Praca Magisterska

Agnieszka Pocha

September 20, 2015

Abstract

The goal of this work is to... ...drug design... This is achieved by applying (deep?) convolutional neural networks to the problem.

Contents

Chapter 1

Introduction

1.1 related work, że to ma oparcie w czymś i o czym jest praca

Chapter 2

The Problem and the Data

2.1 Terminology

One of major areas of study nowadays is **drug design**. It is a quickly developing field, facing challenging problems, such as conducting experiments which are very expensive and extremely time consuming. Therefore, computer modelling is of vital importance. Artificial intellingence and machine learning have been successfuly incorporated to this field. (citation needed?). One of the most common problems in drug design is telling whether **protein** and **ligand** will together produce an **active** or **inactive** compund.

Proteins are molecules built from **amino acid residues** forming a single **chain**. The number of residues defines the length of the chain. Proteins are present in all leaving organisms.

ligand, protein-ligand docking, receptor, donor, active, not active, pharmacophore, interactions, active non/inactive protein

2.2 Fingerprints

One of the first questions that have to be answered before *any modelling task can be started* is: how will I represent my data? Some proteins have less than 100 residues while others might have even few thousands of them. The order of amino acids in the chain and their spatial arrangement carry a lot of information. It might seem that a natural representation of a protein will be a graph containing extra information about how the nodes are arranged in space. Unfortunately, graph algorithms are computationally expensive and it is nowadays not possible to use them (na czym się opieram?). Therefore, *we* need another representation, that will carry as much information as possible and *be computationally effective*. *For this reason* many types of fingerprints have been designed and they meet the criterions metioned.

Conventionally, fingerprints represent a *protein/molecule* as a binary(?) vector. Each element of this vector *tells* whether a specific structure is present in the *protein/molecule* or not, e.g. whether the *protein/molecule* has a .... (oprzeć się na jakiejś publikacji). Vectors are widely used in machine learning as data representation as they can be *compilled* into matrices which allows to *easy computation, easy proofs, happy algebra* «We need a simpler representation - such on which we can quickly apply many function, such that is well designed for computers, for modern algorithms» Even though representing a protein as a vector means loosing a lot of information about it, it enables effective computation and still provides us with reasonable results.

As already said, there are many different fingerprints designed. They vary in length and the features included. Some of them are designed for specific tasks. (citation needed). In this work 2D-SIFt will be used.

2.3 Problem

drug design, innovative data representation from [2], more details, what exactly am I trying to achieve? *Deep* Convolutional neural networks will be applied to the problem.

2.4 Datasets

The data consists of multiple datasets, each describing reactions between a single protein and multiple ligands. Each dataset consists of four dimensions described by: the number of ligands, length of the protein, 6 standard pharmacophore features of ligand and 9 types of interactions with amino acid[2]. One data sample can be seen as a 3-dimensional matrix that describes how a single ligand bounds with a specific protein. The 3 dimensions are: the length of the protein (number of its residues), 6 standard pharmacophore features and 9 types of interactions with amino acid. Most of the data samples are labeled as active or nonactive, the rest is unlabeled. A single data sample is presented on figure 2.1.

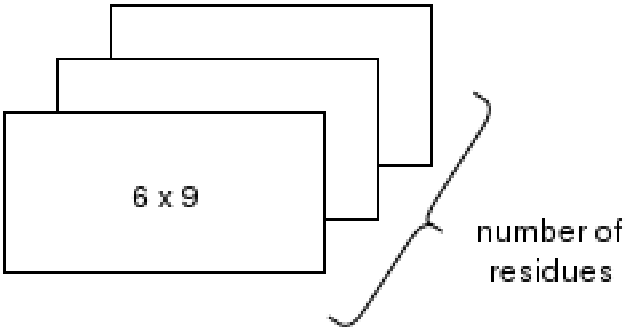


Figure 2.1: A single data sample.

The 6 pharmacophore features are: hydrogen bond acceptor, hydrogen bond donor, hydrophobic, negatively charged group, positively charged group, aromatic. The 9 types of interactions with amino acid are: any, with a backbone, interaction with sidechain, polar, hydrophobic, hydrogen bond acceptor, hydrogen bond donor, charged interaction, aromatic.

The values constituting the dataset are discrete, namely: 0 to 9. 0 means there is no interaction of specific kind. 1 and 2? As stated above the labels are reperesented as ??? (not active), ??? (active), ?? (no information).

