EMPIRICAL STUDY OF DEEP NEURAL NETWORK ARCHITECTURES FOR PROTEIN SECONDARY STRUCTURE PREDICTION

A Thesis presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Ming Du

Dr. Yi Shang, Thesis Supervisor

May 2017

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

EMPIRICAL STUDY OF DEEP NEURAL NETWORK ARCHITECTURES FOR PROTEIN SECONDARY STRUCTURE PREDICTIONx

presented by Ming Du,

a candidate for the degree of Master of Computer Science.

And hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Yi Shang

Dr. Dong Xu

Dr. Ioan Kosztin

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. Yi Shang for all of his guidance, support and patience for all these time. I also would like to thank Dr. Dong Xu for the professional opinions and valuable advises he gave me. And, I would like to thank Dr. Ioan Kosztin for being on the thesis committee. I would like to specially express my gratitude to my friend Pinwen Xu, he provides me the Alienware computer to run all the experiments.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

Protein secondary structure prediction is a sub-problem of protein structure prediction. Instead of fully recovering the whole three dimensional structure from amino acid sequence, protein secondary structure prediction only aimed at predicting the local structures such as alpha helices, beta strands and turns for each small segment of a protein. Predicted protein secondary structure can be used for improving fold recognition, ab initial protein prediction, protein motifs prediction and sequence alignment.

Protein secondary structure prediction has been extensively studied with machine learning approaches. And in recent years, multiple deep neural network methods have pushed the state-of-art performance of 8-categories accuracy to around 69%. Deep neural networks are good at capturing the global information in the whole protein, which are widely believed to be crucial for the prediction. And due to the development of high level neural network libraries, implementing and training neural networks are becoming more and more convenient and efficient.

This project focuses on empirical performance comparison of various deep neural network architectures and the effects of hyper-parameters for protein secondary structure prediction. Multiple deep neural network architectures representing the state-of-the-art for secondary structure prediction are implemented using TensorFlow, the leading deep learning platform. In addition, a software environment for performing efficient empirical studies are implemented, which includes network input and parameter control, and training, validation, and test performance monitoring. An extensive amount of experiments have been conducted using popular datasets and benchmarks and generated some useful results. For example, the experimental results show that recurrent layers are useful in improving prediction accuracy, achieving up to 5% improvement on 8-category accuracy. This work also shows the trade

off between running speed and building speed of the model, and the trade off between running speed and accuracy. As a result, a relatively small size recurrent network have been build and achieved 69.5% 8-category accuracy on dataset CB513.

Chapter 1

INTRODUCTION

Accurately and reliably predicting 3D structures, from protein sequences is one of the most challenging tasks in computational biology, and has been of great interest in bioinformatics. An important intermediate step of predicting the whole 3D structure is correctly predicting the secondary structure of a protein[1]. In recent years, Deep neural networks have been widely apply to the problem of protein secondary structure prediction and continuously pushing the state-of-the-art forward. For example, in[2] a convolutional Generative Stochastic Network achieved 66.4% Q8(8 categories) on CB513 dataset. In [3] a convolutional recurrent network which is a combination of convolutional network and recurrent network achieved 69.7% Q8 accuracy. And in [4] they use a multi-scale convolutional network together with many techniques that help accelerating training and prevent over-fitting, such as dropout[5], batch-normalization[6] and regularization. The best Q8 result they report on CB513 is 70.6%.

One advantage of deep learning and deep neural network is the reusability. On one hand, a certain kind of architecture can apply to different problems without much modification. For example, a convolutional recurrent network similar to the one in [3] have also been apply to video activity recognition and Video Description[7]. And

the convolutional architecture in[4] are originally used in image recognition. On the other hand, most complex networks contain reusable modules such as different layers, regularizer and optimizer. So in this case a good way to build a deep neural network is using a reliable deep learning framework such as Torch, TensorFlow Caffe and Theano. TensorFlow[8] is a python deep learning framework create by Google. Using TensorFlow, it easy for researchers to visualize the graph and training process. And the API designs make researchers code shareable, standardize how software engineers approach deep learning.

The first goal the this work is to design and build a deep learning system that handles the training, validation, evaluation and save/restore of deep neural networks for protein secondary structure prediction. The second goal is to use the system to compare the performances such as accuracy, running speed and building time of different network architectures.

This thesis will show how to build different deep neural networks using TensorFlow to solve the protein secondary structure prediction problem. And this work will particular focus on how to build a recurrent network and the technique difficulties in doing this. Chapter 2 will introduce the related deep neural network architectures that designed for protein secondary structure prediction, and the mechanism of recurrent neural network. And chapter 3 will describe the models in detail. Chapter 4 will cover the design, implementation and technique difficulties. Chapter 5 will show the result of all the experiments.

Chapter 2

RELATED WORKS

2.1 Recurrent networks

Different from other networks such as feed forward neural network(FFNN) and convolution neural network, recurrent network works well on problems that have sequential input such as speech recognition, language modeling, translation and image captioning. Figure 2.1 shows how a typical recurrent network looks like. A loop in the network allows it to pass the information from previous steps to the network to make a decision for the current input. In contrast to the fix length context window used in FFNN, RNN stores the activation of previous time step in hidden state and it provide context information within a undefined window size.

However, training conventional RNNs using back-propagation technique is difficult due to the vanishing gradient and exploding gradient problems[9]. The gradient shifting problem make also make it difficult for the network to remember long time dependency that longer than 5-10 time steps between input and output.

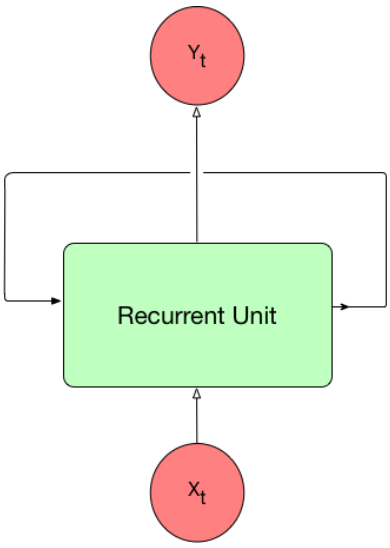

Figure 2.1: Illustration of recurrent network

2.1.1 Long Short term memory

In order to address the problem above, a special recurrent network architecture Long-Short-Term Memory(LSTM) have been proposed[10]. LSTM and it's variations have been successfully applied to sequence prediction, translation and sequence labeling tasks. The rest of this chapter will introduce how LSTM and one of its variantion GRU(gated recurrent unit) works.

2.1.2 Long short term memory

To make it easier to understand the idea of LSTM, in Figure 2.2 the loop in the network been unrolled and results in a fix length static version of the same network. The operation in the two RNN version are the same, the difference is the static version only take fix length input and don't have to consider when to stop the iteration thus it runs faster.

When we look at the inside of each recurrent unit(Cell), a conventional RNN can be represent as Figure 2.3. The cell take the concatenation of input of time t and output of time t-1 as input. Going through a dense layer the cell will output the

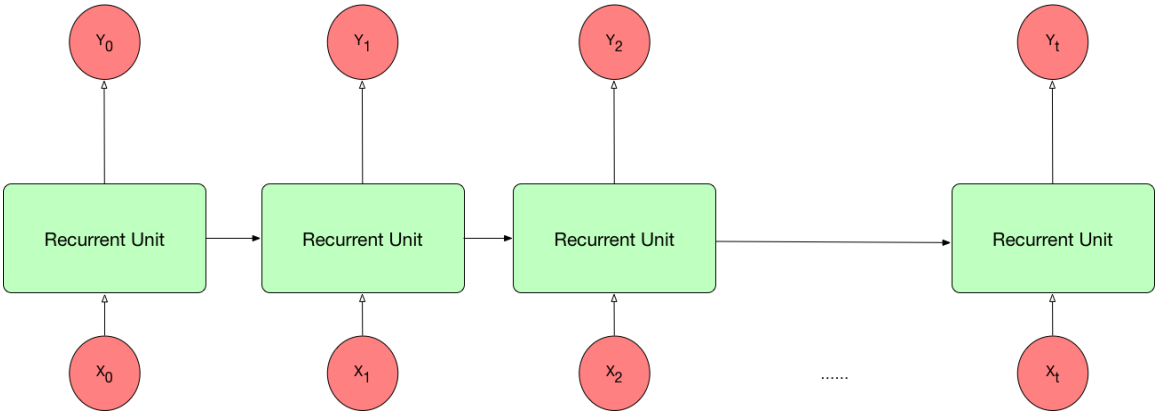

Figure 2.2: unrolled recurrent network

activation of the dense layer.

LSTM also have the same loop structure of conventional RNN, but it has different structure inside the cell. Instead of having only a single dense layer in the cell, the LSTM cell has four dense layers. One of them is the counter-part in conventional RNN cell, the rest three, however, works as three "gates" controlling the behavior of the cell(Figure 2.4).

