PROTEIN AND SECONDARY STRUCTURE PREDICTION WITH CONVOLUTIONS AND VERTICAL-BI-DIRECTIONAL RNNS

Alexander Rosenberg Johansen [a], *Søren Kaae Sønderby* [b], *Ole Winther* [a,b]

[a] Department for Applied Mathematics and Computer Science, Technical University of Denmark (DTU), 2800 Lyngby, Denmark
[b] Bioinformatics Centre, Department of Biology, University of Copenhagen, Copenhagen, Denmark

ABSTRACT

We trained a bi-directional recurrent neural network (RNN) with memory cells (long-short term memory) with convolutional filters and hidden states from previous long-short term memory passes as input. With purpose of mapping amino acid and sequence profiles from the CB513 data set onto secondary protein structures. On the test data we achieved a Q8-accuracy of 68.7% for single model performance and 70.2% with an ensemble, which is a significant improvement on state-of-the-art on 68.3%. The neural network, which has 2.1 million parameters, consists of three convolutional layers with batch normalization, followed by a fully connected layer with batch normalization, bi-directional recurrent layer with long-short term memory cells where the hidden state from the forward pass is used as input in the backwards pass, then feed into a fully connected layer before the final 8-way softmax layer. The convolutional and dense layers uses non-saturating neurons. The network is regularized by dropout and the L_2 norm. Further, gradients where normalised and output prediction clipped to stabilize training.

Index Terms— computational biology, deep learning, recurrent neural network, convolutional neural network, batch normalization, GPU

1. INTRODUCTION

Current approaches to annotating secondary protein structures from amino acid profiles and sequence profiles makes uses of machine learning methods [1]. Recently deep learning methods for modelling secondary protein structures has emerged and received state of the art performance [2, 3].

The approaches has risen as the challenge of annotating secondary protein structures is often predictable from structures along the sequence of the amino acid and sequence profiles **(Ole cite)**. A recurrent neural network with memory cells such as the long-short term memory (LSTM) cell can inform the model about earlier data in a sequence, thus utilizing the sequential structure[4]. Further the local sequence typology has a high importance for predicting secondary protein structures **(Ole cite)**. A Convolutional Neural Network (CNN) is an approach to parametrize filters for detecting defined spaces of data [5], thus convolutions propose a method for utilizing the local structures along the sequence of amino acid and sequence profiles.

As the problem of predicting secondary structures given a sequence of amino acid and sequence profiles is not direction or time dependant, since the entire sequence is given at one time, bi-directional RNNs can be utilized to analyse the sequence from both directions[6].

The contribution of this paper is an extension to the deep learning model proposed by Sønderby et al., 2014 [3] by combining the convolutions ability to model local typology in the input while allowing a bi-directional recurrent neural network to model long term depedancies. We further test using the hidden state of the forward pass as input for the backwards pass, batch normalization and L^2 regularization. [1]

2. MATERIALS AND METHODS

2.1. The Dataset

The test set which we performed predictions on is the CB513+profile test set on secondary protein structures. To train our classifier we used the filtered version of cullpdb+profile data set with no further preprocessing applied [1] [2]. For every point in the sequence there is a recording of the amino acid and sequence profiles coded in one-hot encoding. The purpose of the data set is to predict the secondary structure of every sequence step. For validation purposes we extracted 256 sequences at random in the training set to optimize hyper parameters in our model.

2.2. The Model

The architecture of the network is defined by three different types of learn-able layers, the fully connected, the convolutional layer and the recurrent layer with long-short term mem-

[1]Codebase available at: github.com/alrojo/cb513
[2]http://www.princeton.edu/ jzthree/datasets/ICML2014/

ory (LSTM) cells. Below, we describe how the different layer types used in our network in more detail.