2.5 Sparsity

check if the data is sparse, it yes then state that it is and explain why

2.6 Data representation

Each data sample is represented as a vector of $r * 6 * 9$ length, where r is the length of the protein. Data samples constitute a dataset. Each dataset describes a reaction/bonding between a certain protein and the ligand.

why was this particular fingerprint representation chosen

Chapter 3

The Model or Deep Convolutional Neural Networks

3.1 Deep Neural Networks

DNN

3.2 Convolutional Neural Networks

The simplest definition of convolutional neural networks is probably: neural networks that takes adventage of using the convolution operation. CNN might be seen as a network consisting of two parts - first part is the convolutional part and it is responsible for extracting the features from the data. The second part might be a softmax layer responsible for classifying samples based on features provided by the convolutional part. This schema is shown on figure 3.1.

In this section we will give motivation that stands behind using convolutional neural networks, explain what is the convolution operation, and what is pooling. Types of activation functions that might be used in convolutional neural networks will be presented as well as learning algorithm.

3.2.1 Motivation

Convolutional neural networks take adventage data which topological order carries some information. Therefore, CNNs are often applied to image recognition, video analysis and natural language processing problems.
This attempts *are often succesful/often give better results than (any other) models*.

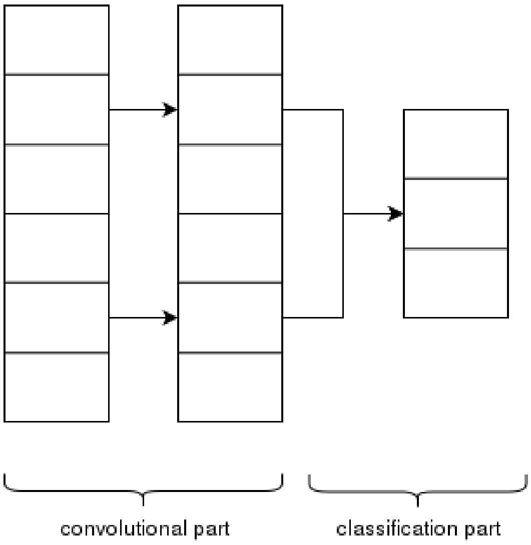


Figure 3.1: An example of convolutional neural network.

3.2.2 Computation Flow

As stated above, CNNs can be conceptually divided into two subnetworks. In this subsection it will be described how the data is processed within the convolutional subnetwork. Not much attention will be put to classifying subnetwork as there is a wide variety of possible approaches.

There are three elemental operations that are performed in each layer of the convolutional subnetwork. At first, the input is convoluted with a kernel matrix. The result of this operation becomes an input to the activation function, which output becomes the input to the pooling function. The output of the pooling function becomes the input to the next layer which might be another convolutional layer and the whole process will begin again. This process is schematically depicted on figure 3.2 and it can be also described by the following formula:

$output = p(\sigma(c(input)))$,

where c is the convolution function, σ is the activation function, and p is the pooling function.

One might also imagine three consecutive seperate layers: a convolutional layer, a classical layer that applies activation function and finally a pooling layer.

Each of these operations will be described in details in the following subsections.

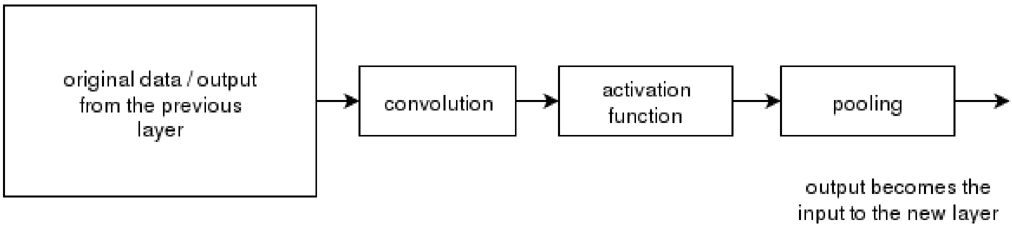

original data / output from the previous layer
convolution
activation function
pooling
output becomes the input to the new layer

Figure 3.2: The three elemental operations performed in each convolutional layer.

3.2.3 Convolution

Convolution operation takes as operands two functions and returns a new function as a result. Mathematically, convolution is defined as:

$$c(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx,$$

where c is a function returned by convolution operation and t is a point in which function c is evaluated. c is defined as an integral over two other functions: f is often called the input, while g is often reffered to as a kernel.

