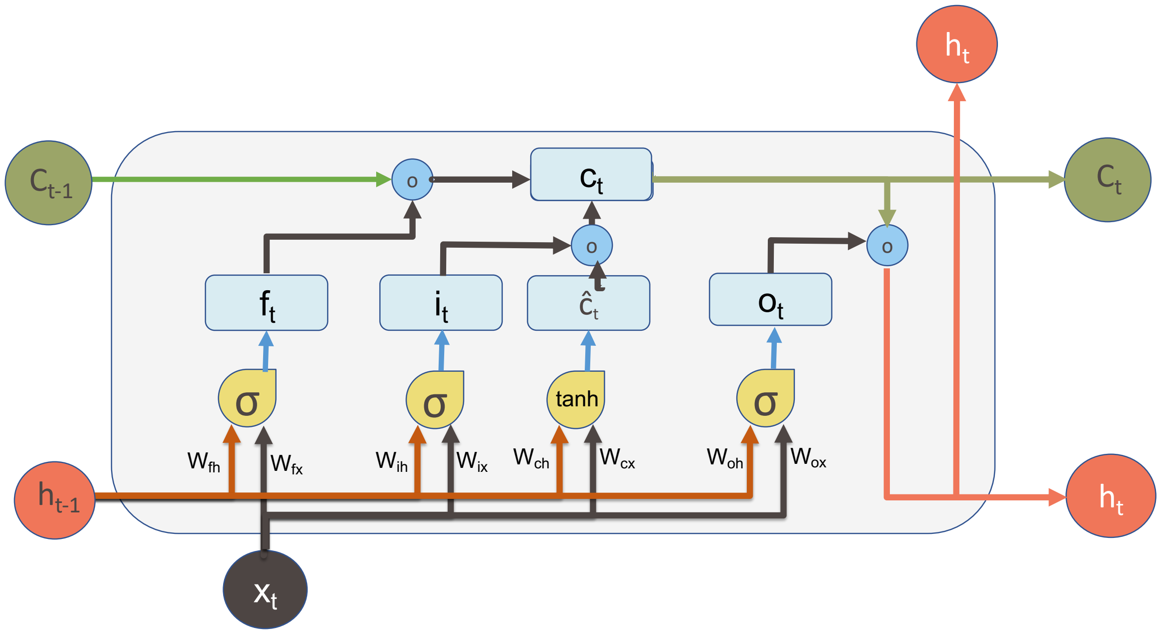


Figure 2.4: LSTM block at time t

2.3.1 Forward Pass of LSTM

The major innovation of LSTM is the memory cell $\mathbf{c}_t$ which essentially acts as an accumulator of the state information. The memory cell is updated by several self-parameterized controlling gates. When a new input $\mathbf{x}_t$ comes at time t, its information will be accumulated to memory cell if the input gate $\mathbf{i}_t$ is activated. Also, the past cell $\mathbf{c}_{t-1}$ could be forgotten in this process if the forgotten gate $\mathbf{f}_t$ is on. Whether the latest cell output $\mathbf{c}_t$ will be propagated to the final state $\mathbf{h}_t$ is further controlled by the output gate $\mathbf{o}_t$. All gates $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{o}_t$ as well as memory cell $\mathbf{c}_t$ and hidden state $\mathbf{h}_t$ are of the same dimension which are $n_{state} \times 1$ vectors.

Hidden state $\mathbf{h}_0$ and memory cell $\mathbf{c}_0$ are all initialized to a zero vector $\mathbf{0}$ at time 0. At time step t ($t = 1, 2, ..., T$), given $\mathbf{h}_{t-1}, \mathbf{c}_{t-1}$ from the last time step $t-1$ and current input $\mathbf{x}_t$, the forward pass of LSTM is operated by first computing the updated memory cell $\tilde{\mathbf{c}}_t$:

$$\tilde{\mathbf{c}}_t = tanh(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{ch}\mathbf{h}_{t-1} + \mathbf{b}_c). \tag{2.10}$$

Then an input gate $\mathbf{i}_t$ is introduced to control how much information in $\tilde{\mathbf{c}}_t$ will flow into the new memory $\mathbf{c}_t$ and a forgotten gate $\mathbf{f}_t$ is also introduced to control how much information in the previous memory cell $\mathbf{c}_{t-1}$ should be remembered:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{b}_i), \tag{2.11}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{b}_f), \tag{2.12}$$

$$\mathbf{c}_t = \mathbf{i}_t \circ \tilde{\mathbf{c}}_t + \mathbf{f}_t \circ \mathbf{c}_{t-1}, \tag{2.13}$$

Finally, the output gate $\mathbf{o}_t$ is also introduced to determine which part of the memory cell $\mathbf{c}_t$ should flow into the hidden state $\mathbf{h}_t$:

$$\mathbf{o}_t = \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{b}_o), \tag{2.14}$$

$$\mathbf{h}_t = \mathbf{o}_t \circ tanh(\mathbf{c}_t). \tag{2.15}$$

One advantage of using the memory cell and gates to control the information flow is that the gradient will be trapped in the cell and be prevented from vanishing too quickly which is a normal problem in simple RNNs. This multi-variate input version of LSTM is also called *fully connected LSTM* (FC-LSTM). Multiple LSTMs can be stacked to form more complex structures as in Figure 2.5. Such model can be applied in some real life sequence modelling problems.

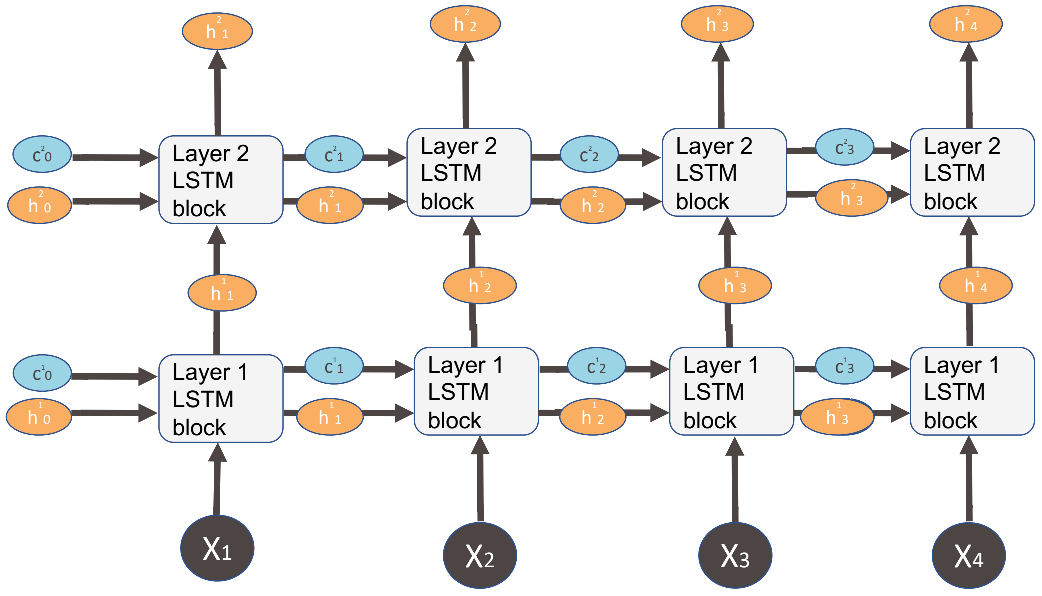


Figure 2.5: Example of a 2-layer stacked LSTM

2.3.2 Back-Propagation of LSTM

Let $\mathbf{W}_f = [\mathbf{W}_{fh}|\mathbf{W}_{fx}]$, $\mathbf{W}_i = [\mathbf{W}_{ih}|\mathbf{W}_{ix}]$, $\mathbf{W}_c = [\mathbf{W}_{ch}|\mathbf{W}_{cx}]$ and $\mathbf{W}_o = [\mathbf{W}_{oh}|\mathbf{W}_{ox}]$ which are $n_{state} \times (n_{state} + n)$ matrices. The back-propagation of LSTM at time step t is to derive $\delta\mathbf{W}^t_f$, $\delta\mathbf{W}^t_i$, $\delta\mathbf{W}^t_c$, $\delta\mathbf{W}^t_o$, $\delta\mathbf{b}^t_f$, $\delta\mathbf{b}^t_i$, $\delta\mathbf{b}^t_c$ and $\delta\mathbf{b}^t_o$ as well as $\delta\mathbf{h}_{t-1}$ and $\delta\mathbf{c}_{t-1}$ for previous time step $t-1$, given $\delta\mathbf{h}_t$ and $\delta\mathbf{c}_t$. A small tricky step is that the derivative of loss function with respect to current memory cell $\mathbf{c}_t^{current} = \mathbf{i}_t \circ \tilde{\mathbf{c}}_t + \mathbf{f}_t \circ \mathbf{c}_{t-1}$ is

$$\begin{aligned}
\delta \mathbf{c}_t^{current} &= \frac{\partial J}{\partial \mathbf{c}_t^{current}} \\
&= \frac{\partial J}{\partial \mathbf{c}_t} + \frac{\partial J}{\mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} \\
&= \delta \mathbf{c}_t + \delta \mathbf{h}_t \circ \mathbf{o}_t \circ [1 - tanh^2(\mathbf{c}_t)]
\end{aligned}$$

since the memory cell $\mathbf{c}_t$ flows into next time step in two routes. The first route is as $\mathbf{c}_t$ does directly and the second route is included into $\mathbf{h}_t = \mathbf{o}_t \circ tanh(\mathbf{c}_t)$.

