CNN-Fold: Protein Fold Recognition by Deep Convolutional Neural Networks

A Project

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

TYLER BANKS

Dr. Jianlin Cheng, Advisor

MAY 2016

The undersigned, appointed by the dean of the Graduate School, have examined the project entitled

CNN-FOLD: PROTEIN FOLD RECOGNITION BY DEEP CONVOLUTIONAL NEURAL NETWORKS

Presented by Tyler Banks

A candidate for the degree of

Master of Science

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

Dr. Jianlin Cheng

Dr. Rohit Chadha

Dr. Jeffrey Uhlmann

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Jianlin Cheng and Jie Hou for their help in completing this project. Dr. Cheng suggested this topic as an outlet for my interest in machine learning and biology; this project could not have been more fun and rewarding to research. Jie was an invaluable resource and friend. Jie's motivation, teaching and help made this project possible in the end. Thank you both. I would also like to thank Dr. Chadha and Dr. Uhlmann for taking time to be on my project committee.

-Tyler Banks

CNN-Fold: Protein Fold Recognition by Deep Convolutional Neural Networks

By Tyler Banks

For Master of Science, Computer Science

University of Missouri – Columbia 2016

Advisor: Dr. Jianlin Cheng

Abstract:

This paper's findings and methodologies are heavily based on the work by Taeho et al. in their paper entitled “Improving Protein Fold Recognition by Deep Learning Networks.” [1]. This paper and its proposed experiment tests the efficacy of Deep Convolution Neural Networks (DCNNs) in the binary protein classification problem. This builds on the knowledge and techniques used by Taeho and Hou et al. [1] and uses it as a control and guide; comparing conventional deep learning neural networks (DLNNs) to deep convolution neural networks. This project will also explore and employ various deep neural network and convolution neural network specific optimization techniques, including the Dropout method [2], network architecture adjustments, and various methods of data preprocessing. Convolution Neural networks have gained a lot of attention in recent years, and have been proven to be extremely effective in image recognition tasks. In a recent paper by Ciresan, CNNs achieved a remarkable 0.23% error rate on the MNIST character dataset [3]. The CNN's success is attributed to its biologically inspiration of receptive fields in the visual cortex. In the brain, inputs are mapped to higher layers of neurons that are receptive to particular overlapping patterns of input, such as lines or shapes.

Similarly, CNNs use a technique called filtering where a smaller matrix is slid, in stride, over the input matrix and the convolution of the two is taken to produce an activation matrix. This activation matrix can then be passed on to higher layers of the neural network to be processed further and classified. We propose that this may provide an advantage in protein classification if the input can be considered with regard to surrounding data points. The dataset that will be fed into the Deep CNN will be the same dataset used in "Improving Protein Fold Recognition by Deep Learning Networks." It consists of 951,600 protein pairs, based on 976 proteins carefully selected SCOP 1.37 dataset. Each protein pairing produces an 84 point comparison vector. Because Convolution Networks have been well suited to matrix based image recognition problems in the past it is a possibility that the vector based input may insufficient. If this is the case, other forms of input like the protein distance matrix may be considered and discussed. While this problem has been attacked from many different angles in the past, we are optimistic that Deep CNNs will provide comparable and possibly better results, based on its prior successes.

Contents

List of Figures:

List of Tables:

Chapter-1 Introduction

1.1 Introduction to Background

A protein's three-dimensional structure largely determines its function, therefore predicting a protein's tertiary structure is of great interest to many disciplines of biology and medicine alike. Protein prediction has lead to great advances in biotechnology and drug discovery [2] in recent years. However, present methodologies such as X-Ray Crystallography, and NMR Spectroscopy are too slow or too expensive to keep up with the widening gap of known protein sequences to known protein structures. Thus new methods of protein prediction are developed every day in an attempt to tackle this growing problem.

Many methods for protein recognition have been developed, including X-Ray Crystallography [4, 5, 6] and NMR Spectroscopy [7, 8, 9]. Both of these approaches have been able to provide some relief in the expanding gap, but even more significant to this paper is the methods provided by Machine learning.

Machine learning methods are data-driven methods that learn a function from the input data representing the protein's structure. This method is free from physical constraints presented in historical methodologies. The data driven nature of the protein recognition makes machine learning an excellent candidate to solve this problem, especially with the wide variety of machine learning methods already available today.

Machine learning has been used in the past to solve this exact same problem. Ding [10] uses Support Vector Machines (SVMs) to classify proteins. Ding was able to achieve convergence quickly and classify proteins with an error rate of less than 50% Taeho and Hou, et al.[1] used Deep Learning Neural Networks were used to achieve a remarkably high recognition rate of 84.5%. Several network architectures were tested and trained to provide the best possible outcome.

The purpose of this project is to focus and expand on the DN-Fold paper, by utilizing the same dataset and applying a Convolutional Neural Network strategy. We hope to determine if the dataset used in the DN-Fold paper is applicable to convolutional networks, and if it is, how well convolutional network perform in the protein recognition problem. We developed and utilized a new deep learning machine learning tool, CNN-Fold (Convolutional Neural Network for protein Fold recognition), to predict the tertiary protein structure based on the protein's primary structure. This tool utilizes some of the most recent developments in the machine learning field, including the dropout technique.