Convolution for neural networks

It is worth considering how this equation can be applied to neural networks. Since representing real values in the computers is rather troublesome, it is desirable to express this equation in a discrete form, which is shown below:

$$\overline{c}(t) = \sum_{x=-\infty}^{\infty} \overline{f}(x)\overline{g}(t-x)$$

If we want to apply this equation to neural networks we might want to think of $\overline{f}$ as a data sample. In such approach $\overline{g}$ would become a matrix that is moving around the data sample producing a single value in each place. From now on, we will call f an input, g a filter or a convolutional matrix and c na output.

Usually, the convolutional window is much smaller then the input and convolution is appplied multiple times. Each time a submatrix of input is multiplied by the convolution window. The result of this operation is a scalar, and a result of a whole process is a matrix. Each layer of neural network might include multiple filters and thus each layer might produce an output of higher dimensionality than the provided input - the additional dimension is produced due to using multiple filters.

It is worth noting, that there are different kernels in each layer. Kernels in the next layer work on the features extracted from the previous layers. Therefore, the later the layer is in the neural network, the more complicated patterns it may discover.

Size and stride, spatial invariance

It is worth taking a closer glance at how convolution works in details. First, we have to define two important parameters of the convolutional windows - their size and stride. The size is simply the size of the convolutional matrix. The stride defines which part of the input will be convoluted next with respect to the part of the input that is currently being convoluted. This concept is shown in figure 3.3.

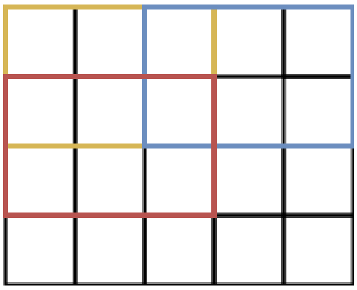


Figure 3.3: The figure shows three consecutive positions of a convolutional window - yellow, blue and red. The input is of size 4 × 5, the convolution window has size 2×3 and stride 1×2, which means the window can move two elements to the right or one to the bottom.

Three consecutive positions of a convolutional window are depicted. Each position defines which submatrix of the input will be multiplied by the convolutional matrix. Each multiplication will produce a single value - these values when combined will produce an output matrix.

It should be noted that each time the convolution window is the same, only the part of the input that is being convoluted changes. Each filter is specialised in finding a specific pattern - because it is convoluted with many submatrices of the input it will find that pattern regardless of where the patter is on the input matrix. This property is called spatial invariance. Figure 3.4 shows this concept more clearly.

As stated above, in each layer there usually is plenty of convolution matrices - each is specialised in finding a specific pattern. Each pattern might be seen as a useful feature of the data and convolution provides us with some sort of map that shows where in the input this feature exists and where it does not. Therefore, it is often useful to see convolution windows as filters or feature extractors.

Data size reduction due to convolution operation

Let the input be a $I \times J$ matrix and the convolution filter a $K \times L$ matrix with a 1×1 stride. Therefore, the first submatrix to multiply by the convolution filter will be $[1 : K] \times [1 : L]$ and it will return a single value. The last submatrix will

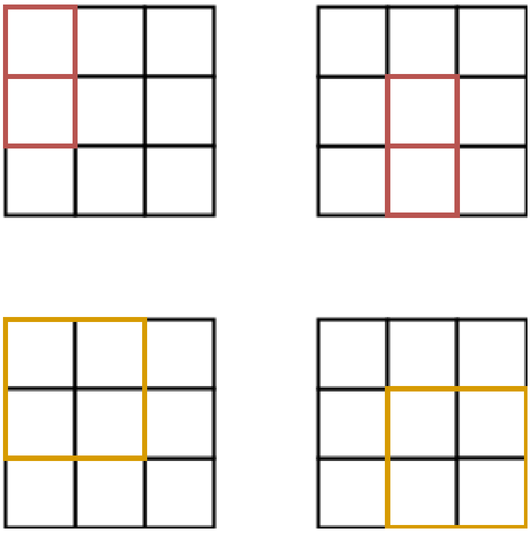


Figure 3.4: In the upper row two locations of a certain feature are shown in red. In the bottom row the location of a convolution window that will discover this feature is shown in yellow. As can be observed, by moving the convolution window it is possible to extract a certain feature regardless of its location.

be $[I-K+1:I]\times[J-L+1:J]$ and it will also produce a single value. As a result the output will be a $[I-K+1]\times[J-L+1]$ matrix. This process is shown on picture 3.5.