Let's define

$$\begin{aligned}
Z(\tilde{\mathbf{c}}_t) &= \mathbf{W}_c \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_c, \\
Z(\mathbf{i}_t) &= \mathbf{W}_i \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i, \\
Z(\mathbf{f}_t) &= \mathbf{W}_f \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f, \\
Z(\mathbf{o}_t) &= \mathbf{W}_o \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_o
\end{aligned}$$

as the linear transformations of the updated memory cell and all gates, then

$$\begin{aligned}
\delta Z(\tilde{\mathbf{c}}_t) &= \frac{\partial J}{\partial Z(\tilde{\mathbf{c}}_t)} = \frac{\partial J}{\partial \mathbf{c}_t^{current}} \cdot \frac{\partial \mathbf{c}_t^{current}}{\partial \tilde{\mathbf{c}}_t} \cdot \frac{\partial \tilde{\mathbf{c}}_t}{\partial Z(\tilde{\mathbf{c}}_t)} \\
&= \delta \mathbf{c}_t^{current} \circ \mathbf{i}_t \circ [1 - \tilde{\mathbf{c}}_t^2] \\
&= (\delta \mathbf{c}_t + \delta \mathbf{h}_t \circ \mathbf{o}_t \circ [1 - tanh^2(\mathbf{c}_t)]) \circ \mathbf{i}_t \circ (1 - \tilde{\mathbf{c}}_t^2),
\end{aligned}$$

$$\begin{aligned}\delta Z(\mathbf{i}_t) = \frac{\partial J}{\partial Z(\mathbf{i}_t)} &= \frac{\partial J}{\partial \mathbf{c}_t^{current}} \cdot \frac{\partial \mathbf{c}_t^{current}}{\partial \mathbf{i}_t} \cdot \frac{\partial \mathbf{i}_t}{\partial Z(\mathbf{i}_t)} \\ &= \delta \mathbf{c}_t^{current} \circ \tilde{\mathbf{c}}_t \circ \mathbf{i}_t \circ (1 - \mathbf{i}_t) \\ &= (\delta \mathbf{c}_t + \delta \mathbf{h}_t \circ \mathbf{o}_t \circ [1 - tanh^2(\mathbf{c}_t)]) \circ \tilde{\mathbf{c}}_t \circ \mathbf{i}_t \circ (1 - \mathbf{i}_t),\end{aligned}$$

$$\begin{aligned}\delta Z(\mathbf{f}_t) = \frac{\partial J}{\partial Z(\mathbf{f}_t)} &= \frac{\partial J}{\partial \mathbf{c}_t^{current}} \cdot \frac{\partial \mathbf{c}_t^{current}}{\partial \mathbf{f}_t} \cdot \frac{\partial \mathbf{f}_t}{\partial Z(\mathbf{f}_t)} \\ &= \delta \mathbf{c}_t^{current} \circ \mathbf{c}_{t-1} \circ \mathbf{f}_t \circ (1 - \mathbf{f}_t) \\ &= (\delta \mathbf{c}_t + \delta \mathbf{h}_t \circ \mathbf{o}_t \circ [1 - tanh^2(\mathbf{c}_t)]) \circ \mathbf{c}_{t-1} \circ \mathbf{f}_t \circ (1 - \mathbf{f}_t),\end{aligned}$$

$$\begin{aligned}\delta Z(\mathbf{o}_t) = \frac{\partial J}{\partial Z(\mathbf{o}_t)} &= \frac{\partial J}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} \cdot \frac{\partial \mathbf{o}_t}{\partial Z(\mathbf{o}_t)} \\ &= \delta \mathbf{h}_t \circ tanh(\mathbf{c}_t) \circ \mathbf{o}_t \circ (1 - \mathbf{o}_t).\end{aligned}$$

So the derivative with respect to each weight and bias at time step t are

$$\delta \mathbf{W}_c^t = \delta Z(\tilde{\mathbf{c}}_t) \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix}^T,$$

$$\delta \mathbf{W}_i^t = \delta Z(\mathbf{i}_t) \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix}^T,$$

$$\delta \mathbf{W}_f^t = \delta Z(\mathbf{f}_t) \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix}^T,$$

$$\delta \mathbf{W}_o^t = \delta Z(\mathbf{o}_t) \cdot \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix}^T,$$

$$\delta \mathbf{b}_c = \delta Z(\tilde{\mathbf{c}}_t), \delta \mathbf{b}_i = \delta Z(\mathbf{i}_t), \delta \mathbf{b}_f = \delta Z(\mathbf{f}_t), \delta \mathbf{b}_o = \delta Z(\mathbf{o}_t)$$

and the $\delta \mathbf{c}_{t-1}$ and $\delta \mathbf{h}_{t-1}$ for previous time step $t-1$ are

$$\begin{aligned}
\delta \mathbf{c}_{t-1} &= \frac{\partial J}{\partial \mathbf{c}_{t-1}} \\
&= \frac{\partial J}{\partial \mathbf{c}_t^{current}} \cdot \frac{\partial \mathbf{c}_t^{current}}{\partial \mathbf{c}_{t-1}} \\
&= \delta \mathbf{c}_t^{current} \circ \mathbf{f}_t \\
&= (\delta \mathbf{c}_t + \delta \mathbf{h}_t \circ \mathbf{o}_t \circ [1 - tanh^2(\mathbf{c}_t)]) \circ \mathbf{f}_t
\end{aligned}$$

and

$$\begin{aligned}
\delta \mathbf{h}_{t-1} &= \frac{\partial J}{\partial \mathbf{h}_{t-1}} \\
&= \frac{\partial J}{\partial Z(\tilde{\mathbf{c}}_t)} \cdot \frac{\partial Z(\tilde{\mathbf{c}}_t)}{\partial \mathbf{h}_{t-1}} + \frac{\partial J}{\partial Z(\mathbf{i}_t)} \cdot \frac{\partial Z(\mathbf{i}_t)}{\partial \mathbf{h}_{t-1}} + \frac{\partial J}{\partial Z(\mathbf{f}_t)} \cdot \frac{\partial Z(\mathbf{f}_t)}{\partial \mathbf{h}_{t-1}} + \frac{\partial J}{\partial Z(\mathbf{o}_t)} \cdot \frac{\partial Z(\mathbf{o}_t)}{\partial \mathbf{h}_{t-1}} \\
&= \mathbf{W}_{ch}^t \cdot \delta Z(\tilde{\mathbf{c}}_t) + \mathbf{W}_{ih}^t \cdot \delta Z(\mathbf{i}_t) + \mathbf{W}_{fh}^t \cdot \delta Z(\mathbf{f}_t) + \mathbf{W}_{oh}^t \cdot \delta Z(\mathbf{o}_t).
\end{aligned}$$

Then we take the average of derivatives with respect to weights and biases from all time steps in one iteration to update the gradient:

$$\begin{aligned}
\delta \mathbf{W}_c &= \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{W}_c^t \,, \delta \mathbf{b}_c = \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{b}_c^t, \\
\delta \mathbf{W}_i &= \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{W}_i^t \,, \delta \mathbf{b}_i = \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{b}_i^t, \\
\delta \mathbf{W}_f &= \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{W}_f^t \,, \delta \mathbf{b}_f = \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{b}_f^t, \\
\delta \mathbf{W}_o &= \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{W}_o^t \,, \delta \mathbf{b}_o = \frac{1}{T} \sum_{t=1}^{T} \delta \mathbf{b}_o^t
\end{aligned}$$

and use Adam algorithm to train LSTM as we train MLP and simple RNN.

2.4 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are specialized neural networks which are well suited for the processing of images. A typical CNN as shown in Figure 2.6 usually consists of multiple convolutional layers and the goal of these convolutional layers is to extract the features from inputs which results in multiple feature maps via filters (or kernels). For example, the 3×3 filter $\mathbf{K}$ in Figure 2.7 can be used to detect all vertical edges in an image. Instead of using defined filters, convolutional neural network will learn these filters through backward pass.

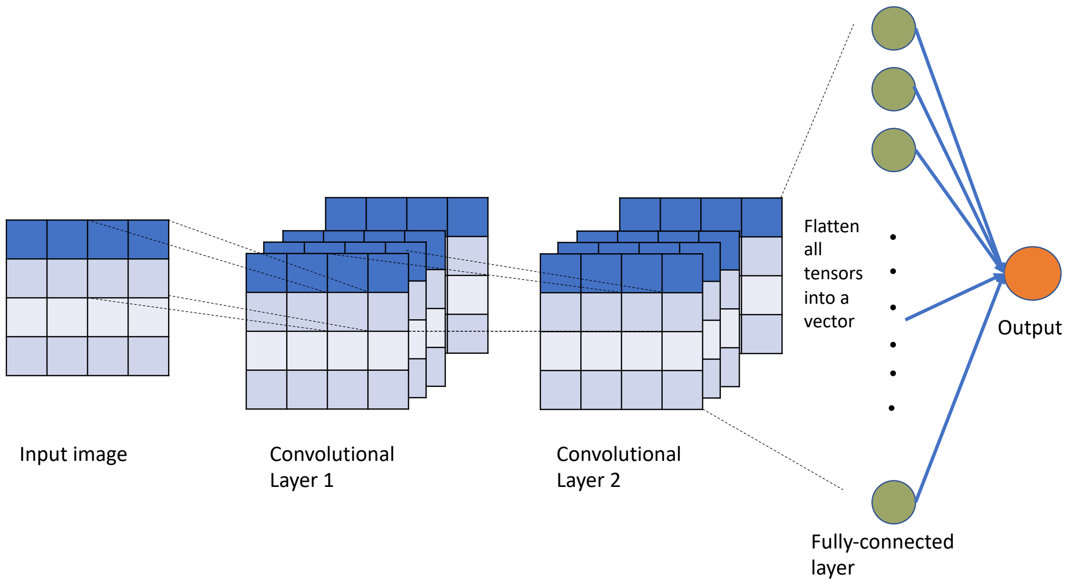


Figure 2.6: Example of CNN

2.4.1 Forward Pass of CNN

Convolution preserves the relationship between pixels by learning image features using the small squares of input data. A convolutional layer consists of a set of learnable filters or kernels, which are the weights of the network. Given an $n \times n$ filter

$$\mathbf{K} = \begin{bmatrix} w_{11} & \dots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{nn} \end{bmatrix}$$

and one $n \times n$ patch of an image

$$\mathbf{I} = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nn} \end{bmatrix}.$$

The convolution of the filter and the image patch is then defined as

$$\mathbf{I} * \mathbf{K} = \sum_{j=1}^{n} \sum_{k=1}^{n} x_{jk} w_{jk}.$$

The idea is that you slide the filter over the image, convolving the filter with each patch of the image as it slides. Let stride s be the number of pixels the filter will move either along the row or the column every step. We can vary the number of pixels the filter moves each time it slides by specifying a stride (so with a stride of 1 the filter sees every possible patch of the image). Each convolution produces an output, and the resulting outputs of all convolutions of a filter with an image produces a new feature map.

