Non-Convex Optimization: RMSProp Based Optimization for Long Short-Term Memory Network

Except where reference is made to the work of others, the work described in this project is my own or was done in collaboration with my advisory committee. Further, the content of this project is truthful in regards to my own work and the portrayal of others' work. This project does not include proprietary or classified information.

Jianzhi Yan

Certificate of Approval:

Bruno Andriamanalimanana
Associate Professor
Department of Computer Science and Software Engineering

Chen-Fu Chiang
Assistant Professor
Department of Computer Science and Software Engineering

Jorge Novillo
Professor
Department of Computer Science and Software Engineering

Non-Convex Optimization: RMSProp Based Optimization for Long Short-Term Memory Network

Jianzhi Yan

A Project

Submitted to the
Graduate Faculty of the
State University of New York Polytechnic Institute
in Partial Fulfillment of the
Requirements for the
Degree of

Master of Science

Utica, New York
May 9, 2020

Non-Convex Optimization: RMSProp Based Optimization for Long Short-Term Memory Network

Jianzhi Yan

_____
Signature of Author

_____
Date of Graduation

Project Abstract

Non-Convex Optimization: RMSProp Based Optimization for Long Short-Term Memory Network

Jianzhi Yan

Master of Science, May 9, 2020
(B.A., Century College,Beijing University of Posts and Telecommunications, 2014)

38 Typed Pages

Directed by Bruno Andriamanalimanana

This project would give a comprehensive picture of non-convex optimization for deep learning, explain in details about Long Short-Term Memory (LSTM) and RMSProp. We start by illustrating the internal mechanisms of LSTM, like the network structure and backpropagation through time (BPTT). Then introducing RMSProp optimization, some relevant mathematical theorems and proofs in those sections, which give a clear picture of how RMSProp algorithm is helpful to escape the saddle point. After all the above, we apply it with LSTM with RMSProp for the experiment; the result would present the efficiency and accuracy, especially how our method beat traditional strategy in non-convex optimization.

Style manual or journal used Journal of Approximation Theory (together with the style known as "sunypolyms"). Bibliography follows van Leunen's *A Handbook for Scholars.*

Computer software used The document preparation package TEX (specifically LATEX2e) together with the style-file `sunypolyms.sty`.

Table of Contents

List of Tables

Chapter 1

Introduction

In the present era, machine learning is one of the most significant technologies which impacts and changes the rules of many fields, like biology, finance, graphic processing and military. However, many machine learning techniques, such as deep learning, create optimization problems with non-convex objective functions. Global optimization of non-convex objectives, finding the global optimum objective function, is a major NP problem in general.

Instead of creating more innovative networks, more researchers prefer to give solutions for the non-convex optimization questions. As we know the first-order methods, like stochastic gradient descent(SGD), is still the most chooses for networks training. However, they failed to exploit second-order structure since storing the large numbers of parameters in the full Hessian is not storage efficient, and the training speed would slow down. Thus, the researches proposed several SGD methods with adaptive learning rate; those algorithm can escape saddle point on the optimization path. Meanwhile, many open source machine learning platforms, like Tensorflow, also offer powerful optimizers, make it convenient for developers to build a designed network.

In this report, we would give a comprehensive picture of non-convex optimization for deep learning. Including the features and advantages of a modified recurrent neural network - Long short-term memory (LSTM), the theory and application of RMSProp and how it works on the LSTM. Further, our experiments show that RMSProp better than plain SGD in terms of escaping saddle point.

Chapter 2

Related Works

2.1 Artificial Neural Network

Deep Learning is an important subfield of machine learning, artificial neural networks with representation learning are the most methods employed. Convolutional neural network (CNN) and recurrent neural network (RNN) are 2 typical artificial neural network architectures, CNNs are convolution operation based neural network, they are consisted of an input layer, an output layer and several hidden layers, CNNs are most commonly applied for image processing. Though CNNs have outstanding performance on the visual field, they cannot handle sequential data, like speech recognition, language modelling and financial analysis, that is why we need to introduce RNNs for our project.

Recurrent neural network (RNN) is a neural sequence model with short-term memory. Derive from the traditional feed-forward neural network; the recurrent neural network allows previous outputs to be the inputs of cells while having hidden states, which make them applicable for the data analysis with sequential data[1].

2.2 Non-Convex Optimization Methods

Non-Convex optimizations are not something innovative. Indirect techniques like convex relaxations, and direct techniques like EM, gAM, gPGD, they are widely applid in statistics, machine learning and other data science relevant fields.

Expectation Maximization (EM) is commonly used learning rhythms like data, such as those used to learn Gaussian mixture models, the Baum-Welch learning algorithm and mixed regression. The EM algorithm also has shared properties with the Lloyd's algorithm for objectively clustering the k-means.

The alternative minimization algorithm (gAM) is illustrated on two variables, respectively limited to the sets X and Y. Alternative minimisation is often used when these residual concerns can be solved with ease.

However, above methods cannot be applied directly in the deep learning, so researchers need to find out other technologies for artificial neural network optimization.

Chapter 3

Preliminaries

3.1 Smooth and Convexity

3.1.1 L-Lipschitz

A function $f(x)$ is L-Lipschitz if

$$|f(x_1) + f(x_2)| \leq L \parallel x_1 - x_2 \parallel$$

For function with the property, the first order derivative f'(x) is not over L, how steep function can get.

3.1.2 Convex Set

Set X is convex set if any two points x, y in X, all points on the line connected them lie inside the set X. Whereas, it is a non-convex set.

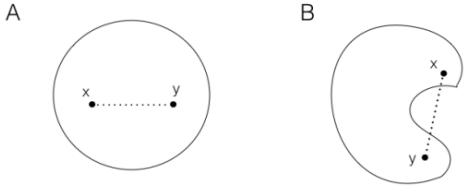


Figure 1. Convex and Non-Convex Sets

3.1.3 Convex function

A convex function has following properties:

1. The domain $x \in X$ is a convex set

2. For all elements of domain of function $f(x)$, there is

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)$$

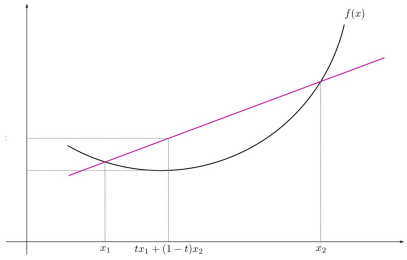


Figure 2. Convex function

In another word, if the chord which connecting any two points of the function always lies above the function, the function is denoted as convex function.

