Bioinformatics Coursework

Predicting the sub-cellular location of eukaryotic proteins

Victor Schmidt - 16095279

MSc. Machine Learning, University College London

Abstract

Inferring a protein's function based on its amino-acid structure is a critical task of biology to understand genomes and their expressions. In this paper we focus on eukaryotic cells' proteins, inferring their sub-cellular location based on their amino-acid sequence. We restrict these locations to 4 major location : Cytosolic, Secreted, Nuclear and Mitochondrial. Using a Recurrent Neural Network with GRU cells, we achieve a cross-validated accuracy of 73%. Moreover, analyzing the network's embedding matrix we find that our feature-free approach still finds good correlations in chemical properties between amino-acids.

1 Introduction

Knowing the sub-cellular location of a protein (i.e. where in the cell it is found) may give some clue as to its possible function. Since organelles are the location of specialized functions in the cell, such as oxidative metabolism in mitochondria, transcription of ribosomal RNA in nucleoli, and maturation of newly synthesized proteins in the endoplasmic reticulum, the determination of sub-cellular location for a protein can yield hypotheses about the metabolism in which it is involved and the proteins with which it interacts, making an automated method that assigns proteins to a certain sub-cellular location a useful tool for analysis(1).

Especially in the situation of a new gene sequence analysis when very little is know, as is the case for our task. For eukaryotes, it is reasonable to define 4 major sub-cellular locations:

- Cytosolic - i.e. within the cell itself, but not inside any organelles
- Secreted - proteins which are transported out of a cell
- Nuclear - proteins found/used within the cell's nucleus
- Mitochondrial - proteins transported to the cell's mitochondria

Proteins are composed of amino acids chained together, in sequences of very varying lengths, from a few hundred to several thousand. These amino acids have various properties (2), they interact both physically and chemically making the specificity of each protein and one could hope to use algorithms to classify proteins based in these properties. Various techniques have been tried with 3 's PSORT (1991), Sarda *et al*. 's SVMs (2004) or Monney *et al*. 's N to 1 Neural Networks (2011).

In this paper we will use Recurrent Neural Networks (RNNs) to take advantage of the proteins' sequence structure while not needing to rely on hand picked features.

2 Approach

2.1 Data

The data in this task is made of 4 *fasta* files, each containing sequences for a specific class. A typical training example is made of 2 elements :

- A *fasta* header describing the protein
- The amino-acid sequence of the given protein using the one-letter code (4)

It is important to notice that every sequence in these sets can be assumed to be unique (non-homologous), therefore allowing us to split it in any way without losing the *i.i.d* assumption of the training data (5).

As RNNs rely on the sequence structure of the data, it is interesting to look at the distribution of lengths across classes :

Table 1. Proportion of sequences longer than 500, 1000 and 200 per class and proportion of each class across the training dataset.

	Cytosolic	Secreted	Mitochondrial	Nuclear
Sequence >500	52.53 %	16.82 %	23.09 %	49.70 %
Sequence >1000	17.68 %	4.55 %	1.93 %	14.67 %
Sequence >2000	2.66 %	1.18 %	0.154 %	1.99 %
Proportion in Dataset	32.57 %	17.40 %	14.08 %	35.95 %

The test data is rather small so statistics as above are not really relevant. Its should however be noted that:

- As they are unknown sequences, the files do not include the *fasta* header
- All sequences are relatively short except for two of them of lengths 1474 and 1876