We can make this more concrete with an example. In the part of the Figure 2.7 below, we see the result of the convolution of a 3×3 filter

$$\mathbf{K} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

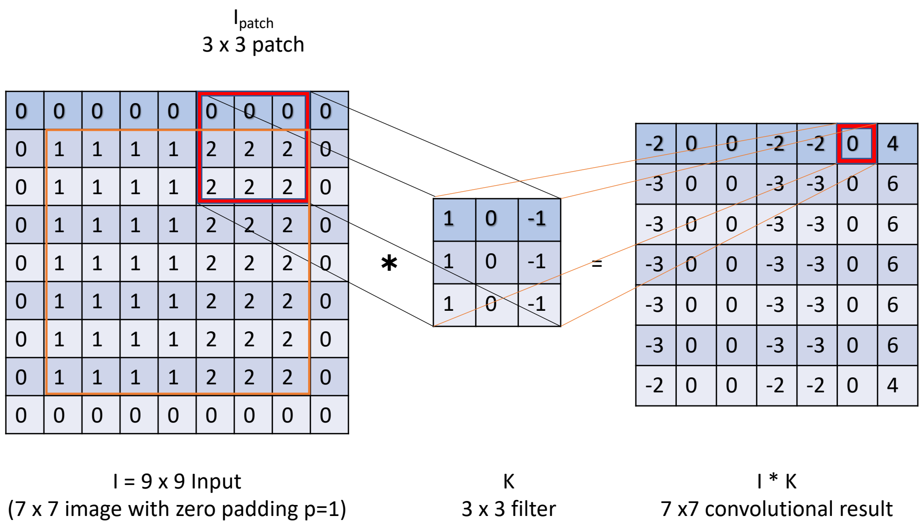
with a patch of the image

$$\mathbf{I}_{patch} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

from

$$\mathbf{I}_{patch} * \mathbf{K} = 0 \times 1 + 0 \times 0 + 0 \times -1 + 2 \times 1 + 2 \times 0 + 2 \times (-1) + 2 \times 1 + 2 \times 0 + 2 \times (-1) = 0.$$

The output $\mathbf{I} * \mathbf{K}$ can be used as a input feature for the next convolution operation which



I_patch
3 x 3 patch
I = 9 x 9 Input
(7 x 7 image with zero padding p=1)
K
3 x 3 filter
I * K
7 x7 convolutional result

Figure 2.7: An example of convolutional layer

is aimed to build multiple-layer CNN.

2.4.2 Zero Padding

Zero padding is a process of adding zero pixels symmetrically to the images or the convolutional layers. It is mostly used in designing the CNN layers when the dimensions of the input volume need to be preserved in the output volume. Let p be number of zero paddings added to a $n \times n$ convolutional layer, then the size of convolutional layer becomes $(n+2p) \times (n+2p)$. As the example shown in Figure 2.7, if we use a 3×3 filter with stride $s = 1$ on a 7×7 image with zero padding $p = 1$, then the convolutional output will be 7×7 which satisfies the usual purpose of zero padding.

2.4.3 Backward Pass of CNN

We are going to show the backward pass of CNN via a simple example. Suppose we have a 3×3 image **I** and a 2×2 filter **K** and an output **O** as shown in figure 2.8. Then the output **O** is computed as

$$\begin{aligned}
o_{11} &= w_{11}x_{11} + w_{12}x_{12} + w_{21}x_{21} + w_{22}x_{22}, \\
o_{12} &= w_{11}x_{12} + w_{12}x_{13} + w_{21}x_{22} + w_{22}x_{23}, \\
o_{21} &= w_{11}x_{21} + w_{12}x_{22} + w_{21}x_{31} + w_{22}x_{32}, \\
o_{22} &= w_{11}x_{22} + w_{12}x_{23} + w_{21}x_{32} + w_{22}x_{33}.
\end{aligned}$$

Let $\delta o_{ij} = \frac{\partial J}{\partial o_{ij}}$ and $\delta w_{ij} = \frac{\partial J}{\partial w_{ij}}$ be the derivatives of cost function with respect to the outputs and weights respectively. For implementing backward pass for the current layer, we assume δo_{ij}'s are given and the goal is to find δw_{ij} as well as δx_{ij} which is the input for the backward pass of the previous layer if we have multiple convolutional layers. The mathematical equations for the derivatives with respect to weights in the filter are:

$$\delta w_{11} = x_{11}\delta o_{11} + x_{12}\delta o_{12} + x_{21}\delta o_{21} + x_{22}\delta o_{22},$$

$$\delta w_{12} = x_{12}\delta o_{11} + x_{13}\delta o_{12} + x_{22}\delta o_{21} + x_{23}\delta o_{22},$$

$$\delta w_{21} = x_{21}\delta o_{11} + x_{22}\delta o_{12} + x_{31}\delta o_{21} + x_{32}\delta o_{22},$$

$$\delta w_{22} = x_{22}\delta o_{11} + x_{23}\delta o_{12} + x_{32}\delta o_{21} + x_{33}\delta o_{22}.$$

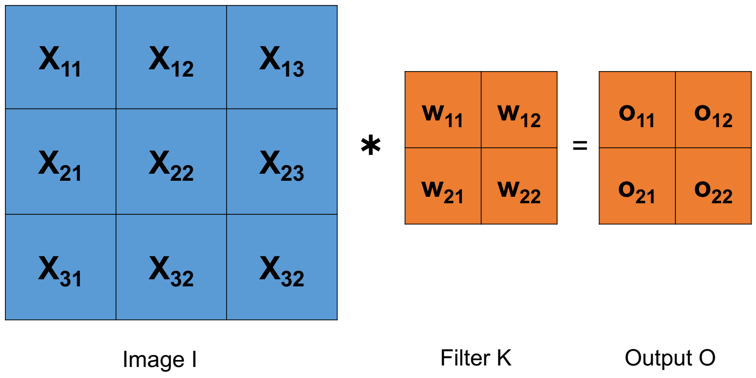


Figure 2.8: A simple example of CNN backward pass

Chapter 3

Model Development for Sequential Images Prediction

We start our model from adding convlutional operation to the FC-LSTM network which has been proved to have the strong ability to learn the temporal relationship. By this innovative modification, our new ConvLSTM network will be able to capture both spatial and temporal relationships from the spatio-temporal input sequence. For this special sequence-to-sequence prediction task, an encoder-forecaster structure will be applied to read the input sequences in variant lengths and predict the output sequence of expected length. A stacked multi-layer ConvLSTM encoder-forecaster network is finally constructed to tackle this complex spatio-temporal sequence prediction problem.

3.1 Add Convolutional Structure to LSTM

Although FC-LSTM has been proved to be powerful for handling the temporal relationship, the major drawback of FC-LSTM is that we lose the spatial information when we unfold the inputs into 1D vectors before we send them to the neural networks. The *convolutional LSTM* (ConvLSTM) captures all spatial information throughout the model. A distinguish feature of ConvLSTM is that all inputs $\mathcal{X}_t$, cell outputs $\mathcal{C}_t$, hidden states $\mathcal{H}_t$ and gates

i_t, o_t, f_t of ConvLSTM, for $t = 1, ..., T$, are 3D tensors whose first two dimensions are rows and columns. One can imagine the inputs and states as vectors standing on spatial grids and each vector has a number of information resources of the corresponding pixel on the grid. The ConvLSTM determines the future values on a grid of memory cells and hidden states by using convolution operation in the input-to-state and state-to-state transitions as illustrated in Figure 3.1. The equations in (3.1) show how a ConvLSTM block at time t is operated, where $*$ is the convolutional operation and W is the filter used in the convolution.

$$\begin{aligned}
\tilde{\mathcal{C}}_t &= tanh(W_{xc} * \mathcal{X}_t + W_{hc} * \mathcal{H}_{t-1} + b_c) \\
i_t &= \sigma(W_{xi} * \mathcal{X}_t + W_{hi} * \mathcal{H}_{t-1} + b_i) \\
f_t &= \sigma(W_{xf} * \mathcal{X}_t + W_{hf} * \mathcal{H}_{t-1} + b_f) \\
\mathcal{C}_t &= f_t \circ \mathcal{C}_t + i_t \circ \tilde{\mathcal{C}}_t \\
o_t &= \sigma(W_{xo} * \mathcal{X}_t + W_{ho} * \mathcal{H}_{t-1} + b_o) \\
\mathcal{H}_t &= o_t \circ tanh(\mathcal{C}_t)
\end{aligned} \tag{3.1}$$

If we view the hidden states and memory cells as representations of the moving objects, then the ConvLSTM with a larger filter W should capture faster motions while a smaller filter should capture slower motions. Also the FC-LSTM can be viewed as a special ConvLSTM as the first two dimensions of the 3D tensors are all 1. We can just treat all features of an image as a 1D vector on a single grid cell.

To ensure the hidden states and memory cells have the same rows and columns as the input, zero-padding is applied before the convolution. Zero-padding does not just provide the fixed dimension states, it can also learn the out-of-domain information. Imagine we are learning a bouncing ball moving in a closed area. Although we can't see the walls, we can infer the existence of the wall by finding the ball bouncing over them again and again, which can be done if there is a strong contrast between the ball and zero-padding boundaries.