2.2.1. *Neural Network*

The fully connected layer, also known as a dense layer or a multi layer perceptron (MLP) [7] is a layer performing a non-linear transformations on the previous layer. The first layer, $l = 0$, is considered the input. In our case a one-hot encoding of the amino acid and sequence profiles. The standard MLP is defined in the following linear algebraic operation,

$$\begin{aligned} z_{\ell+1} &= h_l \theta_{\ell+1} + b_{\ell+1}, \\ h_{\ell+1} &= a(z_{\ell+1}) \end{aligned}$$

where h_l is the current layer and θ is the weight matrix and b the bias used to compute the linear combination of the input; $z_{\ell+1}$. A non-linear activation function, $a(z_{\ell+1})$ is applied to the linear combination of the input which results in the next layer; $h_{\ell+1}$.

Most commonly used functions for the activation function $a(z)$ includes the Logistic Sigmoid $a(z) = 1/(1 + e^{-z})$ and the Hyperbolic Tangent $a(z) = (e^z - e^{-z})/(e^z + e^{-z})$. Non-saturating functions such as the Rectifier Linear Unit (ReLU) $a(z) = max(0, z)$ has become popular as their gradients do not vanish and it is computationally easier, which makes optimizing the network faster using stochastic gradient descent [5]. Unfortunately the ReLU suffers from "dead" gradients (large gradients could cause the weights to update in such a way that the neuron will never activate), which can make the network unable to learn. To avoid this problem we are using the Leaky ReLU $a(z) = max(\alpha z, z)$ with $\alpha = 0.01$[8], where "Leaky" refers to the added slope when $a(z) < 0$.

As regularization technique, to avoid overfitting, we add Bernoulli dropouts[9] as a layer, it works as defined:

$$h_{\ell+1} = h_l \odot p$$

where $p_i \in [0, 1]$ is a random sampled number with a hyper parameter determining bias towards 0 or 1 and $\odot$ is element-wise multiplication.

As another regularization technique, to improve convergence by reducing internal co-variate shifts and to be less volatile to poor initializations we apply batch normalization [10]. Batch normalization is defined as follows

$$\begin{aligned} \hat{\ell}^{(i)} &= \frac{l^{(i)} - E[l^{(i)}]}{\sqrt{\text{Var}[l^{(i)}]}}, \\ u^{(i)} &= \gamma^{(i)} \hat{\ell}^{(i)} + \beta^{(i)} \end{aligned}$$

where l is the distributed vector representation from layer used as input to the batch normalization, such that $l = (l^{(1)}, l^{(2)}, \ldots, l^{(d)})$. When using batch normalization with nonlinear transformations we use the batch normalization layer after the linear transformation and before the nonlinear transformation. Such that the fully connected would look as follows

$$\begin{aligned} z_{\ell+1} &= h_l \theta_{\ell+1}, \\ u_{\ell+1} &= \text{batch-norm}(z_{\ell+1}), \\ h_{\ell+1} &= a(u_{\ell+1}) \end{aligned}$$

notice the bias, $b_{\ell+1}$ removal in the linear combination. This is due of the shifting, β, applied in the batch normalization.

2.2.2. *Convolutional Neural Network*

The 1D convolutional layers are implemented as follows:

$$\begin{aligned} z^{id}_{\ell+1} &= x^i_\ell \theta^d_{\ell+1} + \beta^d_{\ell+1}, \\ h^{id}_{\ell+1} &= a(z^{id}_{\ell+1}) \end{aligned}$$

Where the input $x^i_\ell \in R^{kc \times 1}$ represents a co-located vector of length k in c channels and i refers to the specific co-located vector. The weight matrix $\theta^d_\ell \in R^{1 \times kc}$ corresponds to a spatial weight filter with kc connections, in the d^{th} output channel, between the input layer and each neuron in the output layer and β^d_ℓ is a scalar. As a result, convolutional layers are configurable to their filter size k and number of filters d [5], leading to many possible configurations of the CNN architecture. It has been found that multiple convolutional layers on top of one another allows to reduce parameter space and model nonlinear filters[11]. Also using different filter sizes on the same input can increase performance[12]. In our solution we found inputs of different sizes to have the superior validation Q8-accuracy.