1.2 Introduction to CNN-Fold

Deep learning networks are nearly identical to traditional neural network architectures in that they consist of layers, containing node neurons that are all fully connected to other neurons in adjacent layers. The difference is the number of layers contained in the network the way the network types are trained to learn input data. Deep learning neural networks biggest advantage is the depth of the network. The introduction of additional layers allows for deeper layers to represent more abstract concepts, allowing for better classification of input data. Such a sophisticated and successful[11] machine learning technique is not without problems, however. With the introduction of additional layers as compared to the traditional 3 layer neural network, we run into the vanishing gradient problem, where the back-propagation of error starts to diminish as the layers increase. Traditional neural networks employ a variant of un-supervised stochastic gradient decent or convergence methods to maximize the likelihood of input data. This does not work well for anything many layers and extremely small error rates being back-propogated. This is where deep learning networks come in, as they are defined by their ability to combat this vanishing gradient problem and push past the traditional 3 layer architecture. They accomplish this through the application of unsupervised learning methods[12]; Pretraining the weights between layers individually through unlabeled data can help tune the parameters in advance in order to capture the most information in the original data space. This allows for back-propagated updates to remain minimal and reach the beginning of the network without disappearing.

In this project a special kind of Deep Learning technique was utilized. The model was developed using convolutional layers to achieve input recognition. Typically, convolutional neural networks are used to classify images because of their superior ability of make generalizations about local information. [13] A convolution can be intuitively illustrated as a simple comparison problem. A small filter is slid over an input space, or image, and at each step a snapshot is taken. Steps at which the filter window matches the input the most return a higher value than those that match poorly. Mathematically, this is the integral of the product of two functions. A convolutional layer produces a feature map based on the filter's parameters (kernel, stride). This feature map is the collection of snapshots. It then classifies the feature map at higher network layers.

We expected convolutional networks to exhibit similar or better results than standard deep learning networks due to several advantages provided by the convolutional technique. Conveniently, convolutional layers are similar to standard neural network layers in that data is fed forward through the network and trained via back-propagation, as such there is little to no additional cost for this additional benefit.

We expect that the use of convolutional neural networks will also provide a method to improve the speed at which we can classify protein folds. The advantage of windowing provided by a convolutional layer to downsize the input while retaining much of the information as possible will hopefully provide the speed up without a loss of accuracy.

Conversely, there is a possibility that there may be no improvement to fold recognition accuracy, it may even prove to be detrimental. Convolutional networks derive their befit from assumptions that can be made about the input data. We assume that spatially, data points share weights of their local, surrounding neighbors. This experiment will effectively determine if the Cheng dataset holds hidden spacial properties useful to classification using convolutional networks. We will test a variety of network architectures for effectiveness.

Chapter-2 Methods

2.1 Dataset and Input Features

The dataset utilized in this project was provided by Cheng and Baldi[14], a derivation of the original SCOP dataset, developed by Lindahl and Elofsson[15]. This dataset is comprised of 976 protiens from the SCOP 1.37 dataset such that the identity between any two proteins is less than 40%. By pairing each protein with every other protein in the dataset, Cheng was able to derive 951,600 protein pairings. To ensure that the derived dataset accurately represented the Lindahl dataset as a whole, Cheng and Baldi compared the effectiveness of their networks trained with their data on independent datasets. The final dataset contains 614 proteins that share the same family, 336 that share the same superfamily and 300 that share the same fold.

For each protein pairing a set of similarity features was extracted and calculated from the LINDAHL dataset to characterize the pairing. In total 5 different sequence alignment/protein structure prediction tools were used to derive statistics on sequence-sequence alignment, sequence-family information, sequence-profile alignment, profile-profile alignment and structural information. [1] The result of the feature extraction was a set of 84 features specific to each protein pairing. In Dr. Cheng's DN Fold paper an accuracy of 84.5% was attained on test protein pairings using the dataset. They used Deep Belief Networks to achieve this.

2.2 Deep Convolutional Neural Networks (DCNN)

A Convolutional Neural Network (CNN) is a subset of the neural network classification while the Deep Convolutional Neural Network (DCNN) is a subset of deep learning algorithms or deep learning neural networks. CNNs are comprised of neurons like a standard neural network with the key difference being the shared weights between layers. CNNs are spatially aware to their local environment through local connection patterns to adjacent neurons. See fig. 1.

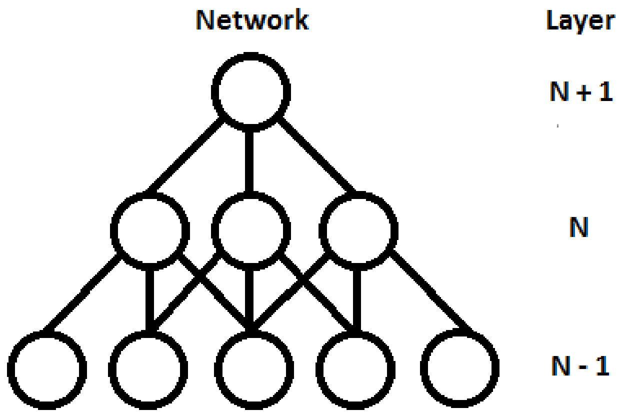


Fig 1. Convolution neural networks exploit local connectivity in adjacent neurons.