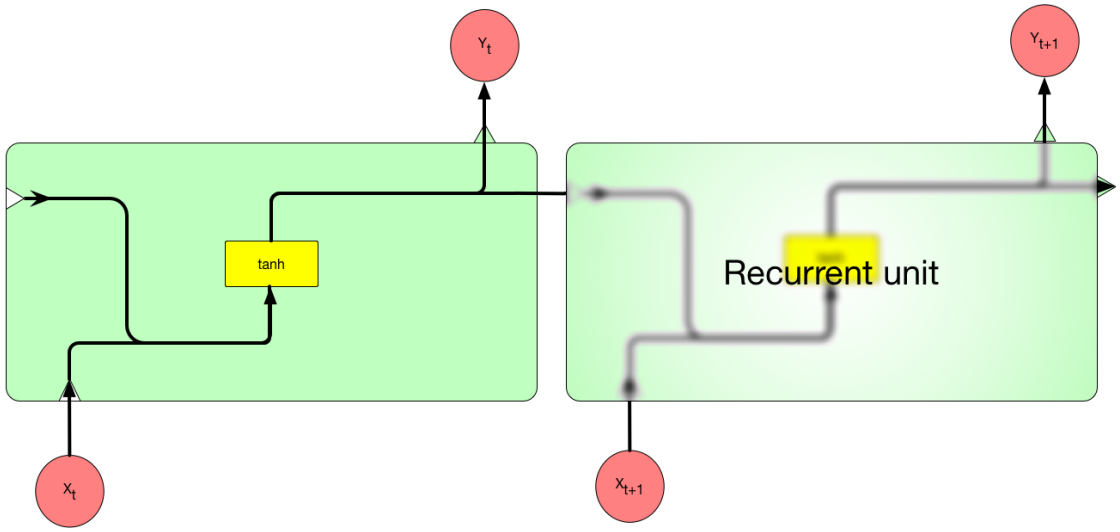

Figure 2.3: Detail inside recurrent unit

The first step of calculating the output h_t of each time step is updating the cell state C_t. Because the output of each time step is essentially the cell state masked by a weight matrix. In order to update the cell state, the output of the first three

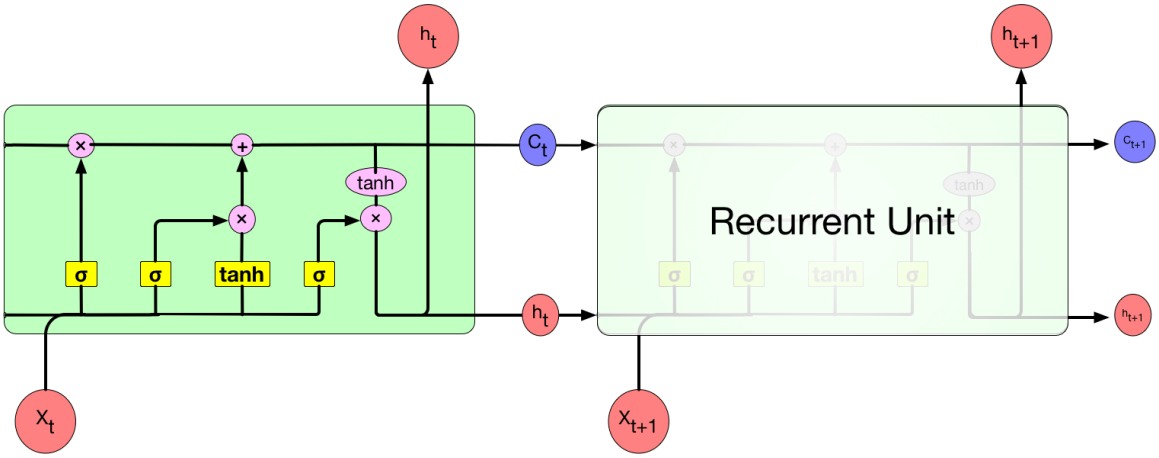

Figure 2.4: Detail inside LSTM

dense layer inside the cell needed to be calculated. They are forget gate layer, input gate layer and *tanh* layer respectively. The activation of these three layers can be calculated by using 2.1a, 2.1b and 2.1c. Then according to 2.2 the network will decide how to update the cell state by forgetting some of the old state and remember some of the new input state.

After updated the cell state, the network will calculate the output of this time step using 2.3a and 2.3b. More specifically speaking, 2.3a calculate the output gate weights o_t and 2.3b apply the weight to the activation of cell state.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.1a}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.1b}$$

$$\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.1c}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C} \tag{2.2}$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.3a}$$

$$h_t = o_t * tanh(C_t) \tag{2.3b}$$

2.1.3 Gated recurrent unit

The gate recurrent unit(GRU)[11] is a variance of LSTM. In fact there are many different version of LSTM. The version described in previous chapter is the simplest one. One problem of this LSTM version is that the number of parameters is about 4 times of a conventional recurrent network, because it have four dense layers inside the cell. And it have to keep the cell state which will consume more memory. These problems slow down the speed of the network. The GRU is a special variance of LSTM that only has three dense layer and doesn't have the cell state.

As illustrate in Figure 2.5, GRU merge the input gate and forget gate in LSTM cell into a single update gate z_t. The update gate z_t selects whether the hidden state is to be updated with a new hidden state $\tilde{h}$. The reset gate r_t decides whether the previous hidden state is ignored. See Eqs. 2.4a-2.4b for the detailed equations of r, z, h and $\tilde{h}$.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \tag{2.4a}$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \tag{2.4b}$$

$$\tilde{h} = tanh(W \cdot [r_t * h_{t-1}, x_t]) \tag{2.4c}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h} \tag{2.4d}$$

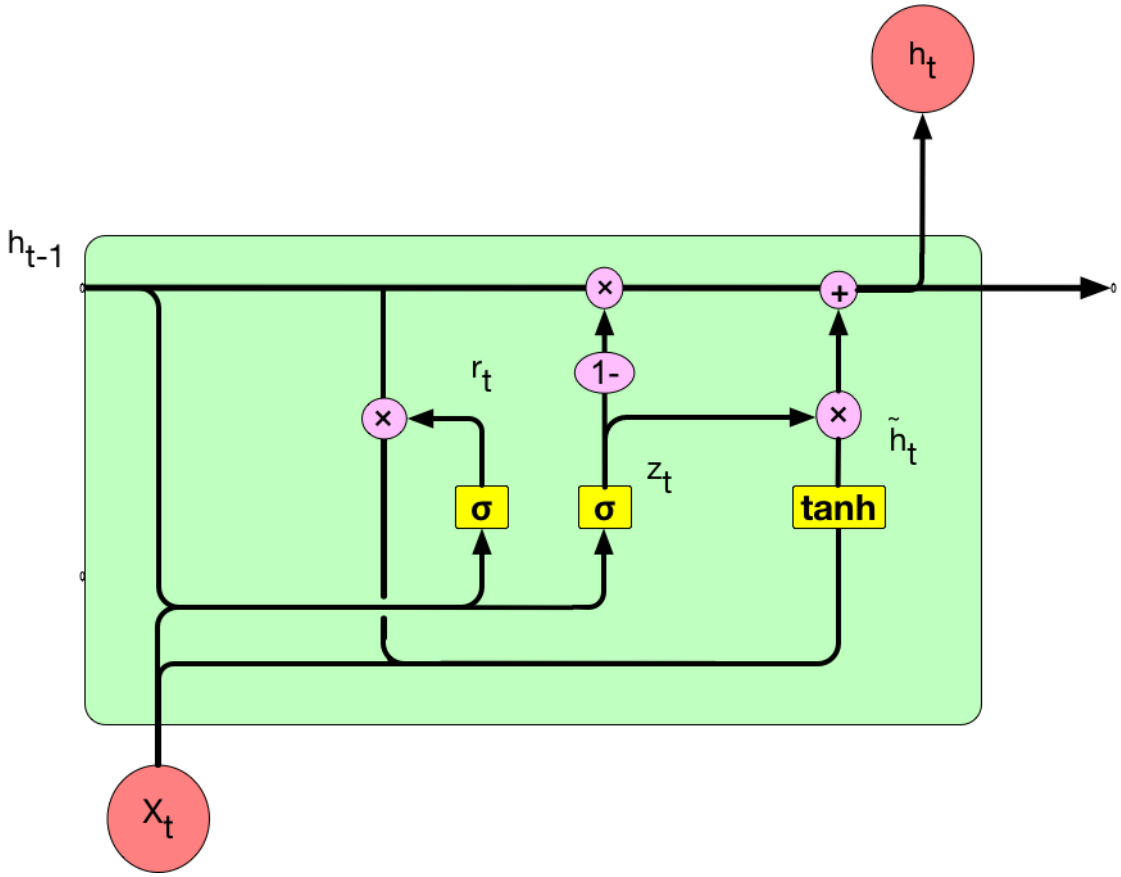

Figure 2.5: details in GRU

2.2 DNNs for protein secondary structure prediction

Despite some early attempts using neural networks as submodules to predict protein secondary structure such as condition conditional neural fields [12]. Several end-to-end trainable deep neural networks have also been proposed to solve the 8-class secondary structure prediction problem. The first one is a convolutional Generative Stochastic Network proposed by Zhou et al.[2]. This method achieved 66.4% 8-class accuracy on CB513. This model is not very effective since simply using multiple convolutional layers can also achieve 66% accuracy. The second method, or second kind of architecture is using recurrent network combine with convolutional layers. Sren[13] and Z.Li et al. [3] both proposed their recurrent neural network and achieved