3.1.4 Smoothness

A function is $\beta - smooth$ if:

$$\| \bigtriangledown f(x_1) - \bigtriangledown f(x_2) \| \leq \beta \| x - y \|$$

A $\beta - smooth$ function is $\beta - Lipschitz$ of $\bigtriangledown f(x)$ as well. For the multivariate second order derivative $\bigtriangledown^2 f(x) = \mathrm{H}$, where H is denoted as Hessian matrix, all eignvalues are less than β. It implies the function cannot change too quickly and the curvature is bounded by defined variable.

3.1.5 Strong Convexity

A function f(x) is α strongly convex if $f(x) - \frac{\alpha}{2} \| x \|^2$ is convex.

3.2 Gradient Descent

Gradient descent algorithm is inspired from the fact that current point a function f(x) decreases most rapidly in the direction of the negative gradient f(x), it starts from a random or defined point x_0, moves to the direction of steepest descent on each iteration.

The sequence of x_n is generated by the rule:

$$x_{n+1} = x_n - \gamma \bigtriangledown f(x_n)$$

where γ is denoted as learning rate or step size.

3.2.1 Main Theory

For convex function $f(x)$, if it is L-lipschitz, the steps number T and learning rate γ:

$$\gamma = L\frac{\parallel x_1 - x_2 \parallel}{L\sqrt{T}}$$

3.2.2 Strict Saddle properties

If any point of function $f(w)$ has one of following properties at least, it is $(\alpha, \gamma, \xi, \delta)$-strict saddle

1. $\parallel \bigtriangledown f(x) \parallel \geq \epsilon$
2. $\lambda_{min}(\bigtriangledown^2 f(w)) \leq -\gamma$
3. For some local minimum x_*,$\parallel x_0 - x_* \parallel_2 \leq \xi$

   For any point with those properties, if the gradient point is small and has the negative eigenvalue of Hessian matrix, it is saddle point.

3.2.3 Backpropagation Through Time

BPTT starts in time with the formation of a recurrent neural network. The unfolded network contains inputs and outputs f, which shares the same parameters for each network copy The cost gradient for all networks is determined with the backpropagation algorithm parameters.

Find an example of a neural network with a recurrent layer and a feed-forward layer. There are various ways of calculating the loss of training, the lost of each time step can be

computed separately, and the overall loss is usually the average contribution of lost time step. For example, we need to compute the lost at time t, by unfolding the recurrent layer for t-time steps and adding the feed-forward layer. Each instance of the time step in the unfolded network shares the same parameters, and the weight updates in each instance are a summary of all steps. The following sections would display the whole progress.

Chapter 4

Method and Application

4.1 RMSProp Optimization

4.1.1 Introduction

RMSProp is the expansion of traditional adaptive learning rate methods; the traditional gradient descent is trapped in saddle point while it can escape. The basic idea is not complicated; for a non-convex function with strict-saddle property, there always exists steep descent direction until it approaches the local minimum. So we add some random perturbations for gradient, then it may go along the steep direction and escape the saddle point.

4.1.2 Algorithm

It disturbs the gradient at each step by using a unit vector pointing in a random direction; if we are currently stuck at a saddle point, the disruptions drive us to discover the steep descent path, allowing us to continue making progress.

Here is the Pseudocode of RMSProp Algorithm

RMSProp Algorithm

Algorithm 2 RMSProp

function RMSPROP $(\mathbf{x}_1, \beta_2, \alpha)$
  Initialize : $\mathbf{v}_0 = \mathbf{0}$
  for $t = 1, 2, \ldots$ **do**
    $\mathbf{g}_t = \nabla f(\mathbf{x}_t)$
    $\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\, \mathbf{g}_t^2$
    $V_t = \mathtt{diag}(\mathbf{v}_t)$
    $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha V_t^{-\frac{1}{2}} \mathbf{g}_t$
  end for
end function

4.2 Data Resource

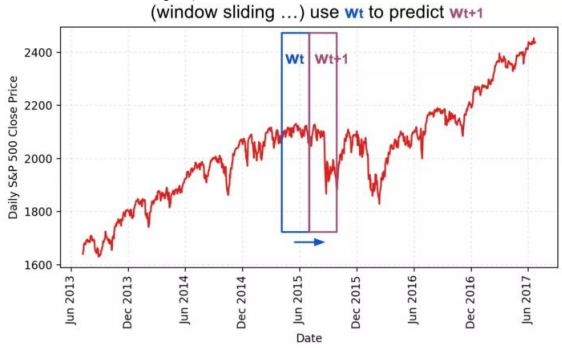
This Section would present the format of the data resource and data pre progressing – the Yahoo stock information. The advantage of the recurrent neural network over the traditional neural network is handling the sequential data, and the stock price index fits the pattern. For this project, the history data of Dow Jones Industrial Average in the last 5 years is the data resource for the experiment.

Date	Open	High	Low	Close	Adj Close	Volume
4/1/2015	17727.480469	27542.779297	Static	Static	Static	Static
4/2/2015	17778.519531	22327.570313	Static	Static	Static	Static
4/3/2015	18052.320313	17965.369141	Static	Static	Static	Static
4/4/2015	25457.210938	24992.359375	Static	Static	Static	Static
4/5/2015	21050.339844	26706.169922	Static	Static	Static	Static

Table 4.1: the formal of stock prices dataset

4.2.1 Data Preprocessing

Mini-batch gradient descent is an alternative gradient descent that splits the dataset into small batches. For this project,we split to T batches, every batch X_t have the daily average price informations for N days, where $0 \leq t \leq T$ and N is fixed.



We can take them as N width windows, which sort in the order of time, and there is no overlap between two consecutive windows. Since we use the unrolled RNN to avoid the difficulty of backpropagation, there are N input elements for epoch, and the label is the next N elements after them. For example, the format of input:

$$X_0 = (p_0, p_1, p_2, ..., p_(n-1))$$

...

$$X_{t+1} = (p_{t+1}n, p_{(t+1)(n+1)}, ..., p_{(t+2)n-1})$$

And if set hyper-parameter N = 3 and Step = 3, the training set input sample:

$$INPUT_1 = [X_1, X_2, X_3] LABEL_1 = [X_4]$$

4.2.2 Normalization

The data preprocessing is not done yet, due to the nature of economic laws, the stock price can exceed the highest price in history; it is unpredictable if we learn the history record. Thus instead of the stock price, it is more reasonable to learn the variation of price.