Like traditional neural networks traditional training methods and error correction techniques like back-propogation can be applied. In our case, using DCNNs we will need to utilize an unsupervised learning technique to reduce the initial error of the network to prevent the diminishing gradient problem. We will be using an unsupervised RBM pretraining method to produce a good set of initial neural weights. Another useful technique to improve results will be the use of the dropout technique. This will reduce redundancy produced by early convolutional layers. To summarize the steps in configuring and training our DCNN we will implement the following procedure:

1. Pre-train the network using Restricted Boltzmann Machines to ensure good initial weights
2. Back-propagate the error produced by the input data to update the initial weights

3. Temporarily drop neurons through the course of training to prevent overfitting
4. Cross validate the results with an independent dataset
5. Repeat on several network architectures

2.2.1 Convolutional Layers

Convolutional Networks are biologically inspired by our eyes and their associated cortical regions in the brain. Studies on the visual cortex, like those recorded by Hubel and Wiesel [16], have determined that patches of cells in the striate cortex are sensitive to smaller areas of the visual field. These smaller regions are known as receptive fields, and they cover the entirety of the visual field. The cells in this region act as filters over the visual field and take advantage of the spatially local correlations found in nature. For example, certain cells may respond strongly to a black bar in a vertical orientation, while another grouping of cells may have a preference for a black bar in a horizontal orientation. This is accomplished though sparse connectivity of the neurons in adjacent layers. In Figure 1, it can be seen that neurons in layer n receive their input from only a few neurons in the previous layer, n-1. In addition to the sparse connectivity the neurons that feed into the neuron in layer n are physically located next to each other. In this case, the neurons in layer n-1 can be represented as the original input or it's biological equivalent, the retina in an eye. Neurons in layer n are trained to produce the strongest response to certain local patterns, while stacking these layers together at a higher level, layer n+1 allows for more advanced features to be learned. As an example, we can imagine layer n to contain one neuron that responds strongly to the previously mentioned vertical bar, and another to respond to the horizontal bar. A neuron in layer n+1 can then be responsive to plus (+) signs in an input space while another can be responsive to a negative (-) sign. This highly simplified explanation does not account for many abstract concepts that can also be derived at higher layers, like absence of light or edges.

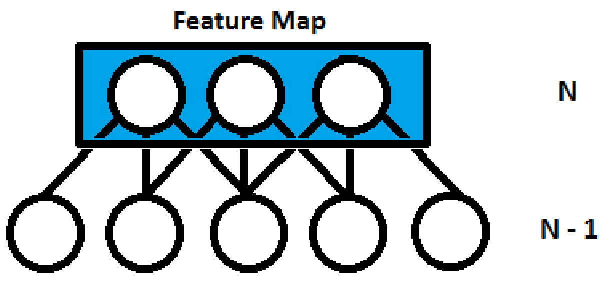


Fig 2. Convolution neural networks exploit local connectivity in adjacent neurons.

In CNNs each receptive field or “filter” is pushed across the entire input space, creating a feature map. As shown in Figure 2, the feature map is the result of pushing a filter of size [1,3] across the input space, or layer n-1, in stride [1, 1]. We can think of this mathematically as a convolution of the integral of two functions. As shown below:

$$(f * g)(i) \equiv \int_{0}^{\tau} f(\tau) * g(i - \tau)\, d\tau$$

However, for the application of convolutional networks, the input and filter are not continuous. Therefore we must reformulate our convolution equation as as two separate discrete functions with discrete sizes and discrete increments multiplied together. This is mathematically shown below:

$$(f * g)(i) - \sum_{j=1}^{m} g(j) * f(i - j + m / 2)$$

In terms of a 1 dimensional convolution, we can name *f* our input and *g* our filter. We can then state that *f* has a length of *n* and *g* has a length of *m*. This will ensure that we will produce a feature map where the higher the value of each function the higher the value of the outputted feature map. This will then produce a feature map of size:

$$f = (n - m + 1)$$

Furthermore we can alter the stride at which the snapshot is taken by altering the rate at which j increases, such that we take jump by k every time, producing a smaller kernel due to the decreased number of snapshots. This will produce a feature map of size:

$$f = (n - m + 1) / k$$

The effect of increasing the kernel size or stride can change the results drastically. By increasing kernel size you effectively reduce the significance of any individual data point. By increasing the stride you take redundancy out of the equation. For example, a kernel size of [1, 8] and a stride of [1, 8] will create a feature map that is 1/8th of the original size input, effectively holding 1/8th of the information previously conveyed. By decreasing the stride in the previous example we can reintroduce some of the lost information through redundancy; a stride of 2 will increase the output to the feature map by a factor of 4.

The kernel and stride length are both variables of the filter matrix which in turn produces the feature or activation matrix. So far we have only discussed the use of a single filter matrix, it imperative to use more than one, as each filter is only capable of learning a single function of the input. This is not dissimilar and in fact draws it's roots, biologically, to the previously mentioned receptive fields in the visual cortex. This is why we have many filters that pass over the data learning as many features as possible. In our case we are looking for features that are correlated between a subset of our 1 dimensional input. This, however, produces a direct, linear increase to the number of output neurons in the layer, and a multiplicative computational increase dependent on the number of neurons in the next layer. For a set of 10 filters with a kernel size of [1, 2], stride of [1, 1] with an input of [1, 84] would produce a staggering 830 neurons in that layer, as compared to a single filter of 83 neurons.