67.4% and 69.7% 8-class accuracy respectively. The major difference of these two method is the first one using 1 layer bidirectional long short term memory(LSTM) cell as recurrent unit, the second one using 3 layers of bidirectional gated recurrent unit(GRU) cell instead. Sren provides the theano source code of the method on github and continues the development, the current best accuracy is 68.9%. The last architecture is proposed by Busia et al.[4]. They stacks multiple convolutional blocks which similar to inception architecture in the network and get 70.6% accuracy on CB513. Their convolutional model is designed to mimic the behavior of a recurrent sequence to sequence model and is evaluated using beam search.

Chapter 3

DEEP NEURAL NETWORK ARCHITECTURES FOR PROTEIN SECONDARY STRUCTURE PREDICTION

3.1 Problem definition

The exact definition of protein secondary structure prediction in this work is the following. Given the amino acid sequence $a_0a_1...a_{l-1}, 0 \leq i < l$ and the profile $p_0p_1...p_{l-1}, 0 \leq i < l$ of the of a protein. The goal is to predict the secondary structure of the protein which can also be represented as a sequence $s_0s_1...s_{l-1}, 0 \leq i < l$. In general there are 21 different kinds of amino acid, so each a_i have 21 different classes. And the profile p_i is a vector of probabilities, $p_i[k]$ representing the possibilities of the ith amino acid to be kth class. As for the s_i, it has following 8 classes.

label	name
H	*alpha*-helix
E	*beta*-strand
L	loop or irregular
T	*beta*-turn
S	bend
G	3_10-helix
B	β-bridge
I	π-helix

Table 3.1: The label and its correspond names of 8-class secondary structure

3.2 Network architecture

Z.Li et al. [3] proposed a deep convolutional and recurrent neural network for predicting protein secondary structure. In their work a multiscale convolutional layer is followed by bidirectional GRU recurrent layers. This work follows their general architecture as illustrate in Figure 3.1 which have preprocessing, multiscale convolution, recurrent and output four major parts. More details of these parts are explained in the following sections.

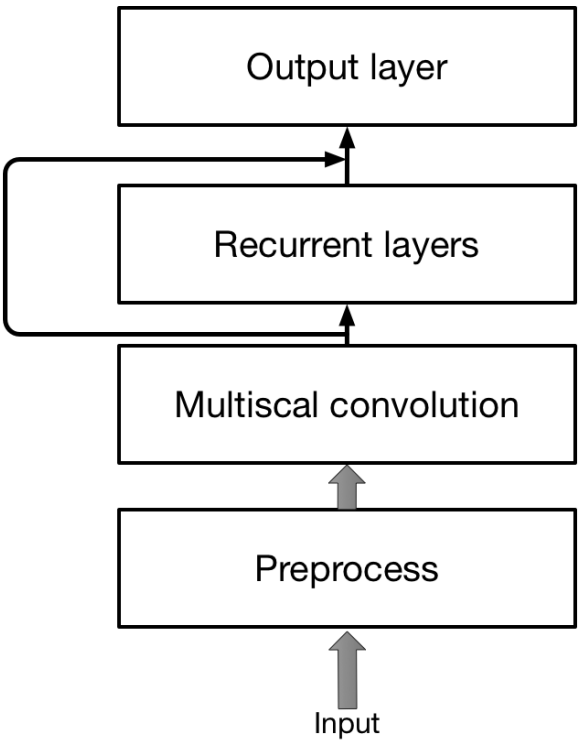

Figure 3.1: Overall architecture

3.2.1 Input and output

Input

The input of the network have the shape of [sequence_length, features]. In this work the number of features is 42, which contains two parts. For each amino acid in the sequence, the first 21 features is a one-hot-encoding indicate which amino acid it is. The last 21 features are the profiles features obtained from PSI-BLAST[14] which is a dense vector.

Output

The output the network have the shape of [sequence_length, 8]. So for each amino acid in the sequence it will output a one-hot-encoding to indication which secondary structure it is.

3.2.2 Preprocessing

As illustrate in Figure 3.2 What the preprocessing module does is converting the sparse one-hot-encoding of amino acid to a dense representation, and merge it with the dense profile feature vector. In order to avoid the inconsistency of feature representations. Same as the original work, a 21-by-50 embedding matrix is used. Another poteintial benefit of using a embedding matrix is the embedding matrix can be initialized with a pretrained matrix which comes from a sequence auto-encoder.

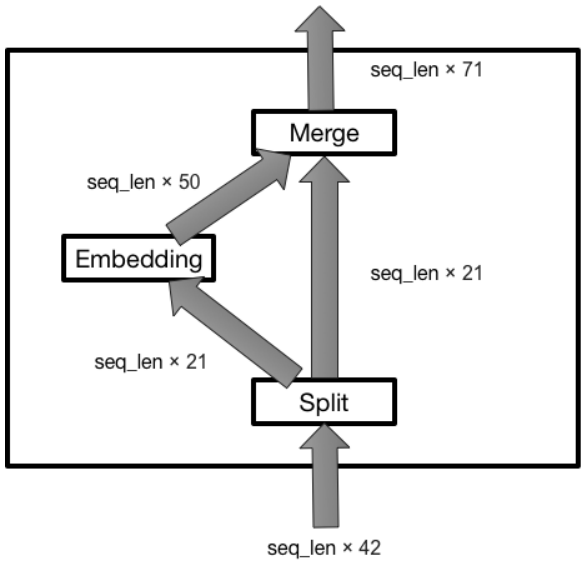

Figure 3.2: preprocess layer

3.2.3 Multiscale convolution

After getting embedded features, the next step is using different 1-D convolutional kernels to extract loacl context information from the sequence. As shown in Figure 3.3, total three kernels are used. Each of them have 64 output channels. And the shape of the kernel are 3, 7, 11 respectively. The kernel will moving along the protein with stride size 1. A concatenate layer will merge the output of three convolution layers and then a RELU layer will generate the activation. So far the module is the same as the one in original work. However in this work, a batch normalization layer

is added after the activation. This layer will normalize the output of the module to have same distribution at each position and speed up the converging make the network learn faster. However, while learning faster, the model is more likely to overfit. Thus a dropout layer will also add at the end of this module.

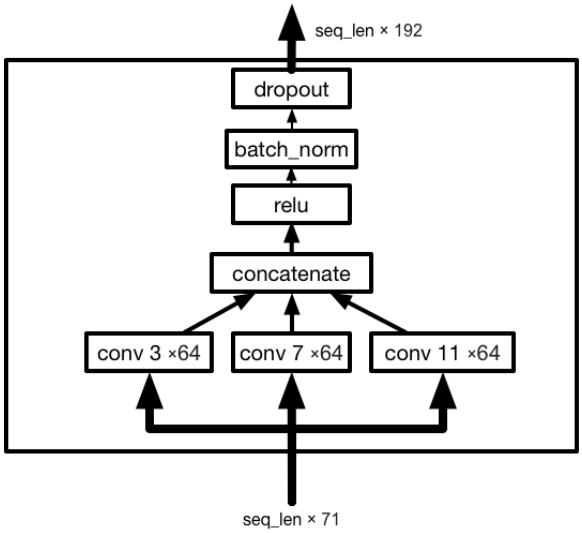

Figure 3.3: multi-scale convolutional layer

3.2.4 Recurrent layers

In addition to the local dependency extracted by convolutional network, Z.Li et al. [3] argued that long-range dependency is also important for secondary structure prediction. However a convolutional layer can not capture dependencies that longer than the kernel size. In order to capture long-range dependencies, bidirectional recurrent layers are placed after the multiscale convolution layer. Although conventional recurrent network are able to capture long dynamic range dependency, it very difficult to train due to gradient vanishing problem. So in recent years, only RNN with LSTM cells and its variance are practically used. In this work, are used instead of standard LSTM since it has less parameters and can achieve comparable result[15]. Figure 3.4 shows the structure of a single bidirectional GRU layer. In this layer, there are

actually two RNN layers, a forward layer scan from the first position of the sequence to the last position, and a backward layer which scan from the last to the first position. And the output features of forward and backward layer at each position will be concatenated together to form the final output of the bidirectional GRU layer.