4.3 Optimization for Recurrent Neural Network

For this section, we introduce Long Short Term Memory Network(LSTM) and RMSProp optimization of LSTM. Including LSTM BPTT, LSTM modelling, non-convex of LSTM, and convergence guarantee of RMSProp for gradient descent or the variation of price.

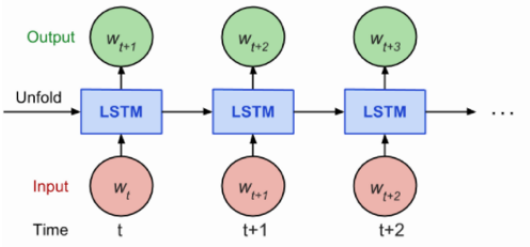


Figure 4.1 Structure of LSTM

4.3.1 Introduction of LSTM

Recurrent neural networks connect previous information to the present task, but it is not reliable when there is a large gap between relevant terms and current one[3]. Since

the traditional RNNs are not robust against the problems of long-term dependency, an alternative RNN model, Long Short Term Memory network(LSTM), is employed for this project. The core idea behind LSTM is cell state; it goes through the entire network chain and regulated by a structure named gate[7]. There are 3 kinds of gate contain sigmoid activations, forget gate filter the relative non-significant information, input gate keep the important information and update the cell state, and output gate decides next hidden state [6].

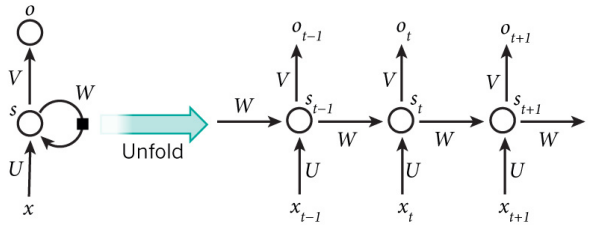


Figure 4.2 Unfolded LSTM

The fig above shows the unrolled version of LSTM since the unrolled network is a DAG, we can train it using back-propagation (BP) just like a standard neural network. We can see how the data flow as time goes by, and there are following several relevant variables:

- x_t is the input on tth time step
- s_t is the hidden status tth time step, it is the memory of LSTM, $s_t = f(Ux_t + W_{t-1})$ and the f is denoted as the activation function which is a non-linear, like tanh or ReLU. We have to choose an appropriate activation function to avoid the vanishing gradient problem, more specifically works on it presented in the following sections.
- Derive from the traditional neural networks, every layer of RNN has shared parameter (U, V, W), while their inputs are different from each other.
- o_t is the output on tth time step, it is a vector of probabilities. Just like all kinds of neural network, the outputs are posterior probabilities. Converting them into one-hot vector which the highest probability would be to one, and rest of them are zero.

4.3.2 LSTM BPTT

For this section, we explain how LSTM BPTT works, especially how it avoids the long-term dependency problem, which makes the traditional RNN unreliable. To clarify this problem, we need to take an insight look of the structure of LSTM Cell first.[2] [Advances in Optimizing Recurrent Networks]

- Forget Gate: The gate is responsible for filtering information from the cell state. The information that is no longer significant for the LSTM to learn data or the information that is not significant will be removed via a filter. Inputting the pairs of previous hidden state and current input X_t are into a sigmoid function. As we know the domain of sigmoid function is [0,1], if the return value is closer to 0 would be forgotten, while the one closer to 1 will be uploaded to cell state C_t. Here is the formula of forget gate:

$$\mathrm{f}_t = \sigma(W_f[h_{t-1}; x] + b_f)$$

- Input Gate: The gate update the cell state. The data processing isa three-step process. First, like forget gate, we input the pairs of hidden state h_{t-1} and current input X_t into a sigmoid function, keep the return value i_t and forget it if it is closer to 0. Then we also input the pairs into the tanh function and the output values C_t from -1 to 1. Finally, multiplying the i_t to the C_t and adding this useful information to the cell state. Here are formulas of input gate:

$$\widetilde{C_t} = tanh(W_c[h_{t-1}; x] + b_c)$$
$$\mathrm{i}_t = \sigma(W_i[h_{t-1}; x] + b_i)$$

- Output Gate: Selecting the critical information and showing it out, it is the hidden state for the next LSTM cell.

$$\mathrm{o}_t = \sigma(W_o[h_{t-1}; x] + b_o)$$

- Cell State:

$$\mathrm{C}_t = i_t \circ \widetilde{C_t} \oplus f_t \circ C_{t-1}$$

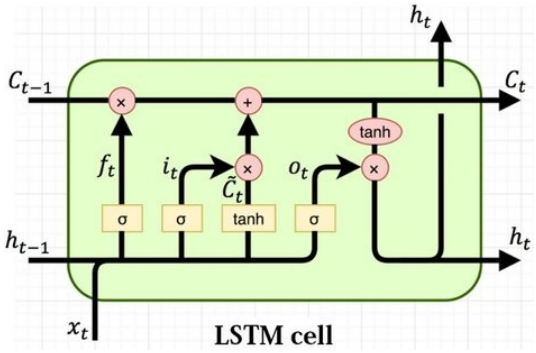


Figure 4.3 LSTM Cell

As we mention in preliminaries, the Backpropagation Through Time is the gradient descent algorithm used to update weights in RNNs. However it is not robust enough to discover contingencies spanning long temporal intervals[2] . Since this project focus on network optimization, so we just briefly talk about how does long-term dependency result in the vanishing gradient problem, instead of showing all details of the theoretical proof.

We use the chain rule of differentiation to calculate these gradients. The backwards start from the error. The goal of gradient descent is finding the global minimum of the loss function. In other words, minimizing the error of training. For the rest of this post we use L which is the summary of error at each time step as an example.

Here is the BP for traditional RNN:

$$\frac{\partial L}{\partial W_x} == \sum_{t=1}^{T}\sum_{k=1}^{t}\frac{\partial L_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\left(\prod_{j=k+1}^{t}\frac{\partial h_j}{\partial h_{j-1}}\right)\frac{\partial h_k}{\partial W_x}$$

We can see $\frac{\partial h_j}{\partial h_{j-1}} = \sigma' W_h$, it would result in the vanishing gradient as the index j increment [4].