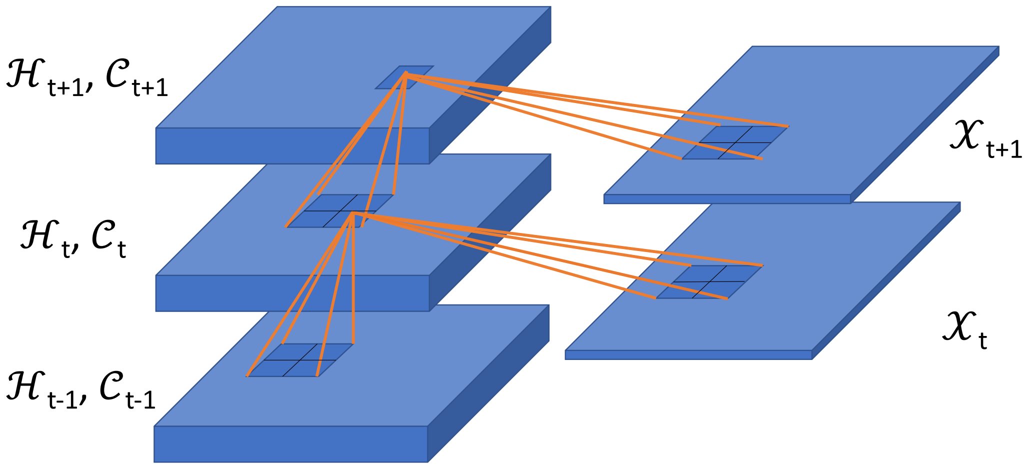

$\mathcal{H}_{t+1}, \mathcal{C}_{t+1}$
$\mathcal{H}_{t}, \mathcal{C}_{t}$
$\mathcal{H}_{t-1}, \mathcal{C}_{t-1}$
$\mathcal{X}_{t+1}$
$\mathcal{X}_{t}$

Figure 3.1: Inner structure of ConvLSTM

3.2 Convolutional LSTM Encoder-Forecaster model

By stacking the multiple ConvLSTM layers and forming the encoder-forecasting structure, we are able to build a network model for the general sequential image prediction problems.

3.2.1 Encoding-Forecasting Structure

For the sequence-to-sequence prediction task, the encoder-forecaster structured network with LSTM units is commonly used. This is a classic approach in languages processing for modelling sequences of words [2]. The model consists of two LSTMs. The encoder LSTM and the forecaster LSTM as shown in Figure 3.2. The encoder LSTM will read the input sequence. After the last time step of input has been read, the forecaster LSTM will take the accumulated hidden states and memory cells as input to predict the future sequence coming after the input sequence. Conditional forecaster LSTM is implemented in our model. A conditional forecaster [16] will take last time step's prediction as the input for current time step's prediction. By incorporating an encoder-forecaster structure into FC-LSTM, the model can take a sequence of arbitrary length as the input and predict the output sequence of any length as wanted.

In order to predict future frames in our spatio-temporal sequence prediction task, the encoder-forecaster model needs information about the background and objects and how objects moved so that the future motions can be extrapolated. The hidden states and memory cells coming from the encoder should capture all necessary information and will be treated as a representation of the input sequence. To implement this LSTM encoder-forecaster network with images, we simply flatten all images row by row into 1D vectors and feed them to the model.

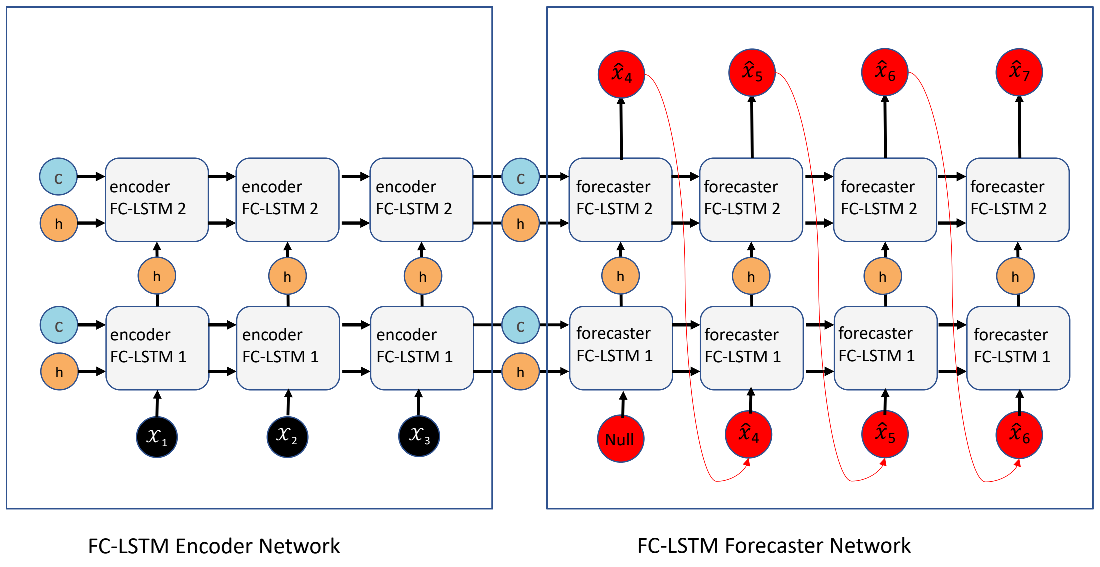


Figure 3.2: Example of a stacked 2-layer FC-LSTM encoder-forecaster network

3.2.2 ConvLSTM Encoder-Forecaster Network

Like FC-LSTM, ConvLSTM can be adopted as building block for more complex structures. For our spatio-temporal sequence forecasting problems, we use the structure shown in Figure 3.3, which consists of encoding and forecasting networks. Like the LSTM encoder-forecaster network in [16], the last memory cell and hidden state from the encoding network are copied as the initial memory cell and hidden state to forecaster network since they have the same dimensions. Both encoder and forecaster networks are stacked with multiple ConvLSTM layers. Since our hidden states have the same dimension as the inputs, we perform a 1×1 convolution on all hidden states in the forecaster network to generate the most likely final predictions:

$$\begin{aligned}\tilde{\mathcal{X}}_{t+1}, \ldots, \tilde{\mathcal{X}}_{t+K} &= \underset{\mathcal{X}_{t+1},\ldots,\mathcal{X}_{t+K}}{\operatorname{argmax}} P(\mathcal{X}_{t+1}, \ldots, \mathcal{X}_{t+K} | \mathcal{X}_{t-J+1}, \mathcal{X}_{t-J+2}, \ldots, \mathcal{X}_{t}) \\ &\approx \underset{\mathcal{X}_{t+1},\ldots,\mathcal{X}_{t+K}}{\operatorname{argmax}} P(\mathcal{X}_{t+1}, \ldots, \mathcal{X}_{t+K} | f_{encoder}(\mathcal{X}_{t-J+1}, \mathcal{X}_{t-J+2}, \ldots, \mathcal{X}_{t})) \\ &\approx g_{forecaster}(f_{encoder}(\mathcal{X}_{t-J+1}, \mathcal{X}_{t-J+2}, \ldots, \mathcal{X}_{t})).\end{aligned}$$

Since the network has multiple stacked ConvLSTM layers, it has stronger representation power which makes it suitable for giving predictions in complex dynamical systems like precipitation nowcasting problems.

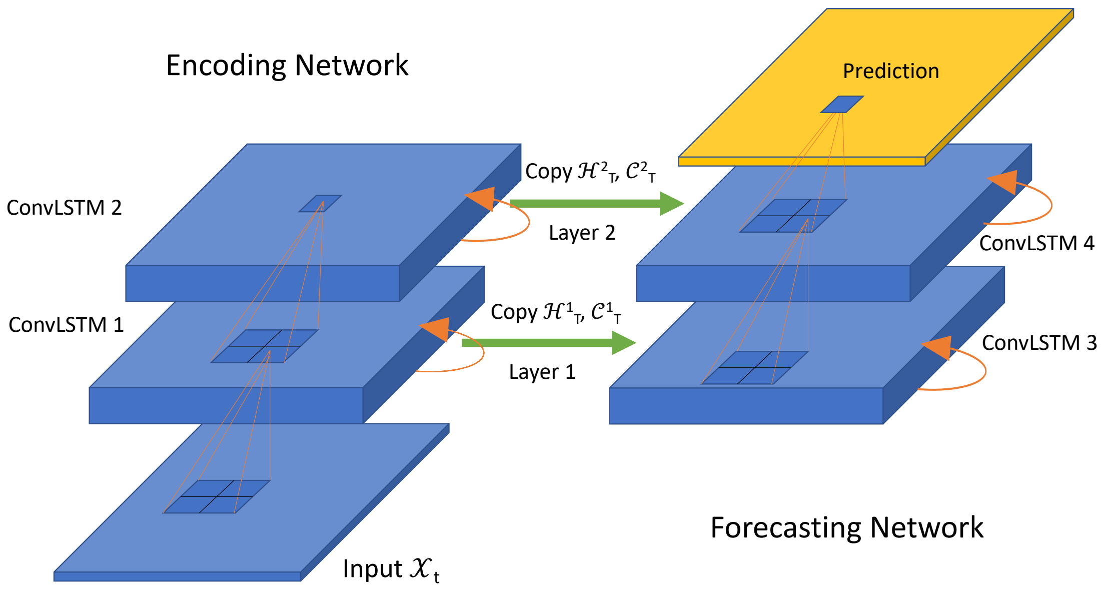


Figure 3.3: A folded example of 2-layer stacked ConvLSTM encoder-forecaster model

3.3 Lost Function for Spatio-temporal Sequence Output

Suppose the spatio-temporal output is a sequence of K grey scale images with each image of dimension $N_1 \times N_2$. We will use the pixel-wise mean squared error among the entire spatio-temporal output sequence in a sample as the loss function. The simplest way to define this loss function is to compute the pixel-wise *mean squared error* (MSE) at the last time step. Let $\mathbf{I}^{(m)}_{k,n_1,n_2}$ and $\hat{\mathbf{I}}^{(m)}_{k,n_1,n_2}$ denote the ground truth pixel value and prediction pixel value on the n_1 row and n_2 column grid of the k-th time step output image in the m-th sample, then the loss function L of this sample is:

$$\begin{aligned} L^{(m)} &= MSE(\mathbf{I}^{(m)}, \hat{\mathbf{I}}^{(m)}) \\ &= \frac{1}{K \cdot N_1 \cdot N_2} \sum_{k=1}^{K} \sum_{n_1=1}^{N_1} \sum_{n_2=1}^{N_2} (\mathbf{I}^{(m)}_{k,n_1,n_2} - \hat{\mathbf{I}}^{(m)}_{k,n_1,n_2})^2. \end{aligned}$$

Then cost function among a set of M samples which is used to train the model is:

$$J = \frac{1}{M} \sum_{m=1}^{M} L^{(m)}.$$

This simplest form of cost function will be used to train the model on our experiments later. The pixel-wise *mean absolute error* (MAE) defined as:

$$MAE(\mathbf{I}^{(m)}, \hat{\mathbf{I}}^{(m)}) = \frac{1}{K \cdot N_1 \cdot N_2} \sum_{k=1}^{K} \sum_{n_1=1}^{N_1} \sum_{n_2=1}^{N_2} |\mathbf{I}^{(m)}_{k,n_1,n_2} - \hat{\mathbf{I}}^{(m)}_{k,n_1,n_2}|,$$

is also used to evaluate the learning result of the model.