3.2.5 Output layers

The main function of the output layers is adjusting the dimension of the feature from previous layer to match the size of the label. In typical classification task, several fully connected layers are served for this purpose. For example in [3] 2 fully connected layers are put together, adjusting the hidden feature to the size of 14 for each residue in the sequence.

However busia et al. [4] use a little different approach. In stead of using the hidden feature of each amino acid as the input of the fully connected layer, they use a sliding window of 11, and use the flattened context features of 11 amino acid as the input of the fully connected layers. Inspired by the sliding window method, this work use a simple convolution layer of kernel size 11 for the same purpose, because it easier to implement.

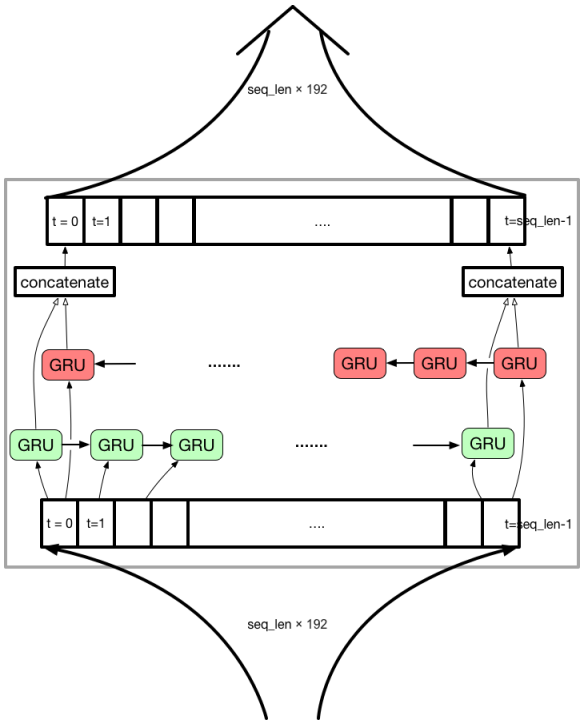

Figure 3.4: Recurrent layer

3.2.6 Loss and optimizer

The loss function are composed of three parts, the average cross-entropy loss between predicted secondary structure and true secondary structure, the average cross-entropy loss of solvent accessibility and the summary of all l2 norm weight regularization. For each protein sequence of length l the loss function formulate as equation 3.1a.

$$Loss = \frac{1}{l}\sum_{i}^{l} L_s(s_i, s_i^*) \quad + \quad \frac{\lambda_1}{l}\sum_{i}^{l} L_a(a_i, a_i^*) \quad + \quad \lambda_2 \Sigma \left\|\theta_j\right\|^2 \tag{3.1a}$$

$$L_s(s_i, s_i^*) = -s_i^* log(s_i) \tag{3.1b}$$

$$L_a(a_i, a_i^*) = -a_i^* log(a_i) \tag{3.1c}$$

Where s_i, a_i represent the predicted secondary structure and solvent accessibility for each residue. s_i^*, a_i^* represent is the true label and θ_j is the weight to be regularized.

Adam optimizer [16] are used for training the network.

Chapter 4

DESIGN AND IMPLEMENTATION OF A DNN LEARNING SYSTEM IN TENSORFLOW FOR PROTEIN SECONDARY STRUCTURE PREDICTION

Deep learning in these days are not just simply building a model, train it on training set and test it on testing set. On one hand, there is a common trend in recent development of new network architectures. That is modularization, instead of finding a best architecture for a certain task, the most exciting and successful techniques are focusing on finding modules or strategy that can be add to current architectures and improve the performance. For example, dropout and batch normalization are layers that can be add after every activation in order to speed up the learning and prevent overfitting. And residue net and inception [17] construct their structure by stacking modules repeatedly. So a good deep learning program should be able to easily add or remove certain module or function to/from the model. On the other

hand, while the network architecture becoming more complex and the dateset getting bigger, the training time are getting longer and often need to run the program on GPU, multi-GPUs or clusters. In order to meet different requirements without changing the program too much, a robust and flexible program are needed. Fortunately, TensorFlow provides excellent building blocks to build such programs and the following sections will introduce the design and TensorFlow implementation of a system that has training, validation, testing, inference, logging, monitoring and save/restore functions.

4.1 System design

4.1.1 A brief introduce of TensorFlow mechanism

The unique way of how TensorFlow works have a huge influence on the system design fashion. So it's important to have some understanding of TensorFow, before introducing the system.

There are two major parts in every TensorFlow program, that is graph and session.

1. Graph: a computational model contains all the operation you want to perform on the dataset.
    - The graph is only a data structure record all the operations that will be executed when data feed into the graph.
    - The graph can't run by itself, it need a session to provide the running environment.
    - Once the graph is finalized, the structure of the graph can not be changed anymore.
2. Session: encapsulates the environment for a graph to be executed.

- the environment include data input/output, multithread coordinator, save/restore mechanism, logging and monitor.
- Training, validation, and early stopping will handled by session

4.1.2 Overall design

As illustrate in Figure 4.1, the system has four module in general:

1. Main module: The most important module in the system, contains the network architecture described in Chapter 3. And inside the main module, there are three sub-modules in order to separate the training validation and inference procedure. This design makes it possible to run the training, validation and inference procedure using different threads and on different devices(CPUs/GPUs).

2. Input module: There are several different data input method in TensorFlow(From memory from files etc.). So the input module handles the different input from different source and different formats. This module separate the main module from the data by providing a standard input interface.

3. Monitor module: This is a utility module that help keep track the training process in real time. This module will run in a separate thread and save requested intermediate results and statistics to log file periodically. And these results can be visualized using TensorBoard in real time.

4. Save and restore module: It's common for a deep neural network to train for days. And there are also many things can interrupt the training(power down, run out of memory, pause by user etc.). So it's really helpful to have a module to save checkpoint files periodically and restore the training procedure when the user want to continue training.

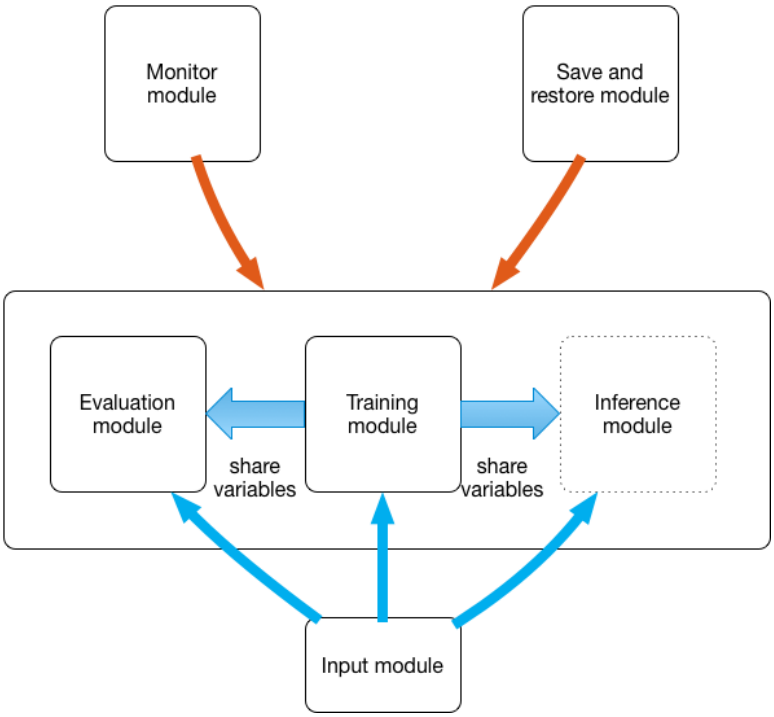

Figure 4.1: System overview

4.1.3 Training, evaluation, inference module design

Now let's further demonstrate the detail of the main module in Figure 4.1. As illustrate in Figure 4.2 the main module have three different sub-modules, they are training, evaluation and inference. According to different functionality, they have different components. The simplest one is the inference module. This module is only used when user already have a trained model and what to predict the secondary structures for new proteins. The only output is the prediction. The other relative simple one is the Evaluation model, this module is useful when user want to calculate the model performance on certain dataset. So beside prediction, this module also calculate the accuracy and loss. The last module is the training module, which calculate the gradient and update the correspond parameters in the model.

The following pseudo code is a pipeline for training a deep neural network. Generally speaking, when training a network, it's better to evaluate the training model

in real time. It can save running time by early stopping the training when the model start overfitting or the improvement is really small. Comparing to the method that save a series of candidate best models. This approach also saves disk space, because at each time the system only need to keep two checkpoint files, one file records the current training process, the other records the best model which is the one with the highest validation accuracy.