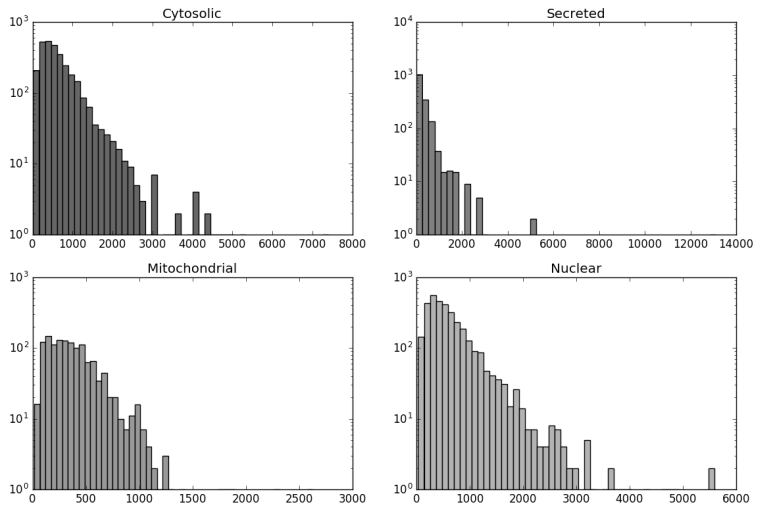

Fig. 1. Distribution of protein lengths across classes. The vertical axis is a log scale to better see outliers. We can clearly see that the Mitochondrial proteins are always quite short when the secreted ones can rarely be quite long. The two other classes share a similar distribution shifted towards longer sequences.

2.2 Model used

2.2.1 Recurrent Neural Networks

Feedforward Neural Networks are very good at understanding underlying structures in static data. However they have a harder time inferring time dependencies such as two following video frames, time series, texts as sequences of words or in our case, proteins as sequences of amino acids.

RNNs use iterative function loops to store information between time steps of sequences. To put it another way, RNNs take the current data point as well as their previous output as input for the current time-step. The following figure shows an RNN unrolled in time to illustrate this.

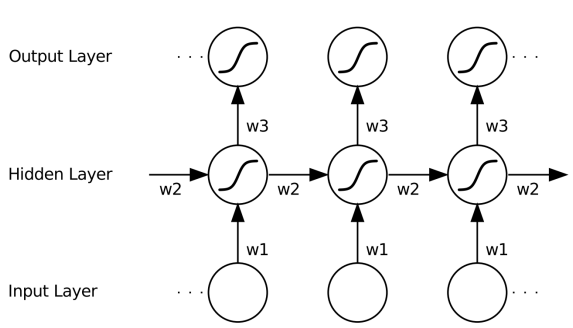

Fig. 2. Illustration of an RNN unrolled in time

An RNN's forward pass is almost the same as a feed-forward's : it computes a weighted sum of inputs and applies a non-linearity to it. Let us consider a sequence $X = \{x^1, \dots x^T\}$. Each x^t is n-dimensional and its coordinates are noted x_i^t. Let us denote $W = (w_{ih})_{i=1..n}^{h=1..m}$ the weight matrix of the RNN with m units. Lastly let s_h^t and a_h^t be the weighted sum of inputs and activation for the unit h at time-step t using the differentiable non-linearity σ :

$$s_h^t = \sum_{i=1}^{n} w_{ih} x_i^t + \sum_{j=1}^{m} w_{jh} a_h^{t-1}$$

$$a_h^t = \sigma(s_h^t)$$

For a classification task with C classes, then the output layer's c^{th} activation is

$$s_c^t = \sum_{i=1}^{m} w_{ic}^{output} a_i^t$$

$$a_c^t = \frac{exp(s_c^t)}{\sum_{i=1}^{C} exp(s_i^t)}$$

In this particular case the network outputs the following probabilities at the end of the sequence:

$$p(X) = p(x^1, \dots, x^T) = p(x^1) \prod_{i=2}^{T} p(x^i | x^{i-1}, \dots, x^1)$$

2.2.2 Gated Recurrent Units

However vanilla RNNs are hard to train as the gradient often either vanishes or explodes (6) which makes them really hard to use in practice (7). The gradient issue is partially solved by the Long Short Term memory recurrent unit (8). In the present task we will use the more recent, faster and more stable Gated Recurrent Units (GRU) that uses the same gating idea as the LSTM (9).

Using a vector notation, here is how the GRU works : its output is h^t, r^t is called the reset gate, z_t the update gate and $\widetilde{h}^t$ is the candidate output:

$$z^t = \sigma(W_{zx}x^t + W_{zh}h^{t-1})$$
$$r^t = \sigma(W_{rx}x_t + W_{rh}h^{t-1})$$
$$\widetilde{h}_t = tanh(W_{hx}x^t + W_{hh}(r^t \circ h^{t-1}))$$
$$h^t = (1 - z^t) \circ h^{t-1} + z^t \circ \widetilde{h}^t$$

This gating mechanism allows for a better memory, and prevents the gradient from vanishing. To solve the exploding gradient issue, a simple heuristic is to clip its values (10).