2.2.2 Downsampling Layers

A technique used in convolutional networks to combat this high computational cost of a convolutional layer is the downsampling layer, otherwise known as the pooling layer. Pooling layers take the output of large convolutional layers and either take the maximum, average, or a variety of other combinations. As such, pooling layers are primarily associated with reducing computational cost. However, this does result in a loss of data. Early testing models showed that by introducing a pooling layer after a convolutional layer performance by 10-20% depending on the pooling parameters. Due the input size consisting of only 84 points a pooling layer will not be used. A max pooling example is shown in figure 3. (Provided by deeplearning4j.org)

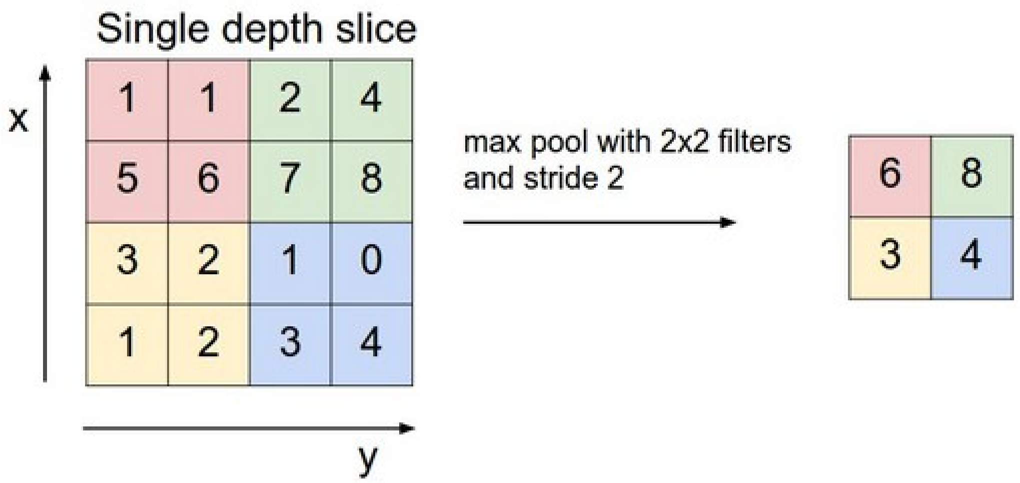


Fig 3. An input is reduced by a factor of 4. The output of a feature map is sub-sampled by taking the max value of the pool.

A typical convolutional neural network using pooling (sub-sampling layers) is shown in figure 4. (Provided by deeplearning4j.org)

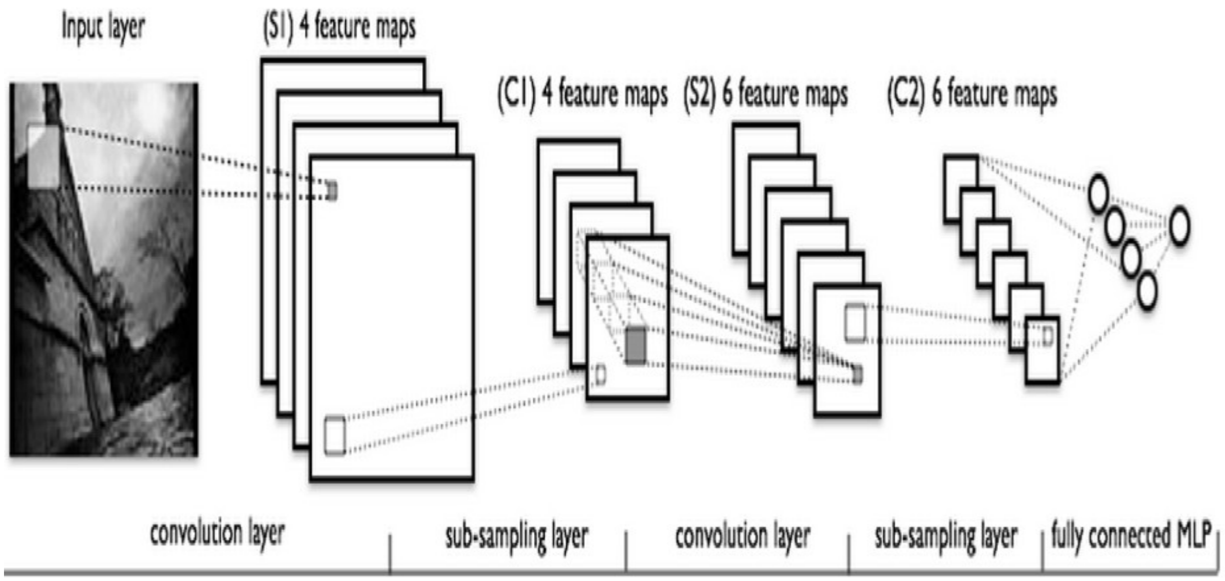


Fig 4. Convolution neural networks on a 2d dataset (image). The image is broken down in to feature maps using a specified kernel size and stride, the feature map is then sub-sampled using a pooling layer. This process is repeated and fed into a fully connected multilayer deep neural network.

Classification is accomplished through traditional means, convolutional neural networks make use of densely connected layers after convolutional layers, in essence, reformat the data.