3.4 Implementation via Keras Under Python 3.6

The ConvLSTM and FC-LSTM encoder-forecaster network models are implemented using TensorFlow as backend under Python 3.6 environment. Keras API which is a model-level library is used to build layers in every model. We use 'ConvLSTM2D' and 'LSTM' functions from Keras to build the recurrent neural network blocks. To incorporate the encoder-forecaster structure into the recurrent neural networks, we need delete line 270 and line 271 in the source code file '*convolutional_recurrent.py*' so that we can store the hidden states and memory cells from the encoder network and then pass them to the forecaster network. An example of how we define a 1-layer ConvLSTM encoder-forecaster network model using Keras 'ConvLSTM2D' function is shown as below:

```
def define_models_1_moving_1(n_filter, filter_size):
# define training encoder
encoder_inputs = Input(shape=(None, 64, 64, 1))
encoder_1 = ConvLSTM2D(filters = n_filter, kernel_size=filter_size,
    padding='same', return_sequences=True, return_state=True,
    kernel_regularizer=l2(0.0005), recurrent_regularizer=l2(0.0005),
    bias_regularizer=l2(0.0005))
encoder_outputs_1, encoder_state_h_1, encoder_state_c_1 = encoder_1(
    encoder_inputs)
# define training decoder
decoder_inputs = Input(shape=(None, 64, 64, 1))
decoder_1 = ConvLSTM2D(filters=n_filter, kernel_size=filter_size, padding=
    'same', return_sequences=True, return_state=True, kernel_regularizer=l2
    (0.0005), recurrent_regularizer=l2(0.0005), bias_regularizer=l2(0.0005)
    )
decoder_outputs_1, _, _ = decoder_1([decoder_inputs, encoder_state_h_1,
    encoder_state_c_1])
decoder_conv3d = Conv3D(filters=1, kernel_size=(1,1,64), padding='same',
    data_format='channels_last', kernel_regularizer=l2(0.0005),
    bias_regularizer=l2(0.0005))
```

```
decoder_outputs = decoder_conv3d(decoder_outputs_1)
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
# define inference encoder
encoder_model = Model(encoder_inputs, [encoder_state_h_1,
    encoder_state_c_1])
# define inference decoder
decoder_state_input_h_1 = Input(shape=(64,64,n_filter))
decoder_state_input_c_1 = Input(shape=(64,64,n_filter))
decoder_output_1, decoder_state_h_1_new, decoder_state_c_1_new = decoder_1
    ([decoder_inputs, decoder_state_input_h_1, decoder_state_input_c_1])
decoder_output = decoder_conv3d(decoder_output_1)
decoder_model = Model([decoder_inputs , decoder_state_input_h_1 ,
    decoder_state_input_c_1], [decoder_output, decoder_state_h_1_new,
    decoder_state_c_1_new])
return model, encoder_model, decoder_model
```

Once a basic ConvLSTM encoder-forecaster network model is defined, one need specify the number of filters for hidden states and memory cells as well as the size of filter by calling the well-defined function above. After the loss function and early-stopping are well set, the model can be trained via TensorFlow as below:

```
train_1_lstm_1, infenc_1_lstm_1, infdec_1_lstm_1 = define_lstm_1_moving_1(
    n_filter=128, filter_size=5)
train_1_lstm_1.compile(loss='mse', optimizer='adam', metrics=['mae'])
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=5)
history_1_lstm_1 = train_1_lstm_1.fit([X1_moving,X2_moving], y_moving,
    batch_size=8, validation_split=0.25, epochs=100, callbacks=[es])
```

The complete python code to build and train all models in my thesis is posted in my github page: `https://github.com/mingkuan94/Thesis_ConvLSTM`.

Chapter 4

Experiments

It is necessary to set a baseline model when testing a new and innovative deep learning model to see if the new model outperforms baseline model. FC-LSTM encoder-forecaster network will be used as the baseline model in the experiments below. We first compare the ConvLSTM encoder-forecaster network with the FC-LSTM encoder-forecaster network on the synthetic moving MNIST dataset to understand the behavior of ConvLSTM based model. We will run the ConvLSTM encoder-forecaster network with different number of layers. We will also study the out-of-domain case as mentioned in [16] on moving MNIST data since it is still an unsolved problem in traditional radar echo extrapolation based method. Then we will build our ConvLSTM networks on more complex radar echo maps dataset.

Both the moving MNIST and radar echo maps data sets contain only grey scale images. Each frame of image has only one channel with integer pixel values from 0 to 255. Both data sets have more than 10000 samples. The moving MNIST data, for example, has 10000 examples with each containing 14 frames of 64×64 images for the training process. This means we have a huge 5-dimension training and validation data denoted by $(10000, 14, 64, 64, 1)$. A popular method of predicting the image data with integer pixel values is to treat all integer pixel values as 266 different classes. Then the information for a single grid on an image is represented by a 266×1 vector with only one element being '1' and the others '0'. So a vector on a grid with only the i-th element being 1 simply means the pixel value of that grid is $i-1$. When predicting the future images for this type of data, a softmax function [11], which is usually used in multi-class classification task, will be applied before the final output. The predicted pixel value on a grid will be determined by the largest value in the predicted vector on that grid.

If we want to study our model using the above method, the dimension of the moving MNIST data in our experiment will be transformed into $(10000, 14, 64, 64, 256)$. This will result in a need of too many parameters which will cause a breakdown of a computer with a limited memory. Thus, we will keep the original data and predict the pixel value on every grid directly. The ReLU activation function will be used for all FC-LSTM and ConvLSTM layers and the final output layer to make sure the predicted pixel values are non-negative. And all predicted values above 255 will be recorded as 255 in the end. Though we have cut the dimension of the data in this way, it still took around 60 hours to train all models on the moving MNIST data in a single experiment by a single NVIDIA GeForce GTX 2070 GPU.

4.1 Moving MNIST data

The generating process of moving MNIST data is similar as described in [16]. Each sample in the dataset is 14 frames long and consists of two digits moving inside a 64×64 patch. These digits are randomly chosen from a set of 500 handwritten digits in the MNIST dataset [9]. Each digit is placed initially at a random location in the first frame. The velocity direction for every digit is uniformly randomly selected and the velocity speed is chosen randomly between 1 and 4 pixels between every two consecutive frames. We repeat this process 12000 times to get a moving MNIST dataset of 8000 training samples, 2000 validation samples and 2000 test samples. We will train all models by minimizing the pixel-wise mean squared error using BPTT via Adam algorithm. Early stopping is implemented on validation set. If the mean squared error cannot be improved in the next five epochs, we will stop training our models.

For the FC-LSTM based model, we will use the FC-LSTM encoder-forecaster model which is similar to the conditional future predictor model mentioned in [16]. This is a two-layer model with 2048 nodes in both hidden states and memory cells in each layer. In practice, the deeper networks always outperform the shallower networks with the same number of parameters on the same task. In our experiments, we want to test if the deeper networks can still improve from the shallower networks with less parameters. We will save time on training the deeper networks with less parameters for the future work if our experiments can prove this hypothesis is true. We will use three variants of the ConvLSTM encoder-forecaster network to make comparisons. The one-layer network has 128 filters for hidden states and memory cells in the ConvLSTM layer. The two-layer network has two ConvLSTM layers with 64 filters for hidden states and memory cells in both layers. The three-layer network has three ConvLSTM layers with 64, 32 and 32 filters for hidden states and memory cells from the first layer to the third layer. All the input-to-state and state-to-state filters have size 5×5 since the movement at every time step is within 4 pixels when we generate the data.

The average pixel-wise mean squared error and mean absolute error shown here for comparison on the training and validation sets are shown in Table 4.1 and an example of the prediction results from each trained model from the test set is illustrated in Figure 4.1. As we can see, this experiment shows that there is a huge improvement from FC-LSTM encoder-forecaster network to ConvLSTM encoder-forecaster network. Even a 1-layer ConvLSTM encoder-forecaster network with 3,311,617 parameters has improved a lot from the deeper 2-layer FC-LSTM encoder-forecaster network with 92,295,168 parameters. Also the deeper ConvLSTM encoder-forecaster networks with less parameters give better predictions though the improvement is not that much.

Finally we test our best trained model, the three-layer ConvLSTM encoder-forecaster network, on the out-of-domain case. We generate another 2000 sequences of the three moving

Model	# parameters	MSE$_{train}$	MAE$_{train}$	MSE$_{val}$	MAE$_{val}$	MSE$_{test}$	MAE$_{test}$
2-layer FC-LSTM $-2048-2048$	92,295,168	814.40	10.12	838.52	9.92	880.46	10.42
1-layer ConvLSTM $(5\times5)-5\times5-128$	3,311,617	694.40	7.82	734.76	8.14	771.51	8.56
2-layer ConvLSTM $(5\times5)-5\times5-64$ $-5\times5-64$	2,475,521	631.53	7.10	659.33	7.32	692.31	7.69
3-layer ConvLSTM $(5\times5)-5\times5-64$ $-5\times5-32$ $-5\times5-32$	1,858,049	582.79	6.62	625.55	6.90	656.84	7.26

Table 4.1: **Comparison results of ConvLSTM and FC-LSTM encoder-forecaster networks on 2-digit moving MNIST data**. '-2048' represents number of nodes in the hidden states and memory cells. '-5×5' represents the filter size for the input-to-state and state-to-state transitions. '-128', '-64' and '-32' represent number of filters for the hidden states and memory cells in the ConvLSTM layers. '(5×5)' means the filter size for the input-to-state transition. Each model is trained and supervised on training and validation set separately. The MSE and MAE on the test set are also provided to evaluate model's future performance.

digits. These handwritten digits are also drawn from MNIST dataset and they are clearly non-overlapping examples from our training set. Since our model is trained on the 2-digit moving MNIST data and has never learned from three moving digits sequences, this would be a good test of the generalization ability of our model. The average mean squared error and mean absolute error of our trained three-layer model on this out-of-domain dataset are 1300.38 and 13.30. From these prediction results, we see that to some extent our model can separate the three digits and predict each digit's movement respectively, although the prediction of moving digits are blurred. An example of the prediction of the out-of-domain case is illustrated in Figure 4.2.