In our task, we will use this ability to generate representations of the amino acids. As this is a feature-free approach, we hope that the embedding of the amino-acids will be able to capture their interactions and these interactions' consequences on the proteins localization. For instance, the learning process of the RNN could cluster hydrophobic amino acids together regardless of any outside knowledge but finding out that proteins localized to the plasma membrane have higher proportions of hydrophobic amino acids or that cytoplasmic proteins have higher proportions of neutral amino acids(11).

3 Method

3.1 Processing the data

The first step to feed the sequences into our RNN-based model is to embed them : each letter in the sequence describing a protein is converted into a number. The overall vocabulary has 25 characters :

- 20 amino acids
- 3 special characters including X for "any amino acid"
- an Out-Of-Vocabulary token to take into account possible unseen characters (those from the test set that were no in the train set).
- a Pad token to fill in the blanks so that each sequence within a batch is of the same length

This process allows us to construct an embedding matrix (EM) of size $25 * input\_dim$. This $input\_dim$ variable is one of the model's hyper-parameters : each token in the vocabulary is associated with a row of the EM and has therefore $input\_dim$ dimensions.
At the end of this process, a batch of size bs is a 3-dimensional tensor with dimensions : $(bs, max\_sequence\_length, input\_dim)$.

Looking carefully at the data, we can see that the longest protein is more than 13 000 amino acids long. But on the other hand, looking at Table 1 we see that most of the data is way shorter than this. It is therefore tempting to crop the sequences and disregard the end of the very long sequences. However this would make us lose a lot of valuable information as the C and N-terminal locations on proteins are extremely informative (12). To cope with this factor, we still crop the long sequences but append the last 200 amino acids. The goal is for the network to sequentially train on the data and build a representation of each amino-acid in its EM, as well as a representation of each sequence at the output of the RNN.

3.2 Network Architecture

To output a classification prediction, we take the last outptut of the RNN (*i.e.* the output after the last amino acid in the sequence has been passed forward), we normalize it (13) and then feed this into a feed-forward neural network (MLP). As the RNN builds a representation of the protein sequence (with k units it means the space in which each protein lies is k dimensional), the MLP's task is to classify within this k-dimensional space, into one of our 4 classes : Cytosolic, Secreted, Nuclear or Mitochondrial. It is composed of two hidden layers with Rectified Linear Unit activations. Its output layer is of size 4 with a Softmax output so overall the model outputs a probability distribution for each specific sequence over the 4 classes.

To account for overfitting we apply 10% of dropout (14).

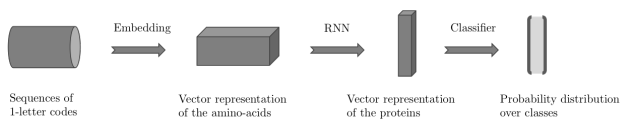

Fig. 3. Overall classification process

3.3 Training the model

We train the model using a variant of the mini-batch stochastic gradient descent algorithm : Adam(16). This algorithm uses yet another hyper-parameter, a learning rate.

To explore the large space of hyper-parameters, we use a random search as described by 15. During this search, the model is assessed using its accuracy on a validation set. This validation set has 800 samples from the training set ($\tilde{1}0\%$) and its statistics are compared to the training set's. It is re-sampled as long as it is not close enough. This random validation set allows us to reduce the risk of overfitting our hyper-parameters to the validation set, in the spirit of cross-validation. However a full k-fold cross-validation is not practical here as the hyper-parameter space is huge and computations are long. It will however be used to chose between a few good models selected from this random search.

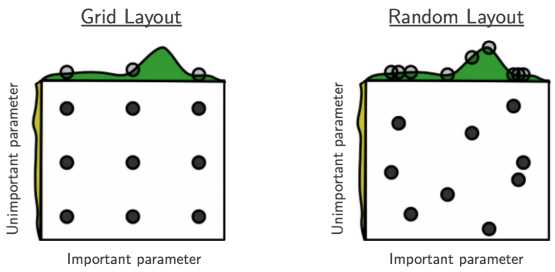

Fig. 4. Illustration of the advantages of a random search compared to a grid search, from 15

4 Experiment

4.1 Metrics

As the training dataset is not balanced between classes, we need to use better metrics than the sole accuracy to compare the cross-validated models. The 2 other metrics selected are related together and rely on (mis-)classification counts T_p, T_n, F_p and F_n : the AUC and the F_1 score.