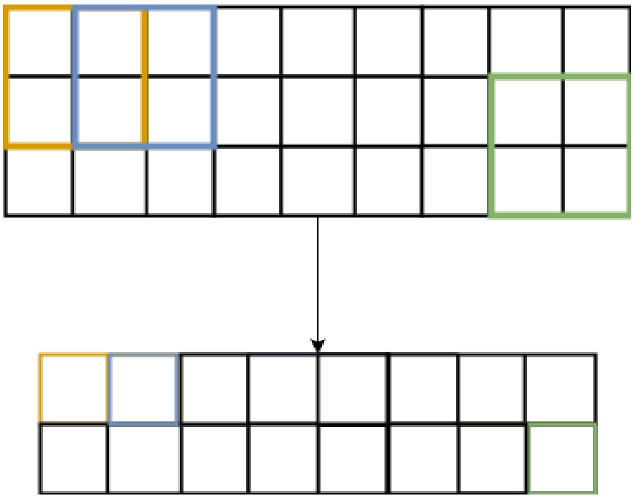


Figure 3.5: On this picture $I=3, J=9, K$ and $L=2$. First submatrix and the output it produces are shown in yellow, second are shown in blue and the last are shown in green. The dimensionality reduction might be observed.

It can be observed that the values laying close to the edge of the input matrix will be underrepresented. The upper left corner of the input matrix will have influence on only one element of the output (the yellow one) while its right neighbour will already have influence on two elements of the output (the yellow and the blue one). The elements in the middle of the input matrix will influence four elements of the output. Such unequalities of the influence might be undesirable. There is a variety of ways to address this problem, e.g. zero-padding, but we will not cover them.

3.2.4 Activation function

There are few possible activation functions that might be used in convolutional neural networks[DUTCH PAPER].

- the sigmoid activation function $\sigma(x) = \frac{1}{1+e^-x}$
- the hyperbolic tangent activation function $tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$
- the rectifier linear function[BENGIO, GLOROT 2011] $rect(x) = max(0, x)$
- the maxout activation function

$$maxout_i(x) = \max_{j\in[1,k]} z_{ij}$$
$$z_{ij} = x^T W_{\dots ij} + b_{ij}$$

3.2.5 Pooling

Pooling is an operation, usually applied on a matrix, that takes as an input multiple values and returns a single value describing the input. Typical pooling functions are[dutch, bengio book]:

- max pooling - the max value of input is returned
- average pooling - the average value of input is returned
- weighted average pooling - the weighted average is returned. Weights are usually *(citation needed?)* defined by the distance from *what?*
- stochastic max pooling - one element is chosen from the input to become the result. Probability of choosing an element is proportional to its value. [Zeiler et all/za dutch]

Pooling is defined not only by its type but also by its size and stride. The size of pooling defines, how many values will be taken as an input - the bigger the size of pooling, the more information is accumulated in a single value. The pooling stride defines, where will be the next submatrix with respect to its present location. Fig 3.6 illustrates this concept.

When pooling is applied on the edges of the input same problems might be encountered as with the convolutions, namely some values will be underrepresented. It is worth noting that the output of the pooling is off smaller size than the input. To address these problems same techniques might be applied.

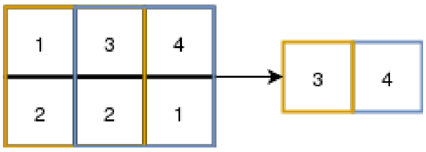


Figure 3.6: The result of applying the max pooling of size 2 × 2 with stride 1 × 1 to the input. Yellow square is the first pooling window, blue square is the second one. The values of the output are coloured accordingly.

3.2.6 Summary

In the convolutional subnetwork each layer applies three operations to the input, namely: convolution, which provides us with spatial invariance, activation function and pooling. As a result of this processing, the features are extracted from the data. These features are then used by the classifying subnetwork.

3.2.7 Learning Algorithm

Even though convolution operation and pooling might seem to introduce a lot of changes, the classical algorithms for learning neural networks, such as stochastic gradient decent, might be used.

Chapter 4

The Model

4.1 Goal

The goal of this work was to build a model that will well perform the task of classification of the provided data. To complete this task multiple obstacles had to be overcomed, i.e. small data size, missing labels, a big number of hyperparameters that had to be adjusted.

4.2 Data

The dataset describes interactions between a single protein and multiple ligands. One might choose to see the dataset as a four dimensional matrix with axes dimensions described by: the number of ligands, length of the protein, 6 standard pharmacophore features of ligand and 9 types of interactions with amino acid[2]. One data sample can be seen as a 3-dimensional matrix that describes how a protein bounds with a specific ligand. The 3 dimensions are: the length of the protein (number of its residues), 6 standard pharmacophore features and 9 types of interactions with amino acid. A single data sample is presented on figure 4.1.