2.2.3 Dense Layers

Fully connected, deep learning neural networks are comprised of an input layer, several hidden layers and an output layer. Deep learning networks typically go beyond the traditional single hidden layer, giving them the ability to learn more complex functions. Each neuron has a set of inputs provided by the previous layer, excluding the input layer, which is a direct representation of the working data. These inputs are run through a function like the one shown below.

$$o_j = f\left(\sum_{i=1}^{m} w_{ij} * a_i + b_j \right)$$

This can be read as: the output (o) of neuron *j* in the current layer is equal to the returned value of the activation function *f* which takes the summation of all *m* neurons output *a* in the previous layer multiplied by a weight, *w,* biased by *b*. The activation function is a way of smoothing the output of many neurons into a unified output that can be reliably read by the next layer's neurons. Activation functions seen in neural networks typically range from -1 to 1, they can be continuous, like the tanh function, or discrete, like the step function. The fully connected neural networks feed the data through all the neurons and output the results from the output layer. During the learning phase an error rate is back-propagated through the network and the weights (*w*) are adjusted to learn the given input better. The error rate, or mean squared error, is back-propagated:

$$MSE = 1/n \sum_{k=1}^{n} e(k)^2$$

These repeated feed-forward, feed-backward runs build a network that can generalize an input set that can be run on data without a known output. This is what gives neural networks their predictive ability. This project will utilize networks consisting of 2-4 hidden layers that will be trained using this method. This feedback technique is fundamentally flawed for errors smaller than what a computer can handle. The way floating point numbers are implemented in a typical computer does not allow for the network to adjust weights the the required precision. This is what lead to the development of pretraining networks.

2.3 Dropout Technique

Similar to how convolutional layers utilize special methods to improving calculation speed, traditional dense layers also have methods to reduce the computational complexity of the problem. Specifically, we used the dropout technique[17] to improve the runtime speed and reduce the chance of overfitting the data. Dropout effectively reduces the computational cost of a single neuron to zero; overfitting is a common problem in training neural networks and occurs when the network has learned the training data so well it is no longer able to make broad generalizations. The dropout technique randomly sets the output of a neuron to zero. The effects of the dropout can be visualized in figure 5.

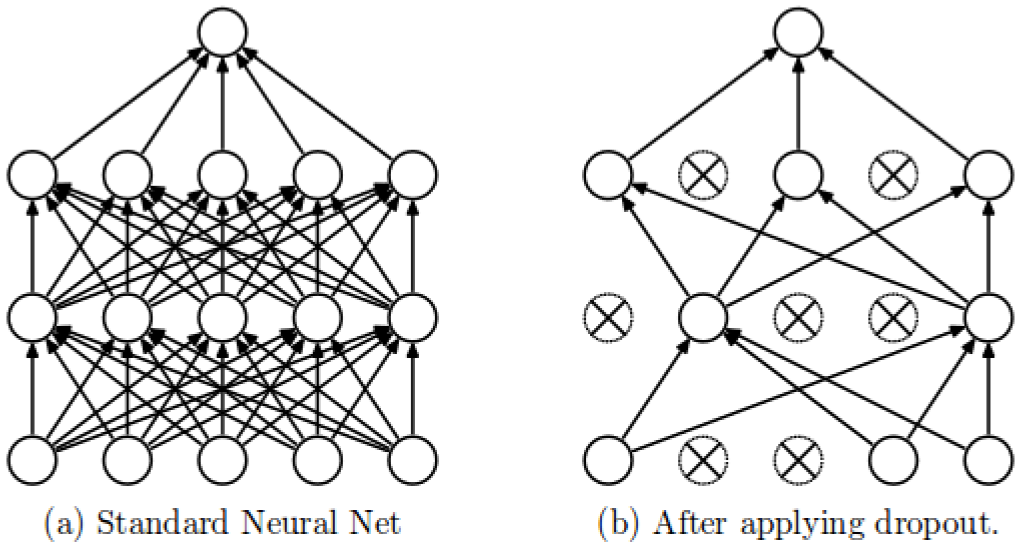


Fig 5. The dropout technique disables select neurons at a constant rate randomly. This disabling is temporary, meaning that the neurons will be reset and turned back on during the next epoch. Figure courtesy of Srivastava et. al.

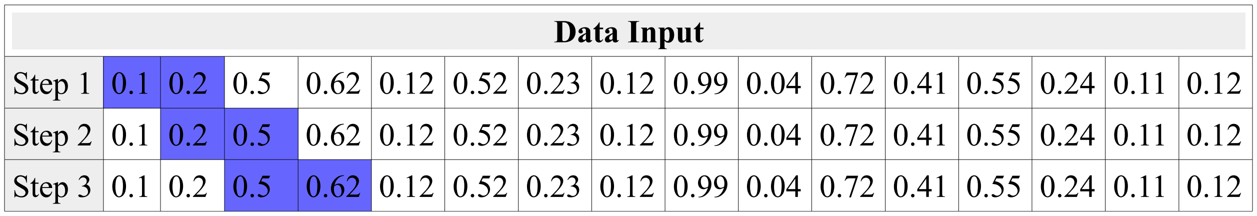
The dropout technique was found to perform better in the dense layers and degrade performance when used in convolutional layers. This is most likely attributed to the dependency of all input neurons applied to a filter. If a filter is missing half of it's input it will not produce a strong activation mapping, thus forwarding on incorrect data to the next classifying layers. It is also important to note that the convolutional layer is the input layer in all of our training networks, making it even more unsuitable.