The AUC is the Area Under the ROC Curve. It basically assesses how much of our classification is due to random choices. A straight linear

ROC curve yields an AUC of 0.5 which indicates full randomness, *i.e.* the classifier does not know anything. On the other hand a perfect step ROC curve integrates to 1 meaning perfect classification.

However the AUC is known to be noisy (17) and sometimes problematic for model comparison (18). This is why we'll also look at the F_1 score of the models.

$$Precision = \frac{T_p}{T_p + F_p}$$

$$Recall = \frac{T_p}{T_p + F_n}$$

$$F_1\ score = \frac{2\ Precision * Recall}{Precision + Recall}$$

Precision and Recall allow us to assess how much we can trust the model when it classifies proteins (Precision) and how much of each class is correctly predicted (Recall). The F_1 score being the harmonic mean of the two, it aggregates these two concurrent metrics into a final score that is their weighted average. In our case we care as much about Precision as Recall. Otherwise one could have used the F_β score to weigh them differently (19).

4.2 Cross-validation results

After a random hyper-parameters search, we selected two models, A and B, that differ as described by Table 2. We then ran a 5-fold cross-validation on each of them to be able to chose the final model. The results are in tables 3 and 4. As these metrics are defined for binary classification, in our multi-class case we averaged over the 4 (class-wise) computed values.

In both cases, the models had numerical issues, and one of the folds yielded random values : AUC scores are very close to 50 and the accuracy scores are close to the proportion of either the Cytosolic class (32%) or Nuclear class (36%). It can be visualized in figure 5. This is why to better compare the models we also provided a comparison of the metrics disregarding the outlying fold (see table 5).

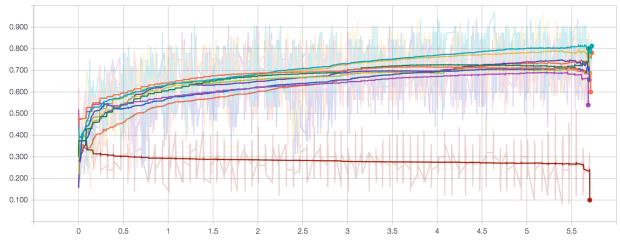

Fig. 5. Model A's learning curves over the cross-validation (Accuracy = f(Time))

Comparing the results (table 5), we see that model B's AUC and F1 scores are higher in mean with lower variances than model A's. These two metrics are the most meaningful and we'll therefore use the model B to classify the test data set. Moreover looking at the confusion matrices in figures 6 and 7, we can also see that chosing the model B improves the classification accuracy of the worst classified localization.

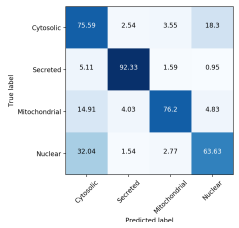

Fig. 6. A's confusion matrix

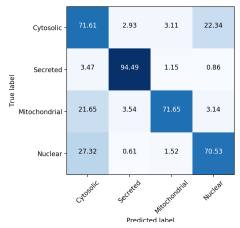

Fig. 7. B's confusion matrix

Table 2. Cross-validated models

	Model A	Model B
Embedding	60	40
RNN layers	[128, 64]	[128, 128]
Classifier layers	[32, 32]	[128, 128]
Learning rate	0.005	0.01

Table 3. Model A's cross-validation metrics

k-fold	Validation Accuracy	AUC score	Precision	Recall	F1 Score
1st	73.222	90.759	75.288	77.177	76.167
2nd	70.277	89.069	72.133	73.828	72.913
3rd	74.277	91.093	77.612	76.942	76.95
4th	71.555	89.714	74.596	73.617	74.014
5th	36.333	56.856	27.476	28.178	25.465
Mean	65.1328	83.4982	65.421	65.9484	65.1018
Std	14.46504	13.34074	19.05273	18.94433	19.87126

Table 4. Model B's cross-validation metrics

k-fold	Validation Accuracy	AUC score	Precision	Recall	F1 Score
1st	72.888	90.515	76.769	76.684	76.642
2nd	75.611	91.563	79.076	77.073	77.777
3rd	73.555	90.658	76.571	75.503	75.914
4th	70.388	89.376	73.569	74.713	74.029
5th	30.888	51.602	26.953	26.636	26.613
Mean	64.666	82.74279	66.58759	66.12179	66.19499
Std	16.97117	15.5859	19.89437	19.76073	19.8285

Table 5. Comparison of metrics accross models for their 4 best cross-validation folds.