2.2.3. *bi-directional Long-Short Term Memory with Vertical Links*

A recurrent neural network (RNN) is a type of neural network layer that works along the sequence of the data and utilizes computation on previous input in the sequence[4]. The LSTM is a type of RNN that uses memory cells to better model long-term dependencies and improve convergence speed. It is defined as described in Graves, 2012[4] without peepholes.

$$\begin{aligned} i_t &= \sigma(x_t W_{xi} + h_{t-1} W_{hi} + b_i), \\ f_t &= \sigma(x_t W_{xf} + h_{t-1} W_{hf} + b_f), \\ o_t &= \sigma(x_t W_{xo} + h_{t-1} W_{ho} + b_o), \\ g_t &= tanh(x_t W_{xg} + h_{t-1} W_{hg} + b_g), \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t, \\ h_t &= o_t \odot tanh(c_t) \end{aligned}$$

Where $\sigma(z)$ is the sigmoid function $1/(1 + e^{-z})$ and x_t is the input, h^{l-1}_t is the previous layer's hidden state and c^{l-1}_t is the previous layer's hidden cell. The LSTM only works in one direction. To utilize information from both directions two LSTM's working from each direction of the sequence are applied. Their hidden states, h_t for $t = [1, 2, ..., T]$ where T is sequence length, are concatenated such as suggested by Schuster & Paliwal, 1997 [6], such that

$$h_t = \begin{bmatrix} \overrightarrow{h_t} \\ \overleftarrow{h_t} \end{bmatrix}$$

where $\overrightarrow{h_t}$ is the forward pass and $\overleftarrow{h}$ the backwards pass.

Further to incorporate sharing of hidden states, we propose the "Vertical Links" which is, illustrated in figure 1, stacking the input, x_t, to the forwards pass hidden state, $\overrightarrow{h_t}$, of the previous pass. This operation allows the backwards pass to have a hidden state from both directions while computing. The concept could be extended to stacking several passes on top of one another with "Vertical Links", however, we leave this for further research.

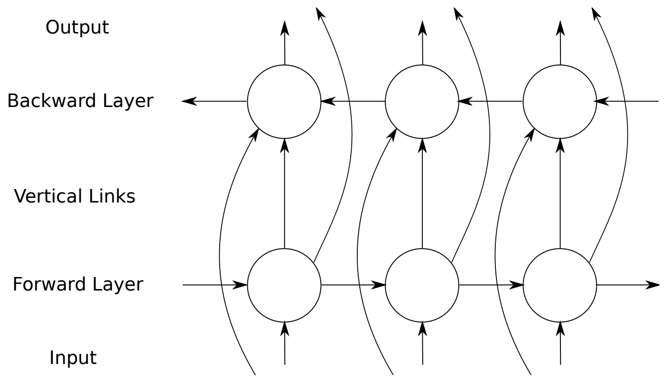

Fig. 1: bi-directional RNN with Vertical Links

2.2.4. *Overall architecture*

As depicted in figure 2 the network contains nine layers. The first three are convolutional layers with different filter sizes on the input and using batch normalization, these are then concatenated with the input and feed to a fully connected network with batch normalization representing the fourth layer. The fifth is a RNN with LSTM cells working in the forward direction, which has its hidden states concatenated with the fourth layer. The four and fifth layer is then feed into the sixth layer also being a RNN with LSTM cells working in the backwards direction. The RNNs(fifths and sixth layers) hidden states are concatenated and feed to a seventh layer being a dropout layer and then into the eight layer, a dense layer. The ninth layer is an 8-way softmax function (to normalize the output) which provides us with a probability of the secondary protein structures in a sequence.

The convolutional layers all have 16 filters with receptive field sizes of 3, 5 and 7. Both of the fully connected layers have 200 hidden units. Both of the recurrent LSTM layers have 400 hidden units.