A standard dropout rate, that has been known to outperform other dropout rates is 0.5 [17]. We applied this rate to all of our networks and all of their layers except for convolutional layers and/or input layers. Empirically, the networks were much quicker to reach a minimum and train overall.

Chapter-3 Models and Results

3.1 Model

The data tested was presented as a 1 dimensional array with a width of 84. As such, the data can be directly read and manipulated by convolutional layers with kernel heights and stride heights of 1. A visualization of a filter being applied over the data can be seen below in figure 6.



Data Input																
Step 1	0.1	0.2	0.5	0.62	0.12	0.52	0.23	0.12	0.99	0.04	0.72	0.41	0.55	0.24	0.11	0.12
Step 2	0.1	0.2	0.5	0.62	0.12	0.52	0.23	0.12	0.99	0.04	0.72	0.41	0.55	0.24	0.11	0.12
Step 3	0.1	0.2	0.5	0.62	0.12	0.52	0.23	0.12	0.99	0.04	0.72	0.41	0.55	0.24	0.11	0.12

Fig 6: A visualization example of a smaller version of our input. This displays a dataset with a width of 16, a single filter (in blue), a kernel width of 2 and a stride of 1. At each step a convolution is taken of the filter and the input to produce the activation matrix (not shown).

The process of selecting a good neural network model with the correct kernel size, stride width, filter number and depth is largely empirical and specific to the problem. It's important to train and test several network architectures.

In an attempt to improve the results of what was obtained by traditional neural networks we tried several different network architectures with varying kernel sizes and strides, see Table 1.

Model Number	Network Architecture
Model 1	C21K2S1-D100-D30-O1_30
Model 2	C21K4S1-D100-D30-O1_30
Model 3	C42K8S1-D100-D30-O1_30
Model 4	C42K2S2-D150-D35-O1_30
Model 5	C63K4S2-D150-D35-O1_30
Model 6	C63K8S2-D150-D35-O1_30
Model 7	C84K4S2-D150-D25-O1_30
Model 8	C84K8S2-D150-D25-O1_30
Model 9	C105K16S2-D150-D25-O1_30
Model 10	C105K2S2-D150-D25-O1_30
Model 11	C84K2S2-D100-D100-D30-O1_30

Table 1: A list of all network models tested to find the best architecture. The networks are formatted as follows: Convolutional Layer (C)<number of filters> Kernel Size (K)<width of kernel> Stride Length (S)<width of strides> - Dense, fully connected Layer (D)<number of node nurons> - Output Layer (O)<number of outputs>_<epochs> used to train the data.

3.1.1 Optimal Network Selection

The optimal networks were chosen by training and testing on a small random subset of our overall dataset. We then determined the network's F1 score and ranked the networks from best to worst and took the top 3. The F1 score is a good indicator because it considers the precision and recall of the test. The precision is the number of true positives divided by the number of positive results possible. The recall is a calculation of true positives divided by the number of positives that should have been positive. A higher F1 score means a better model. The top 3 preforming models were then trained and tested on the full dataset provided by the DNN-Fold paper. Those top performing networks are highlighted in Table 1.

3.2 Results and Comparison

Overall, the deep convolutional network did not perform as well as we had hoped but did classify at a rate comparable and even better than some historical methods. CNN-Fold's results can be seen in table 2 below.

Network	Family		Superfamily		Fold	
	Top 1	Top 5	Top 1	Top 5	Top 1	Top 5
C63K8S2-D150-D35-O1	25.4	51.7	3.7	66.4	4.1	46.3
C84K4S2-D150-D25-O1	33.2	56.8	8.1	67.9	15	58.5
C105K16S2-D150-D25-O1	24.1	37.8	5.6	42.4	10	36.2

Table 2: A list of all networks performance levels measured in sensitivity or recognition success rate. Proteins with the same family are easiest to recognize, while proteins who share the same superfamily or fold are typically more difficult to match.

The results were inferior to what was achieved in the DN-Fold paper. In the best of cases for the Top 5 protein folds we achieve a correct identification rate of 56.8% at the family level, 67.9% at the superfamily level, and 58.5% at the fold level. Top 1 did not exceed 33.2% at the family level, 8.1% at the superfamily level, and 15% at the fold level. We can conclude that this dataset lacked a spacial property that convolutional networks are built to take advantage of. Perhaps by reconstructing the data convolutional networks could outperform DN-Fold. This would require additional preprocessing of the data. For comparison to several other popular methods and DN-Fold see Table 3.

Network	Family		Superfamily		Fold	
	Top 1	Top 5	Top 1	Top 5	Top 1	Top 5
PSI-Blast [18]	71.2	72.3	27.4	27.9	4	4.7
THREADER [19]	49.2	58.9	10.8	24.7	14.6	37.7
CNN-FOLD	33.2	56.8	8.1	67.9	15	58.5
DN-FOLD [1]	84.5	91.2	61.5	76.5	33.6	60.7

Table 3: A list of a few previous prediction methods and their performance [1].