How does LSTM solve this problem? The point is cell state, here is the expanded formula of cell state C_t:

$$C_t = \sigma(W_i[h_{t-1}; x] + b_i) \circ \tanh(W_c[h_{t-1}; x] + b_c) \oplus \sigma(W_f[h_{t-1}; x] + b_f) \circ C_{t-1}$$

It is clear that C_t depends on C_{t-1}, W_c, W_i and W_f. We need to get calculate the gradients of loss L with respect to parameters W_c, W_i and W_f, then update them until the error under the threshold, which is the basic idea behind gradient descent.

Define hidden state $h_t = o_t \circ tanh(C_t)$ and output function $\hat{y}_t = \sigma(W_o h_t)$

$$\frac{\partial L}{\partial W_c} = \sum_{1<t<T} \frac{\partial L_t}{\partial W_c} = \sum_{1<t<T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial C_t} \frac{\partial C_t}{\partial W_c}$$

$$\frac{\partial L}{\partial W_i} = \sum_{1<t<T} \frac{\partial L_t}{\partial W_i} = \sum_{1<t<T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial C_t} \frac{\partial C_t}{\partial W_i}$$

$$\frac{\partial L}{\partial W_f} = \sum_{1<t<T} \frac{\partial L_t}{\partial W_f} = \sum_{1<t<T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial C_t} \frac{\partial C_t}{\partial W_f}$$

Taking W_c as example:

$$\frac{\partial L}{\partial W_c} = \sum_{1<t<T} \frac{\partial L_t}{\partial W_c} = \sum_{1<t<T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial C_j} \sum_{1<k<t} \Big(\prod_{k+1<j<t} \frac{\partial C_j}{\partial C_{j-1}} \Big) \frac{\partial C_1}{\partial W_c}$$

$$\frac{\partial L}{\partial W_c} = \sum_{1<t<T} \frac{\partial L_t}{\partial W_c} = \sum_{1<t<T} \sum_{1<k<t} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial C_j} \Big(\prod_{k+1<j<t} \frac{\partial C_j}{\partial C_{j-1}} \Big) \frac{\partial C_1}{\partial W_c}$$

In this formula, we take derivative of a vector function with respect to a vector, the result is a Jacobian matrix whose elements are first-order partial derivatives.

$$\frac{\partial C_j}{\partial C_{j-1}} = \sigma(W_f[h_{t-1}; x] + b_f)$$

As we know, the range of sigmoid function is $[0, 1]$, so the derivative would never vanish as the index j increment. For the following secitons, we will discuss the optimization of BPTT,

4.3.3 Modeling LSTM

To build the LSTM model with Tensorflow, we start with building up a computational graph, there are several LSTM cells for each LSTM layer, and the inputs and outputs of the graph are tensors[4]. The generation of outputs is LSTM produces a vector of probabilities of prediction for the next symbol, and the index of the element with the highest probability is the predicted index of the symbol of the one-hot label vector.

Moreover, we need three input parameters, they are all $tf.placeholder$ since we have no idea about their value at the graph construction stage.

- Inputs: the training data X, a tensor of shape (data examples, numsteps, inputsize); the number of data examples is unknown. For and it would be batch size in training session.
- Targets: The training label y, a tensor of shape (data examples, inputsize).
- Learning rate: a simple float. Which would update in the gradient descent.
- Weights Biases: The training model we want to get in the training, then we input the test data to test. They initialize as random values, which would update for each batch.

We also need to define hyper-parameters like max epoch; it would go through the full pass of all the training data set for each epoch. In one epoch, the training data points are split into mini-batches of size batch size, then input one mini-batch to the model for one BPTT learning. The initial learning rate is β_2 for the first epochs and then decay by learning rate decay during every succeeding epoch.

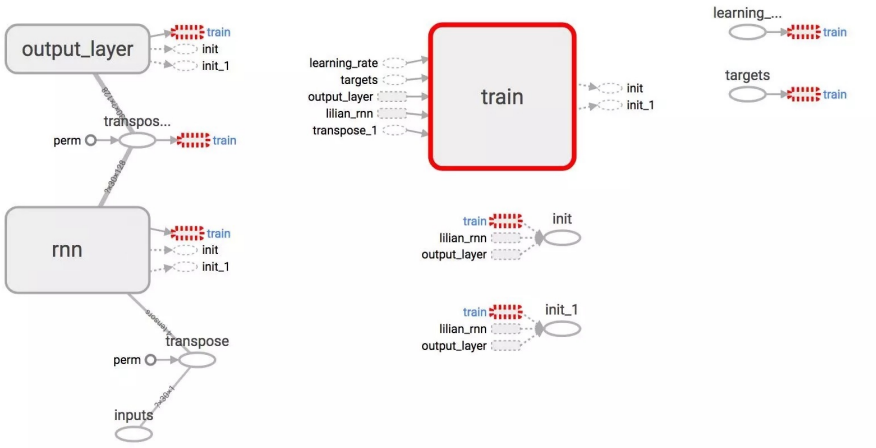


Figure 4.4 TensorBoard showing network frame

In the end, we also need to consider the overfitting problem, the production of an analysis that corresponds too closely or exactly to a particular set of data; it may result in future prediction unreliable. Thus, we set up a dropout mask with keep drop probability for output of every LSTM cell. It would be helpful to alleviate potential strong dependency.

4.3.4 Non-Convex and Smoothness Analysis

- **Non-convex properties**

The optimizations of the neural network are challenging since they are non-convex and NP-hard, LSTM and other RNNs fit this pattern as well. However, people barely talk about the non-convex analysis, why the neural networks are non-convex? For this section, we give a convincible answer.

We introduced some properties of convex function in preliminaries, so let's make an expansion what the properties non-convex function are with? Tactically, a function can not have all properties of convex is denoted as non-convex function, it can have following geometrical properties, many local minimum, saddle points, flat regions with no gradient and widely varying curvature[5].

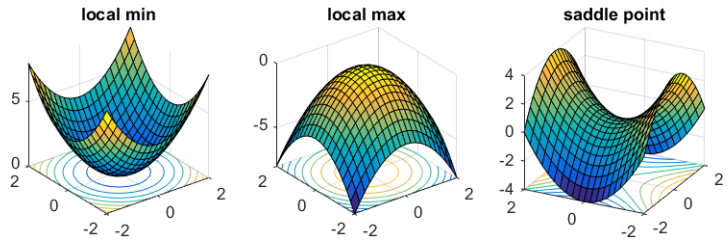


Figure 4.5 Types of Stationary Point: Local min; Local max; Saddle point

- **Tensorized LSTM**

As we mentioned in the previous section, LSTM cell recursively calulate the hidden state $h_t = o_t \circ tanh(C_t)$, and the output function $\hat{y}_t = \sigma(W_o h_t)$.