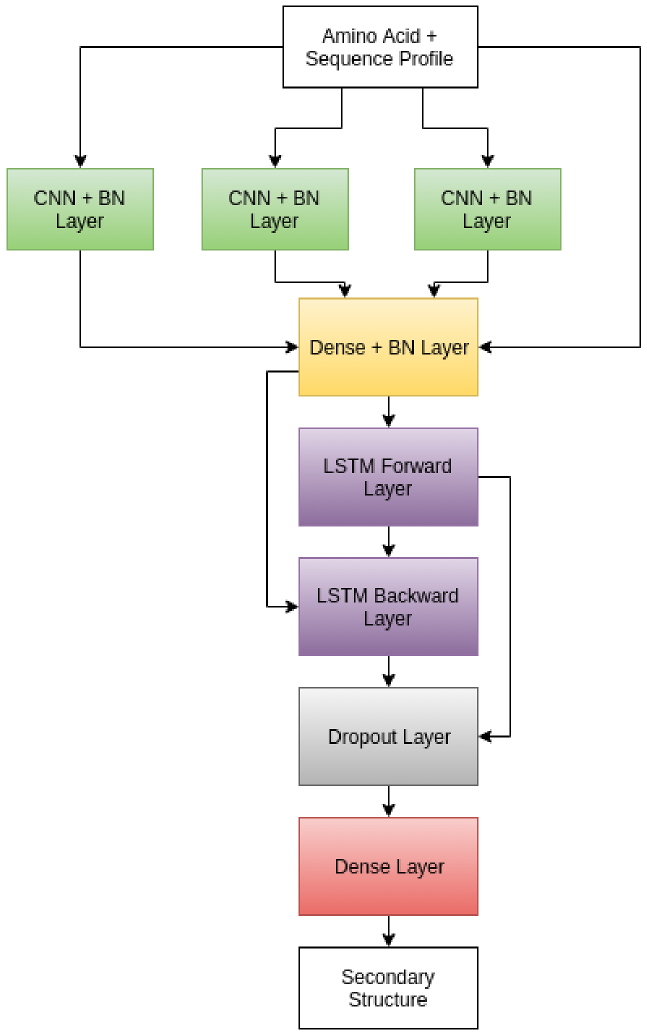

Fig. 2: Neural network model

To optimize, also known as training, our network we minimize the multi class cross-entropy objective on the basis of the probability output from the neural network:

$$L(x, y) = -\sum_c y_c \ln(f_c(x))$$

Where x is the input vector, y the true label, $f(x)$ the neural networks probability prediction and c the class. The probability prediction is clipped to be between τ and $1 - \tau$ to stabilize training and avoid exploding gradients.

We further apply the L_2 norm such that.

$$regterm(\lambda, \theta) = \lambda\left(\sum_{n=1}^{N} \theta_n^2\right)$$

Where λ is a tuneable hyperparameter, N is the number of non-bias weights and θ is the weights in the neural network. This is then added to the cost function.

$$L_{reg}(x, y) = L(y, f(x)) + regterm(\lambda, \theta)$$

2.2.5. *Details of learning*

We trained our model with the first order method: stochastic gradient descent (SGD). SGD works by utilizing chain ruling to take the partial derivative of the loss function with respect to each weight vector in the network. Such that

$$g = \frac{\partial L(x,y)}{\partial \theta} = \frac{\partial L(x,y)}{\partial f(x)} \frac{\partial f(x)}{\partial \theta},$$

where the partial derivative for each weight, g, is used to update the weights. Such that

$$\theta_{b+1} = \theta_b - \alpha g_b,$$

where α is a tunable parameter, known as the learning rate, determining the size of gradients updated with and b is the current training batch. We use a version of SGD called RMSProp[13], which uses historic information to adapt the learning rate for every parameter while training. Such that

$$\theta_{b+1} = \theta_b - \alpha G_b^{-1/2} \odot g_b,$$

where G is a moving average of squared gradients. Such that,

$$G_{b+1} = \rho G_b + (1-\rho) g_b^2,$$

where ρ is the gradient moving average decay factor.

To avoid exploiting gradients we normalized the gradient, g, if the norm exceeds a threshold of . Given such case, all of the gradients would be scaled by

$$norm_2 = \| \frac{g}{batch\_size} \|_2.$$

In our model we $\tau = 1e - 5$ for prediction clipping, $\lambda = 0.0001$ for regularization, $\alpha = 0.005$ and $\rho = 0.9$ for RMSProp and $= 20$ for norm thresholding our gradients.