It's important to note that the data used has already gone through some initial preprocessing to improve performance. Regarding the performance of CNN-Fold, there is no label balancing in place which could hurt the overall score. This could be a potential area for further research.

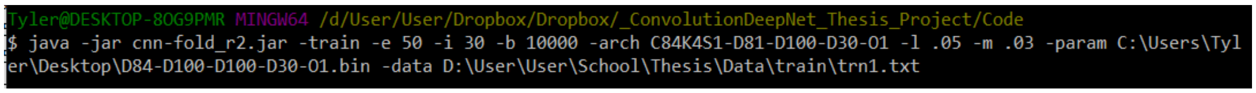
Chapter-4 CNN-Fold

4.1 CNN-Fold Program

The pre-compiled CNN-Fold program can be downloaded at:
[https://goo.gl/PGBV1E] and the source code can be found on bitbucket at:
[https://bitbucket.org/tjb1991/cnn-fold]. The program allows for you to either define a network used in the DN-Fold paper, networks used in this paper, create your own architecture from the command line or use a json styled network configuration file. The help screen can be seen below.

4.2 CNN-Fold Usage

CNN-Fold usage is a two stage process: training and evaluating. First we train a network architecture and then evaluate the network's capability. The program has the ability to handle 4 different types of network specifications. First, a user is able to select from a collection of pre-compiled networks based on the models given in the DN-Fold paper. To do this, use the -dnfold <n> switch. The user is also able to select any of the network architectures previously mentioned in this paper by using the -cnnfold <n> switch. The final two configuration options allow the user to define their own network architecture. The -arch <string> command takes in a string argument that represents the network. For example, the option C30K4S1-D100-D30-O1 would give you a Convolutional layer with 30 filters, a kernel size of 4, and a Stride of 1, followed by a 100 node dense layer, 30 node dense layer and an output layer with a single output neuron. Finally, by using the config option the user is able to specify a json formatted network configuration file. It is important to note that by using any of the other options a json file will be automatically created and saved to your disk. This file can be edited and used with -config and allows for more options to be exploited. An example training command is shown below:

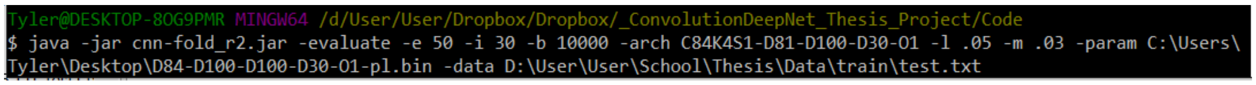


```
Tyler@DESKTOP-8OG9PMR MINGW64 /d/User/User/Dropbox/Dropbox/_ConvolutionDeepNet_Thesis_Project/Code
$ java -jar cnn-fold_r2.jar -train -e 50 -i 30 -b 10000 -arch C84K4S1-D81-D100-D30-O1 -l .05 -m .03 -param C:\Users\Tyl
er\Desktop\D84-D100-D100-D30-O1.bin -data D:\User\User\School\Thesis\Data\train\trn1.txt
```

By running this command a series of output logs will be displayed on the screen and inform you on the progress of training the network. Shown below:

```
Saving C84K4S1-D81-D100-D30-O1.json
04:31:25.716 [main] INFO  edu.missouri.banks.Network - Train model....
04:31:27.004 [main] WARN  o.d.optimize.solvers.BaseOptimizer - Objective function automatically set to minimize.
04:34:37.131 [main] INFO  o.d.o.l.ScoreIterationListener - Score at iteration 0 is 0.6396009765625
```

Next we test our network. An example usage command displaying the evaluation method is shown below:



```
Tyler@DESKTOP-8OG9PMR MINGW64 /d/User/User/Dropbox/Dropbox/_ConvolutionDeepNet_Thesis_Project/Code
$ java -jar cnn-fold_r2.jar -evaluate -e 50 -i 30 -b 10000 -arch C84K4S1-D81-D100-D30-O1 -l .05 -m .03 -param C:\Users\
Tyler\Desktop\D84-D100-D100-D30-O1-pl.bin -data D:\User\User\School\Thesis\Data\train\test.txt
```

At the conclusion of this testing a set of output is produced, giving the user the TP, TN, FP, FN rates as well as Accuracy, Precision, Recall, and an F1 score. Similar to what's shown below:

```
Examples labeled as 0 classified by model as 0: 38582 times
Examples labeled as 0 classified by model as 1: 56221 times
Examples labeled as 1 classified by model as 0: 164 times
Examples labeled as 1 classified by model as 1: 583 times

==========================Scores========================================
 Accuracy:  0.4099
 Precision: 0.503
 Recall:    0.5937
 F1 Score:  0.5446
========================================================================
```