Almost every neural network training problem can be reduced to tensor decomposition problem, taking the case of the fully-connected network. To represent a fully connected operation, we use the product of matrix-vector multiplication Wx_t, where W is the weight tensor, and x_t is the feature vector. As we know, the hidden layers automatically learn inter-parameter correlations in the training process. However, such correlations are almost "invisible"; if the network wants to learn those latent variables models, it has to extract them from higher dimensional information. So where those higher dimensional are?

Using the Truck decomposition operation can obtain the tensorized weight tensor W. However, due to the permutation symmetry of tensor multiplication, the optimum solutions may not be unique. In that situation, the average of optimal solutions is always optimal? The answer is negative, it could be suboptimal ,and that doesn't fit the pattern of a convex function. Here is the decomposition of weight tensor W:

$$W_{i_1 \ldots i_P} = \sum_{\alpha_1 \ldots} A^1_{\alpha_0 i_1 \alpha_1} A^2_{\alpha_1 i_2 \alpha_2} \ldots A^P_{\alpha_{P-1} i_P \alpha_P}$$

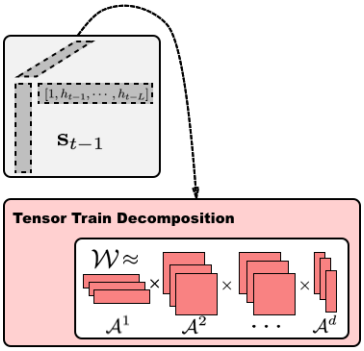


Figure 4.6 Tensor decomposition for weight tensor

So the objective(loss) function for LSTM is a non-convex function; we have to overcome this difficulty to optimize the training progress and make it more efficiently escape from the saddle points.

4.3.5 RMSProp Convergence Guarantee

Since the loss funtion f is L-smoothess, we can conclude inequality of x_{t+1} and x_t, they are inputs of consecutive iterates:

$$f(x_{x+1}) \leq f(x_t) - \alpha_t \langle \nabla f(x_t), V_t^{-\frac{1}{2}} \nabla f(x_t) \rangle + \frac{L}{\alpha_t^2} ||V_t^{-\frac{1}{2}} \nabla f(x_t)||^2$$

and

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

Since v_t is recursion, we need to can get the formula:

$$v_t = \beta_2(\beta_2 v_{t-2} + (1 - \beta_2) \nabla f(x_{t-1})^2) + (1 - \beta_2) \nabla f(x_t)^2$$

$$= (1 - \beta_2) \sum_{1<k<t} (\beta_2^{t-k} \nabla f(x_t)^2)$$

Let us review the square root of Moore - Penrose matrix inverse. $V_t^{-\frac{1}{2}} := \sum_{i \in Support(v)} \frac{1}{\sqrt{v_i}} e_i e_i^T$, where e_i is standard basis of R^d. So we can get the inequality $||V_t^{-\frac{1}{2}}|| \geq \frac{1}{max_{i \in Support(v_t)} \sqrt{(v_t)_i}} =$

$\frac{1}{max_{i \in Support(v_t)} \sqrt{(v_t)_i (1-\beta_2) \sum_{1<k<t} (\beta_2^{t-k} \nabla f(x_t)^2)}} = \frac{1}{max_{i \in Support(v_t)} \sigma \sqrt{(v_t)_i (1-\beta_2) \sum_{1<k<t} \beta_2^{t-k}}} = \frac{1}{\sigma \sqrt{1-\beta_2^t}}$,
where σ is the summary of each time gradient.

The Euclidean inner product of $\nabla f(x_t)$ and $V_t^{-\frac{1}{2}} \nabla f(x_t)$ is:

$$\langle \nabla f(x_t), V_t^{-\frac{1}{2}} \nabla f(x_t) \rangle = ||V_t^{\frac{1}{2}}|| * ||\nabla f(x_t)||^2$$

And we replace vector v_t, the inequality turn out to be:

$$\frac{1}{\sigma \sqrt{1-\beta_2^t}} ||\nabla f(x_t)||^2$$

$$\sum_{1<i<T} \frac{1}{a_t} [f(x_t) - f(x_{t+1})] \leq \frac{B_u}{\alpha} - \frac{B_l \sqrt{T+1}}{\alpha}$$

So it is clear to see we can found $||\nabla f(x_{result})||^2$ in $T = O(\frac{1}{e^4})$, which means the RMSProp can converge if we set designated error for it.

Chapter 5

Experiment and Result

5.1 Experiments

There are two goals for the experiment. The first goal is loss comparison, does RMSProp reduce the learning loss and improve the predicting performance? We compare the training loss with and without the RMSProp optimizer, which could give us a clear view of the performance improvement. Another goal is escaping saddle point; we would trace the eigenvalues of loss function's Hessian matrix if it is neither positive-definite matrix nor positive semi-definite matrix at one point, the point is a saddle point for the function. We need to check whether RMSProp has the capacity to move from "bad" saddle point a relative good saddle point regions where are close to the global optimum.

Before the training starts, we need to split the data set to a training subset and predicting subset, 10 % data would be randomly picked for training. Then we employ RMSProp optimizer with an initial learning rate of 0.001 to minimize the Mean Squared Error loss and learning rate decay 0.99 for gradient descent. Here is the formula of how the learning rate update based on learning rate decay for each epic:

For Learning Rate β, Learning Rate Decay δ and epic epc:

$$\alpha = \prod_{1<i<epc_{max}} MAX(i + 1 - epc_{init}, 0.0)$$

$$\beta = \beta_{init} * (\delta * \alpha)$$

And here are some revelent configurations for the training :

-num_layers = 3

-keep_prob = 0.8

-batch_size = 64

-init_learning_rate = 0.001

-learning_rate_decay = 0.99

-init_epoch = 5

-max_epoch = 100

-num_steps = 3

We would modify some of them for different epics, like layers number, keep probability, step and epoch, which would show how those parameters and hyperparameters have effect on the performances.