We trained the network until we minimized the validation error.

The training was performed on a Nvidia GeForce GTX Titan X GPU, we used the python built Theano library[14][15] to compile to CUDA (a GPU interpretable language). On top of Theano we applied the neural network specific Lasagne library to built and configure our models[16].

The models took on average 24 hours to train on the Nvidia GPU.

3. RESULTS AND APPLICATIONS

Our results on the CB513 dataset is summerized in table 1. Both our single model performance of **0.687** ($p = 0.0763$) and model ensemble of **0.702** ($p < 0.0001$) outperforms best known previous results, 0.683, from Wang et al., 2016 [2]. Where the p value was determined by using a Chi square with Yates' correction. Our model ensemble was made from six different initializations of the same model with the three best performing epochs chosen by evaluated on validation set. For our baseline we recreated the Bi-Directional LSTM as used by Sønderby et al., where we managed to reproduce their results with 0.674 Q8-accuracy.

Models	**Q8-accuracy**
Zhou & Troyanska, 2014 [1]	0.664
Sønderby et al. 2014 [3]	0.674
Wang et al. 2016 [2] (current SOTA)	0.683
Bi-LSTM (reproduction of Sønderby)	0.674
Our model	**0.687**
Our model ensemble	**0.702**

Table 1: Benchmark experiment amongst various neural networks models.

4. DISCUSSION

Our results show that convolutional filters, bi-directional recurrent neural networks with long-short term memory cells, vertical links and L^2 regularization are all able to improve state-of-the-art performance with 1.9% ($p < 0.0001$). During the training we further found that vertical links with no convolutional filters did not have any significant improvement when using L^2 regularization. However, when convolutional filters where applied L^2 regularization became critical, as the network otherwise would overfit. So adequate testing of hyperparameters is essential for constructing an optimal fit between convolutional filters, vertical links and L^2 regularization. Additionally we found that batch normalization[10] made our model converge four times faster and increased the validation performance of all models.

5. REFERENCES

[1] Jian Zhou and Olga G Troyanskaya, "Deep supervised and convolutional generative stochastic network for protein secondary structure prediction," *arXiv preprint arXiv:1403.1347*, 2014.

[2] Sheng Wang, Jian Peng, Jianzhu Ma, and Jinbo Xu, "Protein secondary structure prediction using deep convolutional neural fields," *Scientific reports*, vol. 6, 2016.

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

[4] A. Graves, "Supervised sequence labelling with recurrent neural networks," *Springer*, 2012.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, "Imagenet classification with deep convolutional

neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, Eds., pp. 1097–1105. Curran Associates, Inc., 2012.

[6] Schuster & Paliwal, "Bidirectional recurrent neural networks," *Signal Processing, 45*, 1997.

[7] Dennis W Ruck, Steven K Rogers, Matthew Kabrisky, Mark E Oxley, and Bruce W Suter, "The multilayer perceptron as an approximation to a bayes optimal discriminant function," *Neural Networks, IEEE Transactions on*, vol. 1, no. 4, pp. 296–298, 1990.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015.

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[10] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[12] Søren Kaae Sønderby, Casper Kaae Sønderby, Henrik Nielsen, and Ole Winther, "Convolutional lstm networks for subcellular localization of proteins," *arXiv preprint arXiv:1503.01919*, 2015.

[13] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[14] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio, "Theano: a CPU and GPU math expression compiler," in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010, Oral Presentation.

[15] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio, "Theano: new features and speed improvements," Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[16] Sander Dieleman, Jan Schlter, Colin Raffel, Eben Olson, Sren Kaae Snderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, diogo149, Brian McFee, Hendrik Weideman, takacsg84, peterderivaz, Jon, instagibbs, Dr. Kashif Rasul, CongLiu, Britefury, and Jonas Degrave, "Lasagne: First release.," Aug. 2015.