Model	Validation Accuracy	AUC score	Precision	Recall	F1 Score
A					
Mean	72.33275	90.15875	74.90725	75.391	75.011
Std	1.53312	0.80908	1.95281	1.67223	1.61954
B					
Mean	73.1105	90.528	76.49625	75.99324	76.09049
Std	1.86491	0.77707	1.956	0.9384	1.36284

4.3 Final Model

According to the previously described analysis, we trained a B-type model to classify the dataset. It was trained on the full dataset (no validation set) with the same hyper-parameters as the cross-validated model, and saved after every epoch. In particular, even though it was trained for as many

epochs (25) it managed to overfit very clearly the data : it reached a final accuracy of 90%. To solve this issue, we use as final model the saved model for which the training accuracy was closest to the cross-validated's best fold.

4.3.1 Analyzing the embedding

Reducing the dimension of the EM (each amino acid is projected in a 40 dimensional space) and spherizing it we obtain the visualization in figure 8. Though we can not observe a very clear structure nor distinct clusters, we can still recover interesting properties looking at each amino acid's nearest neighbors. Using the Euclidian Norm, we can see that our model does project closely amino acids with similar properties. It is however far from understanding them perfectly and therefore projects closely amino acid that share no common chemical property.

Here are a few examples of successful amino acid projections, listing amino acids, their 3 closest neighbours and their shared properties :

- Cystein :
  - Threonine : small, polar, hydrophobic
  - Glysine : tiny, hydrophobic
  - Alanine : tiny, hydrophobic
- Leucine
  - Isoleucine : aliphatic, hydrophobic
  - Phenylalaline : hydrophobic
  - Valine : aliphatic, hydrophobic
- Serine
  - Aspargine : small, polar
  - Alanine : tiny
  - Proline : small
- Arginine
  - Lysine : positively charged
  - Serine : polar
  - Histidine : positively charged

These examples are quite encouraging but the model also fails by putting together (nearest neighbour) Proline and Glutamine, amongst other, though they have no common properties.

4.3.2 Sequence labelling

Here are the final model's classification results for the test set. The table 6 also shows the model's classification confidence : these percentages are the result of the softmaxed output layer of the network. Taking the $argmax$ of each probability distribution yields the final classification.

We can see from these results that the model is rarely sure of its prediction : 15 predictions have a 50%+ confidence score (evenly distributed over classes) and only 8 have a 70%+ confidence score (evenly distributed across classes except for Mitochondrial for which it is never that sure).

5 Discussion

The analysis of the embedding matrix could be assimilated to an unsupervised way of understanding the amino acids' interactions within proteins. However the dataset we have used is probably too small for a large RNN to pick up meaningful or even novel properties. However, if such a dataset had bee used, a couple improvements would have been considered.

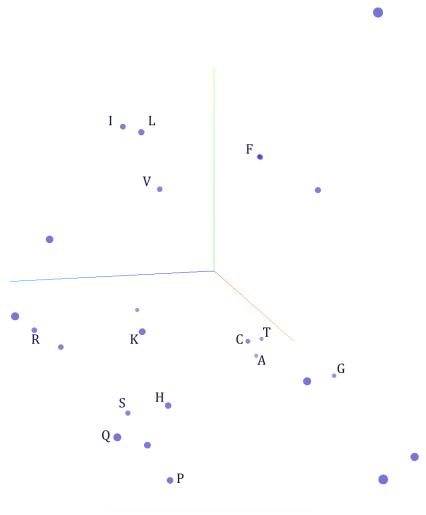

Fig. 8. EM visualization. The data is normalized by shifting each point by the centroid and making it unit norm. Only the mentioned amino acids are labeled for clarity.