4.2 Radar Echo Maps Data

The radar echo maps data used in this thesis is from the Short-term Quantitative Precipitation Forecasting Competition at CIKM AnalytiCup in 2017 [1] and it was originally provided by Shenzhen Meteorological Bureau and Alibaba.

The training set of radar echo maps data has 10,000 samples. Each sample is a sequence of 15 radar echo maps with a 6-minute time span between every two consecutive maps and a target site's rainfall amount in the short future. Each radar echo map covers a target site which locates at the center of the map and its surrounding areas. It is marked as an image of 101×101 grids where each grid point has a radar reflectivity value z. The organizer of this competition has used the dBZ to measure the value z: $dBZ = log(z/z_0)$ where $z_0 = 1mm^6/m^3$. Then a linear transformation is applied for anonymization purpose. In the end, we have each radar echo map in grey scale with pixel values from 0 to 255.

Unlike predicting the future rainfall amount by using all 15 frames in a sequence in the competition, we are training the ConvLSTM encoder-forecaster networks by using the first 10 frames of radar echo maps to predict the last 5 frames. If we take a close look at these 10,000 samples, we would find out that the radar echo maps do not change from the first frame to the last frame in some samples. Since we want to learn the sudden change of these radar echo maps, it would be quite time-efficient and necessary to get rid of these barely changing samples. We compute the correlation between the first and last frame of radar echo maps in each sample and only keep training samples with a smaller correlation. The correlation between frame P and frame Q is defined as:

$$\frac{\sum_{i,j} P_{i,j} Q_{i,j}}{\sqrt{(\sum_{i,j} P_{i,j}^2)(\sum_{i,j} Q_{i,j}^2)} + \epsilon},$$

which is a value between 0 and 1 and ϵ is set as 10^{-9}. Small correlation means two frames are different and larger correlation means we have two similar frames. As we also want to have enough samples to train our model, we keep 5485 samples with correlations between the first and last frames less than 0.75 (The correlation can also be used to evaluate the prediction with respect to the corresponding ground truth as well.). We compare the FC-LSTM encoder-forecaster network with variants of the ConvLSTM encoder-forecaster network on the pre-processed radar echo maps data.

Model	# parameters	MSE$_{train}$	MAE$_{train}$	MSE$_{val}$	MAE$_{val}$	MSE$_{test}$	MAE$_{test}$
2-layer FC-LSTM $-2048-2048$	154,816,473	289.87	8.08	304.09	8.27	319.30	8.69
1-layer ConvLSTM $(3\times 3)-3\times 3-128$	1,191,937	242.78	7.61	265.49	7.74	278.77	8.13
2-layer ConvLSTM $(3\times 3)-3\times 3-64$ $-3\times 3-64$	894,465	220.12	7.21	258.92	7.54	271.87	7.92
3-layer ConvLSTM $(3\times 3)-3\times 3-64$ $-3\times 3-32$ $-3\times 3-32$	670,209	213.72	7.07	251.37	7.32	263.94	7.69

Table 4.2: **Comparison results of ConvLSTM and FC-LSTM encoder-forecaster networks on Radar Echo Maps data**.'-2048' represents number of nodes in the hidden states and memory cells. '-3×3' represents the filter size for the input-to-state and state-to-state transitions. '-128', '-64' and '-32' represent number of filters for the hidden states and memory cells in the ConvLSTM layers. '(3×3)' means the filter size for the input-to-state transition. Each model is trained and supervised on training and validation set separately. The MSE and MAE on the test set are also provided to evaluate model's future performance

After trying filters of different sizes in the input-to-state and state-to-state convolution transition, we found the 3×3 convolutional filters were the best to capture the spatial information. The average pixel-wise mean squared error and mean absolute error on training and validation sets are shown in Table 4.2 and the prediction results from each model on a radar echo map growing example and a decaying example are illustrated in Figure 4.3 and Figure 4.4 separately. As can be seen, the FC-LSTM encoder-forecaster network does not perform well for this more complicated precipitation nowcasting task, mainly caused by the strong spatial correlation in the radar echo maps. The ConvLSTM encoder-forecaster network outperforms FC-LSTM encoder-forecaster network to have smaller MSE and MAE. We also compare the prediction results from each model by using MSE, MAE and correlation at every time step in prediction on the validation set as shown in Figure 4.5. Though the prediction accuracy is decreased as prediction time step goes on, the more layer stacked ConvLSTM network still performs better at every single time step. The ConvLSTM network is able to capture the boundary information since we add convolutional operation and zero-padding techniques in building the model. In real-life cases, there are many sudden changes on the boundary of the radar echo maps, like clouds will grow or decay on the boundary. If our ConvLSTM network has seen these cases during training process, it is able to predict correctly in the future, while the traditional radar echo map based method cannot. However, the ConvLSTM network has a bad blurring effect which makes prediction on the edges of the moving objects unreliable.

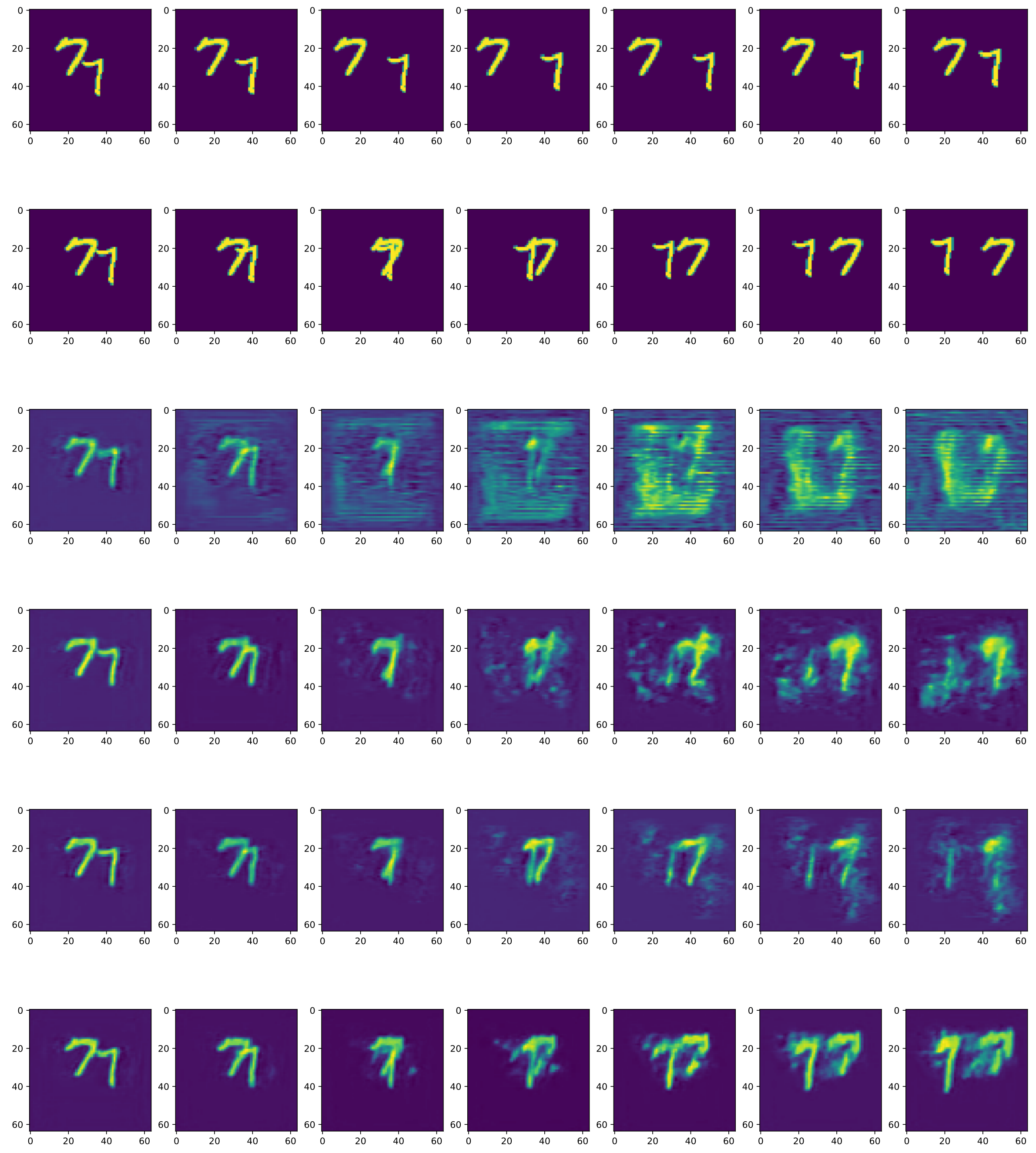


Figure 4.1: **An example of prediction results on the 2-digit moving MNIST data test set**. From top to bottom: input frames 1 to 7; ground truth frames 8 to 14; prediction results from the FC-LSTM with 2048 nodes in hidden states and memory cells, 1-layer ConvLSTM encoder-forecaster network with 5×5 filter, 2-layer ConvLSTM encoder-forecaster network with 5×5 filter and 3-layer ConvLSTM encoder-forecaster network with 5×5 filter for the next 4 rows, respectively.