4.3 CNN-Fold Help

```
usage: cnn-fold.jar [-arch <architecture> | -cnnfold <model number> |
       -config <file> | -dnfold <model number>] [-b <n>]   [-data <file>]
       [-e <n>] [-evaluate | -train] [-i <n>] [-l <n>] [-m <n>] [-param
       <file>]
 -arch <architecture>      create a model from the command line, format
                           should be in C30K4S1-D100-D30-O1, For
                           Convolutional layer with 30 filters, Kernel
                           size of 4, and Stride of 1, dense layer size
                           100... Output layer size 1
 -b <n>                    optional batch size for the dataset
                           [default:1000]
 -cnnfold <model number>   load a model corresponding to the CNNFold paper
 -config <file>            use given file loading a network configuration
 -data <file>              use given file to train a network or test a
                           trained network
 -dnfold <model number>    load a model corresponding to the DN-Fold paper
 -e <n>                    optional epochs to run the network [default:30]
 -evaluate                 test the network
 -i <n>                    optional iterations to run the network each
                           epoch [default:60]
 -l <n>                    optional learning rate, works with arch only
 -m <n>                    optional momentum, works with arch only
 -param <file>             use given file loading or saving a trained
                           network parameters
 -train                    train the network

examples:
 1a. create a trained network (-param will create a new file):
  cnn-fold.jar -train -config conf-5-8-8-1.json -data trn1.txt -param
outFileparam-5-8-8-1.bin

 1b. create a trained network using built in model (-param will create a new
file):
  cnn-fold.jar -train -dnfold 1 -data trn1.txt -param outFileparam-5-8-8-
1.bin

 2a. evaluate a trained network (-param will open a file):
  cnn-fold.jar -evaluate -config conf-5-8-8-1.json -param inFileparam-5-8-8-
1.bin -data test-all.txt

 2b. evaluate a trained network using built in model (-param will open a
file):
  cnn-fold.jar -evaluate -dnfold 1 -param inFileparam-5-8-8-1.bin -data test-
all.txt
```

Chapter-5 Conclusion and Future Work

5.1 Conclusion

In this project, we developed and presented a convolutional deep learning network to predict if a given input protein pairing shared the same family, superfamily or fold. A combination of network architectures and convolutional layers were built, trained, and tested on the Cheng and Baldi dataset. We found that CNN-Fold does not perform as well as the approach employed by DN-Fold. This was due to the lack of spatial qualities presented by the dataset. There may still be hope for CNN-Fold when applied to proteins formatted differently though.

5.2 Future Work

In the future we plan to take a look at alternate methods of conveying information about proteins. By formatting the protein in a way that resembles an image or sound wave perhaps Convolutional neural networks will provide favorable performance. Additionally, we may look at methods of data normalizing in the range of 0 – 1 to closer resemble an image. As for the CNN-Fold tool, we plan to make it a little more general and less dependent to the 84 point dataset used. We plan to add the ability to alter more variables of the network from the command line and make the json file a little easier to use.

Chapter-6 References

1. Jo, Taeho, et al. "Improving Protein Fold Recognition by Deep Learning Networks." Scientific reports 5 (2015).
2. Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." The Journal of Machine Learning Research 15.1 (2014): 1929-1958.
3. Ciresan, Dan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification." Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on. IEEE, 2012.
4. Kendrew J. C., et al. (1958-03-08). "A Three-Dimensional Model of the Myoglobin Molecule Obtained by X-Ray Analysis". Nature 181 (4610): 662–6. Bibcode:1958Natur.181..662K. doi:10.1038/181662a0. PMID 13517261.
5. Svergun, Dmitri I., Maxim V. Petoukhov, and Michel HJ Koch. "Determination of domain structure of proteins from X-ray solution scattering." *Biophysical journal* 80.6 (2001): 2946-2953.
6. Deisenhofer, Johann, et al. "X-ray structure analysis of a membrane protein complex: electron density map at 3 Å resolution and a model of the chromophores of the photosynthetic reaction center from Rhodopseudomonas viridis." *Journal of molecular biology* 180.2 (1984): 385-398.
7. Wüthrich K (December 1990). "Protein structure determination in solution by NMR spectroscopy". J. Biol. Chem. 265 (36): 22059–62. PMID 2266107.
8. Wüthrich K (November 2001). "The way to NMR structures of proteins". Nature Structural & Molecular Biology 8 (11): 923–5. doi:10.1038/nsb1101-923. PMID 11685234.
9. Wüthrich, Kurt. "Nuclear Magnetic Resonance (NMR) Spectroscopy of Proteins." *eLS* (2001).
10. Ding, Chris HQ, and Inna Dubchak. "Multi-class protein fold recognition using support vector machines and neural networks." *Bioinformatics* 17.4 (2001): 349-358.
11. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.
12. Elman, Jeffrey L. "Learning and development in neural networks: The importance of starting small." *Cognition* 48.1 (1993): 71-99.
13. LeCun, Yann, and Yoshua Bengio. "Convolutional networks for images, speech, and time series." *The handbook of brain theory and neural networks* 3361.10 (1995): 1995.
14. Cheng, J. & Baldi, P. A machine learning information retrieval approach to protein fold recognition. *Bioinformatics* **22,** 1456–1463 (2006).
15. Lindahl, Erik, and Arne Elofsson. "Identification of related proteins on family, superfamily and fold level." *Journal of molecular biology* 295.3 (2000): 613-625.

16. Hubel, David H., and Torsten N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex." *The Journal of physiology* 160.1 (1962): 106-154.

17. N Srivastava, G Hinton, and A Krizhevsky. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Reasearch 15 2014;1929-1958.

18. Altschul, S. F. et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic acids research* 25, 3389–3402 (1997).

19. Jones, D. T., Taylort, W. & Thornton, J. M. A new approach to protein fold recognition. *Nature* 358, 86–98 (1992).