Training pipeline

```
begin
  while iteration < max_iteration do
      call training.run_one_batch()
      if timetovalidate
        call valid_accuracy = validation()
        if valid_accuracy > best_accuracy
          call save_model()
          best_accuracy = valid_accuracy fi
        if call should_early() == True
          break
```

The most straightforward way to implement the training pipeline with validation and early stopping is only build one model and switch the input dataset for training or validation. However it's not a good idea in practice. The disadvantages are the following:

1. There are different input methods for data in memory and data on hard disk. In FensorFlow you can not switch between this two methods for a single model. Because the input method is part of the Graph and can not be modified once the graph finish building.

2. If only one model is built for training and validation, the output and statistics of training and validation will be mixed together, since the monitor and output operator is also parts of the TensorFlow Graph, and can not be modified after the building phase.

So with different modules for different purposes in the graph, users can have separate input, output and monitor method for each module. When users want to train a model with validation, they can first build the training module, and then build a Evaluation module with validate dataset. And the two modules are set to share the same model parameters, to make sure the validation module are actually using learned model. When users need test result on test dataset, only a evaluation module will be build with test dataset, and the model will be initialized using saved model on hard disk.

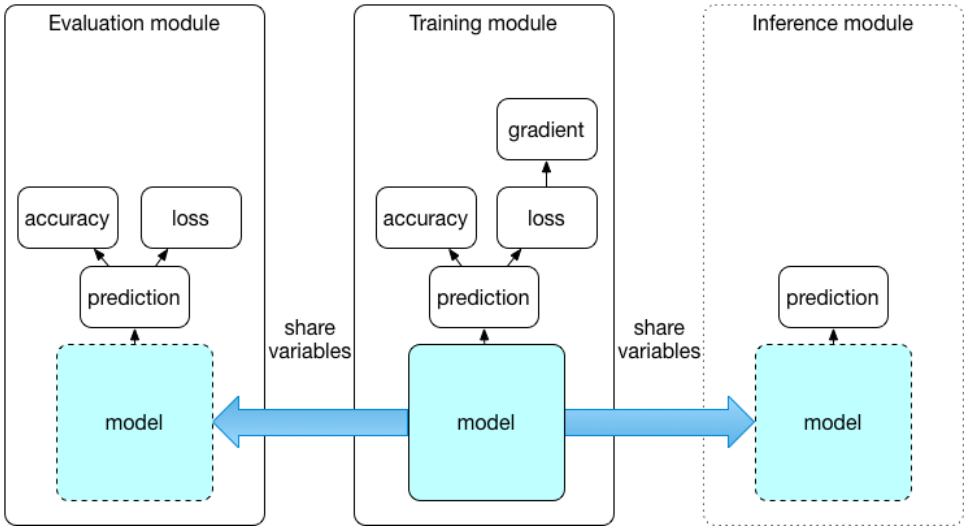

Figure 4.2: Detail inside modules

4.1.4 Implementation details

4.1.5 Input

In the network architecture discussed in Chapter 3, the input is a single training example. However in practice, the input should always be a batch of multiple examples. In this case, the batch contain 64 protein sequences. And each protein sequence are normalized to 700 hundred length(protein shorter than 700 hundred pad with zeros and protein longer than 700 are cut into segments). Figure 4.3 shows the distribution of protein length, the mean value is 208, most of the proteins are shorter than 300. So in each batch, there will be more than half of the data in it are zero paddings. And these zero paddings are actual affect the performance of the network, which will be shown in following sections. So in order to deal with zero paddings, besides the original input data, the length of each protein will also add to each batch. So each batch will have a shape of [64, 700, 42 + 1].

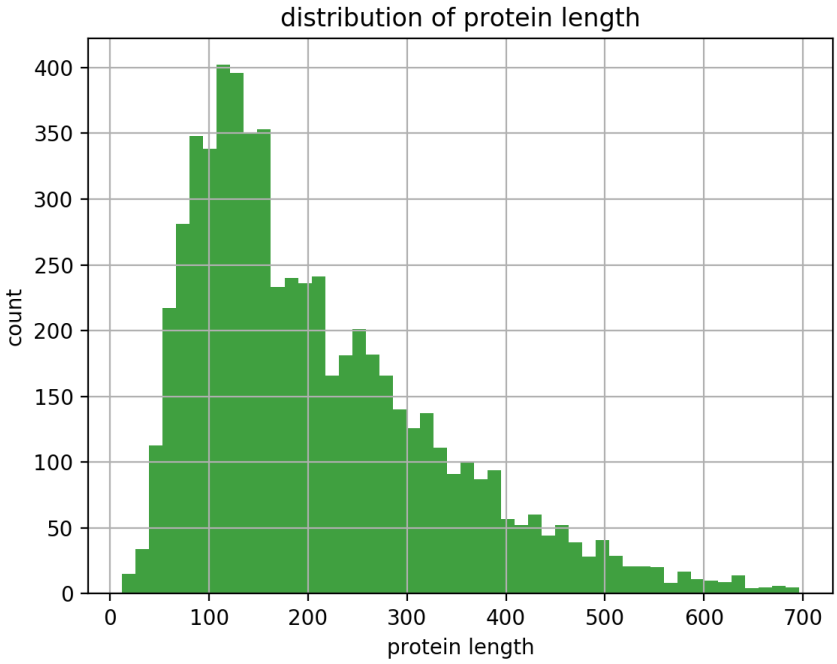

Figure 4.3: Distribution of protein length

There are two major data input method in general. Input data from memory and input data from disk. The most popular way you will found in TensorFlow tutorials or examples are input data from memory. That is read all the data into memory and prepare each batch by your self. This method is easy to use, and user have more control of the input process. However when you can not put all the data into memory, this method is inadequate. TensorFlow also provides a build-in method for reading data from file. Users can convert all data into TensorFlows default binary format, or provide reader functions. Then TensorFlow will help you handle the data input process, as long as user provides the list of input file names. As illustrate in Figure 4.4, a file name queue and a data queue are built as input buffer. For each step the model will read a batch from the data queue.

In this project, training module is using the file input pipeline. However evaluation module is using the light weight memory input method, since testing and validation data are usually have small size and only need to scan through the data only once for evaluation.

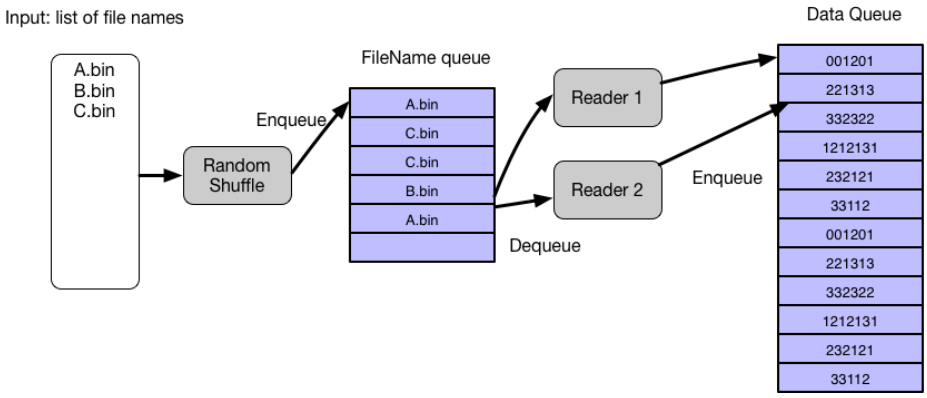

Figure 4.4: Input Queues

4.1.6 Multi-scale convolution

4.1.7 Recurrent layer

When implementing the recurrent layer, there are two things should be considered.

1. Use dynamic RNN or a static RNN.

2. When shoul RNN stop calculating for each batch.

In TensorFlow and most other deep learning libraries, there are two common RNN versions: static RNN and dynamic RNN. Dynamic RNN is pretty much what RNN should be like, it use a loop structure to process variable length of sequences. The static one, however, unrolls the loop and have a fixed number of computational nodes to process fixed length sequences, just as what Figure 2.2 shows.

The benefits of using static RNN is it runs faster than a dynamic RNN, since it doesn't have to test the terminate condition for each time step. The actual speed comparison is listed in Table 4.1.7. What's more it is the only RNN implementation in early version of TnesorFlow. The problem of static RNN is that it makes the model too big. A big model have a lot of disadvantages. It use too much memory, both GPU and CPU memory, it consume more disk space to store the best model, and it takes more time to build the model. In this project the 700 sequence length 3 layers with 64 units model will some how use 20G memory and over 4G GPU memory. Comparing to the 4MB model file of dynamic RNN, It took 275MB to store the static model described above, which will slow down the training process every time it same the current best model. And the build time is longer than 10 minutes comparing to the several seconds of dynamic RNN's build time.