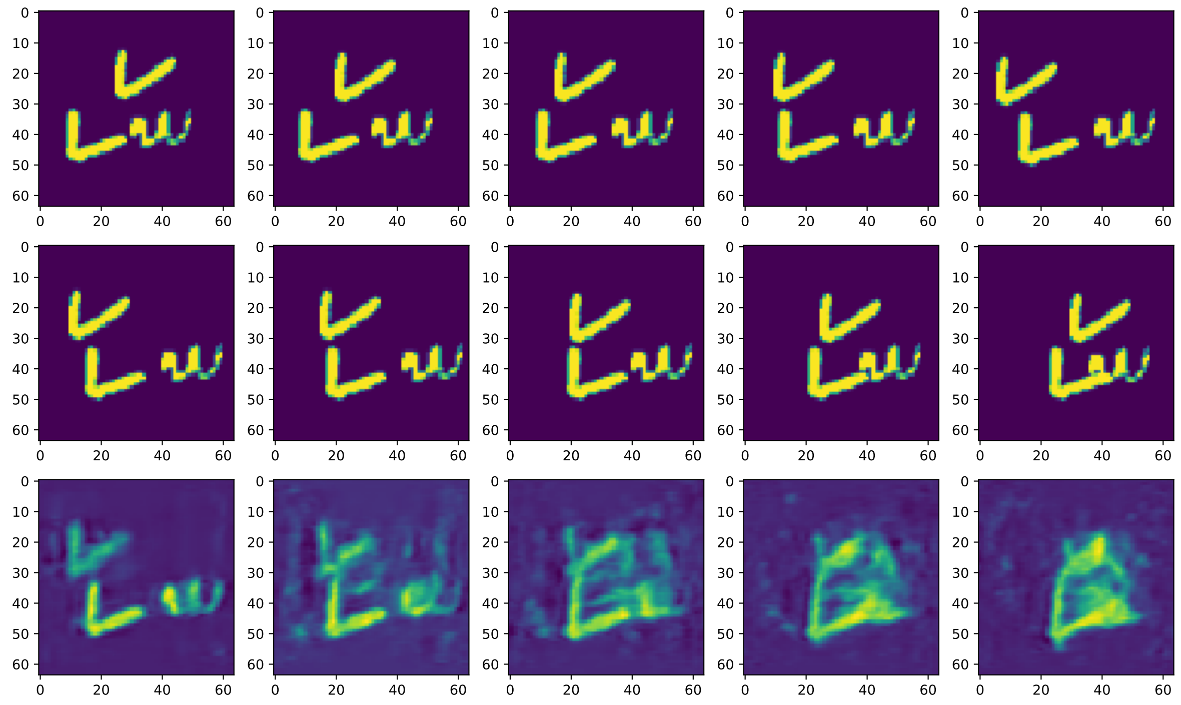


Figure 4.2: **An example of the 3-layer ConvLSTM encoder-forecaster network with 5×5 filter, which is well trained on the 2-digit moving MNIST data, predicts on the 3-digit moving MNIST sequence.** First row: Input frames 1 to 5. Second row: ground truth frames 6 to 10. Third row: prediction results.

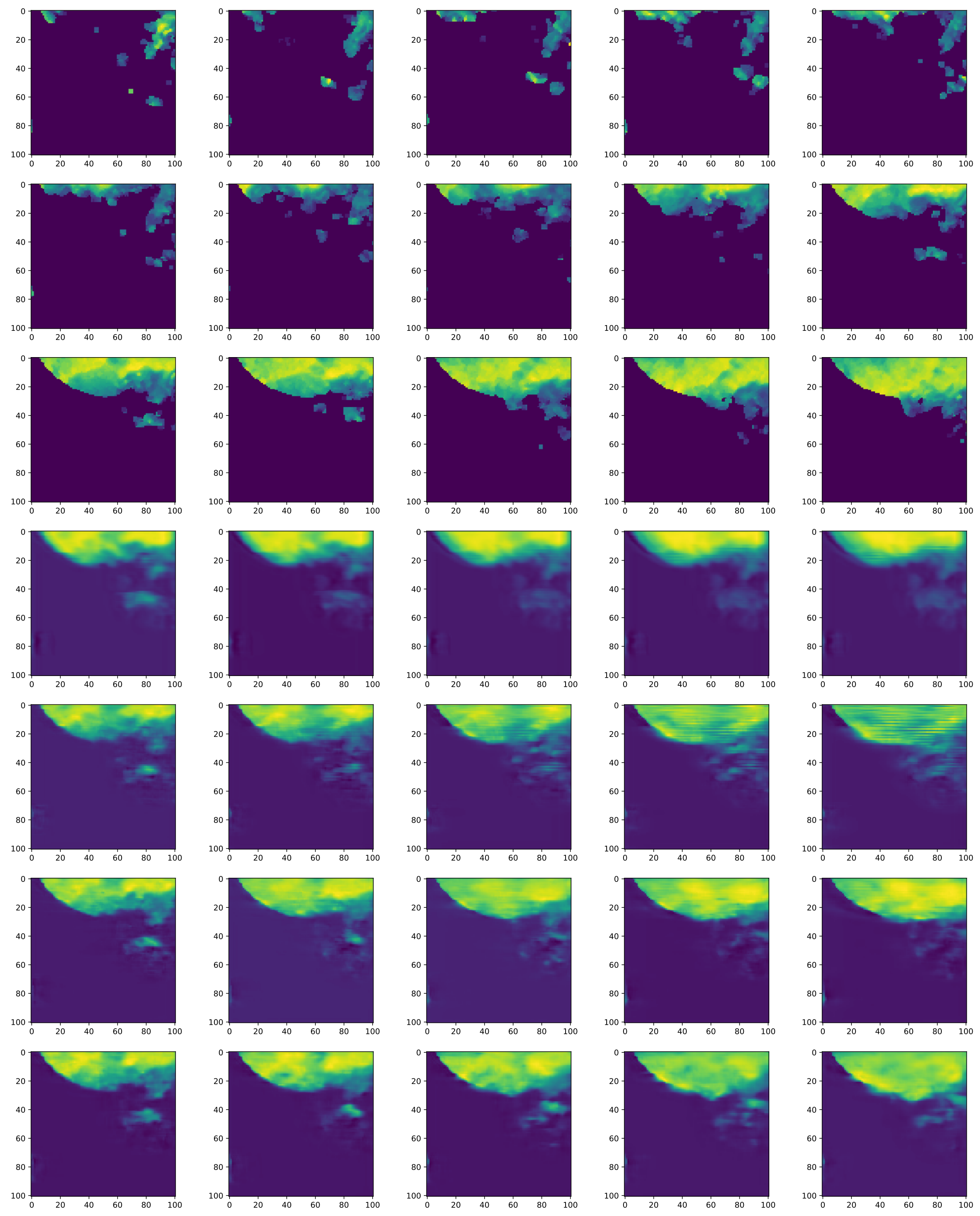


Figure 4.3: **An example of prediction results on the growing radar echo maps**. From top to bottom: input frames 1 to 5; input frames 6 to 10; ground truth frames 11 to 15; Prediction results from the FC-LSTM encoder-forecaster with 2048 nodes in hidden states and memory cells, 1-layer ConvLSTM encoder-forecaster network with 3×3 filter, 2-layer ConvLSTM encoder-forecaster network with 3×3 filter and 3-layer ConvLSTM encoder-forecaster network with 3×3 filter for the next 4 rows, respectively.

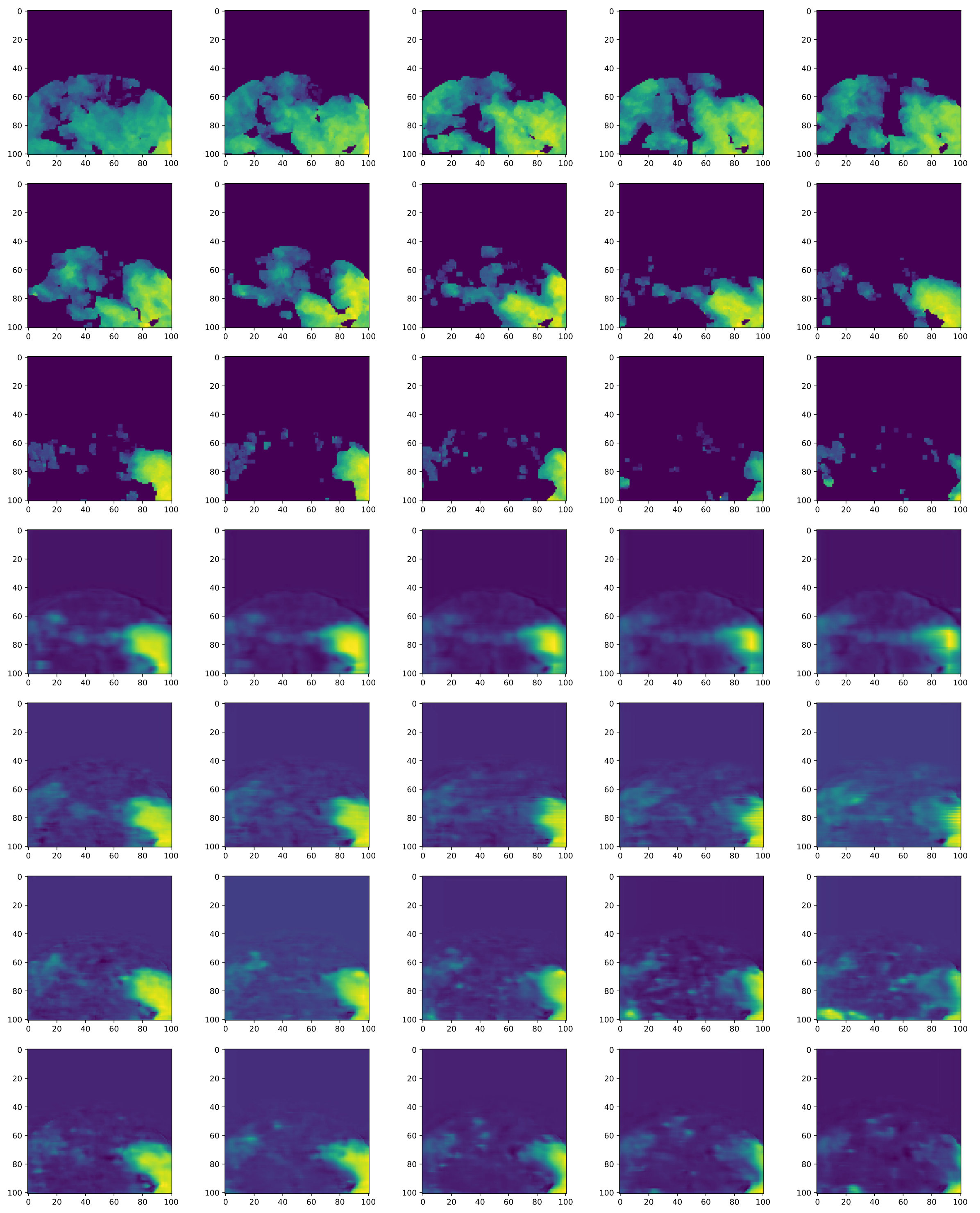


Figure 4.4: **An example of prediction results on the decaying radar echo maps**. From top to bottom: input frames 1 to 5; input frames 6 to 10; ground truth frames 11 to 15; Prediction results from FC-LSTM encoder-forecaster with 2048 nodes in hidden states and memory cells, 1-layer ConvLSTM encoder-forecaster network, 2-layer ConvLSTM encoder-forecaster network and 3-layer ConvLSTM encoder-forecaster network encoder-forecaster network with 3×3 convolution filter for the next 4 rows, respectively.