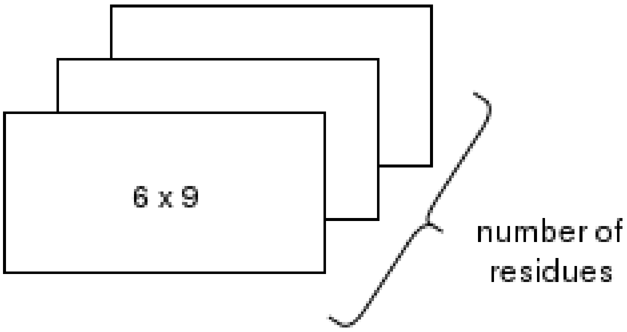


Figure 4.1: A single data sample.

The 6 pharmacophore features are: hydrogen bond acceptor, hydrogen bond donor, hydrophobic, negatively charged group, positively charged group, aromatic. The 9 types of interactions with amino acid are: any, with a backbone, interaction with sidechain, polar, hydrophobic, hydrogen bond acceptor, hydrogen bond donor, charged interaction, aromatic.

Even though it might be intuitive to look at this data as if it were 4-dimensional, it was stored in the memory in a 3-dimensional form by placing the 6 x 9 matrices adjacent to each other. If we had a protein with only three residues, name them A, B and C, a single sample would look like on figure 4.2

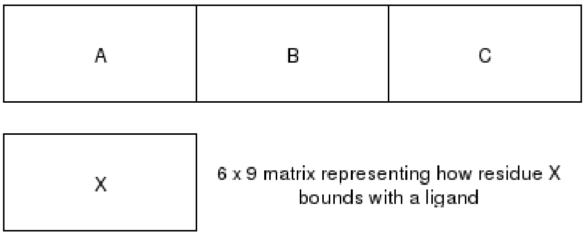

A
B
C
X
6 x 9 matrix representing how residue X bounds with a ligand

Figure 4.2: Data stored in the memory

The values constituting the dataset are discrete numbers in range 0 to 9. 0 means there is no interaction of specific kind. 1 and 2? The data was very sparse - more than 99% of all values were zeros.

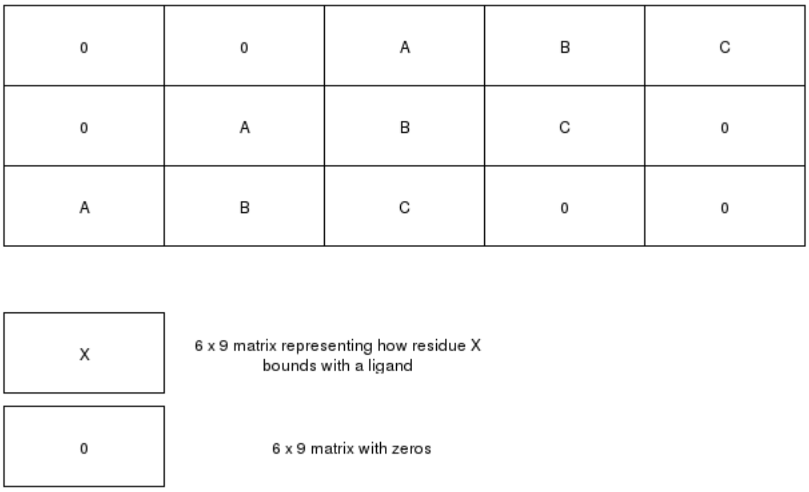
The dataset was stored in 3 files - each file contained samples of only one type: active, inactive, middle (not labled). Out of 5844 samples 2655 were labeled as active, 1945 was labeled as inactive and there were also 1244 unlabeled examples.

4.2.1 Data preprocessing

In order to extract most promising features, the data has been preprocessed. We decided to perform a preprocessing in such a way that the model would be able to build such patterns that could detect whether a bound of a specific kind was present in both adjacent residues. We expected that such approach might lead to discovering of interesting correlations.

To achieve such a form of the dataset that would enable this approach, the dataset was extended in the following way: three copies of the dataset have been created and combined, each copy stored just below the previous one. Each copy was shifted in such a way that going from top to bottom and from left to right would preserve the order of the residues. The shift forced us to either complement each row with zeros or to cut off the residues that would stick out. We decided not to cut off any residues so each of them will be present in the whole dataset the same number of times - this way no residue will be underrepresented. As a result each sample had 18 instead of 6 rows and 18 columns more.

The schema of this approach is shown on figure 4.3 which depicts the simple example of a protein with only three residues. It can be observed that a convolution window broader than 9 would be able to detect whether an interaction of some type is present in both adjacent residues while convolution window higher than 6 would be able to detect if two adjacent residues have same pharmacophore features.