5.2 Results

5.2.1 Training and Testing Loss

- **Without RMSProp Optimizer**

Figure 5.1 shows actual and predictive normalized prices for 100 days, and blue lines show the predictive normalized prices, and red lines show the actual normalized prices. It is clear to see that there is no intersection of blue lines and red lines, which means there is no right prediction at all.

As Table 5.1 shows, the testing loss is 10 times of training error, which is underfitting. So what is wrong with it? The [Yann N. Dauphin] gives the answer, due to the non-convex properties of LSTM, the traditional SGD would stop progressing quickly and trap in a plateau, possibly around a saddle point. And the recurrent neural networks are more challenging to train than feedforward neural networks because there are more parameters and transactions involved.

We made an analysis of the non-convex properties and RMSProp convergence in previous sections. Theoretically, RMSProp could alleviate this hard optimization problem to improve performance, but we still need to more experiments for the guarantee, and that is our work for the next section.

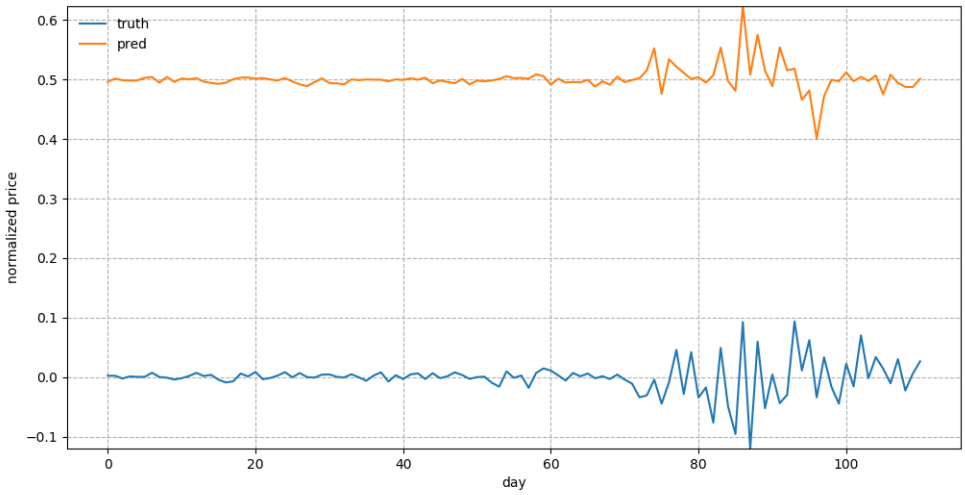


Figure 5.1 Result of experiment without RMSProp Optimizer

Date	Layers	LSTM Size	Loss
Training	1	32	0.0097
Testing	1	32	0.0108

Table 5.1: the formal of stock prices dataset

- **With RMSProp Optimizer**

- **Experiment NO.1**

For this section, we do 3 experiments with RMSProp Optimizer. They have the same data resource but different parameters since we need to know what parameters barely affect the performance, and what parameters have a significant impact on the final loss. There are two types of parameter: model parameters and hyperparameters. Model parameters, like weight and bias, are learned automatically in training progress. Hyperparameters are external parameters set by users. We would modify the Input Layers, LSTM Size and Max Epoch for different experiments and compare their loss.

For the first experiment, we set only 1 input layer, 32 for LSTM Size and 50 for max epoch, which is as same as the previous experiment, which is not optimized by the RMSProp

optimizer. The figure shows actual and predictive normalized prices for 100 days, the blue lines show the predictive normalized prices, and the red lines show the actual normalized prices. We can see there are several intersections of blue lines and red lines this time, which means there is some perfect fitting prediction. Comparing to the previous experiment without RMSProp, the final training loss decline from 0.0097 to 0.00010, and the testing loss is only 0.00092, which is almost $\frac{1}{10}$ of old ones 0.0108. So it is evident that RMSProp does alleviate the non-convex relevant problems and improve the performance of LSTM.

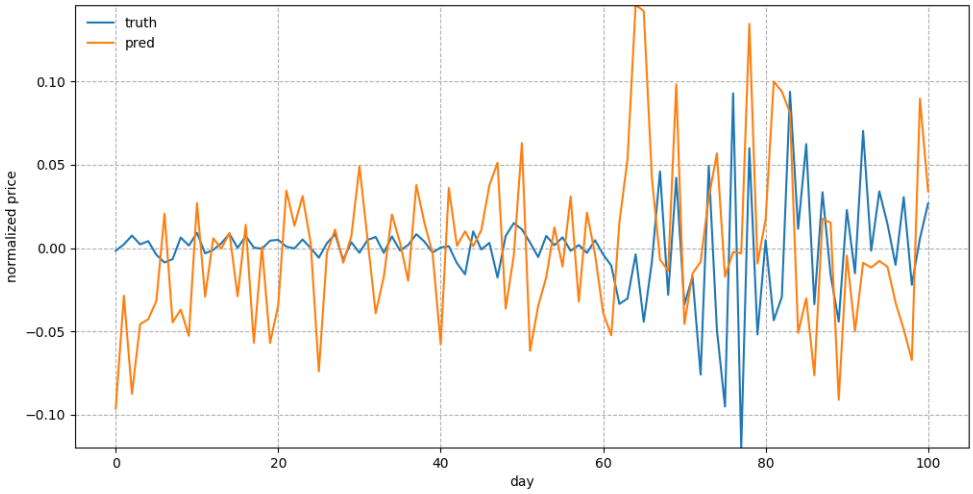


Figure 5.2 Result of experiment NO.1

Date	Layers	LSTM Size	Loss
Training	1	32	0.00010
Testing	1	32	0.00092

Table 5.2: the formal of stock prices dataset

- **Experiment NO.2**

For the second experiment, we only change LSTM Size to 128 and keep the rest of the parameters as same as the first one. As a result, both training loss and test loss decline, and there was a marked drop in test loss, which is almost $\frac{1}{10}$ of old. So what is the mechanism behind this remarkable improvement? LSTM Size is the number of nodes in one LSTM

layer; for a fully connected LSTM, more nodes in one layer, more hidden variables will be traced, which update the fitting ability of the network. However, once we increase the LSTM Size over 512, the testing loss goes worse while the training loss is still in good condition. So we still need the LSTM Size in the appropriate range to make a fair prediction.