The reason using a dynamic RNN is that it can adapt to different input and network architectures. Such as sequence to sequence model, which needs the encoder and decoder to be able to handle variable length sequences. When dealing with real

life data, a model that accepts variable length input are general easier to use. The dynamic RNN is light weight, flexible and can achieve comparable speed of a static RNN(In table 4.1.7 the dynamic rnn is about 8% slower than the static one).

Model			Performance		
Multi conv	RNN	Output layers	Build time (s)	speed (batch/s)	test accuracy(%)
32 channels	-	conv 11	0.69	5.28	65.3
64 channels	-	conv 11	0.72	3.44	65.7
64 channels	1 layer static 32	conv 11	166.61	0.77	-
64 channels	1 layer dynamic 32	conv 11	1.43	0.71	68.7
64 channels	1 layer dynamic 2	conv 11	1.43	0.76	67.8
64 channels	1 layer dynamic 10	conv 11	1.47	0.758	68.3
64 channels	1 layer dynamic 128	conv 11	1.41	0.54	69.2
64 channels	2 layer dynamic 128	conv 11	1.94	0.27	69.5

Table 4.1: Comparison of Static and Dynamic RNN

Another important detail about RNN is when to stop the iteration. Continuing the iteration after reach the end of the sequence is inefficient, and may cause error when doing back propagation, since the network will try to learn the zero paddings that are irrelevant to the protein sequence. The problem is that proteins in each batch will have different length, so in order to stop the iteration correctly, it is required to provide the length information for every protein in the batch.

In summary, RNN layers in this project are dynamic bidirectional RNN, and provided with sequence length information for each sequence in the batch. The unit number of the RNN cell is 128.

4.1.8 Output layers

There are two 1D convolution layer are build after RNN layers. The first one use kernel shape [1] and output channel size is half of the input channel size. The second layer use kernel shape [7] and the output channel is 14.

4.1.9 loss function

Different from convolution networks, RNN need several additional steps to calculate the correct average loss value for each batch. Because RNN always deals with variable length data, such as English sentences and ,in this project, proteins. As illustrate in Figure 4.5 the output of the network would be a 3D array with shape [batch_size, sequence_length, feature_length]. As you may notice in the figure, there are blanks at the end of each sequence, those are zero paddings. Due to the variable length input data, the blanks are always exist. And consider the distribution in Figure 4.3, over half of the data in a batch are zeros. The correct loss function for individual sequence is Equation 3.1a. However in practice, due to the parallel feature of TensorFlow, it is required to calculate the loss for all the sequence in a batch at the same time. The result of the batch loss operation is a 2D array with shape [batch, sequence]. Each element in this array is the loss of a single amino acid. After calculating the loss array, a mask array is constructed using the length vector on the left of Figure 4.5. Apply the mask array to the loss matrix will set the loss of the blanks to zero. Finally the average loss per amino acid will be $sum(LOSS\_ARRAY)/sum(length\_vector)$.

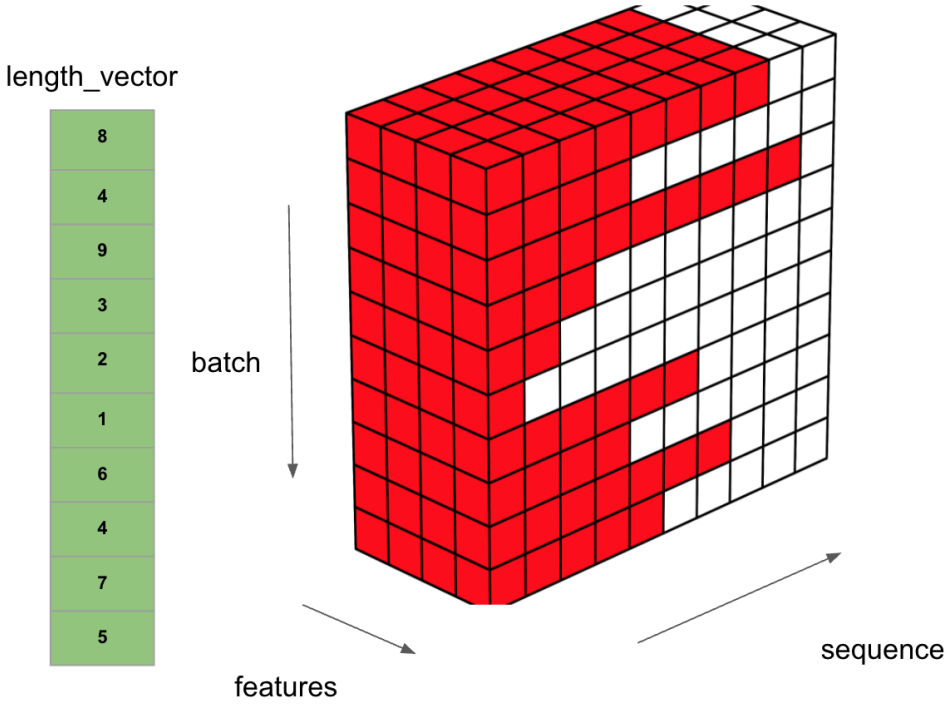

Figure 4.5: Variable length batch Input

4.1.10 Accuracy operator

The evaluation of the project is 8 classes accuracy(Q8). The algorithm of calculating Q8 accuracy from predictions and labels is the following.

Data:

Prediction [batch_size, seq_len, feature_len]

label [batch_size, seq_len, feature_len]

length_vector [batch_size, 1]

Result: Q_8: real number

conf_mat = Confusion_Matrix(label, prediction);

true_positive = sum(diag(conf_mat));

Q_8 = true_positive/sum(length_vector);

Algorithm 1: Q8 accuracy

The implementation of this part is some how tricky. Because when users train the network, they want to monitor the training and validation accuracy in TensorBoard and TensorBoard only monitors the operator inside the Graph. So whatever calculate the accuracy should be an operator inside the TensorFlow Graph. And it should not be a python function outside the Graph. So the correct implementation here is constructing a accuracy operator using basic TensorFlow operators, and add it to the model. Then the calculation will perform inside the Graph and inside GPU. For more details see the code in Appendix.

4.1.11 Training Monitor

The actual monitor of the system has two separate parts. The first part is the summary operators inside the TensorFlow Graph which save the value of desired tensors in log files periodically. The tensors(variables) that are monitored in this projects are the following:

1. training module
    - training loss
    - training accuracy
2. validation module
    - validation loss
    - validation accuracy
3. Input module
    - input queue occupancy
    - running speed (batch/sec)
4. Distributions and histogram
    - output of Multi CNN
    - output of RNN layers

The second part is a visualization tool called TensorBoard. It a command line tool come with the TensorFlow library. What it dose is reading the log file from the path you provide, and start to host a web page(Figure 4.6) which have all the graphs on it. The web page will refresh periodically, so users can use the web page as a

real time monitor of the training process. The advantage of using TensorBoard is the flexibility. Users can choose when they want the visualization, one can easily turn off the monitor without interrupting the training. And users can also choose where they wan to put the visualization, a model may be trained on a server, and the use can monitor the training by visiting the web page on other computers.

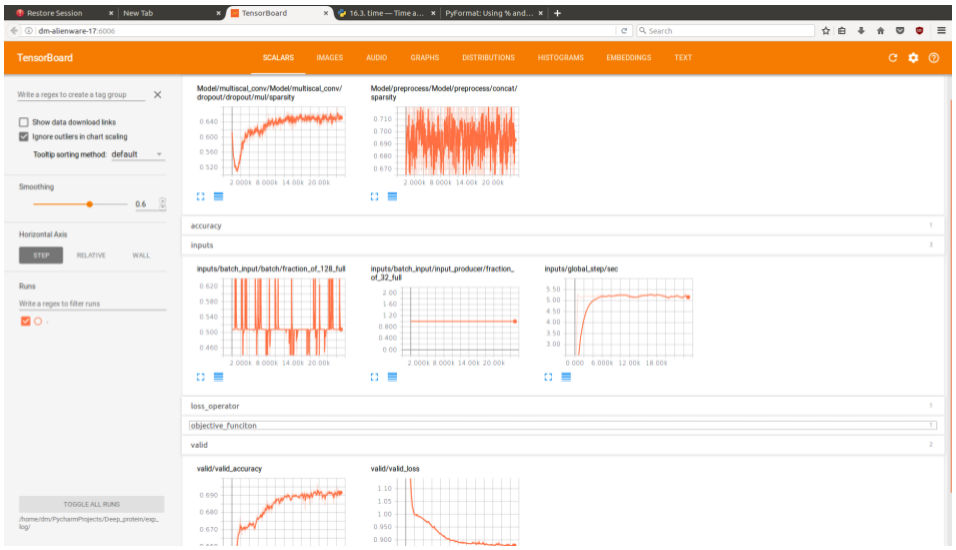

Figure 4.6: Tensorboard visualization

4.1.12 Save and Restore

The system saves checkpoint files periodically, since the training process usually take hours to complete, it's important to only keep the newest N checkpoints files. In this project, when training the system will check if there are more than 5 checkpoint files, if true then delete the oldest one. When the system restore training, it will search for the newest checkpoint file and start the training from there. Another file need to save is the best model file, it similar to a checkpoint file, but it only save the trainable parameters instead of all the variables inside the model that are needed to restore training.