0	0	A	B	C
0	A	B	C	0
A	B	C	0	0

X	6 x 9 matrix representing how residue X bounds with a ligand
0	6 x 9 matrix with zeros

Figure 4.3: Data after preprocessing.

4.3 The Architecture

In this section we will describe the type of architecture which we used for experiments. All the models we have trained were convolutional neural networks with one or two convolutional rectified linear layers. Each layer had 16 or 32 output channels. The number of layers have been chosen in such a way that the learning process will not take too much time and the data size will not be reduced too much. Rectifier activation function was used because of its good properties[Dutch paper].

The convolution window's values were chosen in such a way that would enable the model to find filters catching correlation between same types of interaction in two adjacent residues, what was described in section 4.2.1. The convolution windows were of size $(width, height) \in \{6, 8, 10, 12\} \times \{4, 5, 6, 7, 8\}$. The convolution window's strides were of size $(width, height) \in \{2, 4, 6\} \times \{2, 3\}$. If both convolution window size and stride had big values in the first layer, it could happen that the data size in the second layer would be too small to satisfy the conditions above. In such cases the convolution window's size and stride were reduced in such a way that the convolution window's size would always be smaller than the data size and the convolution window's stride would always be smaller than the convolutional window and, if it only was possible (i.e. all dimensions of the convolutional window were smaller then the corresponding dimensions of the data), small enough to enable existence of at least two "windows" in each dimension.

The shape of pooling windows was (1, 1), (2, 1) or (2, 2) and smaller by at least one than the data size in each dimension so moving the pooling window was always possible. Pooling stride was always equal to or smaller by half than the pooling window in each dimension. Max pooling was used.

The last layer of the network was a softmax layer with two neurons. It was used to classify the sample based on features extracted by the convolutional part of the network.

4.3.1 Finding best architecture

Due to many hyperparameters, there exist many models that fulfill our architecture restrictions and therefore it is not possible to train and measure the performance of all possible architectures. To find the best one we used the tree of Parzen estimators algorithm provided by a Python library - Hyperopt[hyperopt] and let it sample 20 models.

Each architecture that was tested was chosen by hyperopt module. Based on the performance of the already tested architectures hyperopt was choosing another one. Each architecture was passed from hyperopt to the objective function responsible for measuring the performance of the model.

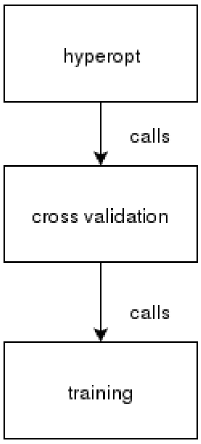


Figure 4.4: The control flow of the experiments.

Objective function for hyperopt

Each time the objective function was creating five models of a given architecture and then trained and measured the performance of each. Cross validation procedure was used to obtain different training data for each model. Validation and test set included only labeled examples because classifying unlabeled examples would not be possible. All the unlabeled samples were added only to the training set. The cross validation proceeded as follows: the active and inactive samples were split into five parts of even size. One part was becoming the validation set, one part was becoming the test set and the other three parts along with all the unlabeled examples were becoming the training set. The whole procedure was repeated five times.

Each time after training the model its performance on testing data was measured and stored. At the end the mean value of scores of all five models was returned to the hyperopt module. The score used for measuring the performance of the model was receiver operating characteristic (ROC) with Youden's J statistic. We will cover this topic in details later in this section. Algorithm 1 shows pseudo code for the cross validation procedure.

Algorithm 1 Cross validation

```
procedure OBJECTIVE_FUNC(sample, data_labeled, data_unlabeled)

    scores ← empty list

    for k ∈ range(0,K) do
        train_set, validation_set, test_set ← split(data_labeled, k)
        train_set ← train_set + data_unlabeled
        model ← build_model(sample)
        model ← train(model, train_set, validation_set)
        score ← measure_performance(model, test_set)
        scores.append(score)

    return mean(scores)
```

Training of the model

During each iteration the model provided by the hyperopt module was trained on the data computed during the iteration. After each epoch the optimal threshold was calculated on the validation set. All samples, which activation value was above the threshold, were then classified as active samples and those, which activation value was below the threshold, were classified as inactive. Keep in mind, that there were no unlabeled examples in the validation set.