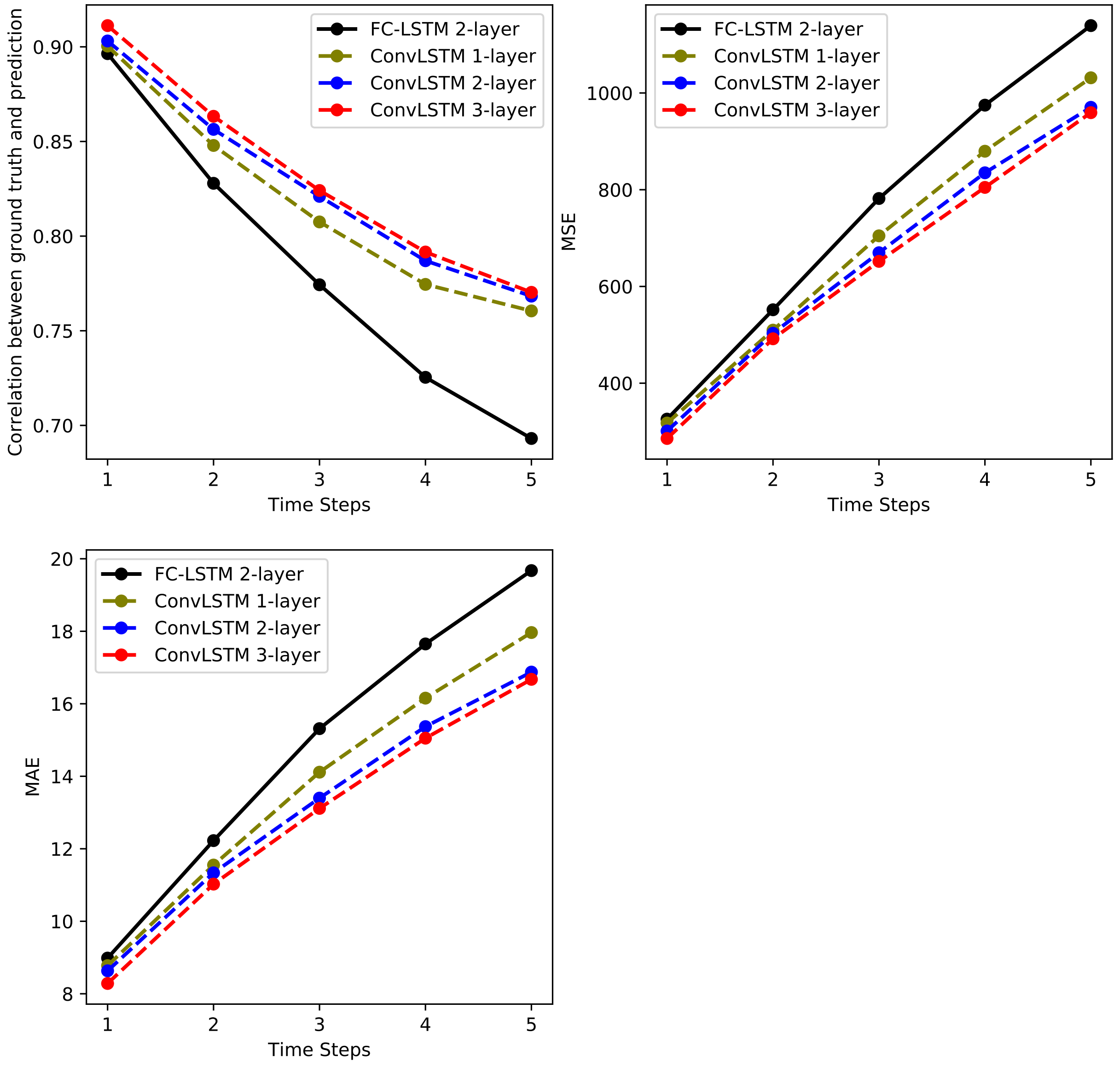


Figure 4.5: **Comparison of model performance at first 5 time steps separately in prediction by MSE, MAE and correlation on the validation set of Radar Echo Maps data.**

Chapter 5

Conclusion and Future Work

In this thesis, we have successfully applied the neural network model for the precipitation nowcasting task from a machine learning perspective. We first formulated the precipitation nowcasting task as a spatial-temporal sequence prediction problem. Then we followed the work done by Shi et al. [15] to build the ConvLSTM by adding convolutional operation to the FC-LSTM. The ConvLSTM does not just preserve the strong ability in capturing the temporal relationship in a sequence but is also well-behaved in dealing with spatial-temporal data because of the innovative convolution operation in the input-to-state and state-to-state transitions. By incorporating the enocder-forecaster structure into ConvLSTM, we built the end-to-end ConvLSTM encoder-forecaster models which were trainable for the precipitation nowcasting task. The ConvLSTM encoder-forecaster network does have the potential to be applied into more different sequential image prediction tasks.

However, the blurring effect is still a problem in our model. The predicted images become more and more blurred as time step increases. One may apply more sophisticated layers at the output layer or incorporate the ConvLSTM with other structures like mechanism attention to get more accurate predictions. Another potential way which to reduce the blurring effect is to define and compute our loss function step by step within the forecaster network. Instead of computing the pixel-wise MSE all at once at the end of the prediction, we may penalize our model by comparing the prediction images and ground truth step by step in a backward order and accumulate the loss while doing the BPTT. In this way, we may focus on the prediction images separately and hopefully this may reduce the blurring effect on the forecaster network when a long sequence of future images is wanted.

Bibliography

[1] CIKM AnalytiCup 2017 shenzhen meteorological bureau-alibaba challenge: Short-term quantitative precipitation forecasting. `https://tianchi.aliyun.com/competition/entrance/231596/information`. Accessed: 2019-02-01.

[2] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL `http://arxiv.org/abs/1406.1078`.

[3] Edward Choi, Andy Schuetz, Walter F Stewart, and Jimeng Sun. Using recurrent neural network models for early detection of heart failure onset. *Journal of the American Medical Informatics Association*, 24(2):361–370, August 2016. ISSN 1527-974X. doi: 10.1093/jamia/ocw112. URL `https://doi.org/10.1093/jamia/ocw112`.

[4] R. J. Frank, N. Davey, and S. P. Hunt. Time series prediction and neural networks. *Journal of Intelligent and Robotic Systems*, 31(1-3):91–103, May 2001. ISSN 0921-0296. doi: 10.1023/A:1012074215150. URL `https://doi.org/10.1023/A:1012074215150`.

[5] Urs Germann and Isztar Zawadzki. Scale-dependence of the predictability of precipitation from continental radar images. part i: Description of the methodology. *Monthly Weather Review*, 130(12):2859–2873, 2002. doi: 10.1175/1520-0493(2002)130⟨2859:SDOTPO⟩2.0.CO;2. URL `https://doi.org/10.1175/1520-0493(2002)130<2859:SDOTPO>2.0.CO;2`.

[6] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL `http://arxiv.org/abs/1308.0850`.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `http://dx.doi.org/10.1162/neco.1997.9.8.1735`.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL `http://arxiv.org/abs/1412.6980`. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[9] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

[10] Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. ISSN 0028-0836. doi: 10.1038/nature14539.

[11] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018. URL `http://arxiv.org/abs/1811.03378`.

[12] Marc'Aurelio Ranzato, Arthur Szlam, Joan Bruna, Michaël Mathieu, Ronan Collobert, and Sumit Chopra. Video (language) modeling: a baseline for generative models of natural videos. *CoRR*, abs/1412.6604, 2014. URL `http://arxiv.org/abs/1412.6604`.

[13] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *CoRR*, abs/1402.1128, 2014. URL `http://arxiv.org/abs/1402.1128`.

[14] H. Sakaino. Spatio-temporal image pattern prediction method based on a physical model

with time-varying optical flow. *IEEE Transactions on Geoscience and Remote Sensing*, 51(5):3023–3036, May 2013. ISSN 0196-2892. doi: 10.1109/TGRS.2012.2212201.

[15] Xingjian Shi, Zhourong Chen, Hao Wang, DitYan Yeung, WaiKin Wong, and Wangchun Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *CoRR*, abs/1506.04214, 2015. URL `http://arxiv.org/abs/1506.04214`.

[16] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using lstms. *CoRR*, abs/1502.04681, 2015. URL `http://arxiv.org/abs/1502.04681`.

[17] Juanzhen Sun, Ming Xue, James W. Wilson, Isztar Zawadzki, Sue P. Ballard, Jeanette Onvlee-Hooimeyer, Paul Joe, Dale M. Barker, Ping-Wah Li, Brian Golding, Mei Xu, and James Pinto. Use of nwp for nowcasting convective precipitation: Recent progress and challenges. *Bulletin of the American Meteorological Society*, 95(3):409–426, 2014. doi: 10.1175/BAMS-D-11-00263.1. URL `https://doi.org/10.1175/BAMS-D-11-00263.1`.

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL `http://arxiv.org/abs/1409.3215`.

[19] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990. ISSN 0018-9219. doi: 10.1109/5.58337.

[20] Wang-chun Woo and Wai-kin Wong. Operational application of optical flow techniques to radar-based rainfall nowcasting. *Atmosphere*, 8(3), 2017. ISSN 2073-4433. doi: 10.3390/atmos8030048. URL `https://www.mdpi.com/2073-4433/8/3/48`.