Chapter 5

EXPERIMENTS AND RESULTS

5.1 Data

In this project, two public available datasets are used, CB6133 produced with PISCES CullPDB[18] and CB513. To better evaluate the performance of the network, a smaller filtered version of CB6133 is formed by removing redundant sequences in CB6133 that have over 25% similarity with some sequence in CB513. 80% of the filtered CB6133 dataset are random selected as training set and the rest are the validation set. CB513 here is the standard test set. Both CB513 and filtered CB6133 can be found at http://www.princeton.edu/ jzthree/datasets/ICML2014/.

5.2 Experiments

In the following experiments, all network structures are trained using the pipeline described in Chapter 4. The system calculate the validation performance every 50 iterations(batch). The early stop strategy is based on the valid accuracy, if the system doesn't get better accuracy for 30 times of validation it stops training. That

approximates to a tolerance of 23 epochs. And the best model get from training will be evaluated using CB513. All the experiment are run on an Alienware desktop with Intel core i7-4700MQ CPG @ 2.4GHz8, 23.5 GB memory and Geforce GTX 780M GPU.

5.2.1 Experiment on loss function

To demonstrate the effectiveness of the modified loss function described in chapter 4, comparison is made between the test result of several different architectures with regular loss and modified loss function. All the architectures been tested and the results are listed in Table 5.2.1. The first four columns are models without RNN layer, with 1, 2 and 3 RNN layers respectively. They are trained with regular loss function. The other fours are the same models which been trained with modified loss functions. As you can see in Figure 5.1, all the model with modified loss function(red bars) achieved higher test accuracy than models with regular loss function(blue bar). Which means the modified loss function performs better when the input data has variable length. So in other experiments, only modified loss function are used.

Model				Performance
Multi conv	RNN	Output layers	loss	test accuracy(%)
64 channels	-	fc	regular	63.1
64 channels	1 layer static 50	fc	regular	63.1
64 channels	2 layer static 50	fc	regular	65.7
64 channels	3 layer static 50	fc	regular	66.2
64 channels	-	fc	modified	63.5
64 channels	1 layer static 50	fc	modified	68.0
64 channels	2 layer static 50	fc	modified	68.0
64 channels	3 layer static 50	fc	modified	67.8

Table 5.1: result of different loss functions

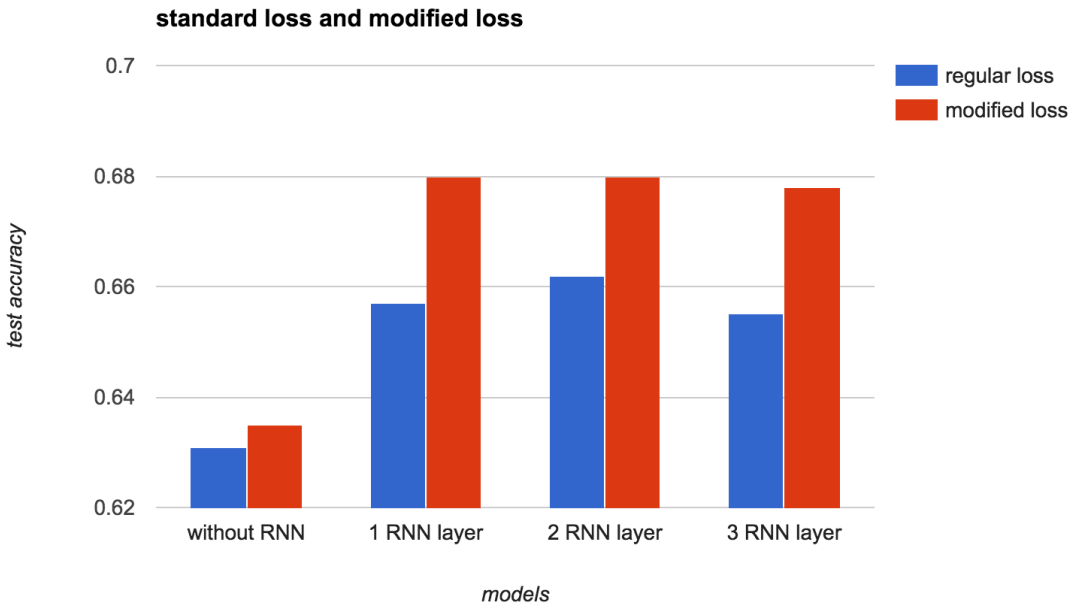

Figure 5.1: result of different loss functions

5.2.2 Experiment of different output layers

This experiment compares the performance of models using fully connected layers and convolutional layers at the end of the network, as described in section 3.2.5. As listed in Table 5.2.2, the first three models use fully connected layers as output layers. The other three are the same model replacing the fully connected layers with convolutional layers. The first kernel size is 3 and the second kernel size is 11. As illustrate in Figure 5.2 the convolutional layers improve the performance of all three models by at least 1 percent. The difference between fully connected a layer and a convolutional layer here is the kernel size. The fully connected layer in this project is equivalent to a convolutional layer with kernel size 1. So this means that when the network making final prediction of secondary structures, considering more context information is helpful. In addition, this kind of context information can not be fully captured by the previous RNN layers.

Model			Performance
Multi conv	RNN	Output layers	test accuracy(%)
64 channels	-	fc	63.1
64 channels	1 layer dynamic 128	fc	68.13
64 channels	2 layer dynamic 128	fc	68.02
64 channels	-	conv	65.4
64 channels	1 layer dynamic 128	conv	69.2
64 channels	2 layer dynamic 128	conv	69.5

Table 5.2: result of different output layers

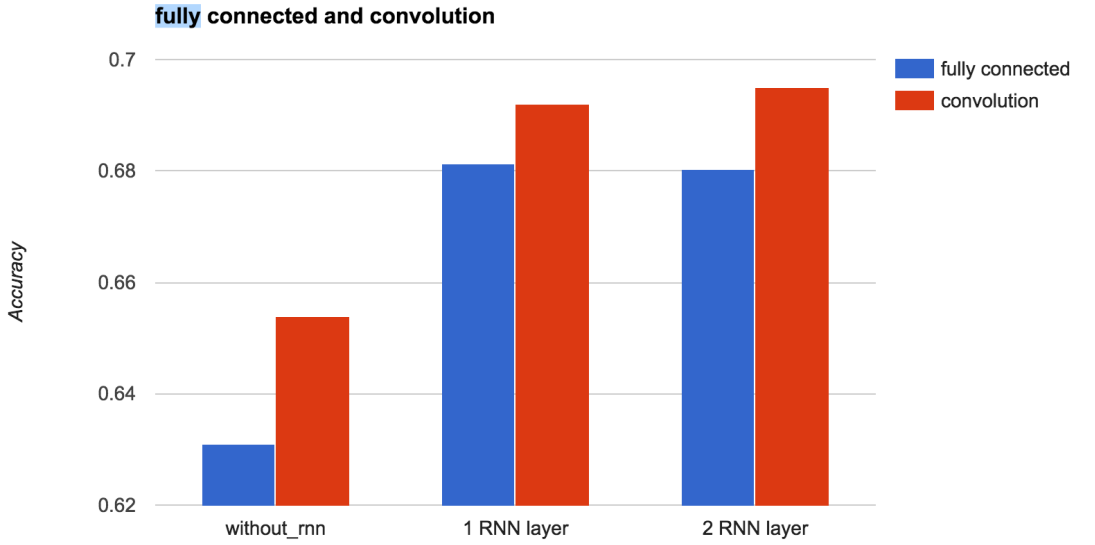

Figure 5.2: result of different output layers

5.2.3 Experiment of different hidden unit in RNN layer

This experiment focuses on how does the hidden unit number in RNN layer affect the performance of the model. Only 1 layer bidirectional RNN are used here because a series of model need to be tested and using more than 1 layer of RNN the experiment would take too long. As shown in table 5.2.3, a model without RNN layer are the baseline, then four models with 2, 10, 32 and 128 hidden units RNN layer have been evaluated. Figure 5.3 is the line chart of accuracy vs hidden unit number. As you can see, with a very small number of 2 hidden units 1 RNN layer can improve the accuracy by 2%. From 2 to 30 units it improve the accuracy by 0.8%, and from 30 to 128 the improvement is only 0.5%.