To compute the optimal threshold the receiver operating characteristic (ROC) procedure was used. To measure the quality of the threshold the Youden's J statistic was used. Afterwards, the model's performance on validation data was measured. The score was the Youden's score. If the performance for this model was best until this point of time, the whole model (i.e. all its parameters along with the computed threshold) was stored on disk. As a result, at the end of the learning process the model's best version was stored on disk. This version was read and its performance on the testing set was measured and stored in the cross validation procedure. The threshold calculated during the learning phase was used. The score to measure the performance of the model was the Youden's score.

The details of the learning algorithm are covered in section 4.4.

4.4 The Learning Algorithm

The provided data included unlabeled examples. Two approaches that would enable using these examples to training the model were tested.

4.4.1 Naïve Approach

Training set was constructed in the following way: all examples were included in the training set two times: the labeled samples were labeled correctly, the unlabeled examples were once labeled as active samples, and once labeled as inactive samples. This way the impact on classification of unlabeled data was minimised while the unalabeled data could have been used to improve parameters in the convolutional part of the model.

Example:

If there were the following examples: [A, B, C] along with the following labels: [act, inact, unlabeled] then the training set would look like this: [A, A, B, B, C, C] and the labels would be [act, act, inact, inact, act, inact].

For this approach the stochastic gradient descent algorithm included in pylearn2 package (include version) was used.

4.4.2 Fancy Approach

In this approach each sample was included in the dataset only once. In order to train the model, a variation of stochastic gradient descent algorithm was written. This enabled using unlabeled examples during the learning process. The SGD implementation provided in pylearn2 (version here) was used as a base and major changes were introduced in the training function in such a way that the unlabeled examples were used to adjust the parameters of the convolutional part of the model but had no impact on the classification part. The pseudo code of this algorithm can be found below as Algorithm 2.

For labeled samples the learning process was performed with no changes. When the sample was unlabeled the network parameters were stored and then the sample was presented to the network as if it was labeled as inactive. During this process the network parameters were updated. The difference in the network parameters was stored and old parameters were restored. Afterwards, the sample was presented to the network again - this time as an active sample. The procedure was the same as before. After calculating the difference and restoring the old parameters the two vectors of differences were compared to produce the

Algorithm 2 Learning

```
procedure TRAIN(sample, label)
    if sample is unclassified then
        parameters_ on_ enter ← current_ parameters

        SGD(sample, inactive)
        diff_ vec_ 1 ← current_ parameters − parameters_ on_ enter
        current_ parameters ← parameters_ on_ enter

        SGD(sample, active)
        diff_ vec_ 2 ← current_ parameters − parameters_ on_ enter
        current_ parameters ← parameters_ on_ enter

        update_ vector = new vector of length same to difference vectors
        for el1, el2, up_ el ∈ zip(diff_ vec_ 1, diff_ vec_ 2, update_ vec) do
            if sign(el1) == sign(el2) then
                up_ el ← combination_ function(el1, el2)
            else
                up_ el ← 0

        for up_ el ∈ update_ vec do
            if up_ el is responsible for updating the classification part then
                up_ el ← 0

        current_ parameters ← parameters_ on_ enter + update_ vector
    else
        SGD(sample, label)

```

final vector of updates.

The final vector had the following properties: the elements responsible for updating the classification part of the network were all zeros, therefore the unlabeled examples had no impact on training the classification part of the model. The elements responsible for updating the convolutional part of the model were calculated in the following way: if the corresponding elements of the two vectors had the opposite sign, then the corresponding element in the final vector was zero. As a result, the unlabeled samples were used by the network to learn only these filters that were useful for classifying samples of both classes. Finally, if the corresponding elements in both vectors had the same sign, then the corresponding element in the final vector was calculated using the values of the two elements. The final value could be:

- minimum by absolute value of the two elements
- maximum by absolute value of the two elements
- mean of the two elements
- softmax mean of the two elements, i.e. having $x, y \in \mathbb{R}$ the softmax mean σ is equal to $x \cdot \frac{e^x}{e^x+e^y} + y \cdot \frac{e^y}{e^x+e^y}$.

  Remark

  It can be observed that $\frac{e^x}{e^x+e^y} \in [0,1]$ for any x, $y \in \mathbb{R}$ and that $\frac{e^x}{e^x+e^y} + \frac{e^y}{e^x+e^y} = 1$, therefore $\sigma = x \cdot \frac{e^x}{e^x+e^y} + y \cdot \frac{e^y}{e^x+e^y}$ is a convex combination of x and y, so σ will be between x and y.

Finally, all parameters corresponding to the classifying part of the network were zeroed, therefore the unlabeled examples had only impact on learning filters of the network and did not influence the classification part of the network.

Concluding: the update vector had zeros in part responsible for classification. If two corresponding values in the vectors of differences had opposite sign, then the corresponding value of the update vector was zero. All other elements were calculated using one of the combination functions.