Table 6. Classes and confidence for each protein in the test dataset

Protein	Cytosolic	Secreted	Mitochondrial	Nuclear	Classification
1	17.7%	51.3%	23.8%	7.1%	Secreted
2	4.2%	82.8%	11.6%	1.3%	Secreted
3	9.5%	21.3%	65.0%	3.9%	Mitochondrial
4	31.1%	14.1%	5.5%	49.1%	Nuclear
5	42.6%	16.9%	8.2%	32.1%	Cytosolic
6	11.5%	1.0%	0.3%	87.0%	Nuclear
7	49.3%	14.9%	12.7%	22.9%	Cytosolic
8	79.8%	5.3%	3.1%	11.6%	Cytosolic
9	55.5%	12.7%	12.8%	18.8%	Cytosolic
10	14.1%	19.5%	58.6%	7.6%	Mitochondrial
11	25.5%	14.7%	50.7%	8.9%	Mitochondrial
12	22.6%	10.0%	2.0%	65.2%	Nuclear
13	2.4%	94.0%	2.6%	0.8%	Secreted
14	7.3%	86.6%	3.4%	2.5%	Secreted
15	7.6%	10.5%	0.1%	81.7%	Nuclear
16	5.2%	5.0%	0.0%	89.5%	Nuclear
17	73.9%	7.3%	4.9%	13.8%	Cytosolic
18	57.1%	6.3%	4.5%	32.0%	Cytosolic
19	18.5%	16.3%	30.4%	34.7%	Nuclear
20	35.2%	19.9%	24.3%	20.3%	Cytosolic

Confidence has been rounded for clarity so it may not sum exactly to 1

Firstly, the RNN reads the sequences of amino acid in the natural sequence way. However because RNNs tend to struggle with long-term dependencies, the end of the protein will prevail in the model's inner representation of the sequence. To improve on this, we could use Bidirectionnal RNNs (figure 9) or even Attention models. These improvement would certainly improve the model's understanding of the underlying structure of amino acids, but it comes at a cost : both computations-wise and data-wise. Granted the RNNs can learn longer-term dependencies, we could also use the full proteins without needing to crop very long sequences.

The second improvement is to include more prior knowledge in the model. To do so we could add human-generated features to the classifier's input, alongside with the RNN's representation of the protein. This

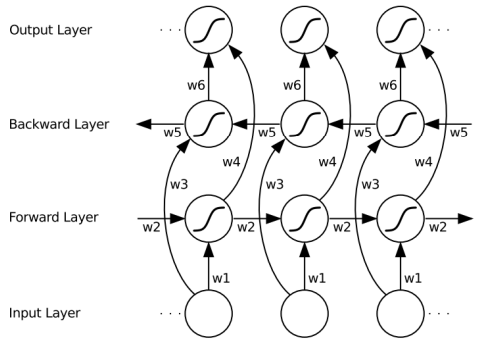

Fig. 9. Illustration of a Bidirectional RNN unrolled both in time and reversed-time.

would allow the classifier to chose from both sources of information what are the most relevant features. Disregarding the computational cost, this architecture can only improve the performances : in the worst-case scenario, the classifier will simply learn to ignore one of these sources of information.

6 Conclusion

Because the localization of a protein is both extremely important to our understanding of the human body and an extremely complex task, one could hope to build an automated system classifying them.

To do so, we use a Feed-forward Neural Network on top of a Recurrent Neural Network to classify proteins. The embedding process used to feed the amino acid sequences in the model is a very valuable step : this is the essence of what the model understands from the data. In our case, it seems that it does pick up a few important relationships between the amino acids. This allows it to reach an average of 73% accuracy on the validation data, as well as a 90% AUC score and 76% F1 score.

Though the dataset was quite small, the deep-learning architecture still managed to learn encouraging features. This could be improved in various ways as described in the Discussion but is already a good result for RNNs in subcellular location of eukaryotic proteins prediction.

References

[1]Automated Subcellular Location Determination and High-Throughput Microscopy. In *Developmental Cell*, Estelle Glory, Robert F. Murphy, 2007

[2] Amino acid properties and consequences of subsitutions. In *Bioinformatics for Geneticists*, M.R. Barnes, I.C. Gray eds, Wiley, 2003

[3] Expert system for predicting protein localization sites in gram-negative bacteria. In *Proteins: Structure, Function, and Genetics*, Vol. 11, K. Nakai and M. Kanehisa, 1991

[Sarda *et al*.] pSLIP: SVM based protein subcellular localization prediction using multiple physicochemical properties. In *BMC Bioinformatics*, Deepak Sarda, Gek Huey Chua, Kuo-Bin Li and Arun Krishnan, 2005

[Monney *et al*.] Prediction of Subcellular Localization by Neural Networks. In *Bioinformatics*, Catherine Mooney Yongâ˜Hong Wang Gianluca Pollastri, 2011

[4] The cell: a molecular approach. In *Washington, D.C: ASM Press*, Robert E. Hausman, Geoffrey M. Cooper, 2004

[5] Neural Networks for Pattern Recognition. C. M. Bishop, Oxford University Press, 1995

[6] Learning long-term dependencies with gradient descent is difficult. In *IEEE Transactions on Neural Networks*, Bengio, Y., Simard, P., and Frasconi, P. (1994).