Model			Performance
Multi conv	RNN	Output layers	test accuracy(%)
64 channels	-	conv	65.4
64 channels	1 layer dynamic 2	conv	67.9
64 channels	1 layer dynamic 10	conv	68.2
64 channels	1 layer dynamic 32	conv	68.7
64 channels	1 layer dynamic 64	conv	68.8
64 channels	1 layer dynamic 128	conv	69.2

Table 5.3: result of different hidden unit numbers

To further demonstrate the performance on individual secondary structures, a precision vs number of unit graph is ploted in Figure 5.4. The y axis is the precision for each class of secondary structure, the x axis is the number of units in RNN layer, zero represent without RNN layer. The number after the label in legend is the frequencies of these secondary structures appear in protein structures. As shown, basically the higher the frequency is, the higher the precision is. The model without the RNN layer can achieve comparable precision on the high frequency secondary structures. However it fails to predict class B, G and I. The RNN layer, even with very few hidden units, significantly improves the performance on low frequency classes. While the number of hidden units increase, the model tend to have higher precision on those rare classes. However the model still fail to predict the class I which is super rare in training dataset.

5.2.4 Final result

The best model from all these experiments in previous sections is the model with 2 layers of 128 hidden units bidirectional RNN with convolutional output layer. The test accuracy on CB513 is 69.5%. To train this model, it took about 20 hours on the

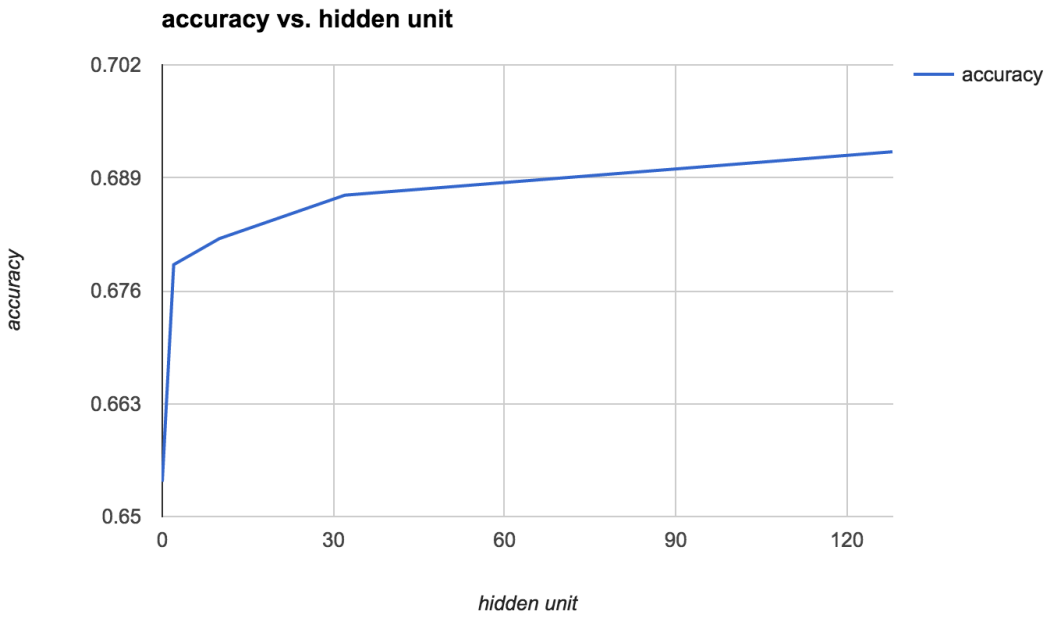

Figure 5.3: result of different hidden unit numbers

computer described at the beginning of this Chapter.

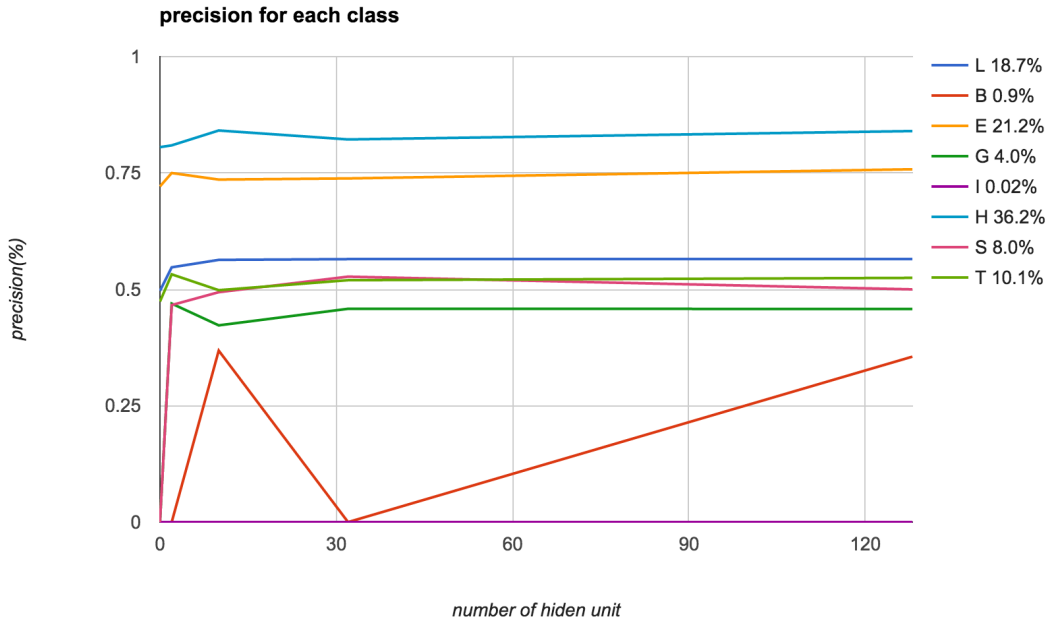

Figure 5.4: Detail result of different hidden unit numbers

Chapter 6

SUMMARY

The major achievements of this work is the following:

1. Design and implement a DNN learning system in TensorFlow for PROTEIN SECONDARY STRUCTURE PREDICTION.

2. Provide detailed information on how to use RNN correctly.

3. Test and explore the trade off between speed and accuracy of the RNN.

4. Achieve 69.5% accuracy on CB513, which is comparable to current state-of-the-art.

This work, following the basic network architecture in [3], use bidirectional GRU RNN after multiscale convolutional layers. Instead of using the exact architecture in their work which has 3 RNN layer and 600 hidden units in each layer. Only 2 128-hidden-unit RNN layers have been used, due to the insufficient GPU memory. However, by using less layers and less hidden units and add batch normalization to the network. The network managed to achieve comparable performance on dataset CB513. The best Q8 accuracy is 69.5% which is 0.2% lower than theirs. However, because of the smaller model, it need less time to train the model. This thesis also

shown how to design and build a system to train a recurrent network with variable length inputs.

The recurrent structure can capture the global context information in the protein sequence and boost the performance of protein secondary structure prediction. However it can not fully capture the context information, simply stack the RNN layer can not out perform a single RNN layer with convolutional layers after it. A possible reason is the current off-the-shelf RNN structures can not deal with extremely long dependencies such as the average 230 length proteins. But there are huge potential in RNN networks. More powerful but complex RNN architectures such as RNN with attention mechanism[19] and RNN with batch normalization[20] may be able to solve this problem.

Appendix A

Title of first appendix

A.1 Source Codes

Source code can be found on Github: https://github.com/duming/Deep_protein.

Bibliography

[1] Ashraf Yaseen and Yaohang Li. Context-based features enhance protein secondary structure prediction accuracy. *Journal of chemical information and modeling*, 54(3):992–1002, 2014.

[2] Jian Zhou and Olga Troyanskaya. Deep supervised and convolutional generative stochastic network for protein secondary structure prediction. In *International Conference on Machine Learning*, pages 745–753, 2014.

[3] Zhen Li and Yizhou Yu. Protein secondary structure prediction using cascaded convolutional and recurrent neural networks. *arXiv preprint arXiv:1604.07176*, 2016.

[4] Akosua Busia, Jasmine Collins, and Navdeep Jaitly. Protein secondary structure prediction using deep multi-scale convolutional neural networks and next-step conditioning. *arXiv preprint arXiv:1611.01503*, 2016.

[5] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[7] Jeffrey Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2625–2634, 2015.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[9] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[11] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[12] Zhiyong Wang, Feng Zhao, Jian Peng, and Jinbo Xu. Protein 8-class secondary structure prediction using conditional neural fields. *Proteomics*, 11(19):3786–3792, 2011.

[13] Søren Kaae Sønderby and Ole Winther. Protein secondary structure prediction with long short term memory networks. *arXiv preprint arXiv:1412.7828*, 2014.

[14] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast:

a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[15] Wojciech Zaremba. An empirical exploration of recurrent network architectures. 2015.

[16] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[18] Guoli Wang and Roland L Dunbrack. Pisces: a protein sequence culling server. *Bioinformatics*, 19(12):1589–1591, 2003.

[19] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[20] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.