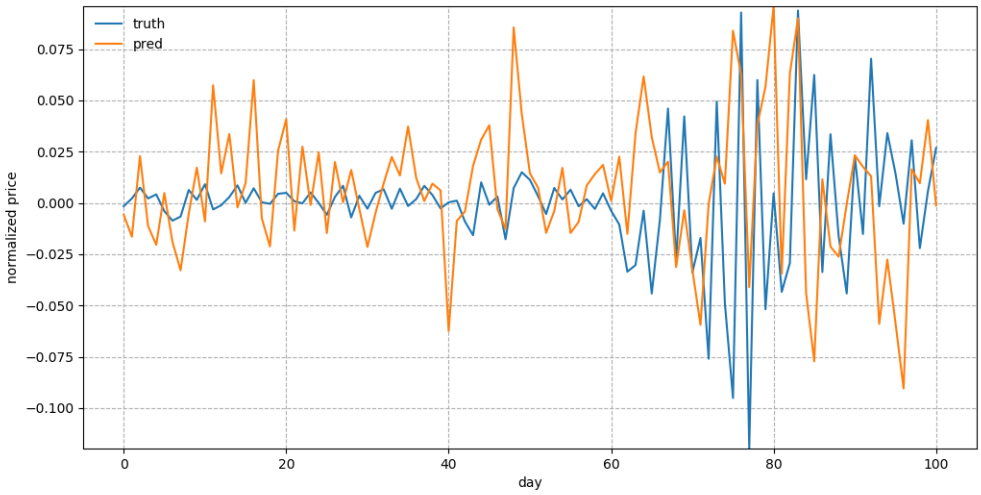


Figure 5.3 Result of experiment NO.2

Date	Layers	LSTM Size	Loss
Training	1	128	0.00007
Testing	1	128	0.00050

Table 5.3: the formal of stock prices dataset

- **Experiment NO.3**

For the third experiment, we increase the layers of LSTM; then, we get the optimal result among 3 experiments. Apparently, properly increasing hidden layers can make a better performance, because deep learning systems enable machines to process data with a nonlinear approach, and more hidden layers can increase the complexity of nonlinear behaviour, which would give LSTM better network generalization ability.

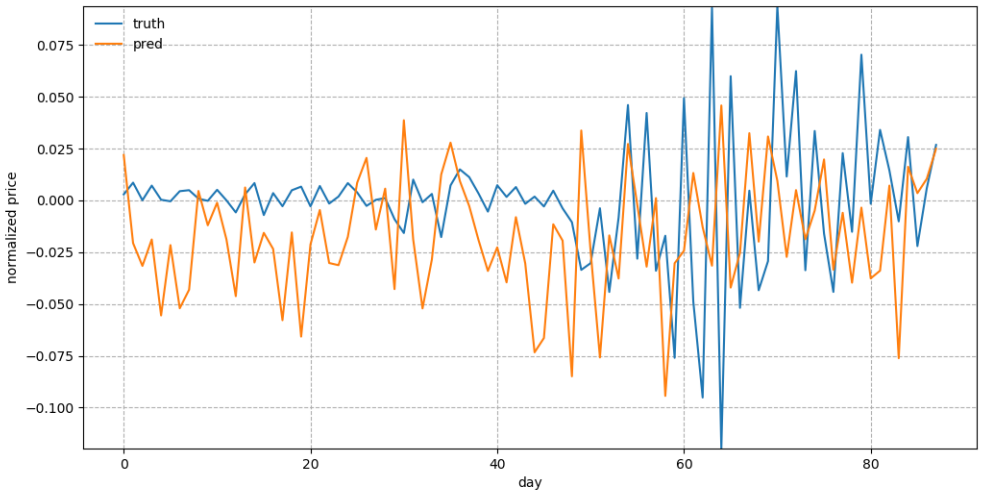


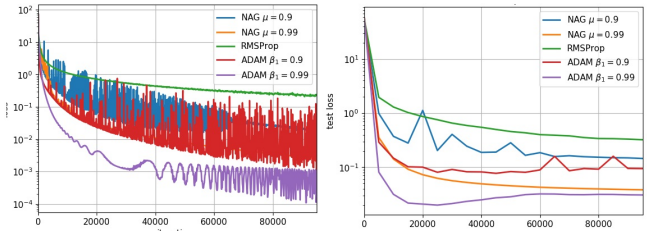
Figure 5.4 Result of experiment NO.3

Date	Input Layers	LSTM Size	Loss
Training	5	128	0.00004
Testing	5	128	0.00018

Table 5.4: the formal of stock prices dataset

5.2.2 Hessian Matrix of Loss function

We need to track the most negative eigenvalue of the Hessian matrix. We find how considering the presence of negative eigenvalues of the Hessian could help us design better suited adaptive learning rate schemes.



Hessian Matrix diagram

Chapter 6

Future Work and Conclusion

6.1 Future Work

Through we make RMSProp as a fair solution to improve the efficiency, when we input the high dimensional data, the training speed would slow down. In the previous sections, we use the tensor decomposition for non-convex analysis. Moreover, tensor decomposition can also be employed to build correlation modelling in higher dimensions, which is more efficient feature extraction, because the lesser parameters are involved in the training progress. As we know, LSTM is the neural network with massive parameters, so such method is helpful for efficiency improvement.

6.2 Conclusion

In this project, we give a comprehensive picture of a deep learning non-convex optimization. We introduce the features and merits of a modified recurrent neural network - Long short-term memory. Meanwhile, we present the mechanism of RMSProp and how it works on the LSTM to make better performance. Further, our experiment results show that RMSProp beat up plain SGD in terms of accuracy and efficiencies, which convinced us that RMSProp is robust optimizer to solve the non-convex problem of LSTM though it is a 1st order derivative measure. However, that is not the end of the job; we would use more updated tensor decomposition to build variant network, which can further improve the efficiency of training.

Bibliography

[1] Fundamentals of recurrent neural network (rnn) and long.

[2] Razvan Pascanu andTomas Mikolov and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[3] Y. Bengio, P. Frasconi, and P. Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, pages 1183–1188 vol.3, 1993.

[4] Yoshua Bengio, Nicolas Boulanger Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. *CoRR*, abs/1212.0901, 2012.

[5] Rong Ge. *Escaping From Saddle Points Online Stochastic Gradient for Tensor Decomposition*. PhD thesis, Microsoft, Seattle, Washinton, 2017.

[6] Schmidhuber J Hochreiter S. Long short-term memory. `https://www.bioinf.jku.at/publications/older/2604.pdf`, 1997.