[7] Gradient flow in Recurrent Nets: the Difficulty of Learning Long-term Dependencies. In *A Field Guide to Dynamical Recurrent Neural Networks* IEEE Press, S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, 2001

[8] Long short-term memory. In *Neural Computation*, Sepp Hochreiter, JÃ¼rgen Schmidhuber, 1997.

[9] Neural machine translation by jointly learning to align and translate. In *Technical report*, D. Bahdanau, K. Cho, and Y. Bengio. , 2014.

[10] Generating Sequences With Recurrent Neural Networks. Alex Graves, 2014

[11] Predicting the Subcellular Localization of Human Proteins Using Machine Learning and Exploratory Data Analysis. In *Genomics Proteomics Bioinformatics* George K. Acquaah-Mensah, Sonia M. Leach, and Chittibabu Guda, 2006

[12] A novel representation of protein sequences for prediction of subcellular location using support vector machines. In Protein Science, Setsuro Matsuda, Jean-Philippe Vert, Hiroto Saigo, Nobuhisa Ueda, Hiroyuki Toh, Tatsuya Akutsu, 2005

[13] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, S. Ioffe and C. Szegedy, 2015

[14] Dropout : A Simple Way to Prevent Neural Networks from Overfitting. In *Journal of Machine Learning Research*, N. Srivastava, G. Hinton, A. Krizhevsky, I Sutskever and R. Salakhutdinov, 2014

[15] Random Search for Hyper-Parameter Optimization. In *Journal of Machine Learning Research*, J. Bergstra and Y. Bengio, 2012

[16] Adam: A Method for Stochastic Optimization. Diederik P. Kingma, Jimmy Ba, 2014

[17] Small-sample precision of ROC-related estimates. In *Bioinformatics*, Hanczar, Blaise; Hua, Jianping; Sima, Chao; Weinstein, John; Bittner, Michael; and Dougherty, Edward R. ; 2010

[18] AUC: a misleading measure of the performance of predictive distribution models. In *Global Ecology and Biogeography*, Lobo, Jorge M.; JimÃ©nez-Valverde, Alberto; and Real, Raimundo ; 2008

[19] Information Retrieval, C. J. Van Rijsbergen, 1979

7 Appendix

Here is a short description of the code used to generate these models. We used Tensorflow 1.0 with Python 3.5. The code can be found at *vict0rsch.github.io/bioinfo/code.zip*

1. Loading the data
   a. Data processed with `Process_data.py`: it is formatted into a `list` of strings
   b. Before they are fed to the network they are embedded : the list of letters is tunrned into a Numpy array with numbers associated with their index in the vocabulary
2. Constructing the graph
   a. placeholers are created to hold the sequences, their lengths and labels at runtime
   b. we project the sequences into the 40 dimensional space of the embedding with the `embedding_lookup` function
   c. the embedded sentences are then given to a `dynamic_rnn` that uses `GRUCells`
   d. we take the last `state` of the `dynamic_rnn` as input to the multi-layer perceptron
   e. The MLP is a built looping over the following sequence of functions:
      (1) `tf.contrib.layers.linear`
      (2) `DropoutWrapper`
      (3) `Relu`
   f. the MLP's output is fed to a 4-units linear layer
   g. we compute the accuraty
   h. we compute the loss with `softmax_cross_entropy_with_logits`
   i. finaly we optimize the model with the function `optimize_loss` from `contrib` setting it to use the `Adam` algorithm.
3. Training the network
   a. The training and validation loss and accuracy are logged using `Tensorboard` to overviw the training procedure
   b. The data is fed into the `Session`
   c. every 50 batches we compute the validation loss to log into `Tensorboard`
   d. after every epoch we compute the overall training accuracy and shuffle the dataset
   e. the model is saved into a `checkpoint` finle after each epoch and when the user whishes to interrupt the training by catching a `KeyboardInterrupt` exception.