Example

If the vectors of differences were: $[2.5, 1, -3, 5, 7]$ and $[-2, 3, -1, -7, -7, 7]$, elements 1 to 4 were responsible for updating the convolutional part, elements 5 and 6 were responsible for updating the classifying part of the network and the combination function used was minimum, then the final vector would be $[0, 3, 0, -3, 0, 0]$. Elements 5 and 6 are zeros because they are responsible for updating the classification part of the network. Elements 1 and 3 are zeros because the corresponding values in two vectors have opposite signs. Elements 2 and 4 are minimums by absolute value of the two corresponding values. This example is ilustrated in figure 4.5.

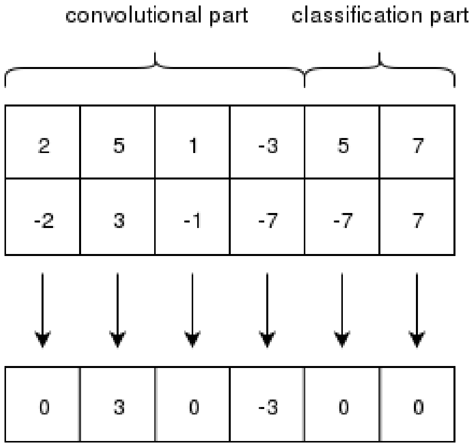


Figure 4.5: Example of the min combining function

Chapter 5

What can be improved

5.1 Everything...

...can be improved.

- data in 3D not 2D - possible new filters
- new combination functions can be tested
- other ways of finding threshold
- maybe finding two thresholds and leaving some examples unlabeled? - why is this useful?

Bibliography

[1] Yoshua Bengio, Ian J. Goodfellow, Aaron Courville - *Deep Learning*

[2] Stefan Mordalski, Igor Podolak, Andrzej J. Bojarski - *2D SIFt - a matrix of ligand-receptor interactions*

Chapter 6

Irrelevant

6.1 zero-pad methods in detail

The easiest way is to let these values stay underrepresented (in MATLAB *citation?* this methodology is called valid), another one is to enlarge the input matrix by adding zeros *at the edges* - this is called zero-pad. One can either add enough zeros for each element of the original matrix to be convolutet exactly the same number of times (in MATLAB *citation?* this methodology is called full) or take only enough zeros for the output matrix to have the same size as the input matrix (in MATLAB *citation?* this methodology is called same).

{obrazek} ilustrujący te przykłady

One can question *legitimacy* of such approach. Adding zeros invites new information into the matrix and might cause additional noise. Instead of adding zeros one might try to change a matrix into torus or instead of zeros use the values that already are present in the original matrix. The added values might be symmetrical *lustrzane odbicie.*

{obrazek} ilustrujący te przykłady

6.2 pooling

is pooling subsampling and if yes then why is polling subsampling?

6.2.1 implementationally awesome things

Fast computation

(spatial invariance) → temu nie musimy też mieć osobnych macierzy na feature w każdym miejscu - oszczędzamy pamięć na parametry (i efektywnośc, bo im więcej paramterów, tym wolniej się uczymy). Small kernel = litlle parameters.

sparse interactions

because kernel is smaller then data so it not kazdy z kazdym but some with some (sparse) → computational boost, kernel is small and moved around input - less parameters, instead of a big matrix we store a small one that runs over the data

{obrazek}

parameter sharing

Connected to the fact thet we move the convolution kernel around

{obrazek} a moze nie?

equivariant representations

equivariance - property of *what?* meaning that if the input changes that output changes the same way. $f(g(x)) = g(f(x))$ Intuition about it: detecting feature in a particuler place - feature elsewhere - we find it elsewhere. To what types of transformation is convolution equivariant and to which transformations it isn‘t?

6.2.2 Extensions

dropout/dropconnect method, activation functions for dropout, other things from the Dutch paper

6.2.3 Learning Algorithm

The problems with a classical backpropagation

diminishing gradient flow, niedouczanie się, przeuczanie się, obczaić co o tym mówił Larochelle, on to chyba jednak mówił o głębokich. Wtedy to i tak napisać i przerzucić do głębokich.

Diminishing gradient flow

co to jest, skąd się bierze, można się wesprzeć wykładami Larochelle, on poleca dużo paperów zawsze.

Backpropagation

See (Goodfellow, 2010) from Bengio

6.3 Why was this model chosen

... Having said that kernels might be used as feature extractors it's worth considering what kinds of features might be discovered in the provided data. ...