[7] Wojciech Zaremba. *Recurrent Neural Network Regularization*LaTeX. PhD thesis, New York University, 2015.

```
import json
import os
import sys; sys.path.append("..")
import tensorflow as tf
from build-graph import build-lstm-graph-with-config
from config import DEFAULT-CONFIG, MODEL-DIR
from data-model import StockDataSet
def load-data(stock-name, input-size, num-steps): stock-dataset = StockDataSet(stock-
name, input-size=input-size, num-steps=num-steps, test-ratio=0.1, close-price-only=True)
print "Train data size:", len(stock-dataset.train-X)
print "Test data size:", len(stock-dataset.test-X)
return stock-dataset
def -compute-learning-rates(config=DEFAULT-CONFIG): learning-rates-to-use = [ config.init-
learning-rate * ( config.learning-rate-decay ** max(float(i + 1 - config.init-epoch), 0.0) ) for
i in range(config.max-epoch) ]
print "Middle learning rate:", learning-rates-to-use[len(learning-rates-to-use) // 2]
return learning-rates-to-use
def train-lstm-graph(stock-name, lstm-graph, config=DEFAULT-CONFIG):
stock-dataset = load-data(stock-name, input-size=config.input-size, num-steps=config.num-
steps)
final-prediction = []
final-loss = None
graph-name = ( stock-name, config.init-learning-rate, config.learning-rate-decay, config.lstm-
size, config.num-steps, config.input-size, config.batch-size, config.max-epoch)
print "Graph Name:", graph-name
learning-rates-to-use = -compute-learning-rates(config)
with tf.Session(graph=lstm-graph) as sess: merged-summary = tf.summary.merge-all()
writer = tf.summary.FileWriter('-logs/' + graph-name, sess.graph) writer.add-graph(sess.graph)
graph = tf.get-default-graph() tf.global-variables-initializer().run()
inputs = graph.get-tensor-by-name('inputs:0')
targets = graph.get-tensor-by-name('targets:0')
learning-rate = graph.get-tensor-by-name('learning-rate:0')
test-data-feed = inputs: stock-dataset.test-X, targets: stock-dataset.test-y, learning-
rate: 0.0
loss = graph.get-tensor-by-name('train/loss-mse:0')
minimize = graph.get-operation-by-name('train/loss-mse-adam-minimize')
prediction = graph.get-tensor-by-name('output-layer/add:0')
```

for epoch-step in range(config.max-epoch): current-lr = learning-rates-to-use[epoch-step]
for batch-X, batch-y in stock-dataset.generate-one-epoch(config.batch-size):
train-data-feed = inputs: batch-X, targets: batch-y, learning-rate: current-lr
train-loss, - = sess.run([loss, minimize], train-data-feed)
if epoch-step test-loss, -pred, -summary = sess.run([loss, prediction, merged-summary], test-data-feed) assert len(-pred) == len(stock-dataset.test-y)
if epoch-step
print "Predictions:", [(map(lambda x: round(x, 4), -pred[-j]), map(lambda x: round(x, 4), stock-dataset.test-y[-j])) for j in range(5)]
writer.add-summary(-summary, global-step=epoch-step)
print "Final Results:" final-prediction, final-loss =
sess.run([prediction, loss], test-data-feed)
print final-prediction, final-loss
graph-saver-dir = os.path.join(MODEL-DIR, graph-name)
if not os.path.exists(graph-saver-dir): os.mkdir(graph-saver-dir)
saver = tf.train.Saver()
saver.save(sess, os.path.join(graph-saver-dir, "stock-rnn-model-
with open("final-predictions..json".format(graph-name), 'w') as fout: fout.write(json.dumps(final-prediction.tolist()))
def main(config=DEFAULT-CONFIG): lstm-graph = build-lstm-graph-with-config(config=config) train-lstm-graph(", lstm-graph, config=config)
if –name– == '–main–': main()
def build-lstm-graph-with-config(config=None):
tf.reset-default-graph()
lstm-graph = tf.Graph()
if config is None:
config = DEFAULT-CONFIG
with lstm-graph.as-default():
learning-rate = tf.placeholder(tf.float32, None,
name="learning-rate")
Number of examples, number of input, dimension of each input
inputs = tf.placeholder(tf.float32, [None, config.num-steps, config.input-size], name="inputs")
targets = tf.placeholder(tf.float32, [None, config.input-size], name="targets")
def -create-one-cell(): lstm-cell = tf.contrib.rnn.LSTMCell(config.lstm-size, state-is-tuple=True) if config.keep-prob ¡ 1.0: lstm-cell = tf.contrib.rnn.DropoutWrapper(lstm-cell, output-keep-prob=config.keep-prob)
return lstm-cell
cell = tf.contrib.rnn.MultiRNNCell([-create-one-cell() for - in range(config.num-layers)], state-is-tuple=True)
if config.num-layers ¿ 1 else -create-one-cell()
val, - = tf.nn.dynamic-rnn(cell, inputs, dtype=tf.float32, scope="lilian-rnn")
Before transpose, val.get-shape() = (batch-size, num-steps, lstm-size)
After transpose, val.get-shape() = (num-steps, batch-size, lstm-size)
val = tf.transpose(val, [1, 0, 2])

with tf.name-scope(”output-layer”): last.get-shape() = (batch-size, lstm-size) last = tf.gather(val, int(val.get-shape()[0]) - 1, name=”last-lstm-output”)

weight = tf.Variable(tf.truncated-normal([config.lstm-size, config.input-size]), name=”lilian-weights”) bias = tf.Variable(tf.constant(0.1, shape=[config.input-size]), name=”lilian-biases”) prediction = tf.matmul(last, weight) + bias

tf.summary.histogram(”last-lstm-output”, last)

tf.summary.histogram(”weights”, weight)

tf.summary.histogram(”biases”, bias)

with tf.name-scope(”train”): loss = -tf.reduce-sum(targets * tf.log(tf.clip-by-value(prediction, 1e-10, 1.0)))

loss = tf.reduce-mean(tf.square(prediction - targets), name=”loss-mse”)

optimizer = tf.train.AdamOptimizer(learning-rate)

minimize = optimizer.minimize(loss, name=”loss-mse-adam-minimize”)

tf.summary.scalar(”loss-mse”, loss)

for op in [prediction, loss]: tf.add-to-collection('ops-to-restore', op)

return